1/167/170/653/15/18/454/532. Two sum (1~4) /3 sum/4 sum/k-diff pairs

#16 - 3 sum closest
#167 - 2 sum II (the given is sorted)
#532 - K-diff pairs in an array

```
// 1 - two sum
// soln-1: one-pass hashmap, O(n) time, O(n) space.
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> m;
    for (int i = 0; i < nums.size(); ++i) {
        if (m.find(target - nums[i]) != m.end()) {
            return vector<int> {m[target - nums[i]], i};
        }
        m[nums[i]] = i;
    }
    return vector<int>{-1, -1};
}
```

```
// 167 - two sum II (input sorted)
// soln-1: binary search
vector<int> twoSum(vector<int>& numbers, int target) {
    for (int i = 0; i < numbers.size(); ++i) {
        int j = bsearch(numbers, i + 1, numbers.size() - 1, target - numbers[i]);
        if (j != -1) return vector<int>{i, j};      // there is only one soln
    }

    return vector<int>{-1, -1};
}
int bsearch(vector<int>& nums, int low, int hi, int target) {
    while (low < hi - 1) {
        int m = low + (hi - low) / 2;
        if (target == nums[m]) return m;

        target > nums[m] ? low = m : hi = m;
    }
    if (target == nums[low]) return low;
    return target == nums[hi] ? hi : -1;
}
```

```
// 170 - design a data structure supporting add/find
// soln-1: multiset
class TwoSum {
    unordered_multiset<int> _nums;
public:
    void add(int number) {
        _nums.insert(number);
    }
    bool find(int value) {
        for (int i : _nums) {
            int count = i == value - i ? 1 : 0;
            if (_nums.count(value - i) > count) {
                return true;
            }
        }
        return false;
    }
};
```

```
// 653 - two sum if input is bst
// search target exclude in-use node
bool search(TreeNode* node, TreeNode* inuse, int target) {
   if (!node || !inuse) return false;
   if (node->val == target && node != inuse) return true;
   return search(node->val > target ? node->left : node->right, inuse, target);
```

```
}
// check every pair
bool doTraversal(TreeNode* node, TreeNode* inuse, int k) {
   if (!node || !inuse) return false;

   if (doTraversal(node, inuse->left, k)) return true;
   if (doTraversal(node, inuse->right, k)) return true;
   return search(node, inuse, k - inuse->val);
}

bool findTarget(TreeNode* root, int k) {
   return doTraversal(root, root, k);
}
```

```
// 15 - three sum
// soln-1: binary search (fix 2 numbers, use binary search in the rest vector in O(n*n*lgn) time)
vector<vector<int>> threeSum(vector<int>& nums) {
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
    for (int i = 0; i < nums.size(); ++i) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;          // skip duplicates

        // find the two sums in the rest of vector
        for (int j = i + 1; j < nums.size(); ++j) {
            if (j > i + 1 && nums[j] == nums[j - 1]) continue;  // skip duplicates

            if (bsearch(nums, j + 1, nums.size() - 1, 0 - nums[i] - nums[j])) {
                ans.push_back(vector<int>{nums[i], nums[j], 0 - nums[i] - nums[j]});
            }
        }
    }
    return ans;
}

// 15 - three sum
// soln-2: two pointers (fix 1 number, then use 2-pointers with O(n^2) time)
vector<vector<int>> soln2(vector<int>& nums) {
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
    for (int i = 0; i < nums.size(); ++i) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;  // skip duplicates

        for (int j = i + 1, k = nums.size() - 1; j < k; NULL) {
            if (j > i + 1 && nums[j] == nums[j - 1]) {
                ++j;     continue;                      // skip duplicates
            }

            if (nums[i] + nums[j] + nums[k] == 0) {
                ans.push_back(vector<int>{nums[i], nums[j], nums[k]}), ++j, --k;
            } else {
                nums[i] + nums[j] + nums[k] > 0 ? --k : ++j;
            }
        }
    }
    return ans;
}
```

```
// 18 - 4 sum
// soln-2: fix 2 number, then use 2-pointers with O(n^3) time
vector<vector<int>> fourSum(vector<int>& nums, int target) {
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end());
```

```
    int n = nums.size();
    for (int i = 0; i < n - 3; ++i) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;           // skip duplicates
        if (nums[i] + nums[n - 3] + nums[n - 2] + nums[n - 1] < target) continue;   // optimize-1
        if (nums[i] + nums[i + 1] + nums[i + 2] + nums[i + 3] > target) break;      // optimize-2

        for (int j = i + 1; j < n - 2; ++j) {
            if (j > i + 1 && nums[j] == nums[j - 1]) continue;  // skip duplicates

            for (int left = j + 1, right = n - 1; left < right; NULL) {
                if (left > j + 1 && nums[left] == nums[left - 1]) {
                    ++left;  continue;                          // skip duplicates
                }

                int sum = nums[i] + nums[j] + nums[left] + nums[right];
                if (sum == target) {
                    ans.push_back({ nums[i], nums[j], nums[left], nums[right] }), ++left, --right;
                } else {
                    sum > target ? --right : ++left;
                }
            }
        }
    }
    return ans;
}
```

```
// 454 - four sum II
// soln-1: hashmap in O(n^2)
int fourSumCount(vector<int>& A, vector<int>& B, vector<int>& C, vector<int>& D) {
    int ans = 0;
    unordered_map<int, int> mp;     // <sum, count>
    for (int c : C) {
        for (int d : D) mp[c + d]++;
    }
    for (int a : A) {
        for (int b : B) ans += mp[-(a + b)];
    }
    return ans;
}
```

```
// 532 - find uniq k-diff pairs
// soln-1: hashmap
int findPairs(vector<int>& nums, int k) {
   if (k < 0) return 0;                             // absolute difference must be at least 0.
   unordered_map<int, int> mp;
   for (auto x : nums) mp[x]++;

   int ans = 0;
   for (auto p : mp) {
       if ((0 == k && p.second > 1) ||              // k = 0, [1, 1, 1, 1]
           (0 != k && mp.count(p.first - k) > 0)) {    // x + k will be found later.
           ++ans;
       }
   }
   return ans;
}
```

Ref: #15 #18 #167

3/30/76/115/159/340/904/239/480/487/567/1004/727 Substring at most... Sliding window (review)

#3/159/340- longest substring with at most 2/k distinct chars

#395 - longest substring with at least k repeating chars
#76 - min window substring (S="ADOBECODEBANC", T="ABC")
#209 - min size subarray sum
#239 - sliding window max/median (**sliding window + monotone stack/Q/deque**)
#402 - remove k digits to create min number
#424 - longest repeating char replacement
#487/1004 - max consecutive 1s (allow flip 1 or k times in 0/1 vector)
#567 - find if s2 contains the permutation of s1.
#727 - min window subsequence

#115 - distinct subsequence

substring/subarray => sliding window (+ monotone stack/Q if dealing with numbers)
subsequence => dynamic programming

```
// 3 - longest substring w/o repeating chars
// soln-1: sliding window
// the window must be starting after the last position of cur char.
int lenOfLongestSubstr(string s) {
    vector<int> pos(256, -1);                    // last position for each char
    int ans = 0;
    for (int i = 0, start = 0; i < s.length(); ++i) {
        start = max(start, pos[s[i]] + 1), ans = max(ans, i - start + 1);
        pos[s[i]] = i;
    }
    return ans;
}
```

```
// 30 - Substring with Concatenation of All Words
// soln-1: sliding window
vector<int> findSubstring(string s, vector<string>& words) {
    unordered_map<string, int> dict;
    for (auto& w : words) ++dict[w];

    vector<int> ans;
    for (int k = 0, len = words.empty() ? 0 : words[0].length(); k < len; ++k) {
        unordered_map<string, int> seen;
        for (int i = k, l = i, hit = 0; i < s.length(); i += len) {
            auto ss = s.substr(i, len);
            auto it = dict.find(ss);
            if (it == dict.end()) {
                seen.clear(), hit = 0, l = i + len;
            } else {
                ++seen[ss], ++hit;
                for (auto str = s.substr(l, len); seen[ss] > dict[ss]; str = s.substr(l, len)) {
                    --seen[str], --hit, l += len;
                }
                if (hit == words.size()) {
                    ans.push_back(l);
                    --seen[s.substr(l, len)], --hit, l += len;
                }
            }
        }
    }
    return ans;
}
```

```
// 209 - min size subarray sum (given positive integers and target)
// soln-1: sliding window in O(n) time (at most twice each number)
int minSubArrayLen(int s, vector<int>& nums) {
   int ans = INT_MAX, sum = 0;
   for (int i = 0, left = 0; i < nums.size(); ++i) {
       sum += nums[i];
       while (sum >= s) {
```

```
            ans = min(ans, i - left + 1),sum -= nums[left++];
        }
    }
    return INT_MAX == ans ? 0 : ans;
}
```

```
// 76 - min window substring
// soln-1: sliding window (almost same as #159/340)
string minWindow(string s, string t) {
   string ans;
   vector<int> inS(128), inT(128);             // occurrence in s and t
   for (char ch : t) inT[ch]++;

   for (int i = 0, left = 0, matched = 0; i < s.length(); ++i) {
       char ch = s[i];
       if (0 == inT[ch]) continue;             // char not in T

       if (++inS[ch] <= inT[ch]) ++matched;
       while (matched == t.length()) {         // greedy pop front to reduce window size
           int len = i - left + 1;
           if (ans.empty() || len < ans.length()) ans = s.substr(left, len);

           if (--inS[s[left]] < inT[s[left]] && inT[s[left]] != 0) --matched;
           ++left;
       }
   }
   return ans;
}
```

```
// 159/340 - longest substring with at most k distinct char
// soln-1: sliding window
// use sliding window to keep substring, and hashmap to track distinct chars within the substring
int lengthOfLongestSubstringTwoDistinct(string s) {
   int ans = 0, distinct = 0, k = 2;
   vector<int> occurrence(256, 0);

   for (int i = 0, left = 0; i < s.length(); ++i) {
       if (++occurrence[s[i]] == 1) {
           distinct++;
           while (distinct > k) {      // shrink the window if possible
               if (--occurrence[s[i]] == 0) distinct--;
               ++left;
           }
       }
       ans = max(ans, i - left + 1);
   }
   return ans;
}
```

```
// 395 - Longest Substring with At Least K Repeating Characters
// soln-1: backtracking (T(n) = n + T(n/2) assuming only half chars left)
int longestSubstring(const string& s, int k, int start, int end) {
    int cnt[26] = {0};
    for (int i = start; i < end; ++i) ++cnt[s[i] - 'a'];

    int ans = 0;
    for (int i = start; i < end;) {
        while (i < end && cnt[s[i] - 'a'] < k) ++i;     // exclude char if < k
        int r = i;
        while (r < end && cnt[s[r] - 'a'] >= k) ++r;    // include as long as >= k
        if (i == start && r == end) return end - start; // all chars appear more than k times

        ans = max(ans, longestSubstring(s, k, i, r));
        i = r;
```

```
    }
    return ans;
}

// 395 - Longest Substring with At Least K Repeating Characters
// soln-2: sliding window (review)
// 1. to ensure all chars appear k+ times in the window, when to shrink/expand window?
// 2. add 1 more constraint, allow d distinct chars in sliding window, help deciding when to shrink/expand.
// 3. enumerate all possible distinct chars in window, the answer would be 1 in 26 possible distinct chars.
int longestSubstring(string s, int k) {
    int ans = 0;
    for (int d = 1; d <= 26; ++d) {
        vector<int> cnt(26);
        for (int i = 0, l = 0, u = 0, lessThank = 0; i < s.length(); ++i) {
            if (++cnt[s[i] - 'a'] == 1) ++u, ++lessThank;       // 1 more unique and less than k repeated char
            if (cnt[s[i] - 'a'] == k) --lessThank;

            while (u > d) {
                if (--cnt[s[l] - 'a'] == 0) --u, --lessThank;
                if (cnt[s[l] - 'a'] == k - 1) ++lessThank;
                ++l;
            }
            if (0 == lessThank) ans = max(ans, i - l + 1);
        }
    }
    return ans;
}
```

```
// 402 - remove k digits to create smallest number ("1432219", k = 3 => "1219")
// soln-1: sliding window
// keep a sliding window with size (n-k), when we got new comer,
// from last kick position (already increasing before kick pos), if A[i] is bigger than A[i+1], then kick out i.
string removeKdigits(string num, int k) {
   if (k >= num.length()) return "0";

   int sz = num.length() - k;
   string ans(num.begin(), num.begin() + sz);
   for (int i = sz, kick = 0; i < num.length(); ++i) {
       ans.push_back(num[i]);

       while (kick + 1 < ans.length() && ans[kick] <= ans[kick + 1]) ++kick;
       ans.erase(ans.begin() + kick);
       --kick;
   }
   while (!ans.empty() && ans[0] == '0') ans.erase(ans.begin());

   return ans.empty() ? "0" : ans;
}
```

```
// 424 - longest repeating char replacement
// soln-1: sliding window + greedy (tricky)
// 1. let len be the length of sliding window, count be the max number of char in current window
// 2. shrink sliding window if len - max-len-in-sliding-window > k,
// it means we are greedy replace chars to be the one with max occurrence in current sliding window.
// 3. the size of sliding window various over time: k + occurrence
int characterReplacement(string s, int k) {
    vector<int> count(26);
    int ans = 0;
    for (int i = 0, j = 0, occurrence = 0; j < s.length(); ++j) {
        occurrence = max(occurrence, ++count[s[j] - 'A']);          // update max occurance in sliding window
        while (j - i + 1 - occurance > k) --count[s[i] - 'A'], ++i;
        ans = max(ans, j - i + 1);
    }
    return ans;
}
```

```
// 485 - max consecutive ones (brute-force)
int findMaxConsecutiveOnes(vector<int>& nums) {
   int ans = 0;
   for (int i = 0, cur = 0; i < nums.size(); ++i) {
       if (nums[i]) ans = max(++cur, ans);
       else cur = 0;
   }
   return ans;
}
```

```
// 487 - max consecutive ones if you can flip at most one 0.
// soln-1: brute-force
// cur/pre - ones in current/previous segment respectively
// total would be (cur + pre + 1).
int findMaxConsecutiveOnesII(vector<int>& nums) {
  int ans = 0;
  for (int i = 0, cur = 0, pre = 0; i < nums.size(); ++i) {
      if (nums[i]) ans = max(++cur + pre + 1, ans);
      else pre = cur, cur = 0;
  }
  return ans;
}
```

```
// 1072 - Flip Columns For Maximum Number of Equal Rows
// soln-1: hashmap
int maxEqualRowsAfterFlips(vector<vector<int>>& matrix) {
    unordered_map<string, int> mp;
    int ans = 0;
    for (auto& r : matrix) {
        string v1, v2;
        for (auto& x : r) v1.push_back(x ? '1' : '0'), v2.push_back(!x ? '1' : '0');
        ans = max(ans, ++mp[v1]);
        ans = max(ans, ++mp[v2]);
    }
    return ans;
}
```

```
// 567 - Permutation in String (if s2 contains permutation of s1)
// soln-1: sliding window
bool checkInclusion(string s1, string s2) {
   vector<int> cnt1(26), cnt2(26);
   for (auto ch : s1) ++cnt1[ch - 'a'];
   for (int left = 0, i = 0, hit = 0; i < s2.length(); ++i) {
       if (++cnt2[s2[i] - 'a'] <= cnt1[s2[i] - 'a']) ++hit;    // <= cnt1 means a valid hit
       if (i - left + 1 > s1.length()) {                       // move left if too long
           if (--cnt2[s2[left] - 'a'] < cnt1[s2[left] - 'a']) --hit;
           ++left;
       }
       if (i - left + 1 == s1.length() && hit == s1.length()) return true;
   }
   return false;
}
```

```
// 904 - Fruit Into Baskets
// Find out the longest length of subarrays with at most 2 different numbers
// soln-1: sliding window
int totalFruit(vector<int>& tree, int k = 2) {
    int ans = 0;
    unordered_map<int, int> cnt;
    for (int l = 0, i = 0; i < tree.size(); ++i) {
        cnt[tree[i]]++;
        while (cnt.size() > k) {
            if (--cnt[tree[l]] == 0) cnt.erase(tree[l]);
            ++l;
        }
        ans = max(ans, i - l + 1);
```

```
    }
    return ans;
}
```

```
// 995 - Minimum Number of K Consecutive Bit Flips (review)
// soln-1: greedy + sliding window + queue
// 1. naive soln would be in O(Kn) time: greedy flip k bits after me.
// 2. key observation:
//    - even flips ahead me means no flip, if cur = 0, then need a flip
//    - odd flips ahead me and cur = 1, then need a flip
// 3. maintain flipped (up to k size) ahead me to decide if we need to flip current.
// 4. space improvement: modify the input to get O(1) space.
int minKBitFlips(vector<int>& A, int K) {
    int ans = 0;
    queue<int> Q;           // up to k size flipped position
    for (auto i = 0; i < A.size(); ++i) {
        if (!Q.empty() && i - K >= Q.front()) Q.pop();  // pop out if window is too big

        bool flipped = (1 == Q.size() % 2);
        if ((flipped && 1 == A[i]) || (!flipped && 0 == A[i])) {
            if (i + K > A.size()) return -1;
            ++ans, Q.push(i);
        }
    }
    return ans;
}
```

```
// 992 - Subarrays with K Different Integers
// soln-1: sliding window
// ex. [1, 2, 1, 2, 3], k = 2
//      ^  ^  ^
//      l  r  i
// when hit A[2], it won't increase distinct num w/i window, but we can greedy increase r from index 0->1
// so we keep 2 window: [l..i], [r..i], make distinct in [l..i] = k, distinct in [r..i] = k - 1
// then r-l would be newly added subarray for current number.
int subarraysWithKDistinct(vector<int>& A, int K) {
   int ans = 0;
   unordered_map<int, int> cnt1, cnt2;
   for (int i = 0, l = 0, r = 0, d1 = 0, d2 = 0; i < A.size(); ++i) {
       if (1 == ++cnt1[A[i]]) d1++;
       if (1 == ++cnt2[A[i]]) d2++;

       while (d1 > K) {
           if (0 == --cnt1[A[l++]]) --d1;
       }
       while (d2 >= K) {
           if (0 == --cnt2[A[r++]]) --d2;
       }
       if (d1 == K) ans += (r - l);
   }
   return ans;
}
// 992 - Subarrays with K Different Integers
// soln-2: sliding window (#159/340)
// transform the question to subarray with at most k distinct integers, then
// at-most-k-distinct = at-most-(k-1)-distinct + exactly-k-distinct
int atMostK(vector<int>& A, int k) {
   int ans;
   unordered_map<int, int> cnt;
   for (int l = 0, i = 0, d = 0; i < A.size(); ++i) {
       if (1 == ++cnt[A[i]]) ++d;
       while (d > k) {
           if (--cnt[A[l++]] == 0) --d;
       }
       ans += (i - l + 1);
   }
```

```
    return ans;
}
```

```
// 1004 - max consecutive ones III
// soln-1: sliding window + greedy
// 1. length-of-sliding-window = ones-in-sliding-window + K
// 2. similar to #424
int longestOnes(vector<int>& A, int K) {
    int ans = 0;
    for (int l = 0, i = 0, ones = 0; i < A.size(); ++i) {
        ones += A[i];                              // count ones in sliding window
        if (i - l + 1 > ones + K) ones -= A[l++];  // reduce length of sliding window
        ans = max(ans, i - l + 1);
    }
    return ans;
}
```

Ref: #30 #159 #239 #340

4. Median of two sorted arrays

There are two sorted arrays **nums1** and **nums2** of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

#215 - kth largest

```
// 4 - soln-1: binary search (quick selection) in O(lg(m+n))
double findMedianSortedArrays(vector<int>& A, vector<int>& B) {
    int m = A.size(), n = B.size();
    int l = (m + n + 1) >> 1, r = (m + n + 2) >> 1;    // for odd number, L == R, otherwise L + 1 = R
    return (getkth(A.data(), m, B.data(), n, l) + getkth(A.data(), m, B.data(), n, r)) / 2.0f;
}

// get Kth smallest from two sorted array in O(lgk), where k = (m+n)/2
int getkth(int A[], int m, int B[], int n, int k) {
    if (m > n)  return getkth(B, n, A, m, k);      // let m <= n
    if (m == 0) return B[k - 1];                   // hit boundary and don't bother check n because m < n
    if (k == 1) return min(A[0], B[0]);            // return the smallest between A and B

    int i = min(m, k / 2), j = min(n, k / 2);
    // safe to get rid of either A[0..i] or B[0..j], whoever is smaller part
    // to reduce problem size (k/2), but not both
    if (A[i - 1] > B[j - 1]) return getkth(A, m, B + j, n - j, k - j);

    return getkth(A + i, m - i, B, n, k - i);
}
```

```
// 4 - soln-2: binary search in O(lg(min(m, n)) better than soln-1.
// Essentially, we want to find partition-of-x + partition-of-y = (m + n + 1) / 2, such that
// 1. all the left of partition-of-x and partition-of-y <= median,
// 2. all the right of partition-of-x >= median, all the right of partition-of-y >= median
//
// A: x1, x2, | x3, x4, x5, x6, x7       ← x has k = 2 numbers on the left
// B: y1, y2, y3, y4, y5, y6, | y7, y8   ← y must have (m+n+1)/2 - k = 6 numbers on the left.
// A[i] < B[j+1]
//    B[j] < A[i+1], median FOUND as abs(L-R) <= 1. m = (max(A[i], B[j]) + min(A[i+1], B[j+1])/2 if L == R
//    B[j] >= A[i+1], means i is too short, move right: i += (n-i)/2
// A[i] >= B[j+1]
//    A[i+1] >= B[j], means i is too long, move left: i -= i/2
//    A[i+1] < B[j], impossible
//
```

```
// * value-range based binary search (search in between min(x1, y1) and max(xn, ym) then count smallest)
//   will NOT work because of dups.
```

Ref: #347

5/131/132. Palindromic substring/sequence... (review)

Given a string *S*, find the longest palindromic substring in *S*. You may assume that the maximum length of *S* is 1000, and there exists one unique longest palindromic substring.

Solution:
There is Manacher's Algorithm for this question in O(n) time. It seems too complicated to implement. Here is O(n^2) soln.

#115 - distinct subsequence (similar DP idea)
#131/2 - palindrome partition II (similar DP idea)
#214 - shortest palindrome (longest palindromic substring from head/tail)
#267 - palindrome permutation (enumerate only if it could)
#336 - palindrome pairs (hashmap)
#516 - longest palindrome subsequence (similar DP idea)
#564 - find the closest palindrome (enumerate candidates)
#583/712 - delete operation for 2 strings (similar DP idea)
#647 - palindromic substring (extend and count, DP)
#664 - strange printer (DP, bottom-up, $O(n^3)$)
#718 - max length of repeated subarray (= LCS, similar DP idea, but use O(n) space)
#730 - count diff. palindromic subsequences (a little complicated dp)

```
// soln-1: brute force extend from current index in O(n^2) time
// extend string is a practical idea for palindrome question
string longestPalindrome(string s) {
   int start = 0, len = 0;
   for (int i = 0; i < s.length(); nullptr) {
       int l = i;
       while (s[l] == s[i]) ++i;      // i points to char which not equals to left
       int r = i - 1;
       while (l - 1 >= 0 && r + 1 < s.length() && s[l - 1] == s[r + 1]) --l, ++r;

       if (r - l + 1 > len) {
           len = r - l + 1;
           start = l;
       }
   }
   return s.substr(start, len);
}

// soln-2: dynamic programming (not good as soln-1 because of space complexity, just for DP practice)
// dp(i, j) is palindrome if s[i] == s[j] && dp(i+1, j-1)
// because of the dependency, i has to go down and j has to go up.
string longestPalindrome(string s) {
   int n = s.length(), start = 0, len = 0;

   vector<vector<bool>> dp(n, vector<bool>(n));

   for (int i = n - 1; i >= 0; --i) {                 // i has to go down as we depend on i+1
       dp[i][i] = true;                               // dp(i, i) must be palindrome with length 1.
       for (int j = i; j < n; ++j) {                  // j has to go up as we depend on j-1
           dp[i][j] = s[i] == s[j];
           if (i + 1 < j - 1 && dp[i][j]) {           // consider dp(i+1, j-1) only if: have space and s[i] =
s[j]
               dp[i][j] = dp[i][j] && dp[i + 1][j - 1];
           }
```

```
            if (dp[i][j] && j - i + 1 > len) {          // update result
                start = i, len = j - i + 1;
            }
        }
    }
    return s.substr(start, len);
}
```

```
// 131 - Palindrome Partitioning
// soln-1: brute-force backtracking then validate
vector<vector<string>> partition(string s) {
    vector<vector<string>> ans;
    vector<string> p;
    helper(0, s, p, ans);

    return ans;
}

void helper(int start, string s, vector<string>& p, vector<vector<string>>& ans) {
    if (start >= s.length() && p.size()) ans.push_back(p);

    for (int i = start; i < s.length(); ++i) {
        if (!isPalindrome(s, start, i)) continue;

        p.push_back(s.substr(start, i + 1 - start));
        helper(i + 1, s, p, ans);
        p.pop_back();
    }
}
```

```
// 132 - Palindrome Partitioning II (min cut to make palindrome)
// soln-1: dynamic programming
// let dp(n) be the min cut for string length n, pal(i, j) true if s[i..j] is palindrome.
// dp(j) = min{dp(i) + 1 | for each palindrome s[i+1..j]}, exmaple:
// "x y z a b c b a"
// .    ^--- f("xyzabcba") = f("xyz") + 1, because "abcba" is palindrome, only 1 cut is needed after z.
// "abcba" is palindrome iff 'a' == 'a' && "bcb" is palindrome.
int minCut(string s) {
    vector<int> dp(s.length());
    vector<vector<bool>> pal(s.length(), vector<bool>(s.length()));

    pal[0][0] = true;
    for (int j = 1; j < s.length(); ++j) {
        dp[j] = dp[j - 1] + 1, pal[j][j] = true;

        for (int i = 0; i < j; ++i) {
            if (s[i] == s[j] && (i + 1 == j || pal[i + 1][j - 1])) {
                pal[i][j] = true;
                dp[j] = min(dp[j], i == 0 ? 0 : dp[i - 1] + 1);
            }
        }
    }
    return dp.back();
}
```

Ref: #214 #266 #336

7/8/65/166. Reverse integer/String to integer (atoi)/Valid numbers/Fraction

```
// 7 - reverse integer
int reverse(int x) {
    int sign = x > 0 ? 1 : -1;
    int ans = 0;
    for (x = abs(x); x > 0; x /= 10) {
```

```
        if (ans * 10 > INT_MAX - x % 10 || ans > INT_MAX / 10)  return 0;   // overflow

        ans = ans * 10 + x % 10;
    }

    return ans * sign;
}
```

```
// 8 - string to integer - brute-force
// a few cases:
// 0x => 0
// -0123 => -123
// 1.2 => 1
// 1 2 3 => 1                 // space ahead of string
// abc => 0                   // not a valid string, return 0
// -999999999999 => INT_MIN   // overflow
int myAtoi(const char* s) {
    if (s == NULL)  return 0;

    long sign = 1, ans = 0;
    while (*s == ' ') ++s;
    if (*s == '+') ++s;
    else if (*s == '-') { sign = -1; ++s; }

    for (*s; *s >= '0' && *s <= '9'; ++s) {
        ans = ans * 10 + *s - '0';
        if (sign > 0 && ans > INT_MAX)     return INT_MAX;
        if (sign < 0 && -ans < INT_MIN)    return INT_MIN;
    }
    return sign * ans;
}
```

```
// 065 - valid number
// soln-1: brute-force with lots of cases
// True - "0", " 0.1", "2e10", " 2e-9", " 0504e+6", "+1.e+5"
// False - "abc", "1 a", "-e10", "+e1", "1+e", " ", "e9", "4e+", " -.", ".e1", "3.e", ". 1"
//
bool isNumber(string s) {
   // TODO
}
```

```
// 166 - Fraction to Recurring Decimal
// soln-1: hashmap (check if we see the recurring numerator)
string fractionToDecimal(int numerator, int denominator) {
    if (!numerator) return "0";

    long n = numerator, d = denominator;
    string ans;
    if (n > 0 ^ d > 0) ans += "-";
    n = abs(n), d = abs(d), ans += to_string(n / d);
    if (n % d == 0) return ans;

    ans += ".";
    unordered_map<long, int> mp;
    for (auto x = n % d; x; x %= d) {   // ex. n = 10, d = 3 => r = 1
        if (mp.count(x)) {
            ans.insert(mp[x], 1, '('), ans.push_back(')');
            return ans;
        }
        mp[x] = ans.length(), x *= 10, ans += to_string(x / d);
    }
    return ans;
}
```

Ref:

9. Palindrome number

determine whether an integer is a palindrome. do this without extra space.

```
// soln-1: reverse the whole digits
bool isPalindrome(int x) {
    int y = 0;
    for (int i = x; i > 0; i /= 10) {
        y = y * 10 + i % 10;
    }

    return x == y;
}

// soln-2: reverse half of integer digits (clever)
bool isPalindrome(int x) {
    if (x != 0 && x % 10 == 0)  return false;

    int half = 0;
    for (int i = x; x > half; x /= 10) {
        half = half * 10 + x % 10;
    }

    return x == half || x = half / 10;
}

// recursive soln:  it is not constant space, but idea is interesting.
bool isPalindrome(int x) {
    if (x < 0) return false;
    return helper(x, x);
}

bool helper(int x, int& y) {
    if (x == 0) return true;

    if (helper(x / 10, y) && (x % 10 == y % 10)) {
        y /= 10;
        return true;
    }
    return false;
}
```

Ref: #234

10. Regular expression matching

implement regular expression matching with support for '.' and '*'.
'.' matches any single character,
'*' match zero or more ***of preceding element***.
The matching should cover the ***entire*** input string (not partial).
E.g. see here.
isMatch("aab", "c*a*b")   -> true (0 c, 2 a)
isMatch("aab", "c*d*a*b") -> true (0 c, 0 d, 2 a)
isMatch("aab", ".*")   -> true (".*" could be "...")
isMatch("caca", "ca*")   -> false (match c, ca, caa, ...)

Solution:
Hard question.
soln-1: recursive soln. (not practical, just for the idea)
the key idea of recursive is to understand when `p = "x*"`, it means to match 0 or more character from s.

1) if match `0` character, we need to proceed p by two character, which means "x*" is consumed;
2) if match `1` character, it means:
    a) "x*" is still there
    b) 1 character from s has been matched. So the problem size reduces.

if p is other pattern, such as "xyz*", which indicates p[2] is not '*' and p[0] has to match with s[0]. So we advance p and s one character respectively under the condition that p[0] = s[0] || p[0] = '.'

soln-2: dynamic programming
similar to Edit Distance question, we organize the 2d matrix dp[row][col] = length(p) * length(s).
dp[i][j] gives the matching result for p[0...i) with s[0...j). There is nasty alignment problem, so dp[i][j] stores p[i-1] vs. s[j-1].

For p[i] = '*', we have two choices:

1. no match for the previous character, acting like 'x*' does not exist in p, `dp[i][j] == dp[i-2][j]`
2. match one or more characters, `(p[i-1] = s[j] || p[i-1] = '.') && dp[i][j-1]`
   `dp[i][j] = dp[i-2][j] || (dp[i][j-1] && (p[i-1] = s[j] || p[i-1] = '.'))`

For p[i] != '*', it depends on dp[i - 1][j - 1] and p[i] == s[j]:
`dp[i][j] = dp[i-1][j-1] && (p[i] == s[j] || p[i] == '.')`

Corner case - the leftmost column, which means p is trying to match empty s,

1. p[i-1] != '*', dp[i][0] = false
2. p[i-1] = '*', dp[i][0] = dp[i-2][0], which means current 2 characters match empty string.

Running time: O(nm), space O(nm) - could be improved, since we only depend on dp[i-2][j], two rows is enough.

#44 - wildcard matching

```
// soln-1: recursive soln
// if p = "x*", since '*' means 0 or more than once character, so it could match empty string or "xx*"
//              p advances 2 characters if match empty string.
//              s advances 1 character if s[0] == p[0] || p[0] == '.'
// else, to advance p and s, they have to match for the first character, that is, p[0] == s[0] || p[0] == '?'
//
bool isMatch(string s, string p) {
    if (p.empty())    return s.empty();

    if (p.length() > 1 && '*' == p[1]) {
        // x* matches empty string, p advances 2 characters.
        if (isMatch(s, p.substr(2))     return true;

        // or at least one character: x* -> xx*, s advance 1 character, problem size reduced.
        return (!s.empty() && (s[0] == p[0] || '.' == p[0]) && isMatch(s.substr(1), p));
    }

    // match one character: s and p advance 1 character to reduce problem size.
    return !s.empty() && (s[0] == p[0] || '.' == p[0]) && isMatch(s.substr(1), p.substr(1));
}

// soln-2: dynamic programming
// if p = "x*", p could match empty string or "xx*"
//              dp[i][j] = dp[i-2][j], for empty string, equal to p[i - 2].
//              dp[i][j] = dp[i][j-1] && (p[i-1] == s[j] || p[i-1] == '.'), for 1 or more characters match
// else, dp[i][j] = dp[i-1][j-1] && (p[i] == s[j] || p[i] == '.')
//
// running time: O(m.n), space: O(m.n) - could be improved.
bool isMatch(string s, string p) {
    int m = s.size(), n = p.size();

    vector<vector<bool>> dp(n + 1, vector<bool>(m + 1, false));
    dp[0][0] = true;

    for (int i = 1; i <= n; i++) {
        dp[i][0] = (i > 1 && p[i - 1] == '*' ? dp[i - 2][0] : false);    // corner case: only match empty string

        for (int j = 1; j <= m; ++j) {
            if (p[i - 1] == '*') {
                // match one or more characters: check previous result at dp[i][j - 1]
                dp[i][j] = (dp[i][j - 1] && (p[i - 2] == s[j - 1] || p[i - 2] == '.'));    // BE CAREFUL!
```

```
                // match empty string, acting like x* does not exist: dp[i - 2][j]
                if (i > 1) dp[i][j] ||= (dp[i - 2][j]);
            } else {
                // up-left corner and exact match
                dp[i][j] = dp[i - 1][j - 1] && (p[i - 1] == s[j - 1] || p[i - 1] == '.');
            }
        }
    }

    return dp[n][m];
}
```

Ref: #44

11. Container with most water

Given n non-negative integers $a_1, a_2, ..., a_n$, where each represents a point at coordinate (i, a_i). n vertical lines are drawn such that the two endpoints of line i is at (i, a_i) and $(i, 0)$. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container.

Solution:
Very clear explained by Stefan:
"1. Start by evaluating the widest container, using the first and the last line.
2. All other possible containers are less wide, so to hold more water, they need to be higher.
3. Thus, after evaluating that widest container, skip lines at both ends that don't support a higher height.
4. Then evaluate that new container we arrived at. Repeat until there are no more possible containers left."

```
// 2-pointers, skip lines that less than current lower height.
int maxArea(vector<int>& height) {
    int ans = 0;
    for (int left = 0, right = height.size() - 1; left < right; NULL) {
        int w = right - left, h = min(height[left], height[right]);
        ans = max(ans, w * h);

        while (left < right && height[left] <= h) ++left;
        while (left < right && height[right] <= h) --right;
    }
    return ans;
}
```

Ref: #42 #221 #238

12/13/38/273. Integer <-> Roman/Count and say/Integer to english words

```
// 012 - integer to roman
string intToRoman(int num) {
   vector<string> M{"", "M", "MM", "MMM"};
   vector<string> C{"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
   vector<string> X{"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
   vector<string> I{"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
   return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];
}

// 013 - roman to integer
// soln-1: backwards scan
int romanToInt(string s) {
   unordered_map<char, int> T = {{'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}};
   int sum = T[s.back()];
   for (int i = s.length() - 2; i >= 0; --i) {
      (T[s[i]] < T[s[i + 1]]) ? sum -= T[s[i]] : sum += T[s[i]];
   }
```

```
    return sum;
}
```

```
// 038 - count and say
// soln-1: brute-force
string nextRun(const string& str) {
    string ans;
    for (int i = 0; i < str.length(); ++i) {
        int count = 0;
        for (int j = i + 1; j < str.length() && str[i] == str[j]; ++j) ++count;

        ans += to_string(count + 1) + str[i];
        i += count;
    }
    return ans;
}

string countAndSay(int n) {
    string ans = "1";
    while (--n) ans = nextRun(ans);
    return ans;
}
```

```
// 273 - integer to english words
// soln-1: recursion
vector<string> t20{"Zero ", "One ", "Two ", "Three ", "Four ", "Five ", "Six ", "Seven ", "Eight ",
                   "Nine ", "Ten ", "Eleven ", "Twelve ", "Thirteen ", "Fourteen ", "Fifteen ", "Sixteen ",
                   "Seventeen ", "Eighteen ", "Nineteen "};
vector<string> t100{"", "Ten ", "Twenty ", "Thirty ", "Forty ", "Fifty ", "Sixty ", "Seventy ", "Eighty ",
                    "Ninety "};

string h(int num) {
    const int thousand = 1000, million = thousand * 1000, billion = million * 1000;
    if (num >= billion) return h(num / billion) + "Billion " + (num % billion ? h(num % billion) : "");
    if (num >= million) return h(num / million) + "Million " + (num % million ? h(num % million) : "");
    if (num >= thousand) return h(num / thousand) + "Thousand " + (num % thousand ? h(num % thousand) : "");
    if (num >= 100) return t20[num / 100] + "Hundred " + (num % 100 ? h(num % 100) : "");
    if (num > 20) return t100[num / 10] + (num % 10 ? h(num % 10) : "");
    return t20[num];
}

string numberToWords(int num) {
    auto str = h(num);
    str.pop_back();
    return str;
}

// read backwards without reversing numbers, ex. "1234" => "Four Thousand Three Hundred Twenty One"
string rev(string str) {
    while (!str.empty() && str.back() == '0') str.pop_back();
    if (str.length() > 9) {             // >9-digits
        return rev(str.substr(9)) + "Billion " + rev(str.substr(0, 9));
    } else if (str.length() > 6) {      // 9~7-digits
        return rev(str.substr(6)) + "Million " + rev(str.substr(0, 6));
    } else if (str.length() > 3) {      // 4-digits
        return rev(str.substr(3)) + "Thousand " + rev(str.substr(0, 3));
    } else if (str.length() > 2) {      // 3-digits
        return t20[stoi(str.substr(2))] + "Hundred " + rev(str.substr(0, 2));
    } else if (str.length() > 1) {      // 2-digits
        if (str[1] >= '2') return t100[str[1] - '0'] + rev(str.substr(0, 1));
        return t20[10 + str[0] - '0'];  // 11/21/31/41/51/..
    } else if (str.length() > 0) {      // 1-digit
        return t20[str[0] - '0'];
    }
    return "";
}
```

Ref:

16/259/923/26/27/80/283/... Two pointers

```
// 16 - 3 sum closest to target
// soln-1: two pointers
int threeSumClosest(vector<int>& nums, int target) {
   sort(nums.begin(), nums.end());

   int ans = nums[0] + nums[1] + nums[2];
   for (int i = 0; i < nums.size(); ++i) {
       if (i > 0 && nums[i] == nums[i - 1]) continue;

       for (int left = i + 1, right = nums.size() - 1; left < right; NULL) {
           int candidate = nums[i] + nums[left] + nums[right];
           if (candidate == target) return target;
           if (abs(target - candidate) < abs(target - ans)) ans = candidate;

           candidate > target ? --right : ++left;
       }
   }
   return ans;
}
```

```
// 259 - 3 sum smaller (return # of triplets which sum < target)
// soln-1: two pointers in O(n^2) time
int threeSumSmaller(vector<int>& nums, int target) {
    sort(nums.begin(), nums.end());

    int ans = 0;
    for (int i = 0; i < nums.size(); ++i) {
        for (int j = i + 1, k = nums.size() - 1; j < k; NULL) {
            if (nums[i] + nums[j] + nums[k] < target) {
                ans += (k - j); // nums[k] is the largest, must be (k-j) triplets satisfying too
                ++j;
            } else {
                --k;
            }
        }
    }
    return ans;
}
```

```
// 923 - 3Sum With Multiplicity
// soln-1: hashmap (math) with 3 cases:
// 1. i = j = k
// 2. i = j != k (OR i != j=k)
// 3. i < j < k
int threeSumMulti(vector<int>& A, int target) {
    unordered_map<int, long> c;
    for (int a : A) c[a]++;
    long ans = 0, mod = 1e9+7;
    for (auto i1 : c) {
        for (auto i2 : c) {
            auto i = i1.first, j = i2.first, k = target - i - j;
            if (c.find(k) == c.end()) continue;
            if (i == j && j == k) ans += (i1.second - 2) * (i1.second - 1) * i1.second / 6;     // n-choose-3
            else if (i != j && j == k) ans += i1.second * (i2.second - 1) * i2.second / 2;
            else if (i < j && j < k) ans += i1.second * i2.second * c[k];
        }
    }
    return ans % mod;
}
```

```
// 026 - remove dups from sorted array
// soln-1: two pointers
int removeDuplicates(vector<int>& nums) {
   if (nums.size() < 2) return nums.size();
   int ans = 1;
   for (int i = 1; i < nums.size(); ++i) {
       if (nums[i] != nums[i - 1]) nums[ans++] = nums[i];
   }
   return ans;
}
```

```
// 027 - remove all instances of given value and return new length
// soln-1: two pointers
int removeElements(vector<int>& nums, int val) {
   int left = 0, right = nums.size() - 1;
   while (left <= right) {
       while (left <= right && nums[right] == val) --right;
       while (left <= right && nums[left] != val) ++left;
       if (left == right) return right;
       if (left < right) swap(nums[left++], nums[right--]);
   }
   return right + 1;
}
```

```
// 080 - remove dups from sorted array (dups are allowed at most twice)
// soln-1: two pointers
int removeDuplicates(vector<int>& nums, int k = 2) {
   int ans = 0;
   for (int x : nums) {
       if (ans < k || x > nums[ans - k]) nums[ans++] = x;
   }
   return ans;
}
```

```
// 283 - move zeros
// soln-1: two pointers
void moveZeroes(vector<int>& nums) {
   int zero = 0;
   while (zero < nums.size() && nums[zero] != 0) zero++;

   int nz = zero + 1;
   while (nz < nums.size() && nums[nz] == 0) nz++;
   while (zero < nz && nz < nums.size()) {
       swap(nums[zero++], nums[nz++]);
       while (nz < nums.size() && nums[nz] == 0) nz++;
   }
}
```

```
// 1089 - Duplicate Zeros (if see 0, insert an extra one after current position)
void duplicateZeros(vector<int>& arr) {
    int l = 0, shift = 0, n = arr.size();
    for (; l + shift < n; ++l) shift += (arr[l] == 0);

    for (--l; shift > 0; --l) {
        if (l + shift < n) arr[l + shift] = arr[l];
        if (arr[l] == 0) --shift, arr[l + shift] = 0;
    }
}
```

```
// 826 - Most Profit Assigning Work
// soln-1: two pointers
int maxProfitAssignment(vector<int>& difficulty, vector<int>& profit, vector<int>& worker) {
   vector<pair<int, int>> jobs;
   for (auto i = 0; i < difficulty.size(); ++i) jobs.push_back({difficulty[i], profit[i]});
```

```
    sort(jobs.begin(), jobs.end()), sort(worker.begin(), worker.end());

    int ans = 0, i = 0, maxp = 0;
    for (auto& cap : worker) {
        // hard jobs do not necessarily mean high profit, take max-profit as difficulty goes up.
        while (i < profit.size() && cap >= jobs[i].first) maxp = max(jobs[i++].second, maxp);
        ans += maxp;
    }
    return ans;
}
```

```
// 838 - Push Dominoes
// soln-1: two pointers (brute-force)
string pushDominoes(string D) {
    for (int i = 0, pre = 0; i < D.length(); ++i) {
        if ('L' == D[i]) {
            if (D[pre] == 'R') {    // two pointers update left and right
                for (int l = pre, r = i; l < r; ++l, --r) D[l] = 'R', D[l + 1] = '.', D[r] = 'L';
            } else {
                for (int r = i - 1; r >= pre && D[r] == '.'; --r) D[r] = 'L';
            }
            pre = i;
        } else if ('R' == D[i]) {
            pre = i;
            while (i + 1 < D.length() && D[i + 1] == '.') D[i + 1] = 'R', ++i;
        }
    }
    return D;
}
```

```
// 845 - Longest Mountain in Array
// soln-1: two pointers
int longestMountain(vector<int>& A) {
    int n = A.size(), ans = 0;
    vector<int> L(n), R(n);
    for (int i = 1; i < n; ++i) {
        if (A[i - 1] < A[i]) L[i] = L[i - 1] + 1;
        if (A[n - i - 1] > A[n - i]) R[n - i - 1] = R[n - i] + 1;
    }
    for (int i = 1; i < n - 1; ++i) {
        if (L[i] && R[i]) ans = max(ans, 1 + L[i] + R[i]);
    }
    return ans;
}
```

Ref:

17. Letter combinations of a phone number

given a digit string, return all possible letter combinations that the number could represent.
E.g. given “23”, return [ad, ae, af, bd, be, bf, cd, ce, cf].

Solution:
A classic backtracking question:
“Backtracking is a systematic way to iterate through all the possible configurations of a search space. … we must generate each one possible configuration exactly once…. Backtracking constructs a tree of partial solutions…. the process of constructing the solutions corresponds exactly to doing a dfs traversal of the backtrack tree. … yields a natural recursive implementation of the basic algorithm.” - TAMD2e

The difficult part of a backtracking question is to generate the candidate. Time complexity for typical backtracking question is **exponential**, since we need to enumerate all possibilities. For example, subset problem has O(2^n) time complexity because each position has two states: either in the subset or not.

```
// time complexity: O(2^n)
```

```
vector<string> letterCombinations(string digits) {
    vector<string> ans;
    string str;
    bt(digits, 0, str, ans);

    return ans;
}

void bt(string& digits, int k, string& str, vector<string>& ans) {
    if (k == digits.length()) {
        if (!str.empty())   ans.push_back(str);
        return;
    }

    const vector<string> c = { "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz" };

    string& candidates = c[digits[k] - '0'];
    if (candidates.empty()) {
        bt(digits, k + 1, str, ans);        // skip empty string and keep going
    } else {
        for (char ch : candidates) {
            str.push_back(ch);
            bt(digits, k + 1, str, ans);    // depth-first search
            str.pop_back();
        }
    }
}
```

Ref: #22

19/21/23/24/25/88. Merge 2/k sorted lists/array

```
// 19 - remove nth node from end of list
// soln-1: sliding window in 1-pass
ListNode* removeNthFromEnd(ListNode* head, int n) {
   ListNode dummy(0);  dummy.next = head;
   ListNode* cur = &dummy;
   for (int i = 0; cur && i < n; ++i) cur = cur->next;
   if (cur) {
       ListNode* pre = &dummy;
       while (cur->next) cur = cur->next, pre = pre->next;
       pre->next = pre->next->next;
   }
   return dummy.next;
}
```

```
// 21 - merge 2 sorted lists
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
   ListNode dummy(0), *cur = &dummy;
   while (l1 && l2) {
       if (l1->data < l2->data) {
           cur->next = l1, l1 = l1->next;
       } else {
           cur->next = l2, l2 = l2->next;
       }
       cur = cur->next;
   }
   if (l1) cur->next = l1;
   if (l2) cur->next = l2;
   return dummy.next;
}
```

```cpp
// 23 - merge k sorted lists
// soln-2: priority queue
ListNode* mergeKLists(vector<ListNode*>& lists) {
    auto cmp = [](const ListNode* lhs, const ListNode* rhs) { return lhs->val > rhs->val; };
    priority_queue<ListNode*, vector<ListNode*>, cmp> h(cmp);

    ListNode dummy(0), *tail = &dummy;
    for (ListNode* n : lists)   if (n) h.push(n);

    while (!h.empty()) {
        ListNode* n = h.top();  h.pop();
        tail->next = n, tail = tail->next;

        if (n->next) h.push(n->next);
    }
    return dummy.next;
}
```

```cpp
// 24. Swap Nodes in Pairs
ListNode* swapPairs(ListNode* head) {
    if (nullptr == head || nullptr == head->next) return head;
    auto nh = head->next, rest = head->next->next;
    head->next->next = head, head->next = swapPairs(rest);
    return nh;
}
```

```cpp
// 25. Reverse Nodes in k-Group
ListNode* reverseKGroup(ListNode* head, int k) {
    if (nullptr == head || nullptr == head->next) return head;

    ListNode* tail = head;
    for (int i = 1; i < k && tail; ++i) tail = tail->next;
    if (nullptr == tail) return head;   // not long enough, remain as-is.

    ListNode* rest = tail->next, *cur = head, *pre = nullptr;
    while (cur != rest) {
        auto next = cur->next;
        cur->next = pre, pre = cur, cur = next;
    }
    head->next = reverseKGroup(rest, k);
    return tail;
}
```

```cpp
// 88 - merge 2 sorted array into the later
// soln-1: two pointers starting from back
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
   int tail = m + n;
   while (m > 0 && n > 0) nums1[--tail] = (nums1[m - 1] > nums2[n - 1]) ? nums1[--m] : nums2[--n];
   while (n > 0) nums1[--tail] = nums2[--n];
}
```

Ref:

20/22/856/921. Validate/generate parentheses/Parentheses score

```cpp
// 21 - validate parentheses
// soln-1: stack
bool isValid(string s) {
    stack<char> stk;
    for (char c : s) {
        switch (c) {
        case '(':
        case '[':
```

```
        case '{':
            stk.push(c);    break;
        case ')':
            if (stk.empty() || stk.top() != '(')    return false;
            stk.pop();  break;
        case ']':
            if (stk.empty() || stk.top() != '[')    return false;
            stk.pop();  break;
        case '}':
            if (stk.empty() || stk.top() != '{')    return false;
            stk.pop();  break;
        }
    }
    return stk.empty();
}
```

```
// 22 - generate parentheses
// soln-1: backtracking
void helper(int left, int right, int n, string parenth, vector<string>& ans) {
   if (parenth.length() == n * 2) {
       ans.push_back(parenth);     return;
   }
   if (left < n)       helper(left + 1, right, n, parenth + "(", ans);
   if (right < left)   helper(left, right + 1, n, parenth + ")", ans);
}

vector<string> generateParenthesis(int n) {
   vector<string> ans;
   helper(0, 0, n, "", ans);
   return ans;
}
```

```
// 678 - Valid Parenthesis String (* can be viewed as '(', ' ', ')')
// soln-1: backtracking (for *, replace with '(', '', ')')
bool h(string& s, int start, int cnt) {
    if (cnt < 0) return false;

    if (start == s.length())  return 0 == cnt;
    else if ('(' == s[start]) return h(s, start + 1, cnt + 1);
    else if (')' == s[start]) return h(s, start + 1, cnt - 1);

    return h(s, start + 1, cnt + 1) || h(s, start + 1, cnt) || h(s, start + 1, cnt - 1);
}
// soln-2: bfs idea
// when hitting *, our counter would be in following 3 ways: count - 1, count - 0, count + 1.
// so we keep track of [lower, upper] boundary, as long as lower could reach 0, we are good.
// for ex. s = '(((***', at s[2] we have: [3, 3], s[3]: [2, 4] - use ')', ' ' and '(' accordingly.
bool checkValidString(string s) {
    int lo = 0, hi = 0;
    for (auto ch : s) {
        if ('(' == ch) lo++, hi++;
        else if (')' == ch) {
            lo--, hi--;
            if (lo < 0) {
                if (hi < 0) return false;   // ))
                else lo = 0;                // (*
            }
        } else {
            lo--, hi++;         // apply ')' and '('
            if (lo < 0) lo = 0; // apply * as empty string
        }
    }
    return 0 == lo;
}
```

```
// 856 - score of parentheses
// soln-1: recursion in O(n^2) worst case
// 1. f("((...))...") ==> 2 * f("(...)") + f("...")
// 2. f("()...")      ==> 1 + f("...")
// soln-2: stack in O(n)
int scoreOfParentheses(string S) {
    int ans = 0;
    stack<int> stk;
    for (auto ch : S) {
        if ('(' == ch) {
            stk.push(0);
        } else {
            int t = stk.top() ? stk.top() : 1;  stk.pop();
            stk.empty() ? ans += t :  stk.top() += 2 * t;
        }
    }
    return ans;
}
```

```
// 921 - Minimum Add to Make Parentheses Valid
// soln-1: brute-force
// 1. if ')', count - 1, if count < 0, then we need to add '('
//                       else if end-of-string, we need to add |count|.
// 2. if ')', count + 1, if end-of-string, we need to add count.
int minAddToMakeValid(string S) {
    int ans = 0, cnt = 0;
    for (auto ch : S) {
        if (')' == ch) {
            if (--cnt < 0) ++ans, cnt = 0;
        } else {
            ++cnt;
        }
    }
    return ans + (cnt > 0 ? cnt : 0);
}
```

Ref:

28. Implement strStr

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Solution:

String search algorithm:

1. Naive string search in O(mn)
2. typical string compare algorithm like KMP in O(m+n), Boyer-Moore, Rabin-Karp (use rolling hash)

#187 - rolling hash for multiple pattern search. (for single pattern search, use KMP/BM algorithm)

```
// soln-1: naive approach in O(mn) time in 800+ secs
int strStr(string haystack, string needle) {
   int n = haystack.length(), m = needle.length();
   if (!n && !m) return 0;

   for (int i = 0; i < n; ++i) {
       int j = 0;
       for (; i + j < n && j < m; ++j) {
           if (haystack[i + j] != needle[j]) break;
       }
       if (j == m) return i;
   }
   return -1;
}
```

Ref: #187 Topcoder: Intro to string search

29/204/625. Divide two integers / count primes ... Math

#263 - ugly number (positive #s whose prime factors only include 2, 3, 5.)

```
// 29 - divide two integers
// soln-1: math
// exponentially increase divisor instead of minus 1 each time
int divide(int dividend, int divisor) {
   if (INT_MIN == dividend && -1 == divisor) return INT_MAX;   // just for a wierd testcase

   long ans = 0, u = labs(dividend), d = labs(divisor);
   while (u >= d) {
       int pow = 0;
       // reduce u is safer, increase d could overflow potentially
       for (nullptr; (u >> (pow + 1)) >= d; ++pow);
       ans += ((long)1 << pow), u -= d << pow;
   }
   bool sign = (dividend > 0) ^ (divisor > 0);
   return sign ? -ans : ans;
}
```

```
// 204 - count prime
// soln-1: See wiki "Prime Number"
int countPrimes(int n) {
    vector<bool> flag(n, true);
    flag[0] = false, flag[1] = false;
    for (int i = 2; i < sqrt(n); ++i) {
        if (flag[i] == false) continue;
        for (int j = i * i; j < n; j += i) flag[j] = false;
    }
    return count(flag.begin(), flag.end(), true);
}
```

```
// 625 - Minimum Factorization (48 <= 68 (6 * 8))
// soln-1: math
int smallestFactorization(int a) {
   if (a == 1) return 1;
   string ans;
   for (int k = 9; k >= 2; --k) {
       while (a % k == 0) {
           ans = to_string(k) + ans, a /= k;
       }
   }
   if (a > 1) return 0;
   return stoll(ans) > INT_MAX ? 0 : stoll(ans);
}
```

```
// 1103 - Distribute Candies to People
// soln-1: math
// time complexity: the candy was given at following speed:
// 1 + 2 + 3 + ... <= total-candies, so it's O(sqrt(candies)) time
vector<int> distributeCandies(int candies, int N) {
    vector<int> ans(N);
    for (int i = 0; candies > 0; ++i) {
        ans[i % N] += min(candies, i + 1), candies -= (i + 1);
    }
    return ans;
}
```

Ref:

31/1053. Next permutation/Prev permutation with 1 swap

```
// 31 - Next permutation - next value that bigger than current (exactly same as #556)
// soln-1: brute-force
// find '/' shape from right side, then use the smallest but bigger than i to replace and sort right part.
// Steps as follows, Ex. 536974 => 534679
// 1. from right to left scan till a[i] < a[i+1]: 53[69]74
// 2. use the smallest (but bigger than current) from right part to replace current digit: 53[7]964
// 3. sort right part: 537[469]
void nextPermutation(vector<int>& nums) {
   int i = nums.size() - 2;
   for (; i >= 0; --i) {
       if (nums[i] < nums[i + 1]) break;
   }
   if (i < 0) {
       sort(nums.begin(), nums.end());
       return;
   }

   // smallest (but bigger than me) in right part
   int smallest = i + 1;
   for (int j = i + 1; j < nums.size(); ++j) {
       if (nums[j] < nums[smallest] && nums[j] > nums[i]) smallest = j;
   }
   swap(nums[i], nums[smallest]);
   sort(nums.begin() + i + 1, nums.end());
}
```

```
// 1053 - Previous Permutation With One Swap (largest permutation that is smaller than A)
// soln-1: brute-force
// find '\' from right side, we want to replace A[i] with the largest but smaller than A[i] in right part
// Ex. 537469, A[i] = 7, largest but smaller than A[i] is 6. So 53[6]4[7]9.
// If remove the one swap limitation, prev permutation would be 536[974] (sort right part decreasingly)
vector<int> prevPermOpt1(vector<int>& A) {
   int i = A.size() - 2;
   for (; i >= 0; --i) {
       if (A[i] > A[i + 1]) break;
   }
   if (i < 0) return A;

   auto bigger = i + 1;
   for (int j = bigger + 1; j < A.size(); ++j) {
       if (A[i] > A[j] && A[j] > A[bigger]) bigger = j;
   }
   swap(A[i], A[bigger]);
   return A;
}
```

Ref: #46 #60 #496

32. Longest valid parentheses

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.
For `"(()"`, the longest valid parentheses substring is `"()"`, which has length = 2.

For `")()())"`, where the longest valid parentheses substring is `"()()"`, which has length = 4.

For `"()(()"`, where the longest valid parentheses substring is `"()"`, which has length = 2.

Solution:

Parentheses problems:
#20 - valid parentheses (stack)

#22 - generate parentheses (backtracking)
#32 - longest valid parentheses (stack, dp)
#241 - different ways to add parentheses
#301 - remove invalid parentheses (BFS, recursion)

```
// soln-1: stack to pair parentheses, similar to #42
int longestValidParentheses(string s) {
   int ans = 0;
   stack<int> stk;
   for (int i = 0; i < (int)s.length(); ++i) {
       if (s[i] == '(') {
           stk.push(i);
       } else {
           // s[i] = ')', try to pair parentheses, #42
           if (!stk.empty() && s[stk.top()] == '(') {
               stk.pop();
               ans = max(ans, i - (stk.empty() ? -1 : stk.top()));
           } else {
               stk.push(i);
           }
       }
   }
   return ans;
}

// soln-1: dynamic programming
// if s[i] == ')',
// 1. if s[i - 1] == '(', dp(i) = 2 + dp(i - 2).                                   <--- consider "()()"
// 2. if s[i - 1 - dp(i - 1)] == '(', dp(i) = 2 + dp(i - 1) + dp(i - dp(i - 1) - 2). <--- consider "()(())"
//
int longestValidParentheses(string s) {
   int ans = 0;
   vector<int> dp(s.length());
   for (int i = 1; i < s.length(); ++i) {
       if (s[i] != ')')    continue;

       if (s[i - 1] == '(')  dp[i] = 2 + dp[i - 2];
       else if (s[i - 1 - dp[i - 1]] == '(') dp[i] = 2 + dp[i - 1] + dp[i - dp[i - 1] - 2];

       ans = max(ans, dp[i]);
   }
   return ans;
}
```

Ref:

33/34/81/153/154/702. Search in rotated sorted array/With unknown size (review)

```
// 33 - search in rotated sorted array (no dups)
int search(vector<int>& nums, int target) {
    if (nums.empty()) return -1;

    int lo = 0, hi = nums.size() - 1;
    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
        if (nums[m] == target) return m;
        if (nums[lo] < nums[m]) {       // low is sorted
            (nums[lo] <= target && target <= nums[m]) ? hi = m : lo = m;
        } else {                        // hi is sorted
            (nums[m] <= target && target <= nums[hi]) ? lo = m : hi = m;
        }
    }
    if (nums[lo] == target) return lo;
```

```cpp
    return nums[hi] == target ? hi : -1;
}
```

```cpp
// 81 - search in rotated sorted array (with dups)
bool search(vector<int>& nums, int target) {
    if (nums.empty()) return false;

    int lo = 0, hi = nums.size() - 1;
    while (lo + 1 < hi && nums[lo] == nums[hi]) --hi;    // remove dups from hi side

    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
        if (nums[m] == target) return true;
        if (nums[lo] <= nums[m]) {       // low is sorted
            (nums[lo] <= target && target <= nums[m]) ? hi = m : lo = m;
        } else {                        // hi is sorted
            (nums[m] <= target && target <= nums[hi]) ? lo = m : hi = m;
        }
    }
    return nums[lo] == target || nums[hi] == target;
}
```

```cpp
// 34 - find first and last position of element in sorted array
vector<int> searchRange(vector<int>& nums, int target) {
    vector<int> ans{-1, -1};
    if (nums.empty()) return ans;

    int lo = 0, hi = nums.size() - 1;
    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
        nums[m] < target ? lo = m : hi = m;
    }
    if (nums[lo] == target) ans[0] = lo;
    else if (nums[hi] == target) ans[0] = hi;
    else return ans;

    lo = 0, hi = nums.size() - 1;
    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
        nums[m] > target ? hi = m : lo = m;
    }
    (nums[hi] == target) ? ans[1] = hi : ans[1] = lo;
    return ans;
}
```

```cpp
// 153 - find min. in rotated sorted array (no dups)
int findMin(vector<int>& nums) {
    int lo = 0, hi = nums.size() - 1;
    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
        if (nums[lo] < nums[m]) {   // lo is sorted
            (nums[lo] < nums[m] && nums[m] < nums[hi]) ? hi = m : lo = m;
        } else {
            hi = m;                 // min. must on left part
        }
    }
    return min(nums[lo], nums[hi]);
}
```

```cpp
// 154 - find min. in rotated sorted array (no dups)
int findMin(vector<int>& nums) {
    int lo = 0, hi = nums.size() - 1;
    while (lo + 1 < hi && nums[lo] == nums[hi]) --hi;    // remove dups from hi side

    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
```

```
        if (nums[lo] <= nums[m]) {   // lo is sorted
            (nums[lo] <= nums[m] && nums[m] <= nums[hi]) ? hi = m : lo = m;
        } else {
            hi = m;                  // min. must on left part
        }
    }
    return min(nums[lo], nums[hi]);
}
```

```
// 702 - search target in unknown size array
int search(const ArrayReader& reader, int target) {
   int lo = 0, hi = INT_MAX;       // just assume a size then search
   while (lo < hi) {
       int m = lo + (hi - lo) / 2, x = reader.get(m);
       if (x == target) return m;
       x < target ? lo = m + 1 ? hi = m;
   }
   return -1;
}
```

Ref:

35. Search insert position

Given a sorted array and a value, return the position if the target is found, otherwise return the position where it would be if it were inserted in order.

```
int findK(const vector<int>& nums, int low, int hi, int target) {
   if ((low + 1) % nums.size() != hi) {
       int m = low + (hi - low) / 2;
       return target <= nums[m] ? findK(nums, low, m, target) : findK(nums, m, hi, target);
   }
   if (target > nums[hi]) return hi + 1;
   if (target == nums[hi]) return hi;
   if (target > nums[low]) return low + 1;
   if (target == nums[low]) return low;
   return 0;
}

int searchInsert(vector<int>& nums, int target) {
   return findK(nums, 0, nums.size() - 1, target);
}
```

Ref: #34 #153 #154

36/37/498. Sudoku/Diagonal traverse

```
// 36 - Valid Sudoku
// soln-1: bit manipulation
// 1. since each row/col only has 9 digits, we can use 1-integer to mark 1~9. same as each box
// 2. how to calculate which box for (i, j)?   => box[i / 3 * 3 + j / 3]
bool isValidSudoku(vector<vector<char>>& board) {
    vector<int> row(9), col(9), box(9);
    for (int i = 0; i < board.size(); ++i) {
        for (int j = 0; j < board[0].size(); ++j) {
            if ('.' != board[i][j]) {
                int v = 1 << (board[i][j] - '0'), k = i / 3 * 3 + j / 3;
                if ((row[i] & v) || (col[j] & v) || (box[k] & v)) return false;
                row[i] |= v, col[j] |= v, box[k] |= v;
            }
        }
    }
    return true;
```

```
}
```

```
// 37 - Sudoku Solver
// soln-1: backtracking
// 1. mark the used for each row and col, hence the candidates for each position
// 2. systematically enumerate each candidate
void mark(int x, int y, char ch, vector<int>& r, vector<int>& c, vector<int>& b, bool set = true) {
   int n = 1 << (ch - '0'), k = x / 3 * 3 + y / 3;
   r[x] = set ? (r[x] | n) : (r[x] & ~n);
   c[y] = set ? (c[y] | n) : (c[y] & ~n);
   b[k] = set ? (b[k] | n) : (b[k] & ~n);
}
bool used(int x, int y, char ch, vector<int>& r, vector<int>& c, vector<int>& b) {
   int n = 1 << (ch - '0'), k = x / 3 * 3 + y / 3;
   return (r[x] & n) || (c[y] & n) || (b[k] & n);
}
bool helper(vector<vector<char>>& board, int pos, vector<int>& r, vector<int>& c, vector<int>& b) {
   if (pos >= 9 * 9) return true;

   int x = pos / 9, y = pos % 9;
   if (board[x][y] != '.') return helper(board, pos + 1, r, c, b);

   for (auto i = '1'; i <= '9'; ++i) {
       if (!used(x, y, i, r, c, b)) {
           mark(x, y, i, r, c, b),        board[x][y] = i;
           if (helper(board, pos + 1, r, c, b)) return true;
           mark(x, y, i, r, c, b, false), board[x][y] = '.';
       }
   }
   return false;
}
void solveSudoku(vector<vector<char>>& board) {
   vector<int> row(9), col(9), box(9);
   for (int i = 0 ; i< 9; ++i) {
       for (int j = 0; j < 9; ++j) {
           if (board[i][j] != '.') mark(i, j, board[i][j], row, col, box);
       }
   }
   helper(board, 0, row, col, box);
}
```

```
// 498 - Diagonal Traverse (same diagonal has same x + y value)
// soln-1: brute-force
vector<int> findDiagonalOrder(vector<vector<int>>& m) {
    if (m.empty()) return {};
    int row = m.size(), col = m[0].size();
    vector<int> ans(row * col);
    for (int i = 0, k = 0; i < row + col; ++i) {
        if (i % 2) {    // down-left direction
            for (int x = 0; x < row && i - x >= 0; ++x) {
                if (i - x < col) ans[k++] = m[x][i - x];
            }
        } else {        // up-right direction
            for (int x = min(i, row - 1); x >= 0 && i - x < col; --x) {
                if (i - x >= 0) ans[k++] = m[x][i - x];
            }
        }
    }
    return ans;
}
```

Ref: #51

39/40/216/377/241/254/1079. Comb. sum(1~4)/Ways to add parentheses/Factor comb. (review)

#77 - a general combinations (n-choose-k) problem.
#139 - word breaker (similar DP with #377).
#322 - coin change (ask for fewest changes needed, easy DP question)

```
// 39 - combination sum to target (inputs are positive integers and w/o dups, can be chosen repeatedly)
// soln-1: backtracking
void helper(const vector<int>& c, int start, int target, vector<int>& comb, vector<vector<int>>& ans) {
   if (target == 0) ans.push_back(comb);

   for (int i = start; target > 0 && i < c.size(); ++i) {
       if (i > start && c[i] == c[i - 1]) continue;

       comb.push_back(c[i]);
       helper(c, i, target - c[i], comb, ans);
       comb.pop_back();
   }
}

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
   vector<int> comb; vector<vector<int>> ans;
   helper(candidates, 0, target, comb, ans);
   return ans;
}
```

```
// 40 - combination sum (input with dups, can not be chose repeatedly)
// soln-1: backtracking, sort to skip dup #s, starting from next to avoid duplicate ans
void helper(const vector<int>& c, int start, int target, vector<int>& comb, vector<vector<int>>& ans) {
   if (target == 0) ans.push_back(comb);

   for (int i = start; target > 0 && i < c.size(); ++i) {
       if (i > start && c[i] == c[i - 1]) continue;

       comb.push_back(c[i]);
       helper(c, i + 1, target - c[i], comb, ans);
       comb.pop_back();
   }
}
vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
   std::sort(candidates.begin(), candidates.end());

   vector<int> comb; vector<vector<int>> ans;
   helper(candidates, 0, target, comb, ans);
   return ans;
}
```

```
// 216 - sum combination III (given 1~9, k numbers sum to target)
// soln-1: backtracking
void helper(int start, int k, int n, vector<int>& comb, vector<vector<int>>& ans) {
   if (!n && comb.size() == k) ans.push_back(comb);
   if (n < 0 || comb.size() >= k) return;

   for (int i = start; i <= 9; ++i) {
       comb.push_back(i);
       helper(i + 1, k, n - i, comb, ans);
       comb.pop_back();
   }
}

vector<vector<int>> combinationSum3(int k, int n) {
   vector<vector<int>> ans;
   vector<int> comb;
   helper(1, k, n, comb, ans);
```

```
    return ans;
}
```

```
// 377 - ways that sum to target
// soln-3: dynamic programming
// 1. f(n) = x + f(n - x), f(0) = 1 meaning 1 way to get 0 given only 0.
// 2. if numbers could be negative and re-usable, there could be infinitely answer.
// 3. if contains negative number but not reusable, recursion with memo could solve this question.
int combinationSum4(vector<int>& nums, int target) {
    vector<int> dp(target + 1);
    dp[0] = 1;
    for (int i = 1; i <= target; ++i) {             // sum from 1 to target
        for (int x : nums) {
            if (i - x >= 0) dp[i] += dp[i - x];
        }
    }
    return dp[target];
}
```

```
// 241 - ways to add parentheses
// soln-1: backtracking (systematically enumerate)
// f(x) = f(first-half)-op-f(second-half)
vector<int> diffWaysToCompute(string input) {
    vector<int> ans;
    for (int i = 0; i < input.length(); ++i) {
        if ('+' == input[i] || '-' == input[i] || '*' == input[i]) {
            auto first = diffWaysToCompute(input.substr(0, i));
            auto rest = diffWaysToCompute(input.substr(i + 1));
            for (auto& x : first) {
                for (auto& y : rest) {
                    switch (input[i]) {
                        case '+':   ans.push_back(x + y);   break;
                        case '-':   ans.push_back(x - y);   break;
                        case '*':   ans.push_back(x * y);   break;
                    }
                }
            }
        }
    }
    if (ans.empty() && !input.empty()) ans.push_back(stoi(input));    // exit condition
    return ans;
}
```

```
// 254 - factor combination
// soln-1: backtracking (systematically enumerate)
// 1. f(n) = i * f(x), x = n/i
// 2. first starting from factor 2
vector<vector<int> > helper(int start, int n) {
    vector<vector<int> > ans;

    for (int i = start; i <= sqrt(n); ++i) {  // sqrt(n) is enough
        if (n % i) continue;

        auto rest = helper(i, n / i);         // starting from i because of reuse, not start because of dup
results
        rest.push_back(vector<int> {n / i});  // n/i will not be explored because of sqrt(n), add it now.

        // i is the least # factor, so merge the results
        for (vector<int>& res : rest) {
            res.insert(res.begin(), i);       // i is the least #, put it into 1st.
            ans.push_back(res);
        }
    }
    return ans;
}
```

```
// 1079 - Letter Tile Possibilities
// "AAB" => "A", "B", "AA", "AB", "BA", "AAB", "ABA", "BAA"
// soln-1: dfs/brute-force enumerate
int h(vector<int>& m) {
    int ans = 0;
    for (auto& c : m) {
        if (c) {
            ++ans, --c, ans += h(m), ++c;
        }
    }
    return ans;
}
int numTilePossibilities(string tiles) {
    vector<int> m(26);
    for (auto& ch : tiles) m[ch - 'A']++;
    return h(m);
}
```

Ref: #17 #77 #216 #322

41/141/142/287/457/565. First missing positive/Find the dup number/Array nesting

```
// 41 - first missing positive (given unsorted array find the 1st missing positive)
// soln-1: sort-of pigeon hole
// 1. put a[i] into idx-i if i within the range,
// 2. scan from left, the first missing positive is i if a[i] != i
int firstMissingPositive(vector<int>& nums) {
   int n = nums.size();
   for (int i = 0; i < n; ++i) {
       while (nums[i] != nums[nums[i] - 1] && nums[i] < n && nums[i] > 0) {
           swap(nums[i], nums[nums[i] - 1]);
       }
   }
   for (int i = 0; i < n; ++i) {
       if (nums[i] != i + 1) return i + 1;
   }
   return n + 1;
}
```

```
// 287 - find the duplicate number (containing n+1 numbers [1..n])
// soln-1: fast/slow pointer (vector as linked list)
// 1. XOR doesn't work because the dups could be any times
// 2. dups must exist because of pigeon hole principle (n+1 pigeon and n hole)
// 3. because the number is within range [1..n], we can view the vector as linked list:
//    - value is acting like pointer
//    - loop must be exist (we have dups, then 2 value pointing to same place)
//    - apply tortoise/hare algorithm to find the dup
int findDuplicate(vector<int>& nums) {
   int slow = nums[0], fast = nums[slow];
   while (slow != fast) {             // cycle detected, #141
       slow = nums[slow], fast = nums[nums[fast]];
   }

   slow = 0;                          // where the cycle begins, #142
   while (slow != fast) {
       slow = nums[slow], fast = nums[fast];
   }
   return slow;
}
```

```
// 457 - Circular Array Loop
// soln-1: fast/slow pointers (mark all visited if no loop found in the path)
int nextIdx(int idx, vector<int>& nums) {
```

```
    int n = nums.size(), val = nums[idx] + idx;
    return (val % n + n) % n;           // return next index based on the move of nums[idx]
}
bool circularArrayLoop(vector<int>& nums) {
    for (int n = nums.size(), i = 0; i < n; ++i) {
        if (0 == nums[i]) continue;

        int slow = i, fast = nextIdx(slow, nums);
        while (nums[slow] * nums[fast] > 0) {
            if (slow == fast) {
                if (slow == nextIdx(slow, nums)) break;    // loop hop must be > 1
                return true;
            }
            slow = nextIdx(slow, nums), fast = nextIdx(fast, nums);
            if (nums[fast] * nums[nextIdx(fast, nums)] <= 0) break;    // in case of jumping back
            fast = nextIdx(fast, nums);
        }
        slow = i;                                       // mark from i to slow as visited
        while (slow != fast && nums[slow] * nums[fast] > 0) {
            nums[slow] = 0, slow = nextIdx(slow, nums);
        }
    }
    return false;
}
```

```
// 565 - array nesting (find the largest loop in array)
// soln-1: vector as linked list
int arrayNesting(vector<int>& nums) {
    int ans = 0, n = nums.size();
    for (int i = 0; i < n; ++i) {
        int len = 0, j = i;
        while (nums[j] >= 0) {
            int t = nums[j], nums[j] = -1;  // mark wivisted. nums[j] could be 0, so can not negate it
            j = t, ++len;
        }
        ans = max(ans, len);
    }
    return ans;
}
```

```
// 141/142 - linked list cycle detection
bool hasCycle(ListNode *head) {
    if (head == NULL || head->next == NULL) return false;
    ListNode* slow = head, * fast = head;
    while (fast->next && fast->next->next) {
        slow = slow->next, fast = fast->next->next;
        if (fast == slow) return true;
    }
    return false;
}

ListNode *detectCycle(ListNode *head) {
    if (head == NULL || head->next == NULL) return NULL;

    ListNode* slow = head,  * fast = head, * entry = head;
    while (fast->next && fast->next->next) {
        slow = slow->next, fast = fast->next->next;
        if (fast == slow) {             // cycle found
            while (slow != entry) {     // how to prove?
                slow = slow->next, entry = entry->next;
            }
            return entry;
        }
    }
```

```
    return NULL;
}
```

Ref: #141 #142 #202

42/84/85/221/316407/689. Trapping water/Remove dup letters... Monotone stack/Q/deque (review)

```
// 42 - trapping water
// soln-1: monotone stack
int trap(vector<int>& height) {
   int water = 0;
   stack<int> stk;
   for (int i = 0; i < height.size(); NULL) {
       if (stk.empty() || height[stk.top()] > height[i]) {
           stk.push(i), ++i;
           continue;
       }

       // height[i] is higher, there might be some water.
       int bottom = height[stk.top()];     stk.pop();
       if (!stk.empty()) {
           water += (i - stk.top() - 1) * (min(height[stk.top()], height[i]) - bottom);
       }
       // do not push i into stack, we need a second try.
   }
   return water;
}

// 42 - trapping water
// soln-2: two pointers to track safe level: safe - current => trapping water
int trap(vector<int>& heights) {
    int ans = 0, level = 0;
    for (int left = 0, right = heights.size() - 1; left < right; NULL) {
        int lower = min(heights[left], heights[right]);
        level = max(level, lower);      // increase the potential safe level
        ans += level - lower;           // safe level - current level is the amount of water

        heights[left] < heights[right] ? ++left : --right;    // move the lower side
    }
    return ans;
}
```

```
// 84 - largest rectangle in histogram
// soln-1: monotone stack
int largestRectangleArea(vector<int>& heights) {
    int ans = 0;
    stack<int> stk;

    heights.push_back(0);                                      // to force to calculate at the end
    for (int i = 0; i < heights.size(); NULL) {
        if (stk.empty() || heights[i] >= heights[stk.top()]) {    // keep into stack if contributes to width
            stk.push(i++);
        } else {
            int h = heights[stk.top()]; stk.pop();
            int w = stk.empty() ? i : i - stk.top() - 1;           // tricky part: {3, 4, 0, 3, 4, 5}
            ans = max(ans, w * h);
        }
    }
    return ans;
}
```

```
// 85 - max rectangle
```

```
// soln-1: compute max histogram each row
int maximalRectangle(vector<int>& hist) {
   int ans = 0;

   stack<int> stk;
   hist.push_back(0);
   for (int i = 0; i < hist.size(); NULL) {
       if (stk.empty() || hist[i] >= hist[stk.top()]) {
           stk.push(i++);
       } else {
           int h = hist[stk.top()];  stk.pop();            // stack pattern like .:|
           int w = stk.empty() ? i : i - stk.top() - 1;
           ans = max(ans, w * h);
       }
   }
   return ans;
}

int maximalRectangle(vector<vector<char>>& matrix) {
   if (matrix.empty()) return 0;
   int row = matrix.size(), col = matrix[0].size();

   int ans = 0;
   vector<int> hist(col);
   for (int i = 0; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           hist[j] = (matrix[i][j] == '1') ? hist[j] + 1 : 0;
       }
       ans = max(ans, maximalRectangle(hist));
   }
   return ans;
}
```

```
// 85 - max rectangle
// soln-2: get left/right boundary for element in each row (similar to #689)
// lb[j] = max(lb[j], cur_lb), rb[j] = min()
int maximalRectangle(vector<vector<char>>& matrix) {
   int row = matrix.size(), col = matrix[0].size();

   int ans = 0;
   vector<int> lb(col), rb(col), h(col);
   for (int i = 0; i < matrix.size(); ++i) {
       // get left-boundary for each col
       for (int cur_lb = 0, j = 0; j < col; ++j) {
           if (matrix[i][j] == '1') {
               lb[j] = max(lb[j], cur_lb);     ++h[j];
           } else {
               lb[j] = 0;      cur_lb = j + 1;     h[j] = 0;
           }
       }
       // get right-boundary for each col
       for (int cur_rb = 0, j = col; j >= 0; --j) {
           if (matrix[i][j] == '1') {
               rb[j] = min(rb[j], cur_rb);
           } else {
               rb[j] = col;    cur_rb = j;
           }
       }

       for (int j = 0; j < col; ++j) ans = max(ans, (rb[j] - lb[j]) * h[j]);
   }
   return ans;
}
```

```cpp
// 221 - max square in matrix
// soln-1: dynamic programming
// dp[i][j] = 1 + min(dp[i - 1][j - 1], dp[i][j - 1], dp[i - 1][j])
int maximalSquare(vector<vector<char>>& matrix) {
   if (matrix.empty()) return 0;

   int ans = 0;
   vector<vector<int>> dp(2, vector<int>(matrix[0].size() + 1));
   int cur = 0, pre = 1, ans;
   for (int i = 1; i <= matrix.size(); ++i) {
       for (int j = 1; j <= matrix[0].size(); ++j) {
           if (matrix[i-1][j-1] == '1') {
               dp[cur][j] = 1 + min(min(dp[cur][j - 1], dp[pre][j]), dp[pre][j - 1]);
               ans = max(ans, dp[cur][j]);
           } else {
               dp[cur][j] = 0;
           }
       }
       swap(cur, pre);
   }
   return ans * ans;
}
```

```cpp
// 407 - trapping water II (TODO)
```

```cpp
// 316 - remove dup letters (every letter appears only once and in lexicographical order/smallest result)
// soln-1: greedy recursion
// 1. If greedy find the smallest letter as long as the remaining part still contain all the required letters.
// For example, "dbcacda", starting from s[0] = 'd', as we know 'd' exists in later too, it means if there is
// anything < 'd', we can ignore 'd' here and add it later.
// s[1] = 'b', and 'b' < 'd', so greedy pick it up, but as no 'b' in later string, we have to pick it now,
though there is 'a' after.
// 2. f('dbcacda') = 'b' + f('cacda')
// 3. each run is O(n), with at most 26 runs.
string removeDuplicateLetters(string s) {
   if (s.empty()) return "";

   int count[26] = {0};
   for (auto ch : s) count[ch - 'a']++;

   int smallest = 0;
   for (int i = 0; i < s.length(); ++i) {
       if (s[i] < s[smallest]) smallest = i;
       if (--count[s[i] - 'a'] == 0) break;
   }
   string ss;
   for (int i = smallest + 1; i < s.length(); ++i) {
       if (s[smallest] != s[i]) ss.push_back(s[i]);
   }
   return s[smallest] + removeDuplicateLetters(ss);
}

// 316 - remove dup letters (every letter appears only once and in lexicographical order/smallest result)
// soln-2: monotone stack + greedy
// Greedy use current smallest, if s[0 ... smallest - 1] will appear after smallest position.
string removeDuplicateLetters(string s) {
   int count[26] = {0}, visited[26] = {0};
   for (auto ch : s) count[ch - 'a']++;

   string ans;
   for (auto ch : s) {
       --count[ch - 'a'];
       if (visited[ch - 'a']) continue;

       while (!ans.empty() && ans.back() > ch && count[ans.back() - 'a'] != 0) {
```

```
            visited[ans.back() - 'a'] = 0, ans.pop_back();
        }
        visited[ch - 'a'] = 1, ans.push_back(ch);
    }
    return ans;
}

// 1081 - Smallest Subsequence of Distinct Characters
// identical with #316.
```

```
// 689 - max sum of 3 non-overlapping subarray (similar to #85 soln-2)
// soln-1: keep max-so-far from left and right side, then test within middle
//      0   1   2   3    4    5    6     7    (for subarray size k = 2)
//    ---+---+---+---+----+-----+-----+------
//      1   2   1   2    6    7    5     1       <--- nums (input)
// 0    1   3   4   6    12   19   24    25      <--- sum (extra one ahead for convenient)
//      x   x   0   0    0    3    x     x       <--- max sum subarray (index) so far from left
//      x   x   4   4    5    6    x     x       <--- from right side
// middle: [k, n - k - k]
// answer: {left[i], i, right[i+k-1]}
vector<int> maxSumOfThreeSubarrays(vector<int>& nums, int k) {
    int n = nums.size();
    vector<int> sum(n + 1);
    for (int i = 0; i < n; ++i) sum[i + 1] = sum[i] + nums[i];

    // lb[i] keeps the max sum window from left
    vector<int> lb(n);
    for (int i = k, t = INT_MIN; i + k < n; ++i) {
        int leftSum = sum[i] - sum[i - k];
        lb[i] = leftSum > t ? i - k : lb[i - 1];
        t = max(t, leftSum);
    }
    // rb[i] keeps the max sum window from right
    vector<int> rb(n);
    for (int i = n - 1 - k, t = INT_MIN; i >= k; --i) {
        int rightSum = sum[i + 1 + k] - sum[i + 1];
        rb[i] = rightSum > t ? i + 1 : rb[i + 1];
        t = max(t, rightSum);
    }

    vector<int> ans;
    for (int i = k, t = INT_MIN; i + k <= n - k; ++i) {
        int l = lb[i], r = rb[i + k - 1];

        //..................left..............self.....................right
        int tmp = (sum[l + k] - sum[l]) + (sum[i + k] - sum[i]) + (sum[r + k] - sum[r]);
        if (tmp > t) ans = vector<int>{l, i, r};
        t = max(t, tmp);
    }
    return ans;
}
```

Ref: #11 #239

43/67/415. Multiply/Add strings /add binary

```
// 43 - multiply
// soln-1: dynamic programming
//              8 3
//              2 7
//      --------------
//              16, 6
//                  56, 21
```

```
//      ---------------
//              16, 62, 21   (carry from right to left)
//              22, 4, 1     <-- (2241)
string multiply(string num1, string num2) {
    while (!num1.empty() && num1[0] == '0') num1.erase(num1.begin());
    while (!num2.empty() && num2[0] == '0') num2.erase(num2.begin());
    if (num1.empty() || num2.empty()) return "0";

    vector<int> dp;
    for (int i = 0; i < num1.length(); ++i) {
        for (int j = 0; j < num2.length(); ++j) {
            if (dp.size() <= i + j) dp.push_back(0);
            dp[i + j] += (num1[i] - '0') * (num2[j] - '0');
        }
    }

    string ans;
    for (int i = dp.size() - 1; i > 0; --i) {
        ans.insert(ans.begin(), '0' + dp[i] % 10);
        dp[i - 1] += dp[i] / 10;
    }
    if (!dp.empty()) ans = to_string(dp[0]) + ans;

    return ans;
}
```

```
string base2(int N) {
   string ans;
   while (N) {
       ans = to_string(N & 1) + ans;
       N = -(N >> 1);
   }
   return ans.empty() ? "0" : ans;
}

string baseNeg2(int N) {
   string ans;
   while (N) {
       ans = to_string(N & 1) + ans;
       N = -(N >> 1);
   }
   return ans.empty() ? "0" : ans;
}

// 2 - Add Two Numbers
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* ans(nullptr);
    int c = 0;
    while (c || l1 || l2) {
        if (l1) c += l1->val, l1 = l1->next;
        if (l2) c += l2->val, l2 = l2->next;
        auto t = new ListNode(c % 10);
        t->next = ans, ans = t, c /= 10;
    }
    ListNode* pre(nullptr), *cur = ans;
    while (cur) {
        auto nxt = cur->next;
        cur->next = pre, pre = cur, cur = nxt;
    }
    return pre;
}
// 445 - Add Two Numbers II
ListNode *addTwoNumbers(ListNode *l1, ListNode *l2) {
    stack<ListNode*> s1, s2;
    while (l1) s1.push(l1), l1 = l1->next;
```

```
    while (l2) s2.push(l2), l2 = l2->next;

    int c = 0;
    ListNode* ans(nullptr);
    while (c || !s1.empty() || !s2.empty()) {
        if (!s1.empty()) c += s1.top()->val, s1.pop();
        if (!s2.empty()) c += s2.top()->val, s2.pop();
        auto t = new ListNode(c % 10);
        t->next = ans, ans = t, c = c / 10;
    }
    return ans;
}

// 067 - add binary ("11" + "1" = "100")
string addBinary(string a, string b) {
   string ans;
   int c = 0, i = a.length() - 1, j = b.length() - 1;
   while (c || i >= 0 || j >= 0) {
       if (i >= 0) c += a[i];
       if (j >= 0) c += b[j];
       ans.push_back(c & 1);
       c = (c >> 1);
   }
   reverse(ans.begin(), ans.end());
   return ans;
}

// 1073 - Adding Two Negabinary Numbers
vector<int> addNegabinary(vector<int>& A, vector<int>& B) {
   vector<int> ans;
   int c = 0, i = A.size() - 1, j = B.size() - 1;
   while (c || i >= 0 || j >= 0) {
       if (i >= 0) c += A[i--];
       if (j >= 0) c += B[j--];
       ans.push_back(c & 1);
       c = -(c >> 1);
   }
   while (ans.size() > 1 && ans.back() == 0) ans.pop_back();
   reverse(ans.begin(), ans.end());
   return ans;
}
```

```
// 425: soln-1: similar to '67 - add binary'
string addStrings(string num1, string num2) {
   string ans;
   int c = 0, i = num1.length() - 1, j = num2.length() - 1;
   while (i >= 0 || j >= 0 || c > 0) {
       c += i >= 0 ? num1[i--] - '0' : 0, c += j >= 0 ? num2[j--] - '0' : 0;
       ans.push_back('0' + c % 10);
       c /= 10;
   }
   reverse(ans.begin(), ans.end());
   return ans;
}
```

Ref: #66 #67

44. Wildcard matching

Implement wildcard pattern matching with support for `'?'` and `'*'`.

`'?' Matches any single character.`

`'*' Matches any sequence of characters (including the empty sequence).`

```
The matching should cover the entire input string (not partial).

Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

Solution:

soln-1: greedy

Wildcard matching is an entire match. So we need to go through the text string. Here are a few cases:

1. *p == *s || *p == '?'. This is match, p and s move to next, continue to check next character
2. *p != *s, but *p == '*'. This is also a match.
    a. Should we move s to next? No, because '*' means zero or more match.
    b. How about p? p has to move forward, otherwise there will be no progress and, p should move to next to test next match with s.
3. if no match, we need to check if there is '*' in p showed up previously.
    a. If no, return mismatch.
    b. Otherwise, set p right after '*' position, and move s from the previous matched position.

running time: O(mn), consider this case: s = "aaaaaaaaaaaaaaab", p = "a*aaaac". Mismatch happens only at the end.

soln-2: recursion

the key idea is how to reduce the problem size for recursion.

1. `p == '*'` - zero or more character could match.
    a. for zero character to match, we actually advance p one character and keep s unchanged, so the problem size reduced;
    b. for one or more character, we move s to next character and keep p unchanged. Problem size also reduced.
2. `*p != '*'`, to continue, iff `p[0] == s[0] || p[0] == '?'`

Recursion soln is not useful in practice, but to show the idea of solving this problem. Discussion in optimization see here.

soln-3: dynamic programming

let m = length(p), n = length(s), dp[m][n] holds the matching result which initialized as false. The state equation as follows:

1) if p[i] != '*',   dp[i][j] = dp[i - 1][j - 1] && p[i] == s[j],
2) if p[i] == '*', dp[i][j] = dp[i - 1][j] || dp[i][j - 1],

running time: O(mn), space: O(mn) - could be optimized to O(n) since we only need to refer the previous state, which is dp[i-1][j-1].

#10 - regular expression matching

```
// soln-1: greedy
bool isMatch(string& s, string& p) {
    if (p.empty())  return s.empty();   // unnecessary, helper can handle this case.
    return helper(s.c_str(), p.c_str());
}

bool helper(const char *s, const char *p) {
    const char* ss = NULL, *star = NULL;

    while (*s) {
```

```
        if (*p == *s || *p == '?') {
            ++p;    ++s;
        } else if (*p == '*') {
            ss = s;         // s will start here if mismatch happen
            star = p++;     // p will start here if mismatch happen
        } else if (star) {
            p = star + 1;   // p is moving back if mismatch but star '*' showed up
            s = ++ss;       // s will advance from here
        } else {
            return false;   // mismatch & no star
        }

    }
    while (*p == '*') ++p;  // get rid of all '*'

    return *p == '\0';      // entire string match, not partial
}

// soln-2: recursion - not practical without optimization.
bool isMatch(string& s, string& p) {
    if (p.empty())  return s.empty();

    if (p[0] == '*') {
        // match empty string, so advance p one character
        if (isMatch(s, p.substr(1))) return true;

        // match one or more character, move s one character to right
        return !s.empty() && isMatch(s.substr(1), p);
    }
    return !s.empty() && (s[0] == p[0] || p[0] == '?') && isMatch(s.substr(1), p.substr(1));
}

// soln-3: dynamic programming
bool isMatch(string& s, string& p) {
    int m = s.length(), n = p.length();

    vector<bool> dp(m + 1, false);
    dp[0] = true;       // corner case for empty string.

    for (int i = 0; i < n; ++i) {
        bool pre = dp[0];
        dp[0] = pre && p[i] == '*';            // Tricky - consider s = "", p = "*?"

        for (int j = 1; j <= m; ++j) {
            bool tmp = dp[j];                  // keep it before update

            if (p[i] != '*') {
                dp[j] = pre && (p[i] == s[j - 1] || p[i] == '?');   // p[i] == s[j] || p[i] == '?'
            } else {
                dp[j] = dp[j - 1] || dp[j];    // p[i] == '*'
            }

            pre = tmp;
        }
    }

    return dp[m];
}
```

Ref: [#10](#10)

45/55/390. Jump/Elimination ... Game

```
// 45 - Jump game (given max jump length at that position, return min around to the last index)
// soln-1: greedy (check next possible steps, go as far as we can)
int jump(vector<int>& nums) {
   int ans = 0, lastR = 0, R = 0;  // last-reached and max-reached so far
   while (R < nums.size() - 1) {
       ++ans;
       int nR = R;
       for (int i = lastR; i <= R; ++i) {  // from last-reached to max-reached to get further position
           nR = max(nR, i + nums[i]);
       }
       lastR = R, R = nR;
   }
   return ans;
}
```

```
// 55 - Jump game (check reachability)
// soln-1: dynamic programming
// 1. mark reachability for each position, greedy jump from it to all other possible positions
// 2. by building DAG, easy to figure out all jump path
bool canJump(vector<int>& nums) {
   if (nums.size() == 0) return true;

   vector<bool> dp(nums.size(), false);
   dp[0] = true;
   for (int n = nums.size(), i = 0, reached = 0; i < n; ++i) {
       if (!dp[i]) continue;
       for (int j = reached + 1; j < min(nums[i] + i + 1, n); ++j) { // starting from reached + 1
           dp[j] = true, reached = max(reached, j);
       }
   }
   return dp[nums.size() - 1];
}

// 55 - Jump game (check reachability)
// soln-2: just check how far we can get under current condition
bool canJump(vector<int>& nums) {
   int reached = 0;
   for (int i = 0; i < nums.size() && i <= reached; ++i) {
       reached = max(reached, i + nums[i]);
   }
   return reached >= nums.size() - 1;
}
```

```
// 390 - Elimination Game (remove 1st and every other number from left and right side alternatively)
// soln-1: math
// 1. each round, remain /= 2, step *= 2 (starting from 1)
// 1  2  3 . 4 . 5 . 6 . 7 . 8 . 9
//    2 .    4 .     6 .     8      <- round-1, step = 1, ->, point to 2 (move 1-step-right because ->)
//    2 .            6              <- round-2, step = 2, <-, point to 2 (dont need to move, even n)
//                   6              <- round-3, step = 4, ->, point to 6 (move 4-step-right, because ->)
int lastRemaining(int n) {
    int ans = 1, l2r = 1, step = 1;
    while (n > 1) {
        if (l2r || n % 2 == 1) ans += step;
        step *= 2, n /= 2, l2r = !l2r;
    }
    return ans;
}
```

Ref: #55

46/47/996/60/77/78/90/753. Permutations/Combinations/Subsets/II/Cracking the safe

```
// 46/47 - permutations in O(n!) time
// soln-1: p({0, 1, 2, 3}) = {0} - p({0, 1, 2, 3} - {0})
//                         = {1} - p({0, 1, 2, 3} - {1})
//                         = {2} - p({0, 1, 2, 3} - {2})
//                         = {3} - p({0, 1, 2, 3} - {3})
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> ans;
    helper(0, nums, ans)
    return ans;
}

void helper(int start, vector<int>& nums, vector<vector<int>>& ans) {
   if (start >= nums.size()) ans.push_back(nums);

   for (int i = start; i < nums.size(); ++i) {
       bool dups = false;
       for(int j = start; j < i && !dups; ++j) {   // check dups [start..i-1]
           if (nums[j] == nums[i]) dups = true;
       }
       if(dups) continue;
       swap(nums[i], nums[start]);
       helper(start + 1, nums, ans);              // NOT from i + 1, but subset question is.
       swap(nums[i], nums[start]);
   }
}
```

```
// 996 - number of squareful arrays
// soln-1: brute-force (backtracking)
int helper(vector<int>& A, int start) {
    int ans = start >= A.size() ? 1 : 0;
    for (int i = start; i < A.size(); ++i) {
        bool dups = false;                     // skip dups [start, i) if any
        for (int j = start; j < i && !dups; ++j) {
            if (A[i] == A[j]) dups = true;
        }
        if (dups) continue;
        if (start > 0) {                       // check if it is worth to swap
            int k = sqrt(A[start - 1] + A[i]);
            if (k * k != (A[start - 1] + A[i])) continue;
        }
        swap(A[start], A[i]);
        ans += helper(A, start + 1);
        swap(A[start], A[i]);
    }
    return ans;
}
int numSquarefulPerms(vector<int>& A) {
    return helper(A, 0);
}
```

```
// 60 - permutation sequence
// soln-1: math
// 1. for length n string, each index will generate (n-1)! permutations.
// 2. hence the kth permutation is: f(n, k) = A[idx] + f(n-1, k - idx * (n-1)!), idx = k / (n-1)!
string helper(string& str, int k) {
   if (str.empty()) return "";

   auto n1 = fact(str.length() - 1);
   auto ch = str[k / n1];
   str.erase(str.begin() + k / n1);
   return string(1, ch) + helper(str, k - (k / n1) * n1);
}

int fact(int n) {
```

```
   return n < 2 ? 1 : n * fact(n - 1);
}

string getPermutation(int n, int k) {
   string str(n, '1');         // n would be in [1, 9]
   for (int i = 0; i < n; ++i) str[i] += i;
   return helper(str, k - 1);
}
```

```
// 78 - combinations
// soln-1: backtracking
vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> ans;
    vector<int> comb;
    bt(1, n, k, comb, ans);
    return ans;
}

void bt(int start, int end, int k, vector<int>& comb, vector<vector<int>>& ans) {
    if (comb.size() == k) {                // 1-line difference with subset question
        ans.push_back(comb);    return;
    }

    for (int i = start; i <= end; ++i) {
        comb.push_back(i);
        bt(i + 1, n, k, comb, ans);
        comb.pop_back();
    }
}
```

```
// 79/90 - subset/II
// soln-1: backtracking (sort to skip dups)
// Subset({1, 2, 3, 4}) =
// {},
// {1, subset({2, 3, 4})}
// {2, subset({3, 4})}
// {3, subset({4})}
// {4, subset({})}
vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> ans;
    if (nums.size() == 0) return ans;

    vector<int> mask(nums.size(), 0);
    sort(nums.begin(), nums.end());
    helper(nums.size() - 1, mask, nums, ans);

    return ans;
}

void helper(int start, vector<int>& nums, vector<int>& res, vector<vector<int>>& ans) {
    ans.push_back(res);               // only 1-line diff from n-choose-k question #77

    for (int i = start; i < nums.size(); ++i) {
        if (i > start && nums[i] == nums[i - 1]) continue;  // skip dups

        res.push_back(nums[i]);
        helper(i + 1, nums, res, ans);
        res.pop_back();
    }
}

// 79 - subset
// soln-2: iterative
vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> ans;
    ans.push_back(vector<int>{});

    for (int i = 0; i < nums.size(); ++i) {
        int sz = ans.size();
```

```
        for (int j = 0; j < sz; ++j) {
            vector<int> tmp = ans[j];
            tmp.push_back(nums[i]);
            ans.push_back(tmp);
        }
    }
    return ans;
}

// 79 - subsets
// soln-3: bit manipulation in time O(2^n)
vector<vector<int>> subsets(vector<int>& nums) {
    // OJ requires the output in ascending order, otherwise sort is unnecessary
    sort(nums.begin(), nums.end());

    int n = 1 << nums.size();       // 2^n subsets
    vector<vector<int>> ans(n);     // allocate buffer

    for (int i = 0; i < nums.size(); ++i) {
        for (int j = 0; j < n; ++n) {
            if ((j >> i) && 1) ans[j].push_back(nums[i]);
        }
    }

    return ans;
}
```

```
// 753 - cracking the safe
// soln-1: backtracking or dfs/bfs to enumerate all visited
// greedy reuse the last n-1 digits until all are visited.
bool helper(int n, int k, string& ans, unordered_set<int>& visited, int count) {
   if (0 == count) return true;

   for (int i = 0; i < k; ++i) {
       auto t = n > 1 ? stol(ans.substr(ans.length() - (n - 1))) : 0, v = t * 10 + i;
       if (visited.find(v) != visited.end()) continue;

       visited.insert(v), --count, ans.push_back(char(i + '0'));
       if (helper(n, k, ans, visited, count)) return true;
       visited.erase(v), ++count, ans.pop_back();
   }
   return false;
}
string crackSafe(int n, int k) {
   unordered_set<int> visited;
   string ans(n, '0');

   int total = pow(k, n);
   visited.insert(0), total--;
   helper(n , k, ans, visited, total);
   return ans;
}
```

Ref: #39 #40 #46 #60 #61 #90 #216

48. Rotate image

You are given an *n* x *n* 2D matrix representing an image. Rotate the image by 90 degrees (clockwise).

Note:

You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

Solution:

Two steps (clockwise and counterclockwise share similar method): switch diagonal direct, then left-right switch

```
// soln-1: do diagonal switch then left-right switch
```

```
void rotate(vector<vector<int>>& matrix) {
   int n = matrix.size();
   for (int i = 0; i < n; ++i) {
       for (int j = i + 1; j < n; ++j) {
           swap(matrix[i][j], matrix[j][i]);
       }
   }
   for (auto& vi : matrix) reverse(vi.begin(), vi.end());
}
```

Ref:

49/242/249/438/760. Group/Valid anagrams ...Anagram

```
// 49 - group anagram
// soln-1: easy hashmap
vector<vector<string>> groupAnagrams(vector<string>& strs) {
   vector<vector<string>> ans;

   unordered_map<string, int> mp;  // <sorted-string, index>
   for (auto w : strs) {
       auto ss = w;
       sort(ss.begin(), ss.end());

       auto it = mp.find(ss);
       if (it != mp.end()) {
           ans[it->second].push_back(w);
       } else {
           mp[ss] = ans.size();
           ans.push_back(vector<string>{w});
       }
   }
   return ans;
}
```

```
// 242 - valid anagram
// soln-1: easy hashmap
bool isAnagram(string s, string t) {
   if (s.length() != t.length()) return false;

   vector<int> c(256);
   for (int i = 0; i < s.length(); ++i) c[s[i]]++, c[t[i]]--;

   for (int x : c) if (x) return false;
   return true;
}
```

```
// 249 - group shifted strings (["abc","bcd","xyz"], ["az", "ba"])
// soln-1: hashmap by pattern (let pattern = s[i+1] - s[i] + 'a')
vector<vector<string>> groupStrings(vector<string>& strs) {
    unordered_map<string, vector<string>> mp;
    for (string& s : strs) {
        string p;
        for (int i = 0; i < (int)s.length(); ++i) p.push_back(((s[(i + 1) % s.length()] - s[i]) + 26) % 26 +
'a');
        mp[p].push_back(s);
    }
    vector<vector<string>> ans;
    for (auto iter : mp) ans.push_back(iter.second);

    return ans;
}
```

```
// 438: soln-1: sliding window + hashmap
vector<int> findAnagrams(string s, string p) {
   vector<int> ans, cs(26), cp(26);
   if (s.length() < p.length()) return ans;

   for (int i = 0; i < p.length(); ++i) cs[s[i] - 'a']++, cp[p[i] - 'a']++;

   for (int i = p.length(); i <= s.length(); ++i) {
       if (cs == cp) ans.push_back(i - p.length());

       cs[s[i - p.length()] - 'a']--;
       if (i < s.length()) cs[s[i] - 'a']++;
   }
   return ans;
}
```

```
// 760: find anagram mapping
// A and B are anagram list, find the mapping.
// Ex. given A = [12, 28, 46, 32, 50], B = [50, 12, 32, 46, 28]
// return [1, 4, 3, 2, 0]
// soln-1: hashmap
vector<int> anagramMappings(vector<int>& A, vector<int>& B) {
    vector<int> ans;
    unordered_multimap<int, int> mp;    // <val, idx>
    for (int i = 0; i < B.size(); ++i) mp.insert({B[i], i});

    for (int i = 0; i < A.size(); ++i) {
        auto it = mp.find(A[i]);
        ans.push_back(it->second), mp.erase(it);
    }
    return ans;
}
```

Ref:

50/69/372. Power(x, n) / sqrt(x) / Super pow

Your task is to calculate a^b mod 1337 where a is a positive integer and b is an extremely large positive integer given in the form of an array.
Example1: a = 2, b = [3], Result: 8

Example2: a = 2, b = [1,0], Result: 1024

```
// soln-1: binary search
double myPow(double x, long n) {
   if (n == 0) return 1.0f;
   if (n == 1) return x;
   if (n < 0) return myPow(1.0f/x, -n);

   return myPow(x * x, n / 2) * (n % 2 ? x : 1);
}
```

```
// soln-1: binary search, keep the last possible answer.
int mySqrt(int x) {
    int ans = INT_MIN, left = 1, right = x;
    while (left <= right) {
        int m = left + (right - left) / 2;
        int t = x / m;
        if (t == m) return m;

        (t > m) ? left = m + 1, ans = m : right = m - 1;
    }
```

```
    return ans;
}
```

```
// soln-1: math
// a^b = (a^(b/10)) * a^10
// (a^b) % M = (((a^(b/10)) % M) * (a^10) % M) % M
int superPow(int a, vector<int>& b) {
   int ans = 1, M = 1337;
   for (int i = b.size() - 1; i >= 0; --i) {
       ans = ans * pow(a, b[i], M) % M;  // a^(b/10)
       a = pow(a, 10, M) % M;            // a^10
   }
   return ans;
}

// a^x = (a * a)^(x/2)       <--- if x is even.
// a^x = a * (a * a)^(x/2)   <--- if x is odd.
long pow(long a, int b, int M) {
   long ans = 1;
   for (nullptr; b; b >>= 1) {
       if (b & 1) ans = (a * ans) % M;
       a = (a * a) % M;
   }
   return ans;
}
```

Ref: #29

51/52/562. N-Queens/II/Longest line of consecutive 1s in matrix

Given an integer n, return all distinct solns to the n-queens puzzle. Each soln contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space respectively.

Eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens threaten each other. By threaten each other, two queens share the same row, column, or diagonal.

562 - Given a 01 matrix M, find the longest line of consecutive one in the matrix. The line could be horizontal, vertical, diagonal or anti-diagonal.

#37 - Sudoku solver
#348 - tic-tac-toe

```
// soln-1: backtracking with col/diag45/diag135
// diag45 = row + col
// diag135 = row - col + N
vector<vector<string>> solveNQueens(int n) {
   vector<bool> col(n), diag45(n), diag135(n);

   vector<vector<string>> ans;
   vector<string> res(n, string(n, '.'));
   helper(0, n, col, diag45, diag135, res, ans);
   return ans;
}
void helper(int r, int n, vector<bool>& col, vector<bool>& diag45, vector<bool>& diag135,
            vector<string>& res, vector<vector<string>>& ans) {
   if (r >= n) {
       ans.push_back(res); return;
   }

   for (int j = 0; j < n; ++j) {
       if (res[r][j] != '.') continue;
       if (col[j] || diag45[r + j] || diag135[r - j + n]) continue;
```

```
        col[j] = true, diag45[r + j] = true, diag135[r - j + n] = true, res[r][j] = 'Q';
        helper(r + 1, n, col, diag45, diag135, res, ans);
        col[j] = false, diag45[r + j] = false, diag135[r - j + n] = false, res[r][j] = '.';
    }
}
```

```
// 562 - soln-1: brute-force
// diag          ==> (i + j)
// anti-diagonal ==> (i - j + n)
int longestLine(vector<vector<int>>& M) {
   int m = M.size(), n = M[0].size(), ans = 0;
   vector<int> cols(n), diag(m + n), adiag(m + n);
   for (int i = 0; i < m; ++i) {
       int row = 0;
       for (int j = 0; j < n; ++j) {
           if (M[i][j]) {
               row++, cols[j]++, diag[i + j]++, adiag[i - j + n]++;
           } else {
               row = 0;
           }
       }
       ans = max(ans, row);
   }
   for (auto c : cols) ans = max(ans, c);
   for (auto c : diag) ans = max(ans, c);
   for (auto c : adiag) ans = max(ans, c);
   return ans;
}
```

Ref: #37 #52 #348 #351

53/363/918. Max subarray/Max sum of rectangle <= K/Circular array max subarray sum (review)

53 - Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

Given the array `[-2,1,-3,4,-1,2,1,-5,4]`, the contiguous subarray `[4,-1,2,1]` has the largest sum = 6.

Follow Up: max sum of rectangle in 2D matrix in $O(n^3)$, brute-force will be O(n^4).

363 - max sum of rectangle no larger than K

918 - max subarray with circular array (transform 2 cases into 1)

#152 - max product of subarray (keep both max-positive and min-negative product)
#238 - product of array except itself (2-way scan trick)

```
// 53 - soln-1: dynamic programming
// the above problem was 1st posed by Ulf Grenander of Brown University in 1977 (Kadane's algorithm)
// keep max_so_far and max_ending_here
// if max_ending_here + a[i] < 0, it means old part is not worth keeping.
// if asking longest subarray with largest sum, and max_ending_here + a[i] = 0, then index i is worth keeping.
int maxSubArray(vector<int>& nums) {
    int max_so_far = nums[0], max_ending_here = nums[0];
    for (int i = 1; i < nums.size(); ++i) {
        if (max_ending_here < 0 || max_ending_here + nums[i] < 0) {
            max_ending_here = nums[i];             // previous part is not worthy, so keep current.
        } else {
            max_ending_here += nums[i];
        }
        max_so_far = max(max_so_far, max_ending_here);
    }
    return max_so_far;
}
```

```
// Q: max sum of rectangle in 2D matrix (from G4G)
// soln-1: dynamic programming (max sum of rectangle)
// apply Kadane's algorithm for each row cumulatively, to get max sum so far.
// time: O(row * row * col), space: O(row)
int maxSubRectangle(vector<vector<int>>& matrix) {
   int ans = 0, row = matrix.size(), col = matrix[0].size();

   for (int ub = 0; ub < row; ++ub) {                // upper boundary
       vector<int> rowsum = matrix[ub];
       ans = max(ans, maxSubArray(rowsum));

       for (int bb = ub + 1; bb < row; ++bb) {      // bottom boundary
           for (int j = 0; j < col; ++j) rowsum[j] += matrix[bb][j];
           ans = max(ans, maxSubArray(rowsum));
       }
   }
   return ans;
}
```

```
// #363 helper: max sum of subarray less than K (sliding window)
// Kadane's algorithm not working for this problem, since
//     if max_ending_here < 0 then max_ending_here = 0
// imaging we have (left, right) for the subarray, the above statement will
// effectively reset left = right = right + 1, which will skip any item in between left and right.
//
// to find out if there is index-x such that sum of (x, curr] <= k, in other words, curr - k <= x,
// we can keep all the accumulated sum, then search for x. For example, given {2, 2, -1}, k = 3.
// accumulated sum: {0, 2, 4, 3},
//                      ~        ~
//              lower_bound   cur, so we will find the max sum of subarray close and lower than k.
// time: O(nlgn), space: O(n)
int maxKSubArray(vector<int>& nums, int k) {
   int cur = 0, ans = INT_MIN;
   set<int> sums;
   sums.insert(0);
   for (int n : nums) {
       cur += n;
       auto it = sums.lower_bound(cur - k);
       if (it != sums.end()) ans = max(ans, cur - *it);
       sums.insert(cur);
   }
   return ans;
}
```

```
// 363 - soln-1: dynamic programming (reuse max sum subarray less than k)
int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
   if (matrix.empty()) return 0;

   int ans = INT_MIN, row = matrix.size(), col = matrix[0].size();
   for (int ub = 0; ub < row; ++ub) {
       vector<int> rowsum(col);
       for (int bb = ub; bb < row; ++bb) {
           for (int j = 0; j < col; ++j) rowsum[j] += matrix[bb][j];

           ans = max(ans, maxKSubArray(rowsum, k));
       }
   }
   return ans;
}
```

```
// 918 - soln-1: two cases transform to one
// two cases for the target subarray:
// - starting < ending, maxSubarr is same as #53
```

```
// - starting > ending, maxSubarr = total - minSubarr
// - corner case: all negative, return maxSum instead of max(maxSum, total - minSum)
int maxSubarraySumCircular(vector<int>& A) {
    int curMax = A[0], curMin = A[0], sumMax = INT_MIN, sumMin = INT_MAX, total = A[0];
    for (int i = 1; i < A.size(); ++i) {
        curMax = max(curMax + A[i], A[i]), sumMax = max(sumMax, curMax);
        curMin = min(curMin + A[i], A[i]), sumMin = min(sumMin, curMin);
        total += A[i];
    }
    // sumMax < 0 means all negative
    return sumMax > 0 ? max(sumMax, total - sumMin) : sumMax;
}
```

```
// 1043 - Partition Array for Maximum Sum (A = [1,15,7,9,2,5,10], K = 3, ⇒ [15,15,15,9,10,10,10])
// soln-1: dynamic programming in O(nk) time
// let dp(i, j) be the max sum for A[i..j]
// dp(i, j+1) = max{dp(i, j+1-k) + max(A[k..j+1]) | k = 1..K}
int maxSumAfterPartitioning(vector<int>& A, int K) {
    vector<int> dp(A.size());
    dp[0] = A[0];
    for (int i = 1; i < A.size(); ++i) {
        for (int j = i, mx = A[j]; j >= max(0, i - K + 1); --j) {
            mx = max(mx, A[j]);              // rolling max from right to left
            if (j > 0) dp[i] = max(dp[i], mx * (i - j + 1) + dp[j - 1]);
            else       dp[i] = max(dp[i], mx * (i - j + 1) + 0);
        }
    }
    return dp.back();
}
```

Ref: #121 #152 #238

54/59/885. Spiral matrix/II/III

```
// 54/59 - spiral matrix/II. soln-1: brute-force
vector<int> spiralOrder(vector<vector<int>>& matrix) {
   if (matrix.size() == 0) return vector<int>{};

   int rows = matrix.size(), cols = matrix[0].size();
   vector<int> ans(rows * cols);

   int u = 0, d = rows - 1, l = 0, r = cols - 1, k = 0;
   while (true) {
       for (int j = l; j <= r; ++j) ans[k++] = matrix[u][j];
       if (++u > d) break;

       for (int i = u; i <= d; ++i) ans[k++] = matrix[i][r];
       if (--r < l) break;

       for (int j = r; j >= l; --j) ans[k++] = matrix[d][j];
       if (--d < u) break;

       for (int i = d; i >= u; --i) ans[k++] = matrix[i][l];
       if (++l > r) break;
   }

   return ans;
}
```

```
// 885 - spiral matrix III
vector<vector<int>> spiralMatrixIII(int R, int C, int r0, int c0) {
   vector<vector<int>> ans;
   for (int i = r0, j = c0, step = 1; ans.size() < R * C; ++step) {
       for (int x = i, y = j; y < j + step && ans.size() < R * C; ++y) {
```

```cpp
            if (x >= 0 && x < R && y >= 0 && y < C) ans.push_back(vector<int>{x, y});
        }
        j += step;
        for (int x = i, y = j; x < i + step && ans.size() < R * C; ++x) {
            if (x >= 0 && x < R && y >= 0 && y < C) ans.push_back(vector<int>{x, y});
        }
        i += step;

        ++step;

        for (int x = i, y = j; y > j - step && ans.size() < R * C; --y) {
            if (x >= 0 && x < R && y >= 0 && y < C) ans.push_back(vector<int>{x, y});
        }
        j -= step;
        for (int x = i, y = j; x > i - step && ans.size() < R * C; --x) {
            if (x >= 0 && x < R && y >= 0 && y < C) ans.push_back(vector<int>{x, y});
        }
        i -= step;
    }
    return ans;
}
```

Ref: #59

56/57/986. Merge/insert intervals/Interval intersection

```cpp
// 56 - merge intervals
// soln-1: sort by starting time
vector<Interval> merge(vector<Interval>& intervals) {
    vector<Interval> ans;
    if (intervals.size() == 0)  return ans;
    sort(intervals.begin(), intervals.end(), [](Interval& lhs, Interval& rhs) {
        return lhs.start < rhs.start;
    });

    Interval line(intervals[0].start, intervals[0].end);
    for (int i = 1; i < intervals.size(); ++i) {
        if (intervals[i].start > line.end) {
            ans.push_back(line);
            line = intervals[i];
        } else {
            line.start = min(line.start, intervals[i].start);
            line.end = max(line.end, intervals[i].end);
        }
    }
    ans.push_back(line);

    return ans;
}
```

```cpp
// 57 - insert interval
// soln-1: brute-force
// 1. form new intervals until overlap is found
// 2. merge overlap as one bigger interval
// 3. add the rest interval
vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
   vector<Interval> ans;
   int i = 0;
   for (NULL; i < intervals.size() && intervals[i].end < newInterval.start; ++i) ans.push_back(intervals[i]);

   for (NULL; i < intervals.size() && intervals[i].start <= newInterval.end; ++i) {
       newInterval.start = min(newInterval.start, intervals[i].start);
       newInterval.end = max(newInterval.end, intervals[i].end);
```

```
    }
    ans.push_back(newInterval);

    for (NULL; i < intervals.size(); ++i) ans.push_back(intervals[i]);

    return ans;
}
```

```
// 763 - Partition Labels (so that each letter appears in at most one part)
// soln-1: two pointers (mark segment and extend)
vector<int> partitionLabels(string S) {
    vector<pair<int, int>> pos(26, {INT_MAX, INT_MAX});
    for (int i = 0; i < S.length(); ++i) {
        auto& p = pos[S[i] - 'a'];
        (INT_MAX == p.first) ? p.first = p.second = i : p.second = i;
    }
    vector<int> ans;
    for (int i = 0; i < S.length(); ++i) {
        auto start = pos[S[i] - 'a'].first, end = pos[S[i] - 'a'].second;
        for (int j = start; j <= end; ++j) {
            end = max(end, pos[S[j] - 'a'].second);
        }
        ans.push_back(end - start + 1), i = end;
    }
    return ans;
}
```

```
// 986 - interval list intersections
// soln-1: two pointers
vector<Interval> intervalIntersection(vector<Interval>& A, vector<Interval>& B) {
    vector<Interval> ans;
    int i = 0, j = 0;
    while (i < A.size() && j < B.size()) {
        Interval seg(max(A[i].start, B[j].start), min(A[i].end, B[j].end));
        seg.end == A[i].end ? ++i : ++j;
        if (seg.start <= seg.end) ans.push_back(seg);
    }
    return ans;
}
```

Ref: #252 #253

61/189. Rotate list / array

```
// 61: soln-1: sliding window
ListNode* rotateRight(ListNode* head, int k) {
    if (!head) return head;

    int len = 0;
    for (auto h = head; h; h = h->next) len++;
    k %= len;
    if (!k) return head;

    // sliding window to get last k
    auto nh = head, tail = head;
    for (int i = 0; i < k; ++i) tail = tail->next;
    while (tail->next) nh = nh->next, tail = tail->next;

    tail->next = head, head = nh->next; nh->next = nullptr;
    return head;
}
```

```
// 189: soln-1: divide-and-conquer
// [1, 2, 3, 4, 5, 6, 7], k = 3
```

```
// [5, 6, 7, 4, 1, 2, 3]  <== 1st: get k nums in position
//            ~~~~~~~~~~   now we have n-k to rotate k, problem size reduced.
// so what is the exit condition?
// n keeps reducing till n <= k, if k % n == 0, there is not need to rotate.
void rotate(int nums[], int n, int k) {
   k %= n;
   if (k) {
       for (int i = 0; i < k; ++i) swap(nums[i], nums[n - k + i]);
       rotate(nums + k, n - k, k);
   }
}
void rotate(vector<int>& nums, int k) {
   rotate(nums.data(), nums.size(), k);
}
```

Ref:

62/63/980. Unique paths/II/III

#174 - dungeon game (DP from right-bottom to left-up side)
#741 - cherry pickup (two-leg DP)

```
// 62 - unique path
// soln-1: dynamic programming - f(i, j) = f(i-1, j) + f(i, j-1)
// soln-2: n-choose-k question - m+n-2 movements to reach destination
//          m-1 for down, n-1 for right movement
//          let's say the direction is horizontal and we could move down only, then m+n-2 choose m-1
//          if we assume a vertical direction and we could move right, then same answer as above
int uniquePaths(int m, int n) {
   if (0 == m || 0 == n) return 0;

   vector<int> dp(n, 1);
   for (int i = 1; i < m; ++i) {
       for (int j = 1; j < n; ++j) {
           dp[j] += dp[j - 1];
       }
   }
   return dp[n - 1];
}
```

```
// 63 - unique path with obstacles
// soln-1: dynamic programming - f(i, j) = f(i-1, j) + f(i, j-1) if grid[i][j] != 1
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
   if (obstacleGrid.empty() || obstacleGrid[0][0] == 1) return 0;

   int m = obstacleGrid.size(), n = obstacleGrid[0].size();
   vector<int> dp(n, 0);             // initialized as 0
   dp[0] = 1;
   for (int i = 0; i < m; ++i) {     // start from 0
       for (int j = 0; j < n; ++j) {
           if (obstacleGrid[i][j]) {
               dp[j] = 0;
           } else {
               if (j > 0) dp[j] += dp[j - 1];
           }
       }
   }
   return dp[n - 1];
}
```

```
// 980 - unique path III (walk over every cell once)
// soln-1: dfs (brute-force)
int helper(vector<vector<int>>& grid, int x, int y, int step, int total_step) {
```

```
    int row = grid.size(), col = grid[0].size();
    if (x < 0 || x >= row || y < 0 || y >= col || -1 == grid[x][y]) return 0;
    if (2 == grid[x][y]) return step == total_step ? 1 : 0;
    grid[x][y] = -1;
    int ans = helper(grid, x + 1, y, step + 1, total_step) +
              helper(grid, x - 1, y, step + 1, total_step) +
              helper(grid, x, y + 1, step + 1, total_step) +
              helper(grid, x, y - 1, step + 1, total_step);
    grid[x][y] = 0;
    return ans;
}
int uniquePathsIII(vector<vector<int>>& grid) {
    int x, y, steps = 0;
    for (int i = 0; i < grid.size(); ++i) {
        for (int j = 0; j < grid[0].size(); ++j) {
            if (1 == grid[i][j]) x = i, y = j;
            if (-1 != grid[i][j]) ++steps;
        }
    }
    return helper(grid, x, y, 1, steps);
}
```

```
// 1091 - Shortest Path in Binary Matrix
// soln-1: bfs
int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
    if (grid.empty()) return -1;
    queue<pair<int, int>> q;
    if (0 == grid[0][0]) q.push({0, 0});
    int ans = 1, m = grid.size(), n = grid[0].size();
    while (!q.empty()) {
        ++ans;
        for (int i = q.size(); i > 0; --i) {
            auto t = q.front();   q.pop();
            for (auto& d : vector<pair<int, int>>{{0, 1}, {0, -1}, ..., {-1, 1}, {1, -1}}) {
                auto x = t.first + d.first, y = t.second + d.second;
                if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y]) continue;
                if (x == m - 1 && y == n - 1) return ans;
                q.push({x, y}), grid[x][y] = 1;
            }
        }
    }
    return -1;
}
```

Ref: #64

64. Minimum path sum

Given a *m* x *n* grid filled with non-negative numbers, find a path from top left to bottom right which *minimizes* the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:

```
[[1,3,1],
 [1,5,1],
 [4,2,1]]
```

Given the above grid map, return `7`. Because the path 1→3→1→1→1 minimizes the sum.

```
int minPathSum(vector<vector<int>>& grid) {
   int m = grid.size(), n = grid[0].size();
   for (int j = 1; j < n; ++j) grid[0][j] += grid[0][j - 1];
```

```
    for (int i = 1; i < m; ++i) grid[i][0] += grid[i - 1][0];

    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            grid[i][j] += min(grid[i - 1][j], grid[i][j - 1]);
        }
    }
    return grid[m - 1][n - 1];
}
```

Ref:

66/369/989. Plus one/Linked list/Array plus integer

```
// 66 - plus one of vector number
// soln-1: brute-force (from right-to-left trick)
vector<int> plusOne(vector<int>& digits) {
    for (int i = digits.size() - 1; i >= 0; --i) {
        if (9 == digits[i]) digits[i] = 0;
        else { digits[i]++; return digits; }
    }
    digits[0]++, digits.push_back(0);    // case of “999”, avoid insert at beginning

    return digits;
}
```

```
// 160 - intersection of 2 linked lists
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    int la = length(headA), lb = length(headB);
    for (int i = 0; i < (la - lb); ++i) headA = headA->next;
    for (int i = 0; i < (lb - la); ++i) headB = headB->next;

    while (headA != headB) headA = headA->next, headB = headB->next;
    return headA;
}
```

```
// 369 - plus one of linked list
// soln-1: dfs
ListNode* helper(ListNode* head, ListNode* cur, int& carry) {
    if (cur->next) helper(head, cur->next, carry);

    cur->val += carry, carry = cur->val / 10, cur->val %= 10;
    if (carry && cur == head) return new ListNode(carry, head);

    return head;
}
```

```
// 989 - soln-1: brute-force (similar to 'Plus One')
vector<int> addToArrayForm(vector<int>& A, int K) {
    int carry = 0, j = K % 10;
    for (int i = A.size() - 1; (K || carry) && i >= 0; --i, K /= 10, j = K % 10) {
        A[i] += carry + j, carry = A[i] / 10, A[i] %= 10;
    }
    K += carry;
    for (int i = K % 10; K; K /= 10, i = K % 10) {
        A.insert(A.begin(), i);
    }
    return A;
}
```

Ref: #43 #67

68. Text justification

Given an array of words and a length *L*, format the text such that each line has exactly *L* characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces `' '` when necessary so that each line has exactly *L* characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

words: `["This", "is", "an", "example", "of", "text", "justification."]` L: `16`.

Return the formatted lines as:

```
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Note: Each word is guaranteed not to exceed *L* in length.

Ref:

70/746/931. Climbing stairs (with cost)/Min falling path sum (review)

```
// 70 - climbing stairs (allow 1 or 2 steps)
// soln-1: dynamic programming
// f(n) = f(n-1) + f(n-2)
int climbStairs(int n) {
   int f0 = 0, f1 = 1;
   for (int i = 1; i < n; ++i) {
       int f = f0 + f1;
       f0 = f1, f1 = f;
   }
   return f0 + f1;
}
```

```
// 746 - min cost climbing stairs (allow 1 or 2 steps with cost)
// soln-1: dynamic programming
// f(n) = cost[n] + min{f(n-1), f(n-2)}, answer is: min{f(n), f(n-1)} - coming from 1 and 2 steps respectively
int minCostClimbingStairs(vector<int>& cost) {
   int c0 = cost[0], c1 = cost[1];
   for (int i = 2; i < cost.size(); ++i) {
       int c = cost[i] + min(c1, c0);
       c0 = c1, c1 = c;
   }
   return min(c0, c1);
}
```

```
// 790 - Domino and Tromino Tiling
// soln-1: dynamic programming
// f(0) = 1, {}  - there is only one way to construct nothing.
// f(1) = 1, {|}
// f(2) = 2, {||, =}
// f(3) = 5, {f(2) + {|}, f(1) + {||}, f(0) + {LL', L'L}}
// f(n) = f(n-1) + f(n-2) + 2 *(f(n-3) + ... + f(0))  --- E1
```

```
// f(n-1) =        f(n-2) + 2 * (f(n-4) + ...+ f(0))  --- E2
// E1 - E2 => f(n) - f(n-1) = f(n-1) + f(n-3)
// .       => f(n) = 2*f(n-1) + f(n-3)
int numTilings(int N) {
    if (1 == N) return 1;
    if (2 == N) return 2;
    int ans = 0, a = 1, b = 1, c = 2, mod = 1e9+7;
    for (int i = 3; i <= N; ++i) {
        ans = (2 * c % mod + a) % mod;
        a = b, b = c, c = ans;
    }
    return ans % mod;
}
```

```
// 931 - Minimum Falling Path Sum
// soln-1: dynamic programming
int minFallingPathSum(vector<vector<int>>& A) {
    for (int i = 1; i < A.size(); ++i) {
        for (int j = 0; j < A[i].size(); ++j) {
            auto tmp = A[i - 1][j];
            if (j > 0) tmp = min(tmp, A[i - 1][j - 1]);
            if (j + 1 < A[i].size()) tmp = min(tmp, A[i - 1][j + 1]);
            A[i][j] += tmp;
        }
    }
    return *min_element(A.back().begin(), A.back().end());
}
```

Ref:

71/388/394/471/735/1003. Simply path/Decode string/Encode string with shortest length

```
// 71 - simplify path
// soln-1: stack
// - ".", skip
// - "..", pop back
// - others, push back
string simplifyPath(string path) {
   stringstream ss(path);

   vector<string> dirs;
   string ans, d;
   while (getline(ss, d, '/')) {
       if (d.empty() || d == ".") continue;
       if (d == "..") {
           if (!dirs.empty()) dirs.pop_back();
           else break;             // invalid path
       } else dirs.push_back(d);
   }

   for (string& dir : dirs) ans += "/" + dir;
   return ans;
}
```

```
// 388 - Longest Absolute File Path
// soln-1: stack
pair<int, int> getIndent(string& path, bool& file) {
   int indent = count(path.begin(), path.end(), '\t');
   file = count(path.begin(), path.end(), '.') >= 1;
   return {indent, path.length() - indent};
}

int lengthLongestPath(string input) {
   istringstream iss(input);
```

```
    vector<pair<int, int>> stk;     // <indent, length>
    int ans = 0;
    string path;
    while (getline(iss, path, '\n')) {
        bool file = false;
        auto x = getIndent(path, file);
        while (!stk.empty() && stk.back().first >= x.first) stk.pop_back();
        auto p = stk.empty() ? 0 : stk.back().second;
        stk.push_back({x.first, p + x.second});
        if (file) ans = max(ans, int(stk.back().second + stk.size() - 1) /* + levels */);
    }
    return ans;
}
// 388 - Longest Absolute File Path
// soln-2: state machine
int lengthLongestPathII(string input) {
    int ans = 0;
    vector<int> dirs(256, 0);       // support max 256 levels dir
    input.push_back('\n');          // force to terminate
    for (int i = 0, level = 0, len = 0, isFile = 0; i < input.size(); i++) {
        switch (input[i]) {
            case '\n':
                if (isFile) ans = max(ans, accumulate(dirs.begin(), dirs.begin() + level + 1, 0) + level);
                level = 0, len = 0, isFile = 0;
                break;
            case '\t':
                level++;
                break;
            default:
                if ('.' == input[i]) isFile = 1;
                len++, dirs[level] = len;
                break;
        }
    }
    return ans;
}
```

```
// 394 - decode string ("3[a]2[bc]" => "aaabcbc")
// soln-1: two stacks
string decodeString(string s) {
    vector<string> stkNum, stkStr;
    stkStr.push_back("");
    for (int i = 0; i < s.length(); ++i) {
        if ('[' == s[i]) {
        } else if (']' == s[i]) {
            auto x = stoi(stkNum.back());     stkNum.pop_back();
            auto ss = stkStr.back();          stkStr.pop_back();
            while (x--) stkStr.back() += ss;
        } else if (isdigit(s[i])) {
            stkNum.push_back(""), stkStr.push_back("");
            while (isdigit(s[i])) stkNum.back() += s[i++];
            --i;
        } else {
            stkStr.back().push_back(s[i]);
        }
    }
    return stkStr.back();
}
```

```
// 471 - Encode string with shortest length (rule: k[encoded_string])
// soln-1: dynamic programming (TODO)
```

```
// 636 - Exclusive Time of Functions
// soln-1: stack
vector<int> exclusiveTime(int n, vector<string>& logs) {
    vector<int> ans(n);
```

```
    vector<pair<int, int>> stk;         // <id, time-stamp>
    for (auto& log : logs) {
        auto id = stoi(log), ts = stoi(log.substr(log.find_last_of(":") + 1));
        if (log[log.find_first_of(":") + 1] == 's') {
            stk.push_back({id, ts});
        } else {
            auto prev = stk.back(); stk.pop_back();
            ans[id] += (ts - prev.second + 1);
            if (!stk.empty()) ans[stk.back().first] -= (ts - prev.second + 1);  // - taken time
        }
    }
    return ans;
}
```

```
// 1003 - Check If Word Is Valid After Substitutions (replace "abc" => "")
// soln-1: stack
bool isValid(string S) {
    vector<int> stk;
    for (auto ch : S) {
        if ('c' == ch) {
            int len = stk.size();
            if (len >= 2 && stk[len - 1] == 'b' && stk[len - 2] == 'a') stk.pop_back(), stk.pop_back();
            else return false;
        } else {
            stk.push_back(ch);
        }
    }
    return stk.empty();
}
```

```
// 735 - Asteroid Collision
// soln-1: stack
vector<int> asteroidCollision(vector<int>& asteroids) {
    vector<int> ans;
    for (auto x : asteroids) {
        ans.push_back(x);
        while (ans.size() > 1 && ans.back() < 0 && ans[ans.size() - 2] > 0) {
            auto x = ans.back();   ans.pop_back();
            auto y = ans.back();   ans.pop_back();
            if (abs(x) > y) ans.push_back(x);
            else if (abs(x) < y) ans.push_back(y);
        }
    }
    return ans;
}
```

Ref:

72/161. Edit distance/One edit distance

```
// 72 - edit distance (insert/delete/replace ops to convert to target)
// soln-1: dynamic programming
// let dp(i, j) be the min distance transforming from s to t
// s: -----i----
// t: -----j----
//     dp(i, j) = dp(i - 1, j - 1)                                        if s[i] == s[j], else
//              = 1 + min {dp(i - 1, j - 1), dp(i - 1, j), dp(i, j - 1)}
//                               replace          delete    insert after si
// space optimization: 2 rows are good.
int minDistance(string S, string T) {
    int m = T.length(), n = S.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    for (int j = 0; j <= n; ++j) dp[0][j] = j;
```

```
    for (int i = 1; i <= m; ++i) {
        dp[i][0] = i;
        for (int j = 1; j <= n; ++j) {
            if (T[i - 1] == S[j - 1]) dp[i][j] = dp[i - 1][j - 1];
            else dp[i][j] = 1 + min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1]));
        }
    }
    return dp[m][n];
}
```

```
// 161 - one distance (determine if 2 strings are 1 edit distance apart)
// soln-1: recursion
bool isOneEditDistance(string s, string t) {
    int m = s.length(), n = t.length();
    if (abs(m - n) > 1) return false;

    if (abs(m - n) == 0) {                              // same length, try to replace
        bool replaced = false;
        for (int i = 0; i < s.length(); ++i) {
            if (s[i] != t[i] && replaced) return false;
            else replaced = true;
        }
        return true;
    }

    for (int i = 0; i < min(m, n); ++i) {               // length diff. = 1, only choice is to remove the
extra
        if (s[i] == t[i]) continue;
        return m > n ? isOneEditDistance(s.substr(i + 1), t.substr(i)) :  // remove from longer one
                       isOneEditDistance(s.substr(i), t.substr(i + 1));
    }
    return true;
}
```

Ref: #161

73. Set matrix zeros (review)

Given a *m* x *n* matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Follow up:
Did you use extra space? A straightforward solution using O(*mn*) space is probably a bad idea. A simple improvement uses O(*m* + *n*) space, but still not the best solution. Could you devise a constant space solution?

Solution:
Tricky for contant space soln.
#289 - Game of life

```
// use first rol & col of matrix to mark if we need to clean, with 2-tricks:
// 1, special position at (0, 0), need one extra flag.
// 2, we have to go bottom-up way.
void setZeroes(vector<vector<int> > &matrix) {
    int m = matrix.size(), n = matrix[0].size();

    bool zeroCol = false;
    for (int i = 0; i < m; ++i) {
        if (0 == matrix[i][0]) zeroCol = true;  // mark it first
        for (int j = 1; j < n; ++j) {           // start from col-1
            if (0 == matrix[i][j]) matrix[0][j] = matrix[i][0] = 0;
        }
    }
    for (int i = m - 1; i >= 0; --i) {          // bottom-up traversal
        for (int j = n - 1; j >= 1; --j) {      // end to col-1
```

```
            if (0 == matrix[i][0] || 0 == matrix[0][j]) matrix[i][j] = 0;
        }
        if (zeroCol) matrix[i][0] = 0;          // do it last
    }
}
```

Ref: #289

74/204. Search a 2d matrix/II

```
// 74 - search in 2d matrix
// soln-1: binary search
bool searchMatrix(vector<vector<int>>& matrix, int target) {
   if (matrix.empty() || matrix[0].empty()) return false;

   int row = bs(matrix, target);
   if (matrix.size() == row) return false;

   return bs(matrix[row], target);
}

// return 1st position which bigger than target
int bs(vector<vector<int>>& nums, int target) {
   int low = 0, hi = nums.size() - 1, col = nums[0].size() - 1;
   while (low + 1 < hi) {
       int m = low + (hi - low) / 2;
       target < nums[m][col] ? hi = m : low = m;
   }
   if (nums[low][col] >= target) return low;
   if (nums[hi][col] >= target) return hi;
   return hi + 1;
}

bool bs(vector<int>& nums, int target) {
   int low = 0, hi = nums.size() - 1;
   while (low + 1 < hi) {
       int m = low + (hi - low) / 2;
       target < nums[m] ? hi = m : low = m;
   }
   return nums[low] == target || nums[hi] == target;
}
```

```
// 240 - search in 2d matrix II (inputs are sorted, but first num of row isn't bigger than last num of prev row)
// soln-1: from top-right to left-bottom in O(m+n) time.
// binary search will not work for this question.
bool searchMatrix(vector<vector<int>>& matrix, int target) {
   if (matrix.empty() || matrix[0].empty()) return false;

   int row = matrix.size(), col = matrix[0].size();
   for (int i = 0, j = col - 1; i < row && j >= 0; NULL) {
       if (matrix[i][j] == target) return true;
       target > matrix[i][j] ? ++i : --j;
   }
   return false;
}
```

Ref:

75/360/791/977. Sort colors/Custom sort

```
// 75 - sort colors (n objects with 3 colors, sort it by color)
// soln-1: bucket sort
void sortColors(vector<int>& nums) {
```

```
    array<int, 3> bucket;
    for (int x : nums) bucket[x]++;
    for (int pos = 0, i = 0; i < bucket.size(); ++i) {
        for (int k = 0; k < bucket[i]; ++k) {
            nums[pos++] = i;
        }
    }
}
// 75 - sort colors (n objects with 3 colors, sort it by color)
// soln-2: when we found red, it means all white/red need to move right
//         similarly if found white, red need to move right.
void sortColors(vector<int>& nums) {
    int r = -1, w = -1, b = -1;
    for (int i = 0; i < nums.size(); ++i) {
        switch(nums[i]) {
        case 0:
            nums[++b] = 2;    nums[++w] = 1;    nums[++r] = 0;  break;
        case 1:
            nums[++b] = 2;    nums[++w] = 1;  break;
        case 2:
            nums[++b] = 2;  break;
        }
    }
}
// 75 - sort colors (n objects with 3 colors, sort it by color)
// soln-3: red/blue pointers from head/tail, swap current val if it's red/blue
void sortColors(vector<int>& nums) {
    int red = 0, blue = nums.size() - 1;
    for (int i = 0; i <= blue; ++i) {
        switch(nums[i]) {
        case 0:
            swap(nums[red++], nums[i]); break;
        case 2:
            swap(nums[blue--], nums[i--]);  break; // blue could be pointing to 0
        }
    }
}
```

```
// 360 - Sort Transformed Array (given sorted array, apply f(x) and make it sorted)
// soln-1: two pointers (merge sort)
int f(int a, int b, int c, int x) {
    return a * x * x + b * x + c;
}
vector<int> sortTransformedArray(vector<int>& nums, int a, int b, int c) {
    for (auto& x : nums) x = f(a, b, c, x);
    vector<int> ans;
    if (a > 0) {    // e.g. y = x^2, lowest is in the middle, then <-- | -->
        int l = min_element(nums.begin(), nums.end()) - nums.begin(), r = l + 1, n = nums.size();
        while (l >= 0 && r < n) nums[l] < nums[r] ? ans.push_back(nums[l--]) : ans.push_back(nums[r++]);
        while (l >= 0) ans.push_back(nums[l--]);
        while (r < nums.size()) ans.push_back(nums[r++]);
    } else {        // otherwise go with --> | <-- (those 2 cases could be merged in 1, see #977)
        for (int l = 0, r = nums.size() - 1; l <= r; nullptr) {
            nums[l] < nums[r] ? ans.push_back(nums[l++]) : ans.push_back(nums[r--]);
        }
    }
    return ans;
}
```

```
// 977 - Squares of a Sorted Array
// soln-1: two pointers (merge sort)
vector<int> sortedSquares(vector<int>& A) {
     for (auto& x : A) x = x * x;
```

```
    vector<int> ans(A.size());
    for (int n = A.size(), l = 0, r = n - 1, t = r; l <= r; --t) {
        if (A[l] > A[r]) ans[t] = A[l], ++l;
        else ans[t] = A[r], --r;
    }
    return ans;
}
```

```
// 791 - Custom Sort String (per the order of S, sort T)
// soln-1: counting in O(T) instead of O(TlgT)
string customSortString(string S, string T) {
    vector<int> cnt(26);
    for (auto c : T) cnt[c - 'a']++;

    string ans;
    for (auto c : S) {
        while (cnt[c - 'a']-- > 0) ans.push_back(c);
    }
    for (auto c : T) {
        if (cnt[c - 'a']-- > 0) ans.push_back(c);
    }
    return ans;
}
```

```
// 912 - Sort an Array
// soln-1: count sort
vector<int> sortArray(vector<int>& nums) {
    const int R = 50000;
    vector<int> cnt(R + R + 1);
    for (auto& x : nums) cnt[x + R]++;
    for (int i = 0, j = 0; j < cnt.size(); ++j) {
        for (int k = 0; k < cnt[j]; ++k) nums[i++] = j - R;
    }
    return nums;
}
```

Ref: #26 #280 #324

79/212. Word search

```
// 79 - Word Search (if a word exist in matrix)
// soln-1: dfs (go thru. matrix, start searching if match word[0])
bool h(vector<vector<char>>& board, int x, int y, string& word, int k) {
    if (k >= word.length()) return true;
    if (x < 0 || x >= board.size() || y < 0 || y >= board[0].size() || board[x][y] != word[k]) return false;
    char tmp = 0;
    swap(board[x][y], tmp);
    if (h(board, x + 1, y, word, k + 1) ||
        h(board, x - 1, y, word, k + 1) ||
        h(board, x, y + 1, word, k + 1) ||
        h(board, x, y - 1, word, k + 1)) return true;
    swap(board[x][y], tmp);
    return false;
}
bool exist(vector<vector<char>>& board, string word) {
    for (int m = board.size(), n = board[0].size(), i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (board[i][j] == word.front() && h(board, i, j, word, 0)) return true;
        }
    }
    return false;
}
```

```
// 212 - Word Search II (how many words in word list exist in matrix)
// soln-1: dfs + trie
```

```
// 1. if we brute-force check each word from the board, worst case in O(k * n^2 * avg-word-length)
// 2. use words to build trie. for each cell in board, ask if there is any word start from it
//                              if yes, for the next 4 direction, keep asking
// 3. time complexity is O(n^2 * avg-word-length)
void insert(string& word, TrieNode* node) {
    for (auto ch : word) {
        if (0 == node->v[ch - 'a']) node->v[ch - 'a'] = new TrieNode;
        node = node->v[ch - 'a'];
    }
    node->word = word;
}
void dfs(vector<vector<char>>& board, int x, int y, TrieNode* node, vector<string>& ans) {
    if (x < 0 || x >= board.size() || y < 0 || y >= board[0].size() || !node || 0 == board[x][y]) return;
    node = node->v[board[x][y] - 'a'];
    if (node && !node->word.empty()) ans.push_back(node->word), node->word.clear();

    char tmp = 0;
    swap(board[x][y], tmp);          // mark it as visited
    dfs(board, x + 1, y, node, ans), dfs(board, x - 1, y, node, ans);
    dfs(board, x, y + 1, node, ans), dfs(board, x, y - 1, node, ans);
    swap(board[x][y], tmp);
}
vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
    vector<string> ans;
    TrieNode trie;
    for (auto& w : words) insert(w, &trie);

    for (int i = 0; i < board.size(); ++i) {
        for (int j = 0; j < board[0].size(); ++j) {
            dfs(board, i, j, &trie, ans);
        }
    }
    return ans;
}
```

Ref: #417

82/83. Remove duplicates from sorted list II

Delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. E.g. given 1->1->1->2->3, return 2->3.

```
// 82 - remove dups including itself
ListNode* deleteDuplicates(ListNode* head) {
   ListNode dummy(0);
   dummy.next = head;
   for (ListNode* pre = &dummy, *cur = head; cur; cur = cur->next){
       bool dup = false;
       while (cur && cur->next) {
           if (cur->val != cur->next->val) break;
           cur->next = cur->next->next, dup = true;
       }
       dup ? pre->next = cur->next : pre = cur;
   }
   return dummy.next;
}
```

```
// 83 - remove dups not including itself
ListNode* deleteDuplicates(ListNode* head) {
   ListNode dummy(0);
   dummy.next = head;
   ListNode* cur = head;
   while (cur && cur->next) {
       if (cur->next->val == cur->val) cur->next = cur->next->next;
```

```
        else cur = cur->next;
    }
    return dummy.next;
}
```

Ref:

86. Partition list

Partition a linked list by given value x, such that nodes less than x come before nodes greater than or equal to x.
E.g. given 1->4->3->2->5->2, and x = 3, return 1->2->2->4->3->5

```
// recursive soln
ListNode* partition(ListNode* head, int x) {
    if (head == NULL) return head;

    ListNode* h1, *h2, *t1, *t2;
    h1 = h2 = t1 = t2 = NULL;

    helper(head, x, h1, h2, t1, t2);
    if (t1) t1->next = h2;
    if (t2) t2->next = NULL;
    return h1 ? h1 : h2;
}

void helper(ListNode* head, int x, ListNode* &h1, ListNode* &h2, ListNode* &t1, ListNode* &t2) {
    if (head == NULL) return;

    if (head->val < x) {
        if (h1 == NULL) h1 = head;
        if (t1) t1->next = head;
        t1 = head;
    } else {
        if (h2 == NULL) h2 = head;
        if (t2) t2->next = head;
        t2 = head;
    }
    helper(head->next, x, h1, h2, t1, t2);
}

// iterative soln
ListNode* partition(ListNode* head, int x) {
    if (head == NULL) return head;

    ListNode* h1, *h2, *t1, *t2;
    h1 = h2 = t1 = t2 = NULL;

    while (head) {
        if (head->val < x) {
            if (h1 == NULL) h1 = head;
            if (t1) t1->next = head;
            t1 = head;
        } else {
            if (h2 == NULL) h2 = head;
            if (t2) t2->next = head;
            t2 = head;
        }
        head = head->next;
    }

    if (t1) t1->next = h2;
    if (t2) t2->next = NULL;
    return h1 ? h1 : h2;
}
```

Ref:

87. Scramble string

Given a string *s1*, we may represent it as a binary tree by partitioning it to two non-empty substrings recursively.

Given two strings *s1* and *s2* of the same length, determine if *s2* is a scrambled string of *s1*.
Example 1:

```
Input: s1 = "great", s2 = "rgeat", Output: true
```

Example 2:

```
Input: s1 = "abcde", s2 = "caebd", Output: false
```

```
// soln-1: recursion
// f(s1, s2) = f(s1[0..i], s2[0..i]) && f(s1[i+1..n], s2[i+1..n]) for (i = 1~n-1)
//               or
//               f(s1[0..i], s2[i+1..n]) && f(s1[i+1..n], s2[0..i])
// exit condition: count occurrence of chars
bool f(string s1, string s2) {
   if (s1 == s2) return true;
   if (s1.length() != s2.length()) return false;

   int count[26] = {0}, n = s1.length();
   for (int i = 0; i < n; ++i) {
       count[s1[i] - 'a']++;   count[s2[i] - 'a']--;
   }
   for (int x : count) if (x != 0) return false;

   for (int i = 1; i < n; ++i) {
       if (f(s1.substr(0, i), s2.substr(0, i)) && f(s1.substr(i), s2.substr(i))) return true;
       if (f(s1.substr(0, i), s2.substr(n - i)) && f(s1.substr(i), s2.substr(0, n - i))) return true;
   }
   return false;
}
```

Ref:

89. Gray code

The gray code is a binary numeral system where two successive values differ in only one bit. For example, sequence 00/01/11/10 is a valid gray code sequence.
Note: for a given n, a gray code sequence is not uniquely defined.

Solution:
Let's use a few example to find the pattern.

n	Gray code (in binary)	
1	0 1	
2	00 01 11 10	flip the highest bit for the previous results from right to left
3	000 001 011 010 110 111 101 100	flip the highest bit for the previous results from right to left

soln-1: iterative: from the above table, the pattern is not hard to find, and the iterative soln is easy to implement.
soln-2: recursive - how to explain???
soln-3: there is a formula: G(i) = i XOR (i>>1)

```
// iterative soln: right-to-left flip the high bit
vector<int> grayCode(int n) {
    vector<int> ans;
    ans.push_back(0);

    for (int i = 0; i < n; ++i) {
        for (int j = ans.size() - 1; j >= 0; --j) {       // from right-to-left direction
            ans.push_back(ans[j] | (1 << i));             // flip bit-i (the highest bit for current loop)
        }
    }
```

```
    return ans;
}

// recursive soln: how to explain?
void grayCode(int& code, int k, vector<int>& ans) {
    if (k == 0) {
        ans.push_back(code);    return;
    }

    grayCode(code, k - 1, ans);
    code ^= (1 << (k - 1));     // flip bit k-1
    grayCode(code, k - 1, ans);
}

// formula: G(x) = x XOR (x>>1)
vector<int> grayCode(int n) {
    vector<int> ans;
    for (int i = 0; i < (1 << n); ++i) {
        ans.push_back(i ^ (i >> 1));
    }

    return ans;
}
```

Ref:

91/639. Decode ways / II

A message containing letters from `A-Z` is being encoded to numbers using the following mapping:

`'A' -> 1, 'B' -> 2, …, 'Z' -> 26`

Given a non-empty string containing only digits, determine the total number of ways to decode it.

Example 1:

```
Input: "12", Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
```

`Follow up - #639:`

Beyond that, now the encoded string can also contain the character '*', which can be treated as one of the numbers from 1 to 9.

`Example 2:`

`Input: "1*", Output: 9 + 9 = 18`

```
// soln-1: dynamic programming
// f(i) = f(i-1) iff s[i] is decodable, plus
//        f(i-2) iff s[i-1..i] is decodable
// f(0) = 1 - current is decodable and reaching empty string
int numDecodings(string s) {
   vector<int> dp(s.length() + 1);
   dp[0] = 1;                  // current is decodable and reaching empty string

   for (int i = 1; i <= s.length(); ++i) {
       if (s[i - 1] > '0' && s[i - 1] <= '9') dp[i] = dp[i - 1];

       if (i > 1 && s[i - 2] != '0' &&
           stoi(s.substr(i - 2, 2)) > 0 &&
           stoi(s.substr(i - 2, 2)) <= 26) dp[i] += dp[i - 2];
   }
   return dp[s.length()];
```

```
}

// soln-1: dynamic programming
// if s[i] = '*', then f(i) = f(i - 1) * 9, plus
//                            f(i - 2) * (6|9|15) if s[i-1] == '2|1|*'
// else f(i) = f(i - 1) iff s[i] is decodable, plus
//             f(i - 2) if s[i-1..i] is decodable
//                      elif s[i-1] == '*',
//                              if (s[i] == '0') f(i) += f(i-2) * 2     // 10/20
//                              else f(i) += f(i-2)                     // 1x
//                                   f(i) += f(i-2) if s[i] <= '6'      // 2x
// f(0) = 1 - current is decodable and reaching empty string
int numDecodings(string s) {
   vector<unsigned long> dp(s.length() + 1);
   dp[0] = 1;
   const int M = 1e9+7;

   for (int i = 1; i <= s.length(); ++i) {
       char curr = s[i - 1];
       if (curr == '*') {             // ex. - *
           dp[i] = dp[i - 1] * 9;
           if (1 == i) continue;

           if (s[i - 2] == '1')            dp[i] += dp[i - 2] * 9;     // 11..19
           else if (s[i - 2] == '2')       dp[i] += dp[i - 2] * 6;     // 21..26
           else if (s[i - 2] == '*')       dp[i] += dp[i - 2] * 15;    // 9 + 6
       } else {
           if (curr > '0' && curr <= '9')  dp[i] = dp[i - 1];          // s[i] decodable
           if (1 == i) continue;

           if (s[i - 2] > '0' && s[i - 2] <= '9') {   // ex. - 12
               int t = stoi(s.substr(i - 2, 2));
               if (t > 0 && t <= 26)       dp[i] += dp[i - 2];         // s[i-1..i] decodable
           } else if (s[i - 2] == '*') {              // ex. - *1
               if (curr == '0') dp[i] += dp[i - 2] * 2;                // 10/20
               else {
                   dp[i] += dp[i - 2];                                 // 1x
                   if (curr <= '6') dp[i] += dp[i - 2];                // 2x
               }
           }
       }
       dp[i] %= M;
   }
   return dp[s.length()];
}
```

Ref:

92/206. Reverse linked list

[#2](#2) - add two numbers stored in linked list
[#141](#141) - Floyd linked list cycle detection (fast-slow pointer: how to prove correctness?)
[#142](#142) - find the entry node if there is a cycle for linked list
[#143](#143) - re-order linked list

```
ListNode* reverseBetween(ListNode* head, int m, int n) {
   ListNode dummy(0);
   dummy.next = head;

   ListNode* tailBeforeM = &dummy, *cur = head;
   for (int i = 1; cur && i < m; ++i) tailBeforeM = cur, cur = cur->next;
```

```
    // reverse (n - m + 1) nodes starting from cur
    ListNode* pre = nullptr, *tailBeforeN = cur;
    for (int i = 0; i <= n - m && cur; ++i) {
        ListNode* next = cur->next;
        cur->next = pre, pre = cur, cur = next;
    }
    // chain the rest
    if (tailBeforeN) tailBeforeN->next = cur;
    if (tailBeforeM) tailBeforeM->next = pre;

    return dummy.next;
}
```

```
ListNode* reverseList(ListNode* h) {
    ListNode d(0);
    while (h) {
        auto next = h->next;
        h->next = d.next, d.next = h, h = next;
    }
    return d.next;
}

ListNode* reverseList(ListNode* head) {
    if (head == NULL || head->next == NULL) return head;
    auto first = head, * rest = head->next;

    auto nh = reverseList(rest);
    first->next->next = first;    // tricky part: chain the first node to the rests
    first->next = NULL;

    return nh;
}
```

Ref:

93/468/816. Restore/validate IP address/Coordinates

```
// 93 - restore IP addr ("24424411135" -> ["225.255.11.135", "225.255.111.35"])
// soln-1: backtracking
vector<string> restoreIpAddresses(string s) {
    vector<string> ans,  ip;
    helper(s, 3, ip, ans);

    return ans;
}

void helper(string s, int k, vector<string>& ip, vector<string>& ans) {
    if (k == 0) {
        if (s.empty() || s.length() > 3 || || stoi(s) > 255) return;
        if (s.length() > 1 && s[0] == '0') return;
        ans.push_back(ip[0] + "." + ip[1] + "." + ip[2] + "." + s);
        return;
    }

    for (int i = 1; i <= min(3, (int)s.length()); ++i) {
        string prefix = s.substr(0, i);
        if (stoi(prefix) > 255) continue;
        if (prefix.length() > 1 && prefix[0] == '0') continue;         // skip "01x"

        ip.push_back(prefix);
        helper(s.substr(i), k - 1, ip, ans);
        ip.pop_back();
```

```
    }
}
```

```
// 468 - Validate IP Address
// soln-1: brute-force string processing
string validIPAddress(string IP) {
    if (count(IP.begin(), IP.end(), '.') == 3) return IPv4(IP);
    if (count(IP.begin(), IP.end(), ':') == 7) return IPv6(IP);
    return "Neither";
}
string IPv4(string& IP) {
    istringstream iss(IP);
    string ip;
    int seg = 0;
    while (getline(iss, ip, '.')) {
        ++seg;
        if (ip.empty() || ip.length() > 3) return "Neither";
        if (!isdigit(ip[0]) || stoi(ip) > 255) return "Neither";
        if (ip.length() != to_string(stoi(ip)).length()) return "Neither";
    }
    return 4 == seg ? "IPv4" : "Neither";
}
string IPv6(string& IP) {
    istringstream iss(IP);
    string ip;
    int seg = 0;
    while (getline(iss, ip, ':')) {
        ++seg;
        if (ip.empty() || ip.length() > 4)   return "Neither";
        for (auto& x : ip) {
            if (x >= '0' && x <= '9') {
            } else if (x >= 'a' && x <= 'f') {
            } else if (x >= 'A' && x <= 'F') {
            } else {
                return "Neither";
            }
        }
    }
    return 8 == seg ? "IPv6" : "Neither";
}
```

```
// 816 - Ambiguous Coordinates ("(00011)" -> ["(0.001, 1)", "(0, 0.011)"])
// soln-1: brute-force split string into two and try to form a valid number
vector<string> formNumber(string str) {
    vector<string> ans;
    if (str.length() <= 1) return vector<string>{str};
    if ('0' == str[0]) {
        if (str.back() != '0') return vector<string>{"0." + str.substr(1)}; // can insert only 1 "."
    } else {
        if (str.back() == '0') return vector<string>{str};                  // can not insert "."

        ans.push_back(str);
        for (auto i = 1; i < str.length(); ++i) {                           // insert "." into every gap
            ans.push_back(str.substr(0, i) + "." + str.substr(i));
        }
    }
    return ans;
}
vector<string> ambiguousCoordinates(string S) {
    S = S.substr(1, S.length() - 2);
    unordered_set<string> ans;
    for (int i = 1; i < S.length(); ++i) {
        auto xs = formNumber(S.substr(0, i)), ys = formNumber(S.substr(i));
        for (auto& x : xs) {
            for (auto& y : ys) ans.insert("(" + x + ", " + y + ")");
        }
```

```
    }
    return vector<string> {ans.begin(), ans.end()};
}
```

Ref: #65

95/96/823/894. Unique binary search trees/II/Binary tree factors/All full binary trees

Given n, how many structurally unique bst that stores 1...n? generate all structurally unique bst that stores 1...n.

```
// 95 - soln-1: dynamic programming
// dp(n) = ∑ −1 =0  {dp(i) * dp(n - i - 1)}
// dp(0) = 1 (only empty tree)
// dp(1) = 1 (can be only root node)
int numTrees(int n) {
    vector<int> dp(n + 1, 0);
    dp[0] = dp[1] = 1;

    for (int i = 2; i <= n; ++i) {
        for (int j = 0; j < i; ++j) dp[i] += dp[j] * dp[i - j - 1];
    }
    return dp[n];
}
```

```
// 96 - soln-1: brute force enumerate
vector<TreeNode*> generateTrees(int n) {
   return helper(1, n);
}

vector<TreeNode*> helper(int start, int end) {
   vector<TreeNode*> ans;
   for (int i = start; i <= end; ++i) {
       combine(i, helper(start, i - 1), helper(i + 1, end), ans);
   }
   return ans;
}

void combine(int i, vector<TreeNode*>& left, vector<TreeNode*>& right, vector<TreeNode*>& ans) {
   if (left.size() == 0)   left.push_back(NULL);
   if (right.size() == 0)  right.push_back(NULL);
   for (TreeNode* l : left) {
       for (TreeNode* r : right) {
           ans.push_back(new TreeNode(i, copyTree(l), copyTree(r)));
       }
   }
}
```

```
// 823 - binary tree with factors
// soln-1: dynamic programming
// f(x) = dp(i) * dp(j) if x = i * j.
int numFactoredBinaryTrees(vector<int>& A) {
   sort(A.begin(), A.end());

   long ans = 0, mod = 1e9 + 7;
   unordered_map<int, long> dp({{1, 1}});   // <val, ans>
   for (int i = 0; i < A.size(); ++i) {
       int cur = 1;                         // cur itself as root
       for (int j = 0; j < i; ++j) {
           if (A[i] % A[j]) continue;

           int t = A[i] / A[j];
           if (dp.find(t) != dp.end()) cur = (cur + dp[t] * dp[A[j]]) % mod;
       }
       dp[A[i]] = cur, ans = (ans + cur) % mod;
   }
```

```
    return ans;
}
```

```
// 894 - all possible full binary trees
// soln-1: brute-force
unordered_map<int, vector<TreeNode*>> _mp;
vector<TreeNode*> allPossibleFBT(int N) {
    if (1 == N) return vector<TreeNode*>{new TreeNode(0)};
    if (_mp.find(N) != _mp.end()) return _mp[N];

    vector<TreeNode*> ans;
    for (int i = 1; i < N; i += 2) {
        for (auto& l : allPossibleFBT(i)) {
            for (auto& r : allPossibleFBT(N - i - 1)) {
                auto n = new TreeNode(0);
                n->left = copyTree(l), n->right = copyTree(r);
                ans.push_back(n);
            }
        }
    }
    return _mp[N] = ans;
}
```

Ref:

97. Interleaving string (review)

Given *s1*, *s2*, *s3*, find whether *s3* is formed by the interleaving of *s1* and *s2*. For example, given: *s1* = `"aabcc"`, *s2* = `"dbbca"`, When *s3* = `"aadbbcbcac"`, return true. When *s3* = `"aadbbbaccc"`, return false.

```
// soln-1: recursion
bool isInterleave(string s1, string s2, string s3) {
   int l1 = s1.length(), l2 = s2.length(), l3 = s3.length();
   if (!l1 && !l2 && !l3)      return true;
   if (l1 + l2 != l3)    return false;

   return (l1 && s1.back() == s3.back() && isInterleave(s1.substr(0, l1 - 1), s2, s3.substr(0, l3 - 1))) ||
          (l2 && s2.back() == s3.back() && isInterleave(s1, s2.substr(0, l2 - 1), s3.substr(0, l3 - 1)));
}

// soln-2: dynamic programming
// consider s1[i], s2[j] then s3[i + j], let dp[i][j] be the interleaving result:
// 1. s1[i] == s3[i + j] && dp[i - 1][j], or
// 2. s2[j] == s3[i + j] && dp[i][j - 1]
// 3. space could be O(1) - only depend on up and left one.
bool isInterleave(string s1, string s2, string s3) {
   int m = s1.length(), n = s2.length();
   if (m + n != s3.length()) return false;

   vector<vector<bool>> dp(m + 1, vector<bool>(n + 1));
   dp[0][0] = true;              // s1/2/3 are all empty string
   for (int i = 1; i <= m; ++i) dp[i][0] = dp[i - 1][0] && (s1[i - 1] == s3[i - 1]);
   for (int j = 1; j <= n; ++j) dp[0][j] = dp[0][j - 1] && (s2[j - 1] == s3[j - 1]);

   for (int i = 1; i <= m; ++i) {
       for (int j = 1; j <= n; ++j) {
           dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[i - 1 + j]) ||
                      (dp[i][j - 1] && s2[j - 1] == s3[i - 1 + j]);
       }
   }
   return dp[m][n];
}
```

Ref:

98/110/333/530. Validate BST / balanced BST / largest BST/min abs diff. in bst

```
// 98 - Instead of using INT_MIN/INT_MAX as boundary, a better way is to keep node pointers.
bool isValidBST(TreeNode* root, TreeNode* minNode = nullptr, TreeNode* maxNode = nullptr) {
    if (root == NULL) return true;
    if (minNode && minNode->val >= root->val) return false;
    if (maxNode && maxNode->val <= root->val) return false;

    return isValidBST(root->left, minNode, root) && isValidBST(root->right, root, maxNode);
}

// 110 - soln-1: bottom-up in O(n) time
bool isBalanced(TreeNode* root, int* height = nullptr) {
   if (root == nullptr) {
       if (height) *height = 0;
       return true;
   }

   int lh = 0, rh = 0;
   bool lb = isBalanced(root->left, &lh);
   bool rb = isBalanced(root->right, &rh);
   if (height) *height = (1 + max(lh, rh));
   return lb && rb && (abs(lh - rh) <= 1);
}

// 333 - soln-1: bottom-up in O(n) time
// return true if bst, sz holds the largest bst so far
bool helper(TreeNode* r, TreeNode* lb, TreeNode* rb, int& sz) {
   if (!r) return true;
   if (lb && lb->val >= r->val) return false;
   if (rb && rb->val <= r->val) return false;

   int left_n = 0, right_n = 0;
   int left = helper(r->left, lb, r, left_n);  // update boundary
   int right = helper(r->right, r, rb, right_n);
   if (left && right) {
       sz = max(sz, left_n + right_n + 1);     // update largest bst so far
   } else {
       sz = max(sz, max(left_n, right_n));     // update largest bst so far
   }
   return left && right;
}
```

```
int _ans = INT_MAX;
// 530 - soln-1: add left/right boundary then apply validate bst
int getMinimumDifference(TreeNode* root, TreeNode* lb = nullptr, TreeNode* rb = nullptr) {
   if (!root) return _ans;
   if (lb) _ans = min(_ans, abs(root->val - lb->val));
   if (rb) _ans = min(_ans, abs(root->val - rb->val));

   getMinimumDifference(root->left, lb, root);
   getMinimumDifference(root->right, root, rb);
   return _ans;
}
```

Ref: #94

99. Recover binary search tree

Two elements of a bst are swapped by mistake, recover it.

```
// 99 - soln-1: dfs
// if only 2 nodes are mistake, swap them.
// if 3 nodes are mistake, swap 1<->3.
void recoverTree(TreeNode* root) {
    TreeNode* n1 = nullptr, *n2 = nullptr, *pre = nullptr;

    helper(root, pre, n1, n2);
    swap(n1->val, n2->val);
}

void helper(TreeNode* cur, TreeNode*& pre, TreeNode*& n1, TreeNode*& n2) {
    if (cur == nullptr) return;

    helper(cur->left, pre, n1, n2);
    if (pre && pre->val > cur->val) {
        if (n1 == nullptr) n1 = pre;
        n2 = cur;
    }
    pre = cur;
    helper(cur->right, pre, n1, n2);
}
```

```
// 99 - soln-2: TODO Morris Traversal
```

Ref: Morris Traversal

100/101/104/111/226. Same/Symmetric/Max-depth/Min-depth/Invert binary tree/Fip equivalent

```
// 100: is same tree
bool isSameTree(TreeNode* p, TreeNode* q) {
   if (!p && !q) return true;
   if (p && q && p->val == q->val) return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
   return false;
}
```

```
// 101: is symmetric tree
bool helper(TreeNode* r1, TreeNode* r2) {
   if (!r1 && !r2) return true;
   if (r1 && r2 && r1->val == r2->val) {
       return helper(r1->left, r2->right) && helper(r1->right, r2->left);
   }
   return false;
}
bool isSymmetric(TreeNode* root) {
   if (root) return helper(root->left, root->right);
   return true;
}
```

```
// 104: max-depth of binary tree (dfs)
int maxDepth(TreeNode* root) {
    return !root ? 0 : max(maxDepth(root->left), maxDepth(root->right)) + 1;
}
```

```
// 111: soln-1: bfs
int minDepth(TreeNode* root) {
    queue<TreeNode*> q;
    if (root) q.push(root);

    int level = 0;
    while (!q.empty()) {
```

```
        ++level;
        for (int i = q.size(); i > 0; --i) {
            TreeNode* n = q.front(); q.pop_front();
            if (n->left == NULL && n->right == NULL) return level;
            if (n->left) q.push(n->left);
            if (n->right) q.push(n->right);
        }
    }
    return level;
}

// 111: soln-2: dfs
int minDepth(TreeNode* root) {
    if (root == NULL) return 0;

    if (root->left == NULL) return minDepth(root->right) + 1;
    if (root->right == NULL) return minDepth(root->left) + 1;
    return min(minDepth(root->left), minDepth(root->right)) + 1;
}
```

```
// 226: invert binary tree
TreeNode* invertTree(TreeNode* root) {
    if (root) {
        invertTree(root->left), invertTree(root->right);
        swap(root->left, root->right);    }
    return root;
}

TreeNode* invertTree(TreeNode* root) {
    stack<TreeNode*> s;
    s.push(root);    // no need to check root == NULL
    while (!s.empty()) {
        TreeNode* n = s.top(); s.pop();
        if (n) {
            swap(n->left, n->right);
            s.push(n->left), s.push(n->right);
        }
    }
    return root;
}
```

```
// 951 - soln-1: dfs
bool flipEquiv(TreeNode* root1, TreeNode* root2) {
    if (!root1 && !root2) return true;
    if (root1 && root2 && root1->val == root2->val) {
        if (flipEquiv(root1->left, root2->left) && flipEquiv(root1->right, root2->right)) return true;
        if (flipEquiv(root1->left, root2->right) && flipEquiv(root1->right, root2->left)) return true;
    }
    return false;
}
```

Ref:

102/103/958. Binary tree level order/Zig-zag level order/Check completeness

```
// 102 - level order
// soln-1: bfs
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> ans;

    queue<TreeNode*> q;
    if (root) q.push(root);

    while (!q.empty()) {
```

```
        ans.push_back(vector<int>());

        for (int i = q.size(); i > 0; --i) {
            TreeNode* n = q.front();    q.pop_front();
            ans.back().push_back(n->val);

            if (n->left) q.push(n->left);
            if (n->right) q.push(n->right);
        }
    }

    return ans;
}
```

```
// 102 - level order
// soln-1: dfs
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> ans;
    helper(root, 0, ans);
    return ans;
}

void helper(TreeNode* root, int depth, vector<vector<int>>& ans) {
    if (root == NULL) return;
    if (ans.size() == depth) ans.push_back(vector<int>());

    ans[depth].push_back(root->val);
    helper(root->left, depth + 1, ans);
    helper(root->right, depth + 1, ans);
}
```

```
// 103 - zig-zag level order
// soln-1: dfs
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> ans;
    helper(root, 0, ans);
    return ans;
}

void helper(TreeNode* root, int depth, vector<vector<int>>& ans) {
    if (root == NULL) return;
    if (ans.size() == depth) ans.push_back(vector<int>());

    (depth % 2 == 0) ? ans[depth].push_back(root->val) : ans[depth].insert(ans[depth].begin(), root->val);
    helper(root->left, depth + 1, ans);
    helper(root->right, depth + 1, ans);
}
```

```
// 1104 - Path In Zigzag Labelled Binary Tree
vector<int> pathInZigZagTree(int label) {
    int h = 0;
    while (1 << h <= label) ++h;

    vector<int> ans(h, 1);
    for (--h; h > 0; --h) {
        ans[h] = label;
        if (h % 2 == 0) {    // ->
            auto ustart = 1 << (h - 1), uend = (1 << h) - 1;
            label = (ustart + uend) - label / 2;
        } else {             // <-
            auto start = 1 << h, end = (1 << (h + 1)) - 1;
            label = (end - (label - start)) / 2;
        }
    }
    return ans;
}
```

```
// 958 - check completeness of binary tree
bool isCompleteTree(TreeNode* root) {
    queue<TreeNode*> q;
    q.push(root);
    bool complete = true;
```

```
    while (!q.empty()) {
        bool nosiblings = !complete;    // will have no siblings if parent is not completed.
        for (int i = q.size(); i > 0; --i) {
            auto n = q.front(); q.pop();
            if (nosiblings) {
                if (n->left || n->right)        return false;
            } else {
                if (n->left) q.push(n->left);   else nosiblings = true;

                if (nosiblings && n->right)     return false;
                if (n->right) q.push(n->right); else nosiblings = true;
            }
        }
        complete = !nosiblings;
    }
    return true;
}
```

Ref:

94/144/145/722. Binary tree traversal/remove comment ... State machine

#105/6 - construct binary tree from pre/post and inorder
#297/449 - serialization/deserialization of a binary (search) tree
#331 - verify preorder serialization of a binary tree

```
// 94/144/145 - binary tree traversal
// soln-1: state machine (3-states)
void helper(TreeNode* root, vector<int>& ans) {
    enum { ON_DISCOVER, ON_LEFT, ON_RIGHT };
    stack<pair<TreeNode*, int> > s;
    if (root) s.push(make_pair(root, ON_DISCOVER));
    while (!s.empty()) {
        pair<TreeNode*, int> &node = s.top();
        switch(node->second) {
        case ON_DISCOVER:
            node.second = ON_LEFT;
            if (node.first->left) s.push(make_pair(node.first->left, ON_DISCOVER));
            // process result if pre-order traversal
            break;
        case ON_LEFT:
            node.second = ON_RIGHT;
            if (node.first->right) s.push(make_pair(node.first->right, ON_DISCOVER));
            // process result if in-order traversal
            break;
        case ON_RIGHT:
            s.pop();
            // process result if post-order traversal
            break;
        }
    }
}
```

```
// 722 - Remove Comments (line comment and block comment)
// soln-1: state machine
// / - comment start
//     / - comment line
//     * - comment block
//         * - comment block pre end
//             / - comment block end
vector<string> removeComments(vector<string>& source) {
    enum {INIT, COMMENT_START, COMMENT_LINE, COMMENT_BLOCK, COMMENT_BLOCK_PRE_END};
    vector<string> ans;
```

```
    string buf;
    for (int i = 0, st = INIT; i < source.size(); ++i) {
        for (int j = 0; j < source[i].length(); ++j) {
            switch (st) {
                case INIT:
                    if (source[i][j] == '/' && j < source[i].length() &&
                       (source[i][j + 1] == '/' || source[i][j + 1] == '*')) st = COMMENT_START;
                    else buf.push_back(source[i][j]);
                    break;
                case COMMENT_START:                 // /
                    if (source[i][j] == '/')        st = COMMENT_LINE;
                    else if (source[i][j] == '*')   st = COMMENT_BLOCK;
                    break;
                case COMMENT_LINE:                  // //
                    break;
                case COMMENT_BLOCK:                 // /*
                    if (source[i][j] == '*')        st = COMMENT_BLOCK_PRE_END;
                    break;
                case COMMENT_BLOCK_PRE_END:         // /*...*
                    if (source[i][j] == '/')        st = INIT;
                    else if (source[i][j] == '*')   st = COMMENT_BLOCK_PRE_END;
                    else                            st = COMMENT_BLOCK;
                    break;
            }
        }
        if (COMMENT_LINE == st)                     st = INIT;
        else if (COMMENT_BLOCK_PRE_END == st)       st = COMMENT_BLOCK;
        if (INIT == st && !buf.empty()) ans.push_back(buf), buf.clear();
    }
    return ans;
}
```

Ref: #331

105/106/107/385/386/889/1008. Binary tree construct ... stack

```
// 105 - preorder + inorder => binary tree
// T(n) = T(n-1) + n => O(n^2) time, use hashmap would help
TreeNode* helper(int preorder[], int inorder[], int n) {
    if (n < 1)     return NULL;

    TreeNode* root = new TreeNode(preorder[0]);
    int i = 0;
    for (NULL; i < n; ++i) {
        if (inorder[i] == preorder[0]) break;
    }
    root->left = helper(&preorder[1], inorder, i);
    root->right = helper(&preorder[i + 1], &inorder[i + 1], n - i - 1);
    return root;
}
```

```
// 105 - soln-2: stack (preorder + inorder => binary tree)
// build tree based on preorder. inorder must be correct
// if parent is not seen from in-order, meaning we are still on the left side.
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    stack<TreeNode*> s;
    s.push(new TreeNode(preorder[0]));

    auto root = s.top(), parent = root;
    bool see_parent = false;
    for (int i = 1, j = 0; i < preorder.size(); ++i) {
        auto node = new TreeNode(preorder[i]);
        while (!s.empty() && s.top()->val == inorder[j]) ++j, parent = s.top(), see_parent = true, s.pop();

        see_parent ? parent->right = node : parent->left = node;
        s.push(node), parent = node, see_parent = false;
```

```cpp
    }
    return root;
}
```

```cpp
// 889 - postorder + preorder => binary tree (tree is not unique given pre + post)
// T(n) = T(n-1) + n => O(n^2) time
TreeNode* helper(int pre[], int post[], int len) {
    if (len < 1) return nullptr;
    auto node = new TreeNode(pre[0]);
    if (len > 1) {
        int childLen = 0;
        while (post[childLen] != pre[1]) ++childLen;
        ++childLen;

        if (childLen + 1 == len) {
            node->right = helper(&pre[1], post, len - 1);
        } else {
            node->left = helper(&pre[1], post, childLen);
            node->right = helper(&pre[1 + childLen], &post[childLen], len - 1 - childLen);
        }
    }
    return node;
}
```

```cpp
// 889 - soln-2: stack (tree is not unique given pre + post)
// pick current value from pre-order and greedy put it as left child,
// until we see top of stack is current of post-order, which means the left subtree is done.
TreeNode* constructFromPrePost(vector<int>& pre, vector<int>& post) {
    vector<TreeNode*> s;
    s.push_back(new TreeNode(pre[0]));
    for (int i = 1, j = 0; i < pre.size(); ++i) {
        auto n = new TreeNode(pre[i]);
        while (s.back()->val == post[j]) ++j, s.pop_back();

        s.back()->left ? s.back()->right = n : s.back()->left = n;
        s.push_back(n);
    }
    return s[0];
}
```

```cpp
// 1008 - construct BST from pre-order
// soln-1: stack
TreeNode* bstFromPreorder(vector<int>& preorder) {
    if (preorder.empty()) return nullptr;

    stack<TreeNode*> stk;
    stk.push(new TreeNode(preorder[0]));

    auto root = stk.top(), parent = root;
    for (int i = 1, seen_parent = 0; i < preorder.size(); ++i) {
        auto node = new TreeNode(preorder[i]);
        while (!stk.empty() && stk.top()->val < node->val) parent = stk.top(), stk.pop(), seen_parent = 1;

        seen_parent ? parent->right = node : parent->left = node;
        stk.push(node), parent = node, seen_parent = 0;
    }
    return root;
}
```

```cpp
// 107 - binary tree bottom-up level order traversal
// soln-2: level order traversal then reverse result vector
vector<vector<int>> levelOrderBottom(TreeNode* root) {
    vector<vector<int>> ans;
    levelOrder(root, 0, ans);
    return vector<vector<int>>(ans.rbegin(), ans.rend());
}
```

```
// 385 - Mini Parser (string to obj: [[1,1],2,[1,1]])
// soln-1: recursion
NestedInteger deserialize(string s) {
    if ('[' != s[0]) return NestedInteger(stoi(s));

    NestedInteger ans;
    for (int cnt = 0, i = 1, j = i; j < s.length(); ++j) {
        if ('[' == s[j]) cnt++;
        else if (']' == s[j]) cnt--;

        if (',' == s[j] && 0 == cnt && i != j) ans.add(deserialize(s.substr(i, j - i))), i = j + 1;
        else if (cnt < 0 && i != j) ans.add(deserialize(s.substr(i, j - i)));    // last ']' of  expression
    }
    return ans;
}

// 385 - Mini Parser
// soln-2: stack
// [ - add a new current instance
// ] - current is ending, 1) add an integer if have; 2) popout current and add it to its parent
// , - add an integer (if have) to current
NestedInteger deserialize(string s) {
    if (s[0] != '[') return NestedInteger(stoi(s));

    vector<NestedInteger> stk;
    for (int l = 0, i = l; i < s.length(); ++i) {
        if ('[' == s[i]) {          // make an instance as current
            stk.push_back(NestedInteger()), l = i + 1;
        } else if (']' == s[i]) {   // current is ending, add an integer to current
            if (i - l > 0) stk.back().add(NestedInteger(stoi(s.substr(l, i - l))));
            if (stk.size() > 1) stk[stk.size() - 2].add(stk.back()), stk.pop_back();  // pop out current
            l = i + 1;
        } else if (',' == s[i]) {
            if (i - l > 0) stk.back().add(NestedInteger(stoi(s.substr(l, i - l))));   // add an integer to
current
            l = i + 1;
        }
    }
    return stk.front();
}
```

```
// 386 - Lexicographical Numbers (13 => 1, 10, 11, 12, 13, 2, 3, ...)
// soln-1: dfs (view the result as pre-order sequence)
//                                root
//                     1                            2               ...
//     10       11       12    ...   19       20   21 22 ... 29
//  100-109  110-119  120-129     190-199
void dfs(int cur, int n, vector<int>& ans) {
    if (cur > n) return;
    ans.push_back(cur);                     // root
    for (int i = 0; i < 10; i++) {          // then 10 children (pre-order)
        dfs(cur * 10 + i, n, ans);
    }
}
vector<int> lexicalOrder(int n) {
    vector<int> ans;
    for (int i = 1; i < 10; ++i) dfs(i, n, ans);
    return ans;
}
```

```
// 1028. Recover a Tree From Preorder Traversal
// soln-1: stack
TreeNode* recoverFromPreorder(string S) {
    auto root = new TreeNode(stoi(S));
    vector<pair<TreeNode*, int>> stk;
    stk.push_back({root, 0});
```

```
    for (auto i = to_string(root->val).length(); i < S.length(); ++i) {
        int d = 0;
        while ('-' == S[i]) ++d, ++i;

        while (stk.back().second + 1 != d) stk.pop_back();
        auto n = new TreeNode(stoi(S.substr(i)));
        if (nullptr == stk.back().first->left) stk.back().first->left = n;
        else stk.back().first->right = n;

        stk.push_back({n, d});

        i += to_string(n->val).length() - 1;
    }
    return root;
}
```

Ref: #654

108/109/426/538. Convert array/list -> bst

```
// 108 - sort array to bst
TreeNode* helper(int nums[], int n) {
    if (n < 1) return NULL;

    int left = (n - 1) / 2;
    TreeNode* root = new TreeNode(nums[left]);
    root->left = helper(nums, left), root->right = helper(&nums[left + 1], n - left - 1);

    return root;
}
```

```
// 109 - sorted list to bst
TreeNode* sortedListToBST(ListNode* head) {
    return helper(head, length(head));
}

TreeNode* helper(ListNode* &head, int n) {
    if (n == 0) return NULL;

    TreeNode* root = new TreeNode(0);   // val will be updated later
    root->left = helper(head, n/2);
    root->val = head->val, head = head->next;
    root->right = helper(head, n - n/2 - 1);

    return root;
}
```

```
// 426 - bst to doubly linked list (left -> predecessor, right -> successor)
TreeNode* connect(TreeNode* l1, TreeNode* l2) {
   if (nullptr == l1) return l2;
   if (nullptr == l2) return l1;
   TreeNode* l1tail = l1->left, *l2tail = l2->left;
   l1tail->right = l2, l2->left = l1tail;
   l1->left = l2tail, l2tail->right = l1;
   return l1;
}
TreeNode* treeToDoublyList(TreeNode* root) {
   if (nullptr == root) return nullptr;
   auto l = treeToDoublyList(root->left), r = treeToDoublyList(root->right);
   root->left = root->right = root;
   return connect(connect(l, root), r);
}
```

```
// 538 - convert bst to greater tree such that
// every key of original bst is changed to original key + sum of all keys greater than the original key in bst.
// soln-1: traverse from right to left and accumulate bigger one
void helper(TreeNode* r, TreeNode* &successor) {
   if (!r) return;
```

```
    helper(r->right, successor);
    if (successor) r->val += successor->val;
    successor = r;        // update current successor accordingly
    helper(r->left, successor);
}
```

Ref:

112/113/129/988/257/437/666. Path sum/II/Sum root to leaf/Binary tree path/III/IV

112 - determine if there is a path from root to leaf sums to given number.
113 - same as 112 but requires path
129/988 - sum root to leaf/smallest string starting from leaf
257 - binary tree paths
437 - path from up to down, can be starting from any node
666 - tree represented by triplets
If the depth of a tree is smaller than 5, then this tree can be represented by a list of three-digits integers.
For each integer in this list:

1. The hundreds digit represents the depth D of this node, 1 <= D <= 4.
2. The tens digit represents the position P of this node in the level it belongs to, 1 <= P <= 8. The position is the same as that in a full binary tree.
3. The units digit represents the value V of this node, 0 <= V <= 9.

Given a list of ascending three-digits integers representing a binary with the depth smaller than 5. You need to return the sum of all paths from the root towards the leaves.

Example 1:
Input: [113, 215, 221], Output: 12
Explanation:
The tree that the list represents is:

```
  3
 / \
5   1
```

The path sum is (3 + 5) + (3 + 1) = 12.

```
// 112: soln-1: dfs
bool hasPathSum(TreeNode* root, int sum) {
    if (root == NULL) return false;

    if (root->left == NULL && root->right == NULL) return root->val == sum;
    return hasPathSum(root->left, sum - root->val) || hasPathSum(root->right, sum - root->val);
}
```

```
// 113: soln-1: dfs
vector<vector<int>> pathSum(TreeNode* root, int sum) {
    vector<vector<int>> ans;
    vector<int> path;
    helper(root, sum, ans);
    return ans;
}

void helper(TreeNode* root, int sum, vector<int>& path, vector<vector<int>>& ans) {
    if (root == NULL) return;

    path.push_back(root->val);
    if (root->left == NULL && root->right == NULL && root->val == sum) {
        ans.push_back(path);
    } else {
        helper(root->left, sum - root->val, path, ans);
        helper(root->right, sum - root->val, path, ans);
    }
    path.pop_back();
}
```

```
// 988 - smallest string starting from leaf
// soln-1: dfs (bottom up)
string smallestFromLeaf(TreeNode* root) {
```

```cpp
    string ans;
    h(root, "", ans);
    return ans;
}
void h(TreeNode* node, string path, string& ans) {
    if (nullptr == node) return;

    path.insert(path.begin(), (node->val + 'a'));
    if (!node->left && !node->right) {
        if (ans.empty() || path < ans) ans = path;
    }
    h(node->left, path, ans);
    h(node->right, path, ans);
}
```

```cpp
// 129 - sum root to leaf numbers
int sumNumbers(TreeNode* root) {
    int ans = 0;
    h(root, 0, ans);
    return ans;
}
void h(TreeNode* node, int s, int& ans) {
    if (nullptr == node) return;

    s = s * 10 + node->val;
    if (!node->left && !node->right) {
        ans += s;
        return;
    }
    h(node->left, s, ans), h(node->right, s, ans);
}
```

```cpp
// 257: soln-1 - dfs
void helper(TreeNode* node, string path, vector<string>& ans) {
   if (!node) return;

   if (!path.empty()) path += "->";
   if (!node->left && !node->right) {
      ans.push_back(path + to_string(node->val));
   } else {
      helper(node->left, path + to_string(node->val), ans);
      helper(node->right, path + to_string(node->val), ans);
   }
}
vector<string> binaryTreePaths(TreeNode* root) {
   vector<string> ans;
   helper(root, "", ans);
   return ans;
}
```

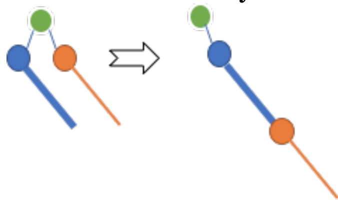
```cpp
// 437: soln-1: dfs to get #. of paths starting from specific node
int fromOneNode(TreeNode* r, int sum, int target) {
   if (!r) return 0;

   int ans = 0;
   if (target == sum + r->val) ++ans;
   ans += fromOneNode(r->left, sum + r->val, target);
   ans += fromOneNode(r->right, sum + r->val, target);
   return ans;
}
int pathSum(TreeNode* root, int sum) {
   if (!root) return 0;
   return fromOneNode(root, 0, sum) + pathSum(root->left, sum) + pathSum(root->right, sum);
}
```

```
// 666 - tree is represented in triplets(height, offset, value)
// soln-1: hashmap to keep node position with its value
// for each leaf, add up to its parent till root.
//       5       (115)          => 11->5
//     /   \
//    2     3    (212, 223)      => 21->2, 22->3
//     \   / \
//      1 6   2  (321, 336, 342) => 32->1, 33->6, 34->2
// given 32x, how to get its parent? 3 - height, 2 - position (starting from 1)
//            parent (height - 1) -> 2         (2 + 1) / 2 <- parent position
// so we could map height-offset (can determine a node) to value
void helper(unordered_map<int, int>& m, int pos, int& ans) {
   if (m.find(pos) == m.end()) return;

   ans += m[pos];
   int parent = (pos / 10 - 1) * 10 + (pos % 10 + 1) / 2;
   helper(m, parent, ans);
}
int pathSum(vector<int>& nums) {
   unordered_map<int, int> m;
   for (int i : nums) m[i / 10] = i % 10;   // height-offset -> node's value

   int ans = 0;
   for (int i = nums.size() - 1, height = nums.back() / 100; nums[i] > height * 100; --i) {
       helper(m, nums[i] / 10, ans);       // from this leaf node, sum up with its parent
   }
   return ans;
}
```

Ref:

114/430/872. Flatten binary tree to linked list / Leaf-similar trees

Assume we already have left and right subtree flattened, the last step is to do following:



this recursive soln is good for flatten tree in other way.

872- Two binary trees are considered *leaf-similar* if their leaf value sequence is the same.

```
// 114: flatten binary tree to linked list
void flatten(TreeNode* root) {
    if (root == NULL) return;

    flatten(root->left), flatten(root->right);
    if (root->left) {
        TreeNode* last = root->left;
        while(last->right) last = last->right;

        last->right = root->right, root->right = root->left, root->left = NULL;
    }
}

// 114: soln-2: iterative
void flatten(TreeNode* root) {
    stack<TreeNode*> s;
    if (root) s.push(root);

    while (!s.empty()) {
        TreeNode* n = s.top();  s.pop();
```

```
        if (n->right) s.push(n->right);
        if (n->left) s.push(n->left);

        n->right = s.empty() ? NULL : s.top();
        n->left = NULL;
    }
}
```

```
// 430: flatten multi level double linked list
// soln-1: dfs
Node* flatten(Node* head, Node** nt = nullptr) {
   Node dummy;
   Node* tail = &dummy;
   for (Node* cur = head, *nxt; cur; cur = nxt) {
       nxt = cur->next;        // save next first as we will update cur directly
       if (cur->child) {
           Node* nt;
           Node* nh = flatten(cur->child, &nt);
           cur->next = nh, nh->prev = cur, cur->child = nullptr;
           tail->next = cur, cur->prev = tail, tail = nt;
       } else {
           tail->next = cur, cur->prev = tail, tail = cur;
       }
   }
   if (nt) *nt = tail;
   if (dummy.next) dummy.next->prev = nullptr;
   return dummy.next;
}
```

```
// 872 -
// soln-1: dfs
// brute-force would be comparing two leaf list node
// depends on tree structure, we can getNextLeaf helper then compare
bool leafSimilar(TreeNode* root1, TreeNode* root2) {
   stack<TreeNode*> stk1, stk2;
   stk1.push(root1), stk2.push(root2);

   while (!stk1.empty() && !stk2.empty()) {
       if (dfs(stk1) != dfs(stk2)) return false;
   }
   return stk1.empty() && stk2.empty();
}

int dfs(stack<TreeNode*>& stk) {
   while (!stk.empty()) {
       auto n = stk.top(); stk.pop();
       if (n->right) stk.push(n->right);
       if (n->left) stk.push(n->left);
       if (!n->left && !n->right) return n->val;
   }
   return INT_MIN; // never reach here
}
```

Ref:

115/300/673/674/944/955/960/1035. Subsequences ... LCS/LIS (review)

#334 - increasing triplet subsequence
#354 - russian doll envelopes (similar DP idea)
#960 - LIS for an array of string

```
// 115 - distinct subsequences (count # of distinct subsequences from S to T)
// soln-1: recursion
//        if S[i] != T[j], then f(i, j) = f(i - 1, j), S[i] has no contribution
```

```
//         else f(i, j) = f(i - 1, j)        <-- NOT consume S[i] even it can contribute
//                          +
//                          f(i - 1, j - 1)    <-- consume s[i] and t[j]
int numDistinct(string S, string T) {
   if (T.empty()) return 1;        // any string to empty string, only 1 possible (none of them included)
   if (S.empty()) return 0;        // no way to transform to T if empty (except T is empty, then return 1.)

   if (T.back() != S.back()) return numDistinct(S.substr(0, S.length() - 1), T);
   return numDistinct(S.substr(0, S.length() - 1), T) +
          numDistinct(S.substr(0, S.length() - 1), T.substr(0, T.length() - 1));
}

// 115 - distinct subsequences (count # of distinct subsequences from S to T)
// soln-2: dynamic programming
// let dp(i, j) be the result for transforming from S to T
//     = dp(i - 1, j),                     if S[i] != T[j]     <-- keep prev result, as s[i] has no contribution
//     = dp(i - 1, j) + dp(i - 1, j - 1)                       <-- not consume s[i] and consume it
// dp(i, 0) = 1, from S to empty string, only 1 way
// dp(0, j) = 0, from S, which is empty string, no way to transform to T.
// dp[0][0] = 1, from empty to empty, only 1 way
int numDistinct(string S, string T) {
   int m = S.length(), n = T.length();
   vector<vector<int>> dp(m + 1, vector<int>(n + 1));
   dp[0][0] = 1;                       // empty to empty, only 1 way

   for (int i = 1; i <= m; ++i) {
       dp[i][0] = 1;
       for (int j = 1; j <= n; ++j) {
           if (S[i - 1] == T[j - 1]) dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1];
           else dp[i][j] = dp[i - 1][j];
       }
   }
   return dp[m][n];
}
```

```
// 300 - longest increasing subsequence
// soln-1: dynamic programming in O(n^2)
//         f(n) = 1 + max{ f(i) if 0 < i < n and a[i] < a[n] }
int lengthOfLIS(vector<int>& nums) {
   int ans = 1;
   vector<int> dp(nums.size(), 1);
   for (int i = 1; i < nums.size(); ++i) {
       for (int j = 0; j < i; ++j) {
           if (nums[j] < nums[i]) dp[i] = max(dp[i], 1 + dp[j]);
       }
       ans = max(ans, dp[i]);
   }

   return ans;
}

// 300 - longest increasing subsequence
// soln-2: maintain all increasing lists and binary search in O(nlgn) time and O(n) space
int lengthOfLIS(vector<int>& nums) {
   vector<int> tails(nums.size());         // at most n increasing lists
   int tails_n = 0;

   tails[tails_n++] = nums[0];
   for (int i = 1; i < nums.size(); ++i) {
       if (nums[i] < tails[0])                tails[0] = nums[i];
       else if (nums[i] > tails[tails_n - 1]) tails[tails_n++] = nums[i];  // extend list
       else {
           // replace the first tail which greater than nums[i]
```

```
            int low = 0, hi = tails_n - 1;
            while (low + 1 < hi) {
                int m = low + (hi - low) / 2;
                if (tails[m] == nums[i]) { hi = m; break; }
                else (tails[m] > nums[i]) ? hi = m : low = m;
            }

            if (tails[low] >= nums[i]) tails[low] = nums[i]; // for strictly increasing, equals will be skipped
            else if (tails[hi] > nums[i]) tails[hi] = nums[i];
        }
    }
    return tails_n;
}
```

```
// 1027 - Longest Arithmetic Sequence
// soln-1: dynamic programming
int longestArithSeqLength(vector<int>& A) {
    int ans = 0;
    vector<unordered_map<int, int>> dp(A.size());
    for (int i = 1; i < A.size(); ++i) {
        for (int j = i - 1; j >= 0; --j) {
            auto diff = A[i] - A[j];
            dp[i][diff] = max(dp[i][diff], dp[j][diff] + 1);
            ans = max(ans, dp[i][diff]);
        }
    }
    return ans + 1;
}
```

```
// 673 - number of longest increasing subsequence
// soln-1: dynamic programming in O(n^2) time and O(n) space
int findNumberOfLIS(vector<int>& nums) {
  vector<pair<int, int>> dp(nums.size(), {1, 1});         // <length, # of LIS>
  pair<int, int> ans;

  for (int i = 0; i < nums.size(); ++i) {
      for (int j = 0; j < i; ++j) {
          if (nums[j] < nums[i]) {
              if (dp[i].first == dp[j].first + 1) dp[i].second += dp[j].second;
              else if (dp[i].first < dp[j].first + 1) dp[i] = {dp[j].first + 1, dp[j].second};
          }
      }
      if (dp[i].first >= ans.first) {
          ans.second = (dp[i].first == ans.first) ? (ans.second + dp[i].second) : dp[i].second;
          ans.first = dp[i].first;
      }
  }
  return ans.second;
}
```

```
// 674 - longest continuous increasing subsequence
// soln-1: greedy (linear scan for longest continuous increasing subsequence)
int findLengthOfLCIS(vector<int>& nums) {
   if (nums.empty()) return 0;

   int ans = 1, sofar = 1;
   for (int i = 1; i < nums.size(); ++i) {
       sofar = (nums[i] > nums[i - 1]) ? sofar + 1 : 1;
       ans = max(ans, sofar);
   }
   return ans;
}
```

```
// 944 - delete columns to make each column sorted (trivial)
// 955 - delete columns to make array sorted (each column might not be sorted)
```

```cpp
// soln-1: greedy
// if A[i][col] > A[i-1][col], then no need to compare A[i][col+1] with A[i-1][col+1].
// use bool sorted[n] to mark if A[i][col] > A[i-1][col].
int minDeletionSize(vector<string>& A) {
    int ans = 0, rows = A.size(), cols = A[0].length();
    vector<bool> pre(rows);
    for (int j = 0, i = 1; j < cols; ++j) {
        vector<bool> cur(rows);      // for cur col, if it is sorted.
        for (i = 1; i < rows; ++i) {
            if (pre[i]) continue;    // line already sorted, no need to compare with prev line.

            if (A[i][j] > A[i-1][j]) {
                cur[i] = true;
            } else if (A[i][j] < A[i-1][j]) {
                ++ans;
                break;
            }
        }
        if (i >= rows) {             // if cur col is sorted, merge state with pre col.
            for (int k = 0; k < rows; ++k) pre[k] = pre[k] || cur[k];
        }
    }
    return ans;
}
```

```cpp
// 960 - delete columns to make each row sorted (whole array might not be sorted)
// soln-1: dynamic programming
// 1. for LIS question, we can do it in O(n^2)
// 2. a variant of LIS, instead of considering dp[cur] = dp[i] + 1 if cur >= Ai for one string,
//    this question needs to consider for all whole column satisfying above condition.
// 3. with LIS, we know the number of columns which needs to be removed.
int minDeletionSize(vector<string>& A) {
    vector<int> dp(A[0].length(), 1);
    for (int j = 1; j < A[0].length(); ++j) {
        for (int jj = 0; jj < j; ++jj) {
            bool ordered = true;    // ordered for every row
            for (int i = 0; ordered && i < A.size(); ++i) {
                if (A[i][jj] > A[i][j]) ordered = false;
            }
            if (ordered) dp[j] = max(dp[j], dp[jj] + 1);
        }
    }
    return A[0].length() - *max_element(dp.begin(), dp.end());
}
```

```cpp
// 1035 - Uncrossed Lines
// soln-1: dynamic programming
// essentially this is same as LCS (longest common subsequence)
// f(i, j) = f(i - 1, j - 1) + 1,         if s[i] == t[j]
//         = max{f(i-1, j), f(i, j - 1)}, if s[i] != t[j]
int maxUncrossedLines(vector<int>& A, vector<int>& B) {
    vector<vector<int>> dp(A.size() + 1, vector<int>(B.size() + 1));
    for (int i = 1; i <= A.size(); ++i) {
        for (int j = 1; j <= B.size(); ++j) {
            if (A[i - 1] == B[j - 1]) dp[i][j] = 1 + dp[i - 1][j - 1];
            else dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
    return dp.back().back();
}
```

Ref: #5

116/117. Populating next right pointers in each node

```
// 116/117 - populate each next pointer to its next right node
// soln-1: populate layer by layer
void connect(TreeLinkNode *root) {
   while (root) {
       TreeLinkNode dummy(0);
       for (auto cur = &dummy; root; root = root->next) {
           if (root->left) cur->next = root->left, cur = cur->next;
           if (root->right) cur->next = root->right, cur = cur->next;
       }
       root = dummy.next;
   }
}
```

Ref:

118/119/120. Pascal's triangle/II/Triangle

```
// 118 - pascal's triangle
// soln-1: populate layer by layer
vector<vector<int>> generate(int numRows) {
   vector<vector<int>> ans;
   if (numRows > 0) ans.push_back(vector<int>{1});
   while (--numRows > 0) {
       vector<int>& lastRow = ans.back();
       vector<int> nr{1};
       for (int j = 1; j < lastRow.size(); ++j) {
           nr.push_back(lastRow[j] + lastRow[j - 1]);
       }
       nr.push_back(1);
       ans.push_back(nr);
   }
   return ans;
}
```

```
// 118 - get kth row of the pascal's triangle
// soln-1: row by row update backwards
vector<int> getRow(int rowIndex) {
   vector<int> ans(rowIndex + 1, 0);
   ans[0] = 1;
   for (int i = 1; i < rowIndex + 1; ++i) {
       for (int j = i; j >= 1; --j) {
           ans[j] += ans[j - 1];
       }
   }
   return ans;
}
```

```
// 120 - min path sum of triangle (each step you may move to adjacent numbers on the row below)
// soln-1: bottom-up dynamic programming
// dp(row, j) = min{dp(row + 1, j), dp(row +, j + 1} + triangle[row][j]
int minimumTotal(vector<vector<int>>& triangle) {
   if (triangle.empty()) return 0;

   vector<int> dp = triangle.back();
   for (int i = triangle.size() - 2; i >= 0; --i) {
       for (int j = 0; j < triangle[i].size(); ++j) {
           dp[j] = triangle[i][j] + min(dp[j], dp[j + 1]);
       }
   }
   return dp[0];
```

```
}
```

Ref: #119 #120

121/1014/122/123/188/309/714. Best time to buy and sell stock (1-Tx, k-Tx, cooldown, with fees) (review)

#416 - partition to equal subset (knapsack variations)

```
// 121 - best time to buy/sell stock (allow at most 1 Tx)
// soln-1: greedy keep lowest price so far
int maxProfit(vector<int>& prices) {
   if (prices.empty()) return 0;

   int ans = 0, lowest = prices[0];
   for (int i = 1; i < prices.size(); ++i) {
       lowest = min(lowest, prices[i]);
       ans = max(ans, prices[i] - lowest);
   }
   return ans;
}

// 121 - best time to buy/sell stock (allow at most 1 Tx)
// soln-2: buy/sell logic
// when selling, sell_portfolio = buy_portfolio + price[i]
// when buying, buy_portfolio = sell_portfolio - price[i]
// since only 1-TX is allowed, sell_portfolio is always 0 in this case.
int maxProfit(vector<int>& prices) {
   if (prices.empty()) return 0;

   int sell_profit = 0, buy_profit = 0 - prices[0];
   for (int i = 1; i < prices.size(); ++i) {
       sell_profit = max(sell_profit, buy_profit + prices[i]);
       buy_profit = max(buy_profit, 0 - prices[i]);    // 1-TX only, no previous profit so far
   }

   return sell_profit;     // max profit must be after selling.
}
```

```
// 1014. Best Sightseeing Pair (max of A[i] + A[j] - (j - i) )
// soln-1: greedy (update cur gain if not worth it)
// this question is similar to best time to buy/sell stock with 1-tx.
int maxScoreSightseeingPair(vector<int>& A) {
    int ans = 0, cur = A[0];
    for (int start = 0, i = 1; i < A.size(); ++i) {
        ans = max(ans, cur + A[i] - i + start);
        if (cur - i + start < A[i]) cur = A[i], start = i;  // discard cur spot if not worth it
    }
    return ans;
}
```

```
// 122 - best time to buy/sell stock (allow infinite Tx even on same day)
// soln-1: greedy grab profit if have
int maxProfit(vector<int>& prices) {
    int ans = 0;
    for (int i = 0; i < (int)prices.size() - 1; ++i) {
        int profit = prices[i + 1] - prices[i];
        if (profit > 0) ans += profit;      // grab the profit if exist
    }
    return ans;
}

// 122 - best time to buy/sell stock (allow infinite Tx even on same day)
// soln-2: buy/sell logic
```

```cpp
// when selling, sell_portfolio = buy_portfolio + price[i]
// when buying, buy_portfolio = sell_portfolio - price[i]
int maxProfit(vector<int>& prices) {
   if (prices.empty()) return 0;

   int sell_profit = 0, buy_profit = 0 - prices[0];
   for (int i = 1; i < prices.size(); ++i) {
       sell_profit = max(sell_profit, buy_profit + prices[i]);
       buy_profit = max(buy_profit, sell_profit - prices[i]);    // profit based on previous selling
   }

   return sell_profit;     // max profit must be after selling.
}
```

```cpp
// 123 - best time to buy/sell stock (allow at most 2 Tx)
// soln-1: 3-loop naive dynamic programming in (k * n * n) time (TLE)
// let f(i, k) be the max profit with i stocks and k tx, we could choose either with Tx or w/o Tx
// profit if without Tx: f(i-1, k)
// profit if with Tx: price[i] - price[j] + f(j, k - 1)  ← choose some day in between [0, i-1]
// dynamic programming: f(i, k) = max {f(i - 1, k), price[i] - price[j] + f(j, k - 1) | j = 0 ~ i-1}
//                              = max {f(i - 1, k), price[i] + max{f(j, k - 1) - price[j]}
int maxProfit(vector<int> &prices) {
    if (prices.empty()) return 0;

    int n = prices.size(), trans = 2;
    vector<vector<int>> dp(n, vector<int>(trans + 1, 0));

    for (int k = 1; k <= trans; ++k) {
        for (int i = 1; i < n; ++i) {
            int no_trans = dp[i - 1][k];

            int with_trans = dp[0][k - 1] + prices[i] - prices[0];  // next soln will make it in O(1) time!
            for (int j = 0; j < i; ++j) {
                with_trans = max(with_trans, dp[j][k - 1] + prices[i] - prices[j]);
            }

            dp[i][k] = max(no_trans, with_trans);
        }
    }

    return dp[n - 1][trans];
}

// 123 - best time to buy/sell stock (allow at most 2 Tx)
// dynamic programming: f(k, i) = max {f(k, i - 1), price[i] - price[j] + f(k - 1, j) | j = 0 ~ i-1}
//                              = max {f(k, i - 1), price[i] + max{f(k - 1, j) - price[j]}
//                              = max {f(k, i - 1), prices[i] + max {preMax, f(k - 1, i - 1) - price[i - 1]}
// time complexity: O(kn). Runtime: 20 ms
int maxProfit(vector<int>& prices) {
    if (prices.empty()) return 0;

    int n = prices.size(), trans = 2;
    vector<vector<int>> dp(trans + 1, vector<int>(n, 0));

    for (int k = 1; k <= trans; ++k) {
        int preMax = dp[k - 1][0] - prices[0];      // at first, j is from 0 to 0.

        for (int i = 1; i < n; ++i) {
            int no_trans = dp[k][i - 1];            // f(k, i - 1)
            int with_trans = prices[i] + preMax;    // prices[i] + max{preMax, f(k - 1, i - 1) - price[i - 1]}
            dp[k][i] = max(no_trans, with_trans);

            preMax = max(preMax, dp[k - 1][i] - prices[i]); // max{preMax, f(k - 1, i - 1) - price[i - 1]}
```

```
        }
    }

    return dp[trans][n - 1];
}

// 123 - best time to buy/sell stock (allow at most 2 Tx)
// soln-3: buy/sell logic in O(n) time and O(1) space
int maxProfit(vector<int>& prices) {
   if (prices.empty()) return 0;

   int sell_profit1 = 0, buy_profit1 = 0 - prices[0];
   int sell_profit2 = 0, buy_profit2 = sell_profit1 - prices[0];
   for (int i = 1; i < prices.size(); ++i) {
       buy_profit1 = max(buy_profit1, 0 - prices[i]);
       sell_profit1 = max(sell_profit1, buy_profit1 + prices[i]);
       buy_profit2 = max(buy_profit2, sell_profit1 - prices[i]);
       sell_profit2 = max(sell_profit2, buy_profit2 + prices[i]);
   }
   return sell_profit2;
}
```

```
// 188 - best time to buy/sell stock (allow at most k Tx)
// dynamic programming: f(k, i) = max {f(k, i - 1), price[i] - price[j] + f(k - 1, j) | j = 0 ~ i-1}
//                              = max {f(k, i - 1), price[i] + max{f(k - 1, j) - price[j]}
// let preMax = max{f(k - 1, j) - price[j]} | j = 0 ~ i-2
//                              = max {f(k, i - 1), prices[i] + max {preMax, f(k - 1, i - 1) - price[i - 1]}
int maxProfit(int trans, vector<int>& prices) {
    if (trans >= prices.size() / 2) return quickSolve(prices);  // have enough transaction times allowed

    vector<vector<int>> dp(2, vector<int>(prices.size(), 0));

    int cur = 0, pre = 1;
    for (int k = 0; k < trans; ++k) {
        int preMax = dp[pre][0] - prices[0];

        for (int i = 1; i < (int)prices.size(); ++i) {
            int no_trans = dp[cur][i - 1];                  // f(k, i - 1)
            int with_trans = prices[i] + preMax;            // prices[i] + max{preMax, f(k-1, i-1) - price[i-1]}
            dp[cur][i] = max(no_trans, with_trans);
            preMax = max(preMax, dp[pre][i] - prices[i]);   // max{preMax, f(k-1, i-1) - price[i-1]}
        }

        swap(cur, pre);
    }
    return dp[pre][prices.size() - 1];
}

// since we have enough tx allowed, grab every possible profit as we can.
// question translates to same as #122.
int quickSolve(vector<int>& prices) {
    int ans = 0;
    for (int i = 1; i < (int)prices.size(); ++i) {
        if (prices[i] - prices[i - 1] > 0) ans += prices[i] - prices[i - 1];
    }
    return ans;
}
```

```
// 309 - Best time to buy and sell stock with cooldown
// soln-1: buy/sell logic state machine
// 1. recall without cooldown, at day-i, buy_profit[i] = max(buy_profit[i - 1], sell_profit[i - 1] - prices[i])
// 2. as we need to cooldown, which means we cannot use sell_profit[i - 1], but sell_profit[i - 2]
// 3. sell_profit[i] = max(sell_profit[i - 1], buy_profit[i - 1] + prices[i]),
```

```
//    since we can sell next day after we bought, no cooldown required.
int maxProfit(int[] prices) {
    if(prices == null || prices.length <= 1) return 0;

    int b0 = -prices[0], b1 = b0;
    int s0 = 0, s1 = 0, s2 = 0;

    for(int i = 1; i < prices.length; i++) {
        b0 = Math.max(b1, s2 - prices[i]);
        s0 = Math.max(s1, b1 + prices[i]);
        b1 = b0, s2 = s1, s1 = s0;
    }
    return s0;
}
```

```
// 714 - best time buy/sell stock with tx fee
// soln-1: buy/sell logic state machine
// greedy is not working, ex. [1, 3, 6, 5, 10], greedy gets (6-1-2) + (10-5-2) = 6, but we could do (10-1-2) =
7.
int maxProfit(vector<int>& prices, int fee) {
   if (prices.empty()) return 0;

   // buy_profit - max profit if buy now (preparation step for selling)
   // sell_profit - max profit if selling (must be based on previous buy action)
   // initially we have nothing to sell so no profit, can only buy to try.
   int sell_profit = 0, buy_profit = sell_profit - prices[0];
   for (int i = 1; i < prices.size(); ++i) {
       // fee can be applied at either buy or sell stage.
       sell_profit = max(sell_profit, buy_profit + prices[i] - fee);
       buy_profit = max(buy_profit, sell_profit - prices[i]);
   }

   // sell_profit must be bigger as only sell can make profit.
   return sell_profit;
}
```

Ref: #53

124. Binary tree maximum path sum

Find the maximum path sum for a given binary tree. A path is defined as any sequence of nodes from some starting node to any node in the tree **along the parent-child connection**. The path does not need to go through the root.

```
int maxPathSum(TreeNode* root) {
    int ans = INT_MIN;
    helper(root, ans);
    return ans;
}

int helper(TreeNode* root, int& ans) {
    if (root == NULL) return 0;

    int left = helper(root->left, ans), right = helper(root->right, ans);
    ans = max(ans, left + right + root->val);

    int sum = root->val + max(left, right);    // either left or right subtree could contribute, NOT BOTH.
    return sum > 0 ? sum : 0;
}
```

Ref:

125/680. Validate palindrome / II

```
bool isPalindrome(string s) {
   for (int l = 0, r = s.length(); l < r; ++l, --r) {
       while (l < r && !isalnum(s[l])) ++l;
       while (l < r && !isalnum(s[r])) --r;
       if (l < r && toupper(s[l]) != toupper(s[r])) return false;
   }
   return true;
}
```

```
// soln-1: allow delete N chars
bool validPalindrome(string s) {
   return h(s, 0, s.length() - 1, 1);
}

bool h(const string& s, int l, int r, int d) {
   if (d < 0)  return false;
   if (l >= r) return true;
   if (s[l] == s[r]) return h(s, l + 1, r - 1, d);
   return h(s, l + 1, r, d - 1) || h(s, l, r - 1, d - 1);
}
```

Ref: #234

126/127/433/797/1048. Word ladder/II/Longest string chain (review)

```
typedef unordered_set<string> t_wordset;
typedef unordered_map<string, t_wordset> t_graph;

// 126 - word ladder (return all shortest paths from begin to end word)
// soln-1: bidirectional bfs in O(k^(d/2)) time where k is adjacent nodes, d is distance from s to t.
// 1. search from begin word and end word respectively and build graph at the same time
// 2. delete visited from word list
// 3. build path after collision
vector<vector<string>> findLadders(string beginWord, string endWord, vector<string> &wordList) {
    t_wordset wordSet;
    for (string& w : wordList) wordSet.insert(w);

    t_graph g; t_wordset s[2];
    s[0].insert(beginWord), wordSet.erase(beginWord);
    s[1].insert(endWord),   wordSet.erase(endWord);
    bool met = false;
    while (!met && !s[0].empty() && !s[1].empty()) {
        t_wordset neighbors;
        int i = s[0].size() <= s[1].size() ? 0 : 1;     // IMPORTANT for bidirectional search
        for (string w : s[i]) {
            if (getNeighbors(w, wordSet, neighbors, s[(i + 1) % 2], g, i == 0)) met = true;
        }
        s[i] = neighbors;
    }
    vector<vector<string>> ans;
    vector<string> path;
    if (met) buildPath(beginWord, endWord, g, path, ans);
    return ans;
}

// return true if collision has been found. directed graph will be built during the process.
bool getNeighbors(string& word, t_wordset& dict, t_wordset& ans, t_wordset& other, t_graph& g, bool fromLeft) {
    bool met = false;
    for (int i = 0; i < word.length(); ++i) {
        string w = word;
        for (char ch = 'a'; ch <= 'z'; ++ch) {
            w[i] = ch;
```

```
            if (other.find(w) != other.end()) met = true;

            // if collision or in the neighbor set, w was once in dict!
            if (dict.find(w) != dict.end() || other.find(w) != other.end() || ans.find(w) != ans.end()) {
                ans.insert(w), dict.erase(w);
                fromLeft ? g[word].insert(w) : g[w].insert(word);   // store outgoing edges
            }
        }
    }
    return met;
}
// return all paths direct to target node in directed graph
void buildPath(string from, string to, t_graph& g, vector<string>& path, vector<vector<string>>& ans) {
    path.push_back(from);

    if (from == to) ans.push_back(path);
    else for (string n : g[from]) {
        buildPath(n, to, g, path, ans);     // directed graph, no need for visited flag.
    }
    path.pop_back();
}
```

```
// 127 - word ladder (shortest length from begin to end word)
// soln-1: bidirectional bfs in O(k^{d/2}) time where k is adjacent nodes, d is distance from s to t.
// 1. search from begin word and end word respectively (brute-force replace each letter)
// 2. delete visited from word list
// 3. return when collision found
int ladderLength(string beginWord, string endWord, unordered_set<string>& wordList) {
    unordered_set<string> s[2];
    s[0].insert(beginWord),  wordList.erase(beginWord);
    s[1].insert(endWord),    wordList.erase(endWord);
    int ans = 0;
    while (!s[0].empty() && !s[1].empty()) {
        ans++;
        unordered_set<string> neighbors;
        int i = s[0].size() < s[1].size() ? 0 : 1;      // smaller size first!
        for (string word : s[i]) {
            if (getNeighbor(word, wordList, neighbors, s[(i + 1) % 2])) return ans + 1;
        }
        s[i] = neighbors;
    }
    return 0;
}
bool getNeighbor(string& word, wordset& wordList, wordset& ans, wordset& s2) {
    for (int i = 0; i < word.length(); ++i) {
        string w = word;
        for (char ch = 'a'; ch <= 'z'; ++ch) {
            w[i] = ch;
            if (s2.find(w) != s2.end()) return true;    // collide with other direction
            if (wordList.find(w) != wordList.end()) {
                ans.insert(w), wordList.erase(w);
            }
        }
    }
    return false;
}
```

```
// 433 - Minimum Genetic Mutation
// soln-1: bfs
int minMutation(string start, string end, vector<string>& bank) {
    unordered_set<string> s{bank.begin(), bank.end()}, seen;
    if (!s.count(end)) return -1;
    queue<string> q;
```

```
    q.push(start), seen.insert(start);
    int ans = 0;
    while (!q.empty()) {
        for (int i = q.size(); i > 0; --i) {
            auto cur = q.front();   q.pop();
            for (auto& x : cur) {
                auto tmp = x;
                for (auto ch : vector<char>{'A', 'C', 'G', 'T'}) {
                    x = ch;
                    if (cur == end) return ans + 1;
                    if (s.count(cur) && !seen.count(cur)) q.push(cur), seen.insert(cur);
                }
                x = tmp;
            }
        }
        ++ans;
    }
    return -1;
}
```

```
// 797 - All Paths From Source to Target
// soln-1: dfs
typedef unordered_map<int, unordered_set<int>> t_graph;
vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
    t_graph g;
    for (int i = 0; i < graph.size(); ++i) {
        for (auto& n : graph[i]) g[i].insert(n);
    }

    vector<vector<int>> ans;
    vector<int> path{0};
    h(g, path, graph.size() - 1, ans);
    return ans;
}
void h(t_graph& g, vector<int>& path, int target, vector<vector<int>>& ans) {
    int start = path.back();
    if (start == target) ans.push_back(path);
    else for (auto n : g[start]) {
            path.push_back(n);
            h(g, path, target, ans);
            path.pop_back();
        }
}
```

```
// 1048 - Longest String Chain
// soln-1: bfs/dfs + dynamic programming (memo)
int helper(unordered_set<string>& s, string& start, unordered_map<string, int>& dp) {
   if (dp.find(start) != dp.end()) return dp[start];

   int ans = 1;
   for (int i = 0; i < start.length(); ++i) {
       auto ss = start.substr(0, i) + start.substr(i + 1);
       if (s.find(ss) == s.end()) continue;
       ans = max(ans, 1 + helper(s, ss, dp));
   }
   dp[start] = ans;
   return ans;
}

int longestStrChain(vector<string>& words) {
   unordered_set<string> s(words.begin(), words.end());
   int ans = 0;
   unordered_map<string, int> dp;
   for (auto& w : words) ans = max(ans, helper(s, w, dp));
   return ans;
}
```

Ref: #675

128. Longest consecutive sequence (review)

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, given `[100, 4, 200, 1, 3, 2]`,

The longest consecutive elements sequence is `[1, 2, 3, 4]`. Return its length: `4`.

Your algorithm should run in O(*n*) complexity.

Solution:
Brute force soln: sort the input and then walk through the sorted numbers. run time: O(nlogn) + O(n).

Soln-2: check each number's neighbors and use hashmap to track the longest sequence for each segment.
For example, [100, 4, 200, 1, 3, 2], for n = 100, check its left adjacent and right adjacent from the result map respectively, if it has left and right part, then conjugate them: map[n] = 1 + left# + right#, and update the leftmost and rightmost part. How to find the leftmost part? since we have the left neighbor's longest consecutive sequence: map[n-1], so the leftmost point is n - map[n-1]. Same for the rightmost. Note, the update for all numbers in the middle will be skipped as we only need leftmost and rightmost.

Soln-3: similar to soln-2 (hashmap + extend longest to leftmost and rightmost part), when going through each number, check its neighbors, unite them if found. Time complexity: union-find with path compression has log*n, not linear in theory but practical, O(n * log*n) in total. Space: O(n) * 2 for Union-find and hashmap.

Soln-4: from Stefan. simply put all number into hashset, which is O(n) time. then walk through each number, if n-1 is not in set which assure n is the first number for this segment, greedily to check the successor until not found. Time complexity: O(n) for hashset and O(n) for walk through the vector. Space: O(n) for hashset.

In sum, soln-2 is the best both in time and space complexity. All 3 solns require O(n) space complexity.

By testing, we find brute-force is the fastest even with higher time complexity. So I would like to conclude the extra *lgn* time complexity is small enough to be able to compensating the extra space complexity.
Union-find soln should be slowest because of its highest space complexity for this question.

```
// brute force: O(nlogn) time, O(1) space.
// 67 / 67 test cases passed. Runtime: 14 ms
int soln1(vector<int>& nums) {
    sort(nums.begin(), nums.end());

    int ans = 0;
    for (int i = 0; i < (int)nums.size(); ++i) {

        int res = 1;
        for (int j = i + 1; j < (int)nums.size(); ++j) {
            if (nums[j] > (nums[j - 1] + 1)) break;
            if (nums[j] == (nums[j - 1] + 1)) res++;
            i = j;
        }

        ans = max(ans, res);
    }
    return ans;
}

// hashmap + extend result to two ends: O(n) time, O(n) space
// 67 / 67 test cases passed. Runtime: 29 ms
int soln2(vector<int>& nums) {
    int ans = 0;
    unordered_map<int, int> mp;

    for (int n : nums) {
        if (mp.find(n) != mp.end()) continue;   // Tricky! skip dups otherwise will get wrong answer

        int left = (mp.find(n - 1) != mp.end() ? mp[n - 1] : 0);      // could use iterator to call find once
        int right = (mp.find(n + 1) != mp.end() ? mp[n + 1] : 0);
```

```
        mp[n] = 1 + left + right;

        // extend the left/right most part, the middle part will be ignored.
        mp[n - left] = 1 + left + right;
        mp[n + right] = 1 + left + right;

        ans = max(ans, 1 + left + right);
    }
    return ans;
}

// union-find + hashmap: O(nlg*n) time, 2 * O(n) space
// 67 / 67 test cases passed. Runtime: 24 ms
int soln3(vector<int>& nums) {
    unordered_map<int, int> mp;                     // value-index map
    UnionFind uf(nums.size());                      // no need to know the max #

    for (int i = 0; i < nums.size(); ++i) {
        if (mp.find(nums[i]) != mp.end()) continue; // skip duplicates

        mp[nums[i]] = i;
        if (mp.find(nums[i] + 1) != mp.end()) uf.unite(mp[nums[i] + 1], i);
        if (mp.find(nums[i] - 1) != mp.end()) uf.unite(mp[nums[i] - 1], i);
    }
    return uf.maxSeg();
}

// hashset + greedy search
// 67 / 67 test cases passed. Runtime: 24 ms
int soln4(vector<int>& nums) {
    unordered_set<int> s;
    for (int n : nums) s.insert(n);     // insert is expected running in O(1) time

    int ans = 0;
    for (int n : nums) {
        if (s.find(n - 1) != s.end())   continue;   // n is not the first number for this segment

        // greedy search next number
        int res = 1;
        while (s.find(++n) != s.end()) ++res;

        ans = max(ans, res);
    }
    return ans;
}
```

Ref: #128 #261 #305

130. Surrounded regions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

```
X X X X
X O O X
X X O X
X O X X
```

After running your function, the board should be:

```
X X X X
X X X X
X X X X
X O X X
```

Solution:

1. mark un-surrounded regions first from 4 directions - top, bottom, left and right most
2. flip the board after that to conform output req.

```cpp
void marker(int x, int y, vector<vector<char>>& board) {
   if (x < 0 || x >= board.size()) return;
   if (y < 0 || y >= board[0].size()) return;
   if (board[x][y] != 'O') return;
   board[x][y] = '#';
   marker(x + 1, y, board);    // right
   marker(x - 1, y, board);    // left
   marker(x, y + 1, board);    // up
   marker(x, y - 1, board);    // down
}

void format(vector<vector<char>>& board) {
   for (int i = 0; i < board.size(); ++i) {
       for (int j = 0; j < board[i].size(); ++j) board[i][j] == '#' ? board[i][j] = 'O' : board[i][j] = 'X';
   }
}

void solve(vector<vector<char>>& board) {
   if (board.size() == 0) return;
   int m = board.size(), n = board[0].size();

   for (int i = 0; i < n; ++i) {
       marker(0, i, board);        // mark from top
       marker(m - 1, i, board);    // mark from bottom
   }
   for (int i = 0; j < m; ++j) {
       marker(i, 0, board);        // mark from leftmost
       marker(i, n - 1, board);    // mark from rightmost
   }
   format(board);
}
```

Ref: #200 #286

133/138. Clone graph/ list with random pointers

```cpp
struct UndirectedGraphNode {
    int label;
    vector<UndirectedGraphNode *> neighbors;
    UndirectedGraphNode(int x) : label(x) {};
};

unordered_map<int, UndirectedGraphNode*> _m;
UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
   if (!node) return nullptr;
   if (_m.find(node->label) != _m.end()) return _m[node->label];
   UndirectedGraphNode* nn = new UndirectedGraphNode(node->label);
   _m[nn->label] = nn;
   for (auto n : node->neighbors) {
       nn->neighbors.push_back(cloneGraph(n));
   }
   return nn;
}
```

Ref:

134/484/659/886/927. Gas station... Greedy (review)

```cpp
// 134 - gas station (given amount of gas and cost in each station, return starting point if could be travelled)
// soln-1: greedy
// observation:
```

```
// 1. if A to X is impossible, any station after A can not reach X, so starting from X + 1.
// 2. to have a soln, sum(gas) must be greater than sum(cost)
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
   int cur_gas = 0, sum_gas = 0, sum_cost = 0, cur = 0;;
   for (int i = 0; i < gas.size(); ++i) {
       sum_gas += gas[i], sum_cost += cost[i];

       cur_gas += gas[i] - cost[i];
       if (cur_gas < 0) cur_gas = 0, cur = i + 1;
   }
   return sum_gas >= sum_cost ? cur : -1;
}
```

```
// 484 - Find Permutation (given "DI" indicators find lexicographically smallest permutation)
// soln-1: greedy (brute-force then validate would TLE)
//  D D I I D I
// 1 2 3 4 5 6 7  <== initial value
// 3 2 1 4 6 5 7  <== swap neighbors if hitting D
vector<int> findPermutation(string s) {
   vector<int> ans(s.length() + 1);
   iota(ans.begin(), ans.end(), 1);
   for (int i = 0; i < s.length(); ++i) {
       if (s[i] == 'D') {
           int j = i;
           while (j >= 0 && s[j] == 'D') swap(ans[j], ans[j + 1]), --j;
       }
   }
   return ans;
}
```

```
// 659 - Split Array into Consecutive Subsequences
// soln-1: greedy pickup next 2 available to form sequence
bool isPossible(vector<int>& nums) {
   unordered_map<int, int> cnt, need;
   for (auto x : nums) cnt[x]++;
   for (auto x : nums) {
       if (cnt[x]) {
           if (need[x]) {
               --need[x], ++need[x + 1];
           } else if (cnt[x + 1] && cnt[x + 2]) {
               --cnt[x + 1], --cnt[x + 2], ++need[x + 3];
           } else {
               return false;   // not needed and no bigger number exists
           }
           --cnt[x];
       }
   }
   return true;
}
```

```
// 861 - Score After Flipping Matrix
// soln-1: greedy
// 1. if A[i][0] == 0, flip entire row
// 2. count 1s for current col, flip if less than 0s
int matrixScore(vector<vector<int>>& A) {
    int ans = 0, row = A.size(), col = A[0].size();
    for (auto& r : A) {
        if (0 == r[0]) {
            for (auto& x : r) x = !x;
        }
    }
    for (int j = 1; j < col; ++j) {
        int ones = 0;
        for (int i = 0; i < row; ++i) {
            if (A[i][j]) ones++;
```

```
        }
        ones = max(ones, row - ones);
        ans += ones * (1 << (col - j - 1));
    }
    return ans + row * (1 << (col - 1));
}
```

```
// 846 - Hand of Straights (return true if consists of W consecutive cards)
// soln-1: greedy + hashmap
bool isNStraightHand(vector<int>& hand, int W) {
    if (hand.size() % W != 0) return false;
    map<int, int> mp;   // <val, count>
    for (auto x : hand) mp[x]++;
    for (auto it = mp.begin(); it != mp.end(); nullptr) {
        if (it->second > 0) {
            --it->second;
            for (int i = 1, x = it->first + 1; i < W; ++i, ++x) {
                if (--mp[x] < 0) return false;
            }
        }
        if (it->second <= 0) ++it;
    }
    return true;
}
```

```
// 927 - three equal parts given 0/1 array
// soln-1: greedy
// 1. number of 1s must be divisible by 3, and each segment must have the share of 1/3.
// 2. looking from backwards we know the number of trailing zeros.
// 3. we now have the number of 1s in each segment and trailing zeros, then use brute-force to validate.
bool validate(vector<int>& A, int start, int end, int& cur) {
   while (cur < A.size() && A[cur] != 1) ++cur;
   if (cur >= A.size()) return false;
   for (int i = start; i <= end; ++i, ++cur) {
       if (A[i] != A[cur]) return false;
   }
   return true;
}
vector<int> threeEqualParts(vector<int>& A) {
   auto ones = accumulate(A.begin(), A.end(), 0);
   if (ones % 3) return {-1, -1};
   if (0 == ones) return {0, (int)A.size() - 1};
   int tail_zeros = 0;
   for (auto i = A.size() - 1; i >= 0 && A[i] != 1; --i) ++tail_zeros;

   ones /= 3;
   int tz = 0, one = 0, start = -1, end = 0;   // 1s and trailing 0s counter
   for (auto i = 0; i < A.size(); ++i, ++end) {
       if (one == ones && 0 == A[i]) ++tz;     // count trailing 0 after seen last 1
       if (A[i]) ++one;                        // count 1s
       if (A[i] && -1 == start) start = i;
       if (one == ones && tz == tail_zeros) break;
   }
   if (one != ones || tz != tail_zeros) return {-1, -1};

   int cur = end + 1;
   if (!validate(A, start, end, cur)) return {-1, -1};

   vector<int> ans{end, cur};          // potential answer after we validate 3rd segment
   return validate(A, start, end, cur) ? ans : vector<int>{-1, -1};
}
```

```
// 1007 - Minimum Domino Rotations For Equal Row
// soln-1: brute-force
int minDominoRotations(vector<int>& A, vector<int>& B) {
    int ans = INT_MAX;
```

```
    for (int x = 1; x <= 6; ++x) {
        int top = 0, bottom = 0, i = 0;
        for (; i < A.size(); ++i) {
            if (A[i] != x && B[i] != x) break;
            if (A[i] != x) top++;
            if (B[i] != x) bottom++;
        }
        if (i == A.size()) ans = min(ans, min(top, bottom));
    }
    return INT_MAX == ans ? -1 : ans;
}

// 1007 - Minimum Domino Rotations For Equal Row
// soln-2: greedy (tricky)
// check for example, we have:
// if count-a[i] + count-b[i] - same[i] == A.size(), we found the answer.
int minDominoRotations(vector<int>& A, vector<int>& B) {
    if (A.size() != B.size()) return -1;

    vector<int> ca(7), cb(7), same(7);
    for (int i = 0; i < A.size(); ++i) {
        ca[A[i]]++, cb[B[i]]++;
        if (A[i] == B[i]) same[A[i]]++;
    }
    for (int i = 1; i < ca.size(); ++i) {
        if (ca[i] + cb[i] - same[i] == A.size()) {
            return min(ca[i], cb[i]) - same[i];
        }
    }
    return -1;
}
```

Ref:

135. TODO

Solution:

Ref:

136/137/260/268/389/540. Single/Missing number/Find the difference ... bit manipulation

#137 - every element appears 3 times
#162 - find peak number (binary search for "rising slope")
#260 - two element appeared once, others twice
#268 - missing number (do XOR)
#287 - one duplicate (pigeon hole with tortoise/hare algorithm)
#389 - find the difference between 2 string
#645 - set mismatch

```
// 136 - single number (every element appears twice except one)
// soln-1: bit manipulation
// Follow-up: what if all the numbers appeared twice are neighbors? do binary search
int singleNumber(vector<int>& nums) {
    int ans = 0;
    for (int n : nums) ans ^= n;

    return ans;
}
```

```
// 540 - Single Element in a Sorted Array  (every element appears twice except one)
// soln-1: binary search
```

```
// just consider [1, 1, 2] and [1, 2, 2]
int singleNonDuplicate(vector<int>& nums) {
    int lo = 0, hi = nums.size() - 1;
    while (lo < hi) {
        auto m = (lo + hi) / 2;
        if (nums[m] != nums[m - 1] && nums[m] != nums[m + 1]) return nums[m];

        if (nums[m] == nums[m - 1]) (m - lo) % 2 ? lo = m + 1 : hi = m;
        else (hi - m) % 2 ? hi = m - 1: lo = m;
    }
    return nums[lo];
}
```

```
// 137 - soln-1: bit manipulation
int singleNumber(vector<int>& nums) {
    int ans = 0;
    for (int i = 0; i < sizeof(int) * 8; ++i) {
        int x = 1 << i;

        int sum = 0;
        for (int n : nums) if (n & x) ++sum;
        if (sum % 3) ans |= x;      // ith bit has been set.
    }
    return ans;
}
```

```
// 260 - soln-1: bit manipulation
// 1. divide the two numbers into 2 group according to the last different bit
// 2. do xor for each group
// follow-up: get the lowest different bit
//    xxxx 0110     a
//    xxxx 0101     a-1
//    xxxx 0100     a & (a-1)
//    0000 0100     a ^ (a & (a-1))
vector<int> singleNumber(vector<int>& nums) {
    int lastBit = 0;
    for (int n : nums) lastBit ^= n;
    lastBit = lastBit & ~(lastBit - 1);

    int a = 0, b = 0;
    for (int n : nums) {
        lastBit & n ? a ^= n : b ^= n;
    }
    return vector<int>{a, b};
}
```

```
// 268 - given 0...n, find missing one
// soln-1: bit manipulation
int missingNumber(vector<int>& nums) {
    int ans = 0;
    for (int i = 0; i <= nums.size(); ++i) ans ^= i;
    for (int n : nums) ans ^= n;
    return ans;
}
```

```
// 371 - sum of two integers
// soln-1: bit manipulation
// a + b = (a^b) + (a&b) << 1
// a^b - bitwise plus
// a&b - get last bit-1 in both a and b, which is discard during a^b
int getSum(int a, int b) {
   if (0 == a) return b;
   if (0 == b) return a;
   auto sum = a ^ b, carry = (a & b) << 1;
   return getSum(sum, carry);
}
```

```
// 389 - added 1 extra char into s to get t, find that char
// soln-1: bit manipulation
char findTheDifference(string s, string t) {
   char ans = 0;
   for (char ch : s) ans ^= ch;
   for (char ch : t) ans ^= ch;
   return ans;
}
```

```
// 1016 - Binary String With Substrings Representing 1 To N
// soln-1: brute-force (bit manipulation)
bool queryString(string S, int N) {
   for (int i = N; i > N / 2;  --i) {        // check upper half part is enough (lower half included already)
       string b = bitset<32>(i).to_string();
       if (S.find(b.substr(b.find("1"))) == string::npos)
           return false;
   }
   return true;
}
```

Ref: #162

139/140. Word break/II

Given a string and a dictionary of words, determine if string can be segmented into a space separated sequence of one or more dictionary words. For example, given `s = "leetcode", dict = [leet, code]`, return true because "`leetcode`" can be segmented as "`leet code`".

#140 - Follow up: return all the segmented sentences.

TADM2e 8.5 presents so-called *linear partition* problem, which is similar to this question in the sense of partitioning.
Problem: Integer partition without rearrangement
Input: An arrangement S of nonnegative numbers and an integer K.
Output: Partition S into K or fewer rangers, to minimize the maximum sum over all the ranges, without reordering any of the numbers.
TODO

```
// soln-1: recursion with memorization (top-down)
bool helper(const string& s, unordered_set<string>& dict, const int maxWordLen, int pos, vector<int>& dp) {
   if (pos <= 0) return true;
   if (dp[pos] != -1) return dp[pos] ? true : false;

   dp[pos] = 0;
   for (int i = max(0, pos - maxWordLen); i < pos; ++i) {      // looking for at most maxWordLen
       string w = s.substr(i, pos - i);
       if (dict.find(w) == dict.end()) continue;

       if (helper(s, dict, maxWordLen, i, dp)) {
           dp[pos] = 1; break;                              // 0..i-1 is breakable and i..pos too, so we are
OK.
       }
   }

   return dp[pos] ? true : false;
}

bool wordBreak(string s, vector<string>& wordDict) {
   int maxWordLen = 0;
   unordered_set<string> dict;
   for (auto& str : wordDict) {
       maxWordLen = max(maxWordLen, (int)str.length());
       dict.insert(str);
```

```
    }

    vector<int> dp(s.length() + 1, -1);
    return helper(s, dict, maxWordLen, s.length(), dp);
}

// soln-2: Dynamic Programming (bottom-up)
// let dp(i) be the result at position-i, that is, true if breakable at i otherwise false
//     dp(i) = dp(j) && s(j, i] if exists, j is [max(0, i - max-word-len), i)
bool wordBreak(string s, vector<string>& wordDict) {
    int maxWordLen = 0, minWordLen = INT_MAX;
    unordered_set<string> dict;
    for (auto& str : wordDict) {
        maxWordLen = max(maxWordLen, (int)str.length()); minWordLen = min(minWordLen, (int)str.length());
        dict.insert(str);
    }

    vector<bool> dp(s.length() + 1, false);
    for (int i = minWordLen; i <= s.length(); ++i) {
        for (int j = max(0, i - maxWordLen); j < i; ++j) {
            string w = s.substr(j, i - j);
            if (dict.find(w) == dict.end()) continue;

            if (j == 0 || dp[j]) {  // if 0..j is breakable too, then it's breakable at position i.
                dp[i] = true;
                break;
            }
        }
    }

    return dp[s.length()];
}
```

```
// soln-2: dynamic programming
// let dp(i) be breakable or not at position-i.
// let matrix(i, j) be breakable or not for s(i, j], to look for s(i, j], dp[i] must be breakable.
// at last we build result according to matrix.
vector<string> wordBreak(string s, vector<string>& words) {
    int minWord = INT_MAX, maxWord = INT_MIN;
    unordered_set<string> dict;
    for (auto word : words) {
        minWord = min(minWord, (int)word.length()); maxWord = max(maxWord, (int)word.length());
        dict.insert(word);
    }

    int n = s.length();
    vector<bool> dp(n);
    vector<vector<bool>> matrix(n, vector<bool>(n));

    for (int i = minWord - 1; i < s.length(); ++i) {
        int start = max(0, i - maxWord);
        int end = min(i, start + (maxWord - minWord + 1));      // max process to index i
        for (int j = start; j <= end; ++j) {
            if (j > 0 && dp[j - 1] == false)    continue;    // before looking s(j, i], j-1 must be breakable

            string str = s.substr(j, i - j + 1);
            if (dict.find(str) != dict.end()) {
                matrix[j][i] = dp[i] = true;
            }
        }
    }

    vector<string> ans, path;
```

```
    buildResult(s, matrix, n - 1, path, ans);

    return ans;
}

// build result from bottom-right to up-left corner
void buildResult(string& s, vector<vector<bool>>& dp, int ending, vector<string>& path, vector<string>& ans) {
    if (ending < 0) {
        string str(path.front());
        for (int i = 1; i < path.size(); ++i) str = path[i] + " " + str;
        ans.push_back(str);

        return;
    }

    for (int i = ending; i >= 0; --i) {
        if (dp[i][ending] == false)  continue;

        path.push_back(s.substr(i, ending - i + 1));
        buildResult(s, dp, i - 1, path, ans);
        path.pop_back();
    }
}
```

Ref: #472

143. Reorder list

Given L_0->L_1->...->L_{n-1}->L_n, return L_0->L_n->L_1->L_{n-1}->...

```
void reorderList(ListNode* head) {
    ListNode* first = head, *second = head;  // first is changed in recursion, but head is not.
    helper(first, second);
}

void helper(ListNode *&first, ListNode *second) {
 if (!first || !second)    return;
 helper(first, second->next); // dfs to hit the end
 if (first) {
   if (first == second || first->next == second) {
     second->next = NULL;
   } else {
     second->next = first->next, first->next = second;
   }
   first = second->next;
 }
}
```

Ref: #234

146/460/588/604/642. LRU/LFU/Autocomplete/In-memory file system/Compressed string iter(review)

```
// 146 - LRU cache
// soln-1: hashmap + linked list
class LRUCache {
    list<pair<int, int>> _list;
    unordered_map<int, list<pair<int, int>>::iterator> _keys;
    int _cap;
public:
    LRUCache(int capacity) : _cap(capacity) {
    }
```

```cpp
    int get(int key) {
        auto it = _keys.find(key);
        if (it == _keys.end()) return -1;
        int v = it->second->second;
        _list.push_front({key, v}), _list.erase(it->second), _keys[key] = _list.begin();
        return v;
    }

    void put(int key, int value) {
        if (get(key) != -1) {
            _keys[key]->second = value;
        } else {
            if (_keys.size() >= _cap) {
                _keys.erase(_list.back().first), _list.pop_back();
            }
            _list.push_front({key, value}), _keys.insert({key, _list.begin()});
        }
    }
};
```

```cpp
// 460 - Least Frequently Used cache
// soln-1: hashmap
// (1) key  | (val, freq)  |
// (2) freq | list of keys |, list of keys which have the same frequency
// (3) key  | iter in list |, iterator of list
// 1. get(x)
//    1.1 x is known   use (1) for val/freq, (3) to remove it from old freq (update min-freq if any), update (2)
//    1.2 x is unknown
// 2. put(x)
//    2.1 x is known   same get(x)
//    2.2 x is unknown use (2) freq[min-freq] to free space then remove key. insert new key and reset min-freq=1

class LFUCache {
   unordered_map<int, pair<int, int>> _keys;       // key -> (v, freq)
   unordered_map<int, list<int>> _freq;            // freq -> list of keys
   unordered_map<int, list<int>::iterator> _pos;   // key -> key's position in freq list
   int _cap, _minFreq;
public:
   LFUCache(int capacity) : _cap(capacity) {
   }

   int get(int key) {
       auto it = _keys.find(key);
       if (it == _keys.end()) return -1;

       int v = it->second.first, freq = it->second.second;
       _freq[freq].erase(_pos[key]);
       if (_freq[freq].empty()) _freq.erase(freq);
       ++freq, ++it->second.second;
       _freq[freq].push_front(key), _pos[key] = _freq[freq].begin();

       if (_freq.find(_minFreq) == _freq.end()) ++_minFreq;
       return v;
   }

   void put(int key, int value) {
       if (_cap <= 0) return;
       if (get(key) != -1) {
           auto it = _keys.find(key);
           it->second.first = value;
       } else {
           if (_keys.size() >= _cap) {
               int del = _freq[_minFreq].back();
               _keys.erase(del), _freq[_minFreq].pop_back(), _pos.erase(del);
               if (_freq[_minFreq].empty()) _freq.erase(_minFreq);
           }
```

```
            _keys.insert({key, {value, 1}}), _freq[1].push_front(key), _pos[key] = _freq[1].begin();
            _minFreq = 1;
        }
    }
};
```

```
// 588 - In-memory file system
// soln-1: hashmap
// 1. to keep it simply, use hash(absolute-path) instead of relative path
// 2. files and folders are using different hashmap because
//    folder contains folders which is a set, but files contain strings.
class FileSystem {
public:
    FileSystem() {
        _dirs["/"];
    }

    vector<string> ls(string path) {
        if (_files.find(path) != _files.end()) {
            int idx = path.find_last_of('/');
            return {path.substr(idx + 1)};
        } else if (_dirs.find(path) != _dirs.end()) {
            auto t = _dirs[path];
            return vector<string>(t.begin(), t.end());
        } else {
            return vector<string>{};
        }
    }

    void mkdir(string path) {
        istringstream iss(path);
        string t, dir;
        while (getline(iss, t, '/')) {
            if (t.empty()) continue;
            if (dir.empty()) dir = "/";
            _dirs[dir].insert(t);
            if (dir.size() > 1) dir += "/";
            dir += t;
        }
    }

    void addContentToFile(string filePath, string content) {
        int idx = filePath.find_last_of('/');
        string dir = filePath.substr(0, idx), file = filePath.substr(idx + 1);
        if (dir.empty()) dir = "/";
        if (_dirs.find(dir) == _dirs.end()) mkdir(dir);
        _dirs[dir].insert(file);
        _files[filePath].append(content);
    }

    string readContentFromFile(string filePath) {
        return _files[filePath];
    }

private:
    unordered_map<string, set<string>> _dirs;
    unordered_map<string, string> _files;
};
```

```
// 604 - compressed string iterator
// soln-1: easy iterator
class StringIterator {
    string _str;
    int _idx, _left;
public:
    StringIterator(string compressedString) {
```

```
        _str = compressedString;
        _idx = 1, _left = _idx < _str.length() ? _str[_idx] - '0' : -1;
    }

    char next() {
        if (!hasNext()) return ' ';
        auto ans = _str[_idx - 1];
        if (--_left <= 0) _idx += 2, _left = _idx < _str.length() ? _str[_idx] - '0' : -1;
        return ans;
    }

    bool hasNext() {
        return _left > 0;
    }
};
```

```
// 642 - Design Search Autocomplete System
// For each character they type except '#', you need to return the top 3 historical hot sentences
// that have prefix the same as the part of sentence already typed.
// soln-1: hashmap + priority queue + trie
// 1. build <sentence, frequency> map
// 2. when get input from user, go thru. each key in hashmap, use priority queue to keep most hot 3 sentence
// 3. when user finished input (seen '#'), update <sentence, frequency> map
// 4. go thru. hashmap to match prefix is expensive, options:
//    4.1 limit the size of hashmap, such as LRU/LFU
//    4.2 build trie for sentence, this would be good for matching prefix, but increase space complexity.
class AutocompleteSystem {
    unordered_map<string, int> _freq;
    string _input;
public:
    AutocompleteSystem(vector<string> sentences, vector<int> times) {
        for (int i = 0; i < sentences.size(); ++i) {
            _freq[sentences[i]] += times[i];
        }
    }

    vector<string> input(char c) {
        vector<string> ans;
        if ('#' == c) {
            ++_freq[_input], _input.clear();
        } else {
            _input.push_back(c);
            auto cmp = [](pair<string, int>& a, pair<string, int>& b) {
                return a.second > b.second || (a.second == b.second && a.first < b.first);
            };
            priority_queue<pair<string, int>, vector<pair<string, int>>, decltype(cmp) > pq(cmp);
            for (auto f : _freq) {                  // go thru. hashmap, could be expensive
                bool matched = true;
                for (int i = 0; matched && i < _input.size(); ++i) {
                    if (_input[i] != f.first[i]) matched = false;
                }
                if (matched) pq.push(f);
                if (pq.size() > 3) pq.pop();
            }
            while (!pq.empty()) ans.push_back(pq.top().first), pq.pop();
        }
        return ans;
    }
};
```

Ref:

147/ 148. Insertion / sort list

```
void insert(ListNode* head, ListNode* n) {
   ListNode* pre = head, *cur = head->next;
   while (cur && cur->val < n->val) pre = cur, cur = cur->next;
   pre->next = n, n->next = cur;
}
ListNode* insertionSortList(ListNode* head) {
   ListNode dummy(0);
   dummy.next = nullptr;
   for (ListNode* n = head; n; nullptr) {
       ListNode* next = n->next;
       insert(&dummy, n);
       n = next;
   }
   return dummy.next;
}
```

```
ListNode* split(ListNode* h) {
   if (NULL == h) return NULL;
   ListNode* slow = h, *fast = h->next;
   while (fast && fast->next) slow = slow->next,  fast = fast->next->next;

   ListNode* r = slow->next;   slow->next = NULL;
   return r;
}
ListNode* merge(ListNode* l1, ListNode* l2) {
   ListNode d(0);
   ListNode* cur = &d;
   for (; l1 && l2; cur = cur->next) {
       if (l1->val < l2->val) {
           cur->next = l1;   l1 = l1->next;
       } else {
           cur->next = l2;   l2 = l2->next;
       }
   }

   if (l1) cur->next = l1;
   else if (l2) cur->next = l2;

   return d.next;
}

// merge sort: T(n) = 2T(n/2) + n
ListNode* sortList(ListNode* head) {
   if (!head || !head->next) return head;
   auto r = split(head);
   return merge(sortList(r), sortList(head));
}
```

Ref: #75

149. Max points on a line

Given *n* points on a 2D plane, find the maximum number of points that lie on the same straight line.

Solution:

soln-1: brute force in O(n^2) time and O(n) space.

If a point with all other points share same slope, they must in same line. To calculate slope k, we have the formula: k = (y1 - y2) / (x1 - x2). 2 corner cases:

1) we set k = INT_MAX if x1 == x2;
2) count duplicate points.

soln-2: accuracy improvement. In soln-1, float data type is used as key in unordered_map, it could be improved using pair<int, int> to represent GCD as slope.

```
map<pair<int, int>, int> slope;
slope[make_pair(dy / gcd, dx / gcd)]++;
```

```
// 27 / 27 test cases passed. Runtime: 20 ms
int maxPoints(vector<Point>& points) {
    int ans = 0;
    for (int i = 0; i < points.size(); ++i) {
        Point& x1 = points[i];

        int dups = 1;
        unordered_map<float, int> slope;
        for (int j = i + 1; j < points.size(); ++j) {
            Point& x2 = points[j];

            if (x1.x == x2.x && x1.y == x2.y) {
                dups++;
                continue;
            }

            float k = x1.x == x2.x ? INT_MAX : float(x1.y - x2.y) / (x1.x - x2.x);
            ++slope[k];
        }
        ans = max(ans, dups);       // in case all are dups
        for (auto iter : slope) ans = max(ans, iter.second + dups);
    }
    return ans;
}
```

Ref: #356

150. Evaluate reverse Polish notation

Example: ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9

#224/227/772/770 - basic calculator / II / III / IV

```
int evalRPN(vector<string>& tokens) {
   stack<int> stk;
   for (auto& tk : tokens) {
       if (isdigit(tk.back())) {           // use back in case of -4
           stk.push(stoi(tk));
       } else {
           int a = stk.top();    stk.pop();
           int b = stk.top();    stk.pop();

           if ("+" == tk) stk.push(b + a);
           else if ("-" == tk) stk.push(b - a);
           else if ("*" == tk) stk.push(b * a);
           else if ("/" == tk) stk.push(b / a);
       }
   }
   return stk.top();
}
```

Ref:

151/186/541/557/916. Reverse words in a string/Word subset

#151 - given s = " the   sky is blue   ", return "blue is sky the".
#186 - given s = "the sky is blue", return "blue is sky the".
#557 - given s = "Let's take LeetCode contest", return: "s'teL ekat edoCteeL tsetnoc"

```
// #186 - given s = "the sky is blue", return "blue is sky the"
// for in-place, we can reverse each word individually, then the whole string
void reverseWords2(string &s) {
```

```
   istringstream is(s);
   for(string tmp; is >> tmp; nullptr) {
       s = tmp + " " + s;
   }
}

// #557 - given s = "Let's take LeetCode contest", return: "s'teL ekat edoCteeL tsetnoc"
string reverseWords(string s) {
   for (int i = 0; i < s.length(); ++i) {
       int start = i;
       while (i < s.length() && s[i] != ' ') ++i;

       reverse(s.begin() + start, s.begin() + i);
   }
   return s;
}
```

```
// 916 - Word Subsets (every b in B must be subset of word in A)
// soln-1: hashmap with trick
vector<string> wordSubsets(vector<string>& A, vector<string>& B) {
    vector<int> u(26), tmp(26);
    for (auto& b : B) {
        tmp = count(b);
        for (auto i = 0; i < u.size(); ++i) u[i] = max(u[i], tmp[i]);
    }
    vector<string> ans;
    for (auto& a : A) {
        tmp = count(a);
        bool match = true;
        for (auto i = 0; i < u.size(); ++i) {
            if (u[i] > tmp[i]) match = false;
        }
        if (match) ans.push_back(a);
    }
    return ans;
}
vector<int> count(string& s) {
    vector<int> cnt(26);
    for (auto ch : s) cnt[ch - 'a']++;
    return cnt;
}
```

```
// 1023 - Camelcase Matching (insert lowercase letters to get queries)
// soln-1: brute-force
vector<bool> camelMatch(vector<string>& queries, string pattern) {
    vector<bool> ans;
    for (auto& q : queries) {
        int i = 0, j = 0;
        for (i = 0, j = 0; j < q.length(); ++j) {
            if (i < pattern.length() && pattern[i] == q[j]) ++i;
            else if (q[j] >= 'A' && q[j] <= 'Z') break;
        }
        ans.push_back(i == pattern.length() && j == q.length());
    }
    return ans;
}
```

Ref:

152/713/795. Max product subarray/Subarray product < K/Subarrays within bound (review)

152 - Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array `[2,3,-2,4]`, the contiguous subarray `[2,3]` has the largest product = 6.

713 - Your are given an array of positive integers nums. Count and print the number of (contiguous) subarrays where the product of all the elements in the subarray is less than k.

795 - We are given an array A of positive integers, and two positive integers L and R(L <= R). Return the number of (contiguous, non-empty) subarrays such that the value of the maximum array element in that subarray is at least L and at most R.

```
// 152 - soln-1: dynamic programming
// Both max/min from previous product could potentially contribute when time current value, so keep both of them.
//     2 3  -2    4   10   -4
// max 2 6  -2    4   40 1920  <= max = max{cur, preMax * cur, preMin * cur}
// min 2 3 -12  -48 -480 -160  <= min = min{cur, preMax * cur, preMin * cur}
int maxProduct(vector<int>& nums) {
   int ans = nums[0], small = ans, big = ans;
   for (int i = 1; i < nums.size(); ++i) {
       int s = small * nums[i], b = big * nums[i];
       small = min(nums[i], min(s, b)), big = max(nums[i], max(s, b));
       ans = max(ans, max(small, big));
   }
   return ans;
}
```

```
// 713 - soln-1: two pointers
// x y z a b c
//     ^
//   ^ ^  for position z, we have at most 3 subarrays (looking backward).
// ^ ^ ^
int numSubarrayProductLessThanK(vector<int>& nums, int k) {
   int ans = 0, left = 0;
   for (int i = 0, prod = 1; i < nums.size(); ++i) {
       prod *= nums[i];
       while (prod >= k && left <= i) prod /= nums[left++];
       ans += i - left + 1;
   }
   return ans;
}
```

```
// 795 - soln-1: greedy or two pointers
// similar to #713 - subarray product < k
int numSubarrayBoundedMax(vector<int>& A, int L, int R) {
   int ans = 0, left = -1, right = -1;
   for (int i = 0; i < A.size(); ++i) {
       if (A[i] > R) left = i;     // cur > upper-boundary, there would be no contribution wrt cur position
       if (A[i] >= L) right = i;   // as long as cur >= lower-boundary
       ans += right - left;
   }
   return ans;
}
```

Ref: #53 #198 #238

155/716. Min / Max stack

155 - Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

```
// 155: soln-1: use 2 stack, the other for min till now
// noticed one stack soln but has to store LONG instead of INT which essentially take same space
```

```
class MinStack {
    vector<int> _stk, _min;
public:
    void push(int x) {
        _stk.push_back(x);
        _min.push_back(_min.empty() ? x : min(_min.back(), x));
    }

    void pop() {
        if (!_stk.empty()) _stk.pop_back(), _min.pop_back();
    }

    int top() {
        return _stk.back();
    }

    int getMin() {
        return _min.back();
    }
};
```

Ref: #239

156. Binary tree upside down

Given a binary tree where all the right nodes are either leaf nodes with a sibling or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Solution:
This question really confuses me. See code.

```
TreeNode* upsideDownBinaryTree(TreeNode* root) {
   TreeNode* curr = root;
   TreeNode* parent = NULL;
   TreeNode* sibling = NULL;

   while (curr) {
      TreeNode* left = curr->left;
      TreeNode* right = curr->right;

      curr->left = sibling;  // sibling turns to be left
      curr->right = parent;  // parent turns to be right (like reverse)
      sibling = right;       // right is going to be sibling for next level
      parent = curr;

      curr = left;
   }
   return parent;     // the left most leaf node
}
```

Ref:

157/158. Read N char given read4 / II - call multiple times

The API: int read4(char *buf) reads 4 characters at a time from a file. The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

```
int read(char* buf, int n) {
   int len = 0;
   for (int r = 0; n > len; len += r) {
```

```
        r = read4(&buf[len]);
        if (0 == r) break;
    }
    return len;
}
```

```
// 158 - soln-1: read the data from local buf first
int read(char* buf, int n) {
    static char _buf[4];
    static int _cur = 0, _len = 0;
    if (n <= 0) return 0;

    if (_len) {
        int r = min(_len, n);
        memcpy(buf, &_buf[_cur], r);

        _len -= r, _cur += r;
        return r + read(&buf[r], n - r);
    } else {
        _len = read4(_buf), _cur = 0;
        return _len ? read(buf, n) : 0;
    }
}
```

Ref: #158

160. Intersection of two linked lists

```
// 160 - intersection of 2 linked lists
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
     int la = length(headA), lb = length(headB);
     for (int i = 0; i < (la - lb); ++i) headA = headA->next;
     for (int i = 0; i < (lb - la); ++i) headB = headB->next;

     while (headA != headB) headA = headA->next, headB = headB->next;
     return headA;
}
```

Ref:

162/941. Find peak element/Validate mountain array

A peak element is an element that is greater than its neighbors. Given an array where nums[i] != nums[i+1], find the peak element index. The given array may contain multiple peaks, return any of them

```
// soln-1: sequential search "rising slope" until going down
int findPeakElement(vector<int>& nums) {
    for (int i = 1; i < nums.size(); ++i) {
        if (nums[i - 1] > nums[i]) return i - 1;
    }
    return nums.size() - 1;
}

// soln-2: binary search "rising slope" essentially
int findPeakElement(vector<int>& nums) {
    int low = 0, hi = nums.size();
    while (low < hi) {
        int m = low + (hi - low) / 2;
        nums[m] < nums[m + 1] ? low = m + 1 : hi = m;    // low and hi are at slope top.
    }
    return low;     // low == hi
}
```

```cpp
// 941 - soln-1: two pointers (two person will meet at the same peak)
bool validMountainArray(vector<int>& A) {
    int i = 0, j = A.size() - 1;
    while (i < j && A[i] < A[i + 1]) ++i;    // climb up from left
    while (j > 0 && A[j - 1] > A[j]) --j;    // climb up from right
    return i > 0 && i == j && j < A.size() - 1;
}
```

Ref: #153

163/228. Missing/Summary ranges

163 - Given a sorted integer array where the range of elements are `[0, 99]` inclusive, return its missing ranges.

For example, given `[0, 1, 3, 50, 75]`, return [“2”, “4->49”, “51->74”, “76->99”]

228 - Given a sorted integer array without duplicates, return the summary of its ranges.

For example, given `[0,1,2,4,5,7]`, return `["0->2","4->5","7"]`.

```cpp
// 163 - find missing range (brute-force)
// given [0, 1, 3, 50, 75], return ["2", "4->49", "51->74", "76->99"] if range is [0, 99]
vector<string> findMissingRanges(vector<int>& nums, int lower, int upper) {
   vector<string> ans;
   int left = lower;
   for (int i = 0; i < nums.size(); ++i) {
       int right = nums[i];
       if (right > left) {
           string str = (right - 1 == left) ? to_string(left) : (to_string(left) + "->" + to_string(right - 1));
           ans.push_back(str);
       }
       left = nums[i] + 1;
   }
   int right = upper;
   if (right >= left) {
       string str = (right == left) ? to_string(left) : (to_string(left) + "->" + to_string(right));
       ans.push_back(str);
   }
   return ans;
}
```

```cpp
// 228 - summary range (brute-force)
// given [0,1,2,4,5,7], return ["0->2","4->5","7"]
vector<string> summaryRanges(vector<int>& nums) {
   vector<string> ans;
   if (nums.empty()) return ans;

   int start = nums[0], pre = start;
   for (int i = 1; i < nums.size(); ++i) {
       if (pre + 1 == nums[i]) {
           pre = nums[i];
       } else {
           ans.push_back(pre == start ? to_string(start) : (to_string(start) + "->" + to_string(pre)));
           start = nums[i], pre = start;
       }
   }
   ans.push_back(pre == start ? to_string(start) : (to_string(start) + "->" + to_string(pre)));
   return ans;
}
```

Ref:

164. Max gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form. Return 0 if the array contains less than 2 elements.

Example 1: `Input: [3,6,9,1], Output: 3`

```
// 164 - soln-1: bucket sort in O(n) time and space
// 1. average-gap = ceiling[(max - min) / (n - 1)], n = numbers in array. For example, [1, 2, 3] => (3-1)/2 = 1
// 2. divide all numbers into n-1 buckets, k-th bucket contains min + [k * gap, (k + 1) * gap)
// 3. for each bucket, store only min/max of a bucket, because max-gap can only happen between 2 buckets.
int maximumGap(vector<int>& nums) {
    if (nums.empty()) return 0;
    int ma = INT_MIN, mi = INT_MAX;
    for (auto& x : nums) ma = max(ma, x), mi = min(mi, x);

    int avg_gap = (ma - mi) / nums.size() + 1;
    vector<pair<int, int>> bucket(nums.size(), {INT_MAX, INT_MIN});
    for (auto& x : nums) {
        int idx = (x - mi) / avg_gap;
        bucket[idx] = {min(bucket[idx].first, x), max(bucket[idx].second, x)};
    }

    int ans = 0;
    for (int i = 1, pre = 0; i < bucket.size(); ++i) {
        if (bucket[i].first == INT_MAX) continue;
        ans = max(ans, bucket[i].first - bucket[pre].second);
        pre = i;
    }
    return ans;
}
```

Ref:

168/171. Excel sheet column title/number

For example, 1 -> A, 2 -> B, 3 -> C, …, 26 -> Z, 27 -> AA, 28 -> AB

```
// 168:
string convertToTitle(int n) {
   if (n <= 0) return "";
   return convertToTitle((n - 1) / 26) + (char)((n - 1) % 26 + 'A');
}
// 171:
int titleToNumber(string s, int level = 1) {
   if (s.empty()) return 0;
   int x = s.back() - 'A' + 1; s.pop_back();
   return x * level + titleToNumber(s, level * 26);
}
```

Ref:

169/229. Majority element/II/III

169 - Given an array of size *n*, find the majority element. Find the element that appears more than `⌊n/2⌋` times. You may assume that the array is non-empty and the majority element **always exist** in the array.

229 - Given an integer array of size *n*, find all elements that appear more than `⌊ n/3 ⌋` times.

III - occurs more than 1/k, given majority number always exists.

```cpp
// 169 - soln-1: Moore voting (wiki) - majority must always exist
// other solns:
// - sort then the majority element must be the middle one if exist.
// - use hashmap in O(n) time and space
int majorityElement(vector<int>& nums) {
   int count = 0, ans = nums[0];
   for (int i = 0; i < nums.size(); ++i) {
       if (ans == nums[i]) count++;
       else if (0 == count) { ans = nums[i]; count = 1;}
       else --count;
   }
   return ans;
}

// 169 - soln-2: bit manipulation (if the bit occurs more than n/2, then keep it)
int majorityElement(vector<int>& nums) {
   int ans = 0;
   for (int i = sizeof(int) * 8 - 1; i >= 0; --i) {
       int count = 0, mask = 1 << i;
       for (int x : nums) {
           if (x & mask) ++count;
           if (count > nums.size() / 2) { ans |= mask; break; }
       }
   }
   return ans;
}
```

```cpp
// 229 - soln-1: modified Moore voting algorithm
vector<int> majorityElement(vector<int>& nums) {
   vector<int> ans;

   int count1 = 0, count2 = 0, c1 = INT_MIN, c2 = INT_MAX;
   for (int x : nums) {
       if (c1 == x)             count1++;
       else if (c2 == x)        count2++;
       else if (0 == count1)    { c1 = x; count1 = 1; }
       else if (0 == count2)    { c2 = x; count2 = 1; }
       else --count1, --count2;
   }

   // extra pass to verify in case there are no majority numbers
   count1 = 0, count2 = 0;
   for (int x : nums) {
       if (x == c1)         ++count1;
       else if (x == c2)    ++count2;
   }
   if (count1 > nums.size() / 3) ans.push_back(c1);
   if (count2 > nums.size() / 3) ans.push_back(c2);
   return ans;
}
```

```cpp
// majorityNumber III - variant of Moore Voting
// occurs more than 1/k, given only 1 exist, in O(n) time and O(k) extra space
int majorityNumber(vector<int> &nums, int k) {
   unordered_map<int, int> mp;
   for (auto& x : nums) {
       mp[x]++;
       if (mp.size() > k) cleanup(mp);
   }

   int ans = INT_MIN, count = 0;
   for (auto& p : mp) {
       if (p.second > count) count = p.second, ans = p.first;
```

```
    }
    return ans;
}

void cleanup(unordered_map<int, int>& mp) {
    vector<int> remove;
    for (auto& p : mp) {
        if (--p.second == 0) remove.push_back(p.first);
    }
    for (auto& x : remove) mp.erase(x);
}
```

Ref:

170. Two sum III - data structure design

Design and implement a TwoSum class. It should support the following operations: `add` and `find`.

`add` - Add the number to an internal data structure.

`find` - Find if there exists any pair of numbers which sum is equal to the value.

For example,

```
add(1); add(3); add(5);
find(4) -> true
find(7) -> false
```

Solution:
Hashmap/multiset would do the job. multiset soln is much faster than hashmap.

```
// 14 / 14 test cases passed. Runtime: 1500 ms
class TwoSum {
    unordered_map<int, int> m_mp;
public:
    void add(int number) {
        m_mp[number]++;
    }

    bool find(int value) {
        for (auto iter : m_mp) {
            int x = iter.first, y = value - x;
            if ((x == y && iter.second > 1) || (x != y && m_mp.find(y) != m_mp.end())) {
                return true;
            }
        }
        return false;
    }
};

// 14 / 14 test cases passed. Runtime: 1232 ms
class TwoSum2 {
    unordered_multiset<int> nums;
public:
    void add(int number) {
        nums.insert(number);
    }

    bool find(int value) {
        for (int i : nums) {
            int count = i == value - i ? 1 : 0;
            if (nums.count(value - i) > count) {
                return true;
            }
        }
        return false;
    }
};
```

Ref: #1 #288

172/233/483/573/793. Factorial trailing zeros/Number of digit one/Factorial zeros/... (review)

```
// 171 - trailing zeros
// soln-1: math
// 1, zeros come from 10 which is from 2 * 5
// 2, 12 has 2 zeros (5, 10), 22 has 4 zeros (5, 10, 15, 20)
// 3, 100 has 100 / 5 = 20 ? NO, 25 * 4 also generates zeros
//            100 / 5^2 = 4  ==> 20 + 4 = 24
// 4, n will have n / 5    \
//                n / 5^2   + sum all of them
//                n / 5^3  /
using ull = unsigned long long;
ull trailingZeroes(ull n) {
   ull ans = 0;
   for (ull i = 5; n / i; i *= 5) ans += (n / i);
   return ans;
}
```

```
// 233 - number of digit one (f(13) = 6 => 1, 10, 11, 12, 13)
// soln-1: math
// f(abcd) = 10^3 + a * f(10^3 - 1) + f(bcd),      if a > 1
//         = (bcd + 1) + a * f(10^3 - 1) + f(bcd), if a == 1
int countDigitOne(int n) {
   if (n < 10) return n > 0 ? 1 : 0;

   int a = 0, len = 0;
   for (int i = n; i; i /= 10) len++, a = i % 10;

   int m = pow(10, len - 1), bcd = n % m, ans = (1 == a) ? (bcd + 1) : m;
   return ans + a * countDigitOne(m - 1) + countDigitOne(bcd);
}
```

```
// 483 - smallest good base
// soln-1: math + binary search
// given n = b^(m-1) + ... + b^1 + b^0, estimate the range of b and m:
// 1. n > b^(m-1) ==> m: [1, lgn/lg2 + 1] (when b = 2, m is the biggest)
// 2. b: [2, n] in general, but n > b^(m-1) ==> b < n^(1/(m - 1)) to avoid overflow
// 3. binary search in range of b to meet: n = b^(m-1) + ... + 1
string smallestGoodBase(string n) {
    using ull = unsigned long long;
    ull num = stoll(n);
    for (int m = log(num) / log(2) + 1; m > 1; --m) {
        ull l = 2, r = pow(num, 1.0 / (m - 1)) + 1;     // fix base range
        while (l < r) {
            ull b = l + (r - l) / 2, sum = 0, val = 1;
            for (int i = 0; i < m; ++i, val *= b) sum += val;
            if (sum == num) return to_string(b);        // biggest m means smallest b.
            sum < num ? l = b + 1 : r = b;
        }
    }
    return to_string(num - 1);                          // the biggest base for every number
}
```

```
// 573 - squirrel simulation (min distance to collect nuts and put into tree)
// soln-1: math
// 1. since each time only 1 nut can be picked, as long as starting from home(tree), the distance is const.
// 2. so the only question is how to choose the first nut.
//    let x be the distance between squirrel and nut
//        y be the distance between tree/home and nut
//    case-1: if we choose the nut, then go home, the distance would be: x + y
//    case-2: if we choose other nut and pick this one later, the distance would be: 2x
// 3. we want to find the nut which gives us the max benefit: 2x - (x + y)
int minDistance(int height, int width, vector<int>& tree, vector<int>& squirrel, vector<vector<int>>& nuts) {
   int ans = 0, offset = 0;
   for (auto& nut : nuts) {
      auto x = abs(nut[0] - tree[0]) + abs(nut[1] - tree[1]);
```

```
        auto y = abs(nut[0] - squirrel[0]) + abs(nut[1] - squirrel[1]);
        ans += 2 * x, offset = max(offset, x - y);
    }
    return ans - offset;
}
```

```
// 793 - factorial zeros function (how many integers have K ending zeros)
// soln-1: math + binary search
// 1. when number increase, trailing zeroes not decrease.
// 2. find a range such that all numbers in between have the same number of trailing zeros.

// return the minimum number which has K trailing zeros
int helper(int K) {
    ull lo = 0, hi = LONG_MAX;
    while (lo < hi) {
        auto m = lo + (hi - lo) / 2;
        trailingZeroes(m) >= K ? hi = m : lo = m + 1;
    }
    return lo;
}
int preimageSizeFZF(int K) {
    return helper(K + 1) - helper(K);
}
```

```
// 878 - Nth magical number (divisible either by A or B)
// soln-1: math + binary search
// given x, nth magic number = x / A + x / B + x / lcm(A, B), then binary search
int nthMagicalNumber(int N, int A, int B) {
    using ull = unsigned long long;
    ull lo = 0, hi = LONG_MAX, mod = 1e9+7, lcm = (A * B) / gcd(A, B);
    while (lo < hi) {
        auto m = lo + (hi - lo) / 2;
        auto n = m / A + m / B - m / lcm;
        n >= N ? hi = m : lo = m + 1;
    }
    return lo % mod;
}
```

Ref:

173/230/671. BST iterator/K-th smallest in BST/2nd smallest in binary tree

```
// 173 - BST iterator
// soln-1: stack
class BSTIterator {
    stack<TreeNode*> _stk;

    void populate(TreeNode* root) {
        for (root; root; root = root->left) {
            _stk.push(root);
        }
    }

public:
    BSTIterator(TreeNode *root) {
        populate(root);
    }

    bool hasNext() {
        return !_stk.empty();
    }

    int next() {
        TreeNode* node = _stk.top(); _stk.pop();
        populate(node->right);

        return node->val;
    }
```

```
};
```

```
// 230 - k-th smallest in BST
// soln-1: stack (in-order traversal)
int kthSmallest(TreeNode* root, int k) {
    stack<TreeNode*> s;
    while (root) s.push(root), root = root->left;
    while (!s.empty()) {
        if (0 == --k) return s.top()->val;
        auto child = s.top()->right;    s.pop();
        while (child) s.push(child), child = child->left;
    }
    return -1;
}
```

```
// 671 - 2nd smallest in binary tree
// soln-1: dfs (brute-force)
int findSecondMinimumValue(TreeNode* root) {
    int m1 = INT_MAX, m2 = INT_MAX;
    h(root, m1, m2);
    return INT_MAX != m2 ? m2 : -1;
}
void h(TreeNode* node, int& m1, int& m2) {
    if (nullptr == node) return;

    h(node->left, m1, m2), h(node->right, m1, m2);
    if (node->val < m1) m2 = m1, m1 = node->val;
    else if (node->val < m2 && node->val != m1) m2 = node->val;
}
```

Ref: #251 #281 #284 341

174. Dungeon game

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess. For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

```
-2 (K)  -3      3
-5      -10     1
10       30    -5 (P)
```

```
// soln-1: dynamic programming
// let dp(i, j) be the minimum points to reach (i+1, j) or (i, j+1),
// 1. right = max(1, dp[i][j+1] - price)
// 2. down = max(1, dp[i+1][j] - price)
// 3. dp[i][j] = min(right, down)
// 4. need special process for last row and col
int calculateMinimumHP(vector<vector<int>>& dungeon) {
    if (dungeon.empty()) return 0;

    int m = dungeon.size(), n = dungeon[0].size();
    vector<vector<int>> dp(2, vector<int>(n));

    int cur = 0, next = 1;
    for (int j = n - 1; j >= 0; --j) {                              // the last row
            dp[next][j] = max(1, (j < n - 1 ? dp[next][j + 1] : 1) - dungeon[m - 1][j]);
    }
    for (int i = m - 2; i >= 0; --i) {
        dp[cur][n - 1] = max(1, dp[next][n - 1] - dungeon[i][n - 1]);   // right most col

        for (int j = n - 2; j >= 0; --j) {
            int right = max(1, dp[cur][j + 1] - dungeon[i][j]);   // per right direction
            int down = max(1, dp[next][j] - dungeon[i][j]);       // per down direction
```

```
            dp[cur][j] = min(right, down);
        }
        swap(cur, next);
    }
    return dp[next][0];
}
```

Ref: #62 #741

179. Largest number

Given a list of non negative integers, arrange them such that they form the largest number.

Example 1: `Input: [10,2], Output: "210"`

```
// soln-1: convert to string then sort
string largestNumber(vector<int>& nums) {
   vector<string> s(nums.size());
   for (int i = 0; i < nums.size(); ++i) s[i] = to_string(nums[i]);

   sort(s.begin(), s.end(), [](const string& a, const string& b){
       return a + b > b + a;           // following code is essentially same as this line, but more efficient
       for (int i = 0; i < max(a.length(), b.length()); ++i) {
           if (a[i] == b[i]) continue;
           if (a[i] && b[i]) return a[i] > b[i];

           if (0 == a[i]) {
               // compare b[i] with b[0..i-1], b can go first only if b[i] > b[0..i-1]
               for (int k = 0; k < i; ++k) {
                   if (b[i] > b[k]) return false;      // b goes first
                   if (b[i] < b[k]) return true;       // a goes first
               }
           } else {
               // compare a[i] with a[0..i-1], a can go first only if a[i] > a[i..i-1]
               for (int k = 0; k < i; ++k) {
                   if (a[i] > a[k]) return true;
                   if (a[i] < a[k]) return false;
               }
           }
           return true;       // doesn't matter who goes first, e.g. 77, 777777
       }
       return true;           // doesn't matter who goes first, e.g. 77, 777777
   });

   string ans;
   for (int i = 0; i < s.size(); ++i) {        // corner case - "00"
       if (s[i].front() == '0' && ans.empty() && i != s.size() - 1) continue;
       ans += s[i];
   }
   return ans;
}
```

Ref:

187. Repeated DNA sequences (review)

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "`ACGAATTCCG`". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the **10-letter-long** sequences (substrings) that occur more than once in a DNA molecule.

For example,

`Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT", Return: ["AAAAACCCCC", "CCCCCAAAAA"].`

Solution:
Soln-1: brute force soln is obvious: take substring with length of 10 and put them into hashset.
Time and space complexity is O(10 * n).

Soln-2: rolling hash
Generally speaking hash function is constant time. But the underlying method typically requires to walk through each character, that is linear time. The idea of rolling hash is to reuse the previous calculated hash code as much as possible. For this question, since there are only 4 different characters, 2-bit space is enough to represent. So 10-letter-long sequence, 20-bits is enough. For example: (let A=00, C=01)

```
A   A   A   A   A   C   C  C  C  C     <-- the sequence
00  00  00  00  00  01  01 01 01 01    <-- the hash code
```

To calculate hash code for a new sequence formed by previous one, we just need to shift 2 bit to left and append the new 2-bit.

Further improvement: the above rolling hash intends to reduce the computation for each hash code. If we throw each hash code into a set and it works ok (112ms, beats 68% submissions).
However, we could actually get rid of hashset. since the hash code for each sequence is within 0 ~ 2^20 - 1, we could use bitset to represent all the space. And test/set the correspond bit flag.

```
// 31 / 31 test cases passed. Runtime: 112 ms
vector<string> findRepeatedDnaSequences(string s) {
    vector<string> ans;
    unordered_set<int> s1, s2;

    int hc = 0;
    for (int i = 0; i < s.length() && i < 10; ++i) {
        hc = (hc << 2) | getCode(s[i]);
    }
    s1.insert(hc);

    int mask = (1 << 20) - 1;
    for (int i = 10; i < s.length(); ++i) {
        hc = ((hc << 2) & mask) | getCode(s[i]);

        if (s2.find(hc) != s2.end()) continue;
        if (s1.find(hc) != s1.end()) {
            s2.insert(hc);
            ans.push_back(s.substr(i + 1 - 10, 10));
        } else {
            s1.insert(hc);
        }
    }
    return ans;
}

// 31 / 31 test cases passed. Runtime: 16 ms
vector<string> findRepeatedDnaSequences(string s) {
    vector<string> ans;
    bitset<(1 << 20) - 1> s1;   // strings appear once - space = 2^20 = 1Mb
    bitset<(1 << 20) - 1> s2;   // strings appear more than once

    int hashCode = 0;
    for (int i = 0; i < s.length() && i < 10; ++i) {
        hashCode = (hashCode << 2) | getCode(s[i]);
    }
    s1.set(hashCode);

    // rolling hash the other patterns
    int mask = (1 << 20) - 1;
    for (int i = 10; i < s.length(); ++i) {
        hashCode = ((hashCode << 2) & mask) | getCode(s[i]);

        if (s2.test(hashCode))  continue;
        if (s1.test(hashCode)) {
            s2.set(hashCode);       // set the bit flag
            ans.push_back(s.substr(i + 1 - 10, 10));
        } else {
```

```
            s1.set(hashCode);
        }
    }

    return ans;
}

int getCode(char ch) {
    switch (ch) {
    case 'A': return 0;
    case 'C': return 1;
    case 'G': return 2;
    case 'T': return 3;
    }
    return 0;
}
```

Ref: #28

189. Rotate array

Given array [1, 2, 3, 4, 5, 6, 7] k = 3, return [5, 6, 7, 1, 2, 3, 4]

Solution:
There is a tricky in-place solution for this:

1) reverse(a[0~n-1-k]) => [4, 3, 2, 1, 5, 6, 7]
2) reverse(a[n-k~n-1]) => [4, 3, 2, 1, 7, 6, 5]
3) reverse(a[0~n-1]) => [5, 6, 7, 1, 2, 3, 4]

```
// 189: soln-1: divide-and-conquer
// [1, 2, 3, 4, 5, 6, 7], k = 3
// [5, 6, 7, 4, 1, 2, 3]  <= 1st round: get k nums in position
//           ~~~~~~~~~~   now we have n-k to rotate k, problem size reduced.
// so what is the exit condition?
// n keeps reducing till n <= k, if k % n == 0, there is not need to rotate.
void rotate(int nums[], int n, int k) {
   k %= n;
   if (k) {
       for (int i = 0; i < k; ++i) swap(nums[i], nums[n - k + i]);
       rotate(nums + k, n - k, k);
   }
}
void rotate(vector<int>& nums, int k) {
   rotate(nums.data(), nums.size(), k);
}
```

Ref: #61

190. Reverse bits

Follow up: If this function is called many times, how would you optimize it?

Solution:

1. hashmap/cache would **hardly** improve the performance

```
uint32_t reverseBits(uint32_t n) {
   uint32_t ans = 0;
   for (int i = 0; i < 32; ++i, n >>= 1) {
       ans <<= 1;
       ans |= n & 1;
   }
   return ans;
}
```

Ref:

[191. Number of 1 bits]

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation `00000000000000000000000000001011`, so the function should return 3.

Solution:
To drop lowest bit: n &= n-1.

#201 - bitwise AND of numbers range

```
int hammingWeight(uint32_t n) {
    int ans = 0;
    for (NULL; n; n &= n - 1) ++ans;
    return ans;
}
```

Ref: #342 #338

198/213/337/968. House robber/II/III/Binary tree cameras

198 - You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

#213 - what if houses are arranged as circle?

#337 - what if houses are arranged as tree?

968 - Given a binary tree, we install cameras on the nodes of the tree. Each camera at a node can monitor its parent, itself, and its immediate children. Calculate the minimum number of cameras needed to monitor all nodes of the tree.

```
// 198 - son-1: dynamic programming
// f(n) = max(a[n] + f(n-2), f(n-1))
int rob(vector<int>& nums) {
   if (nums.size() == 0) return 0;
   int cur = nums[0];
   for (int pre1 = nums[0], pre2 = 0, i = 1; i < nums.size(); ++i) {
       cur = max(pre2 + nums[i], pre1);
       pre2 = pre1, pre1 = cur;
   }
   return cur;
}
```

```
// 213 - soln-1: dynamic programming
// f(n) = max(f(0...n-1), f(1, n))
int h(vector<int>& nums, int l, int h) {
   int pre = 0, cur = 0;
   for (int i = l; i <= h; ++i) {
       int t = max(pre + nums[i], cur);
       pre = cur, cur = t;
   }
   return cur;
}
int rob(vector<int>& nums) {
   int n = nums.size();
```

```
    if (n < 2) return n ? nums[0] : 0;

    return max(h(nums, 0, n - 2), h(nums, 1, n - 1));
}
```

```
// 337 - soln-1: dynamic programming (vertex cover in tree)
// f(n) = max{rob, not-rob}
//     rob: n->val + f(n->left->left) + f(n->left->right) + f(n->right->left) + f(n->right->right)
// not-rob: f(n->left) + f(n->right)
int rob(TreeNode* root, unordered_map<TreeNode*, int>& dp) {
    if (root == NULL) return 0;
    if (dp.find(root) != dp.end()) return dp[root];    // return the cached result

    // rob root and its grandchild
    int robbed = root->val;
    if (root->left) robbed += rob(root->left->left, dp) + rob(root->left->right, dp);
    if (root->right) robbed += rob(root->right->left, dp) + rob(root->right->right, dp);

    // not rob root but its child
    int unrobbed = rob(root->left, dp) + rob(root->right, dp);

    // cache result and return
    return dp[root] = max(robbed, unrobbed);
}
```

```
// 968 - soln-1: greedy + dfs (bottom up)
// This question looks similar to house rob III, but Greedy is straightforward for this one.
// If a node has leaf child(ren), it always better to put camera on it, and its potential parent
// would be covered too.
// Three cases:
//              leaf - need covered by its parent
//  covered_by_child - my parent can be treated as leaf (descendants have been taken care of)
// covered_by_camera - my parent will be covered too.
enum {covered_parent, covered_with_camera, leaf_node};
int helper(TreeNode* n, int& ans) {
    if (nullptr == n) return covered_parent;    // acting like a covered parent node.
    if (nullptr == n->left && nullptr == n->right) return leaf_node;

    int l = helper(n->left, ans), r = helper(n->right, ans);
    if (leaf_node == l || leaf_node == r) {
        ++ans;
        return covered_with_camera;
    } else if (covered_with_camera == l || covered_with_camera == r) {
        return covered_parent;
    }
    return leaf_node;   // l/r both are covered_parent, acting like I am a new leaf.
}

int minCameraCover(TreeNode* root) {
    int ans = 0;
    return helper(root, ans) == leaf_node ? ++ans : ans;
}
```

Ref: [#152](#152) [#628](#628)

199. Binary tree right side view

Given a binary tree, image yourself standing on the right size of it, return the values of the nodes you can see ordered from top to bottom.

```
// soln-1
void helper(TreeNode* n, int h, vector<int>& ans) {
    if (!n) return;
    if (h >= ans.size()) ans.push_back(n->val);
    helper(n->right, h + 1, ans);    // right subtree first
```

```
    helper(n->left, h + 1, ans);
}

// soln-2
vector<int> rightSideView(TreeNode* root) {
    vector<int> ans;
    queue<TreeNode*> q;

    if (root) q.push(root);
    while (!q.empty()) {
        ans.push_back(q.back()->val);

        for (int i = q.size(); i > 0; --i) {
            TreeNode* n = q.front(); q.pop();
            if (n->left) q.push(n->left);
            if (n->right) q.push(n->right);
        }
    }

    return ans;
}
```

Ref:

200/305/827/947. Number of Islands/II/Make larger island/remove stone iff share col/row (review)

```
// 200 - soln-1: dfs
void visit(int x, int y, vector<vector<char>>& grid) {
    if (x < 0 || x >= grid.size() || y < 0 || y >= grid[x].size() || grid[x][y] != '1') return;

    grid[x][y] = 'x';
    visit(x - 1, y, grid), visit(x + 1, y, grid), visit(x, y - 1, grid), visit(x, y + 1, grid);
}
int numIslands(vector<vector<char>>& grid) {
    int ans = 0;
    for (int i = 0; i < grid.size(); ++i) {
        for (int j = 0; j < grid[i].size(); ++j)
        if (grid[i][j] == '1') {
            visit(i, j, grid);
            ++ans;
        }
    }
    return ans;
}
```

```
// 305 - soln-1: union-find (check path compression time complexity)
vector<int> numIslands2(int rows, int cols, vector<pair<int, int>>& positions) {
    UF uf(rows * cols);
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));

    int island = 0;
    vector<int> ans;
    vector<pair<int, int>> dirs{ {1, 0}, {-1, 0}, {0, 1}, {0, -1} };
    for (auto& pos : positions) {
        int x = pos.first, y = pos.second;
        visited[x][y] = true, ++island;

        for (auto dir : dirs) {
            int nx = x + dir.first, ny = y + dir.second;
            if (nx < 0 || nx >= rows || ny < 0 || ny >= cols || !visited[nx][ny])   continue;
            int p = uf.find(x * cols + y), q = uf.find(nx * cols + ny);
            if (p != q) {
```

```
                --island, uf.unite(p, q);          // form a bigger land
            }
        }
        ans.push_back(island);
    }
    return ans;
}
```

```
// 827 - making large island
// soln-1: dfs/bfs
// 1. dfs identify each island and mark the island size
// 2. check 4 neighbors for each water area.
void dfs(vector<vector<int>>& grid, int x, int y, vector<vector<int>>& visited, vector<pair<int, int>>& ans) {
   if (x < 0 || x >= grid.size() || y < 0 || y >= grid[0].size() || visited[x][y] || !grid[x][y]) return;
   visited[x][y] = true, ans.push_back({x, y});
   dfs(grid, x + 1, y, visited, ans), dfs(grid, x, y + 1, visited, ans);
   dfs(grid, x - 1, y, visited, ans), dfs(grid, x, y - 1, visited, ans);
}

int largestIsland(vector<vector<int>>& grid) {
   int row = grid.size(), col = grid[0].size(), ans = 0, mask = (1 << 20) - 1;
   vector<vector<int>> visited(row, vector<int>(col));
   for (int i = 0, k = 1; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           if (grid[i][j] && 0 == visited[i][j]) {
               vector<pair<int, int>> island;
               dfs(grid, i, j, visited, island);
               ++k;
               for (auto& p : island) {
                   visited[p.first][p.second] = (k << 20 | island.size());
                   ans = max(ans, (int)island.size());
               }
           }
       }
   }
   for (int i = 0; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           if (0 == grid[i][j]) {
               int combined = 1;
               unordered_set<int> seen;
               for (auto& dir : vector<pair<int, int>>{{0, 1}, {0, -1}, {1, 0}, {-1, 0}}) {
                   int x = i + dir.first, y = j + dir.second;
                   if (x < 0 || x >= row || y < 0 || y >= col || 0 == grid[x][y]) continue;
                   if (seen.find(visited[x][y] & ~mask) != seen.end()) continue;
                   combined += visited[x][y] & mask, seen.insert(visited[x][y] & ~mask);
               }
               ans = max(ans, combined);
           }
       }
   }
   return ans;
}
```

```
// 947 - can remove stones iff there is a stone with same row/col
// soln-1: union-find / dfs to count island numbers
// 1. stone in same row/col will be treated as one component
// 2. only one stone in each component won't be able to be removed
// 3. max possible remove = stones - components
// 4. union-find would be tricky: how to group stone if they share row/col?
//     * - * - *    <== all stars in this example belong to one group.
//     . ~ * - .
//     . . . . *
// 5. in fact, treat (x, y) which is one point, as two different value to group them solve the issue:
//     for example, (1, 2), (1, 3), (3, 2), (3, 4), plus col with some N incase of conflicting with row.
//     {1, 1002} => {1, 1002, 1003} => {1, 1002, 1003} => {1, 1002, 1003, 3} => {1, 1002, 1003, 3, 1004}
```

```
int islands = 0;
unordered_map<int, int> uf;
int find(int x) {
    if (uf.find(x) == uf.end()) {
        uf[x] = x, ++islands;
    }
    while (x != uf[x]) {
        uf[x] = uf[uf[x]], x = uf[x];
    }
    return x;
}
void uni(int x, int y) {
    x = find(x), y = find(y);
    if (x != y) uf[x] = y, --islands;
}
int removeStones(vector<vector<int>>& stones) {
    for (auto& s : stones) uni(s[0], 10000 + s[1]);
    return stones.size() - islands;
}
```

```
// 419 - Battleships in a Board
// soln-1: array trick
// dfs can do the job too, but it requires one-pass and no modifying content
int countBattleships(vector<vector<char>>& board) {
    int ans = 0;
    for (int i = 0; i < board.size(); ++i) {
        for (int j = 0; j < board[0].size(); ++j) {
            if (board[i][j] == '.' ) continue;
            if (i - 1 >= 0 && board[i - 1][j] == 'X') continue;   // purely by definition of battleship
            if (j - 1 >= 0 && board[i][j - 1] == 'X') continue;
            ++ans;
        }
    }
    return ans;
}
```

Ref: #130 #261 #286 #305 #323

201. Bitwise AND of numbers range (review)

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

Solution:

1. for any m != n, the last bit is always 0. (must be odd & even or in between)
2. get rid of last bit, until m == n, then shift back.

#421 - max XOR of 2 numbers in array (great bit-by-bit considering trick)

```
// soln-1: last bit always = 0 if m != n (odd & even), then right shift.
int rangeBitwiseAnd(int m, int n) {
   int shift = 0;
   while (m != n) {
       m >>= 1;    n >>= 1;    ++shift;
       if (m == n) break;
   }
   return m << shift;
}
```

Ref: #191

202/258/553. Happy number / Add digits / Optimal division

```
// 202 - determine if it's happy number (square of each digit sums to 1)
// soln-1: brute force with Floyd cycle detection
bool isHappy(int n) {
    int fast = n, slow = n;

    do {
        slow = squareSum(slow), fast = squareSum(squareSum(fast));
    } while (fast != slow);

    return slow == 1;
}

int squareSum(int n) {
    int ans = 0;
    while (n) {
        int d = n % 10;
        ans += d * d, n /= 10;
    }
    return ans;
}
```

```
// 258 - soln-1: math - not interesting
```

```
// 553 - optimal division
// soln-1: math - not interesting
// notice for expression a/b/c/d/..., (b/c/d/...) will always be the min, hence a/(b/c/d/...)
// will result in max value, so we simply add parentheses starting from 2nd element.
string optimalDivision(vector<int>& nums) {
    if (nums.empty()) return "";

    string ans = to_string(nums[0]);
    if (1 == nums.size()) return ans;
    if (2 == nums.size()) return ans + "/" + to_string(nums[1]);
    ans += "/(";
    for (int i = 1; i < nums.size(); ++i) {
        ans += to_string(nums[i]);
        if (i + 1 < nums.size()) ans += "/";
    }
    return ans + ")";
}
```

Ref: #141

203. Remove linked list elements

Remove all elements from a linked list of integers that have value x.

```
ListNode* removeElements(ListNode* head, int x) {
    ListNode dummy(0);
    dummy.next = head;

    for (ListNode* pre = &dummy, *cur = head; cur; cur = cur->next) {
        cur->val == x ? pre->next = cur->next : pre = cur;
    }
    return dummy.next;
}
```

Ref: #27

207/210/269/444/953. Course schedule/Alien dict... Topological sort (review)

```
// 207/210 - Course schedule/II
// soln-1: topological sort
// course schedule III asks max amount of courses one can take which is a greedy + priority queue
```

```
bool canFinish(int numCourses, vector<pair<int, int>>& prerequisites) {
   vector<unordered_set<int>> g(numCourses);
   vector<int> indegree(numCourses);
   for (auto& e : prerequisites) {
       g[e.first].insert(e.second), ++indegree[e.second];
   }

   vector<int> stk;        // find nodes to start with (any container works, not necessarily stack)
   for (int i = 0; i < numCourses; ++i) if (0 == indegree[i]) stk.push_back(i);

   vector<int> seq;        // one of study sequence
   while (!stk.empty()) {
       int n = stk.back(); stk.pop_back();
       seq.push_back(n);

       for (int i : g[n]) if (0 == --indegree[i]) stk.push_back(i);
   }
   return seq.size() == numCourses;
}
```

```
// 444 - Sequence Reconstruction (if there is only 1 way for topological sort)
// soln-1: topological sort
bool sequenceReconstruction(vector<int>& org, vector<vector<int>>& seqs) {
   unordered_map<int, int> indeg;              // keep in-degree for each node
   unordered_map<int, unordered_set<int>> g;
   for (auto& seq : seqs) {
       for (auto i = 1; i < seq.size(); ++i) {
           auto fr = seq[i - 1], to = seq[i];
           indeg[fr], indeg[to];               // make sure every node is in indegree
           if (g[fr].find(to) == g[fr].end()) g[fr].insert(to), ++indeg[to];
       }
   }
   queue<int> q;
   for (auto& kv : indeg) if (0 == kv.second) q.push(kv.first);
   if (q.size() > 1) return false;             // more than 1 way for topo sort

   int i = 0;
   while (!q.empty()) {
       auto n = q.front();     q.pop();
       if (n != org[i++]) return false;

       bool found_next = false;
       for (auto nxt : g[n]) {
           if (--indeg[nxt] == 0) {
               if (found_next) return false;   // more ways for topo sort
               q.push(nxt), found_next = true;
           }
       }
   }
   return i == org.size();
}
```

```
// 269 - alien dictionary (given ordered words, determine the order of letters)
// soln-1: topological sort
typedef unordered_map<int, unordered_set<int>> t_graph;
string alienOrder(vector<string>& words) {
    t_graph g;
    vector<int> indeg(26, -1);
    buildGraph(words, g, indeg);
    return topoSort(g, indeg);
}

// check char order from 2 adjacent words
void buildGraph(vector<string>& words, t_graph& g, vector<int>& indeg) {
    if (words.size() == 0) return;
    for (char ch : words[0]) if (indeg[ch - 'a'] == -1) indeg[ch - 'a'] = 0;  // make every char recorded
```

```
    for (int i = 1; i < words.size(); ++i) {
        string& w1 = words[i - 1], &w2 = words[i];

        for (char ch : w2) if (indeg[ch - 'a'] == -1) indeg[ch - 'a'] = 0;    // make every char recorded

        for (int j = 0; j < min(w1.length(), w2.length()); ++j) {
            int x = w1[j] - 'a', y = w2[j] - 'a';
            if (x == y) continue;

            if (g[x].find(y) == g[x].end()) g[x].insert(y), ++indeg[y];
            break;    // other part of words shouldn't be checked because it's undecidable.
        }
    }
}

string topoSort(t_graph& g, vector<int>& indeg) {
    string ans;
    vector<int> topo;
    int alphabets = 0;
    for (int i = 0; i < indeg.size(); ++i) {
        if (indeg[i] == 0) topo.push_back(i);
        if (indeg[i] != -1) ++alphabets;
    }

    while (!topo.empty()) {
        int i = topo.back(); topo.pop_back();
        ans.push_back(i + 'a');

        for (int j : g[i]) {
            if (--indeg[j] == 0) topo.push_back(j);
        }
    }

    return (ans.length() == alphabets) ? ans : "";    // in case of no topological order
}
```

```
// 953 - verify alien dictionary
// soln-1: brute-force
// return 1, if s1 > s2
//        0, if s1 == s2
//       -1, if s1 < s2
int cmp(string& s1, string& s2, vector<int>& order) {
   for (int i = 0; i < min(s1.length(), s2.length()); ++i) {
       if (order[s1[i] - 'a'] > order[s2[i] - 'a']) return 1;
       if (order[s1[i] - 'a'] < order[s2[i] - 'a']) return -1;
   }
   if (s1.length() == s2.length()) return 0;
   return s1.length() < s2.length() ? -1 : 1;
}
bool isAlienSorted(vector<string>& words, string order) {
   vector<int> idx(26);
   int i = 0;
   for (auto ch : order) idx[ch - 'a'] = i++;

   for (int i = 1; i < words.size(); ++i) {
       if (cmp(words[i-1], words[i], idx) == 1) return false;
   }
   return true;
}
```

Ref: #269

208/211/648/676/677/745. trie

#211 - add/search word
#212 - word search II (make trie for given word list)

```
// 208 - trie applications
// autocomplete dictionary (question #211 asks to design a word dictionary data structure)
```

```cpp
// approximate matching algorithms, such as spell checking, find phone number
// prefix matching, such as routing table matching
struct TrieNode {
    string word;        // empty if node is not indicating a full word.
    TrieNode* link[26]; // faster than use vector (no twice heap memory allocation)
    TrieNode() { memset(link, 0, sizeof(link)); }
};

class Trie {
    TrieNode* m_root;
public:
    Trie() : m_root(new TrieNode) { }

    void insert(string word) {
        TrieNode* node = m_root;
        for (char ch : word) {
            if (!node->link[ch - 'a']) node->link[ch - 'a'] = new TrieNode;
            node = node->link[ch - 'a'];
        }
        node->word = word;
    }

    bool search(string word) {
        TrieNode* node = m_root;
        for (char ch : word) {
            if (!node->link[ch - 'a']) return false;
            node = node->link[ch - 'a'];
        }
        return node->word == word;
    }

    bool startsWith(string prefix) {    // no wildcard support
        TrieNode* node = m_root;
        for (char ch : prefix) {
            if (node->link[ch - 'a'] == NULL) return false;
            node = node->link[ch - 'a'];
        }
        return true;
    }
};
```

```cpp
// 211 - Add and Search Word - Data structure design
class WordDictionary {
    TrieNode _root;
public:
    bool search(string& word, int start, TrieNode* node) {
        if (nullptr == node) return false;
        if (start >= word.length()) return !node->word.empty();          // support '.'
        if ('.' == word[start]) {
            for (int i = 0; i < 26; ++i) {
                if (search(word, start + 1, node->v[i])) return true;    // dfs
            }
            return false;
        }
        return search(word, start + 1, node->v[word[start] - 'a']);
    }
};
```

```cpp
// 648 - Replace Words
string replaceWords(vector<string>& dict, string sentence) {
    auto trie = build(dict);
    istringstream iss(sentence);
    string ans, word;
    while (iss >> word) {
```

```
        if (!ans.empty()) ans += " ";
        ans += search(trie, word);
    }
    return ans;
}
TrieNode* build(vector<string>& dict) {
    auto root = new TrieNode;
    for (auto& w : dict) {
        auto node = root;
        for (auto& x : w) {
            if (!node->v[x - 'a']) node->v[x - 'a'] = new TrieNode;
            node = node->v[x - 'a'];
        }
        node->word = w;
    }
    return root;
}
string search(TrieNode* node, string& word) {
    for (auto& x : word) {
        if (nullptr == node->v[x - 'a']) return word;

        node = node->v[x - 'a'];
        if (!node->word.empty()) return node->word;    // return shortest root
    }
    return word;
}
```

```
// 745 - Prefix and Suffix Search (design data structure good for searching prefix and suffix, review #642)
// soln-1: trie                                         <<< space efficient
// 1. build trie with words
// 2. find prefix first, then dfs all leafs under this node and check suffix
// soln-2: trie but insert all combinations of words    <<< time efficient
// 1. build trie with 27 length, give one more space next to 'z', which is '{'
//    use suffix+'{'+prefix, for example, given "apple", insert following:
//    apple{apple, pple{apple, ple{apple, le{apple, e{apple, {apple
// 2. don't need 'word' property anymore.
class WordFilter {
   TrieNode _root;
public:
   WordFilter(vector<string>& words) {
       int i = 0;
       for (auto& word : words) {
           auto node = &_root;
           for (auto& x : word) {
               if (!node->v[x - 'a']) node->v[x - 'a'] = new TrieNode;
               node = node->v[x - 'a'];
           }
           node->word = word, node->weight = i++;
       }
   }

   // all leafs under current node would have the matched prefix, then check it's suffix
   int f(string prefix, string suffix) {
       auto node = &_root;
       for (auto& x : prefix) {
           if (nullptr == node->v[x - 'a']) return -1;
           node = node->v[x - 'a'];
       }
       int ans = -1;
       dfs(node, suffix, ans);
       return ans;
   }

   void dfs(TrieNode* node, const string& suffix, int& weight) {
       if (nullptr == node) return;
       for (int i = 0; i < 26; ++i) {
```

```
            if (!node->word.empty() && node->word.length() >= suffix.length()) {
                bool match = true;
                for (int m = node->word.length(), n = suffix.length(), i = 1; i <= n; ++i) {
                    if (node->word[m - i] != suffix[n - i]) match = false;
                }
                if (match) weight = max(weight, node->weight);
            }
            if (node->v[i]) dfs(node->v[i], suffix, weight);
        }
    }
};
```

Ref: #211 #212

214. Shortest palindrome

Given a string S, you are allowed to convert it to a palindrome **by adding characters in front of it**. Find and return the shortest palindrome you can find by performing this transformation. For example:

Given `"aacecaaa"`, return `"aaacecaaa"`.

Given `"abcd"`, return `"dcbabcd"`.

Solution:
soln-1: reuse palindrome as possible

1. find longest palindrome (similar to #5) starting from 0, because we only allow to add chars from head
2. then reverse the left part and add it in front of head
3. if allow to append chars only, we need to find palindrome from tail.

soln-2: recursion (hard to understand)

```
// soln-1: find longest palindrome substring from 0
string shortestPalindrome(string s) {
   int len = 0;                             // length of palindrome starting from 0
   for (int i = 0; i < s.length(); nullptr) {
       int l = i;
       while (s[l] == s[i]) ++i;
       int r = i - 1;
       while (l - 1 >= 0 && r + 1 < s.length() && s[l - 1] == s[r + 1]) --l, ++r;

       if (0 == l && r - l + 1 > len) len = r - l + 1;
   }
   if (len == s.length()) return s;

   string ss = s.substr(len);
   return string(ss.rbegin(), ss.rend()) + s;
}

// soln-2: hard to understand
string shortestPalindrome(string s) {
   int j = 0;
   for (int i = s.length() - 1; i >= 0; --i) {
       if (s[j] == s[i])  ++j;
   }

   if (j == s.length())  return s;

   string suffix = s.substr(j);
   return string(suffix.rbegin(), suffix.rend()) + shortestPalindrome(s.substr(0, j)) + suffix;
}
```

Ref: #5 #336

215/280/324/358/621/767. kth largest/Wiggle sort/II/Reorganize string (review)

```
// 215 - kth largest element in array
// soln-1: quick-select for kth element in sorted order (kth smallest only if all distinct numbers)
int kthElem(int nums[], int len, int k) {
   if (k < 0 || k > len) return INT_MAX;

   int kth = partition(nums, len);
   if (kth == k) return nums[k - 1];
   if (kth > k) return kthElem(nums, kth - 1, k);
   return kthElem(nums + kth, len - kth, k - kth);
}
// Lomuto partition scheme - return kth element in sorted order (slow but easy to understand)
int partition(int nums[], int len) {
   int low = 0, m = nums[len - 1];
   for (int i = 0; i < len - 1; ++i) {
       if (nums[i] < m) swap(nums[low++], nums[i]);
   }
   swap(nums[low++], nums[len - 1]);

   return low;
}
```

```
// 280 - wiggle sort (given unsorted, no dup nums, make it /\/\/\ shape)
// soln-1: greedy
void wiggleSort(vector<int> &nums) {
   bool asc = true;
   for (int i = 0; i + 1 < nums.size(); ++i) {
       if ((asc && nums[i] > nums[i + 1]) || (!asc && nums[i] < nums[i + 1])) {
           swap(nums[i], nums[i + 1]);
       }
       asc = !asc;
   }
}
```

```
// 324 - wiggle sort (input with dups)
// soln-1: greedy (partition by median, then pick backwards)
void wiggleSort(vector<int>& nums) {
    if (nums.size() < 2) return;

    int n = nums.size();
    nth_element(nums.begin(), nums.begin() + n / 2, nums.end());

    // 3-way-dutch-flag partition: (<=m).m.(>m)
    for (int m = nums[n / 2], i = 0, l = 0, r = n - 1; i <= r; ++i) {
        if (nums[i] > m) swap(nums[i], nums[r--]), --i;
        else if (nums[i] < m) swap(nums[i], nums[l++]);
    }

    vector<int> ans(n);
    for (int i = 1, j = n - 1; j >= 0; --j) {
        ans[i] = nums[j];
        (i + 2 >= n) ? i = 0 : i += 2;
    }
    nums = ans;
}
```

```
// 767 - Reorganize String (make adjacent different)
// soln-1: greedy (sort by frequency, then pick from i = 0, j = (n-1) / 2 + 1)
string reorganizeString(string S) {
   vector<int> cnt(26);
   for (auto ch : S) cnt[ch - 'a']++;
   priority_queue<pair<int, char>> pq;
   for (int i = 0; i < cnt.size(); ++i) {
       if (cnt[i]) pq.push({cnt[i], i + 'a'});
```

```
    }
    string buf, ans;
    while (!pq.empty()) buf += string(pq.top().first, pq.top().second), pq.pop();
    int mid = buf.size() / 2 + 1;
    if (buf[0] == buf[mid]) return "";

    for (int i = 0, j = mid; i < mid; ++i, ++j) {
        ans.push_back(buf[i]);
        if (j < buf.length()) ans.push_back(buf[j]);
    }
    return ans;
}
```

```
// 358 - Rearrange String k Distance Apart (same char at least k distance)
// soln-1: greedy (sort by frequency, then use (k+1) pointers to rearrange)
// this is a general case for #767, which the distance k = 1.
string rearrangeString(string str, int k) {
    vector<int> cnt(26);
    for (auto ch : str) cnt[ch - 'a']++;
    priority_queue<pair<int, char>> pq;
    for (int i = 0; i < cnt.size(); ++i) {
        if (cnt[i]) pq.push({cnt[i], i + 'a'});
    }
    string ans;
    while (!pq.empty()) {
        vector<pair<int, char>> tmp;
        for (int i = 0, left = str.length() - ans.length(); i < min(k, left); ++i) {
            if (pq.empty()) return "";
            auto t = pq.top();  pq.pop();
            ans.push_back(t.second);
            if (--t.first > 0) tmp.push_back(t);    // for next round
        }
        for (auto& t : tmp) pq.push(t);
    }
    return ans;
}
```

```
// 1054 - Distant Barcodes (so that no two adjacent barcodes are equal)
// soln-1: heap + greedy (use high freq. nums first)
vector<int> rearrangeBarcodes(vector<int>& codes) {
    unordered_map<int, int> m;
    for (auto x : codes) m[x]++;
    set<pair<int, int>> s;        // <freq, val>
    for (auto& kv : m) s.insert({kv.second, kv.first});

    int pos = 0;
    for (auto it = s.rbegin(); it != s.rend(); ++it) {
        for (int k = 0; k < it->first; ++k, pos += 2) {
            if (pos >= codes.size()) pos = 1;
            codes[pos] = it->second;
        }
    }
    return codes;
}
```

```
// 621 - Task Scheduler
// soln-1: greedy (sort by frequency then rearrange task)
// tasks = ["A","A","A","B","B","B"], n = 2
// A -> B -> idle -|-> A -> B -> idle -|-> A -> B
// \-----n+1----/      \-----n+1---/       \----/
int leastInterval(vector<char>& tasks, int n) {
    vector<int> cnt(26);
    int mx = 0, ans = 0;
    for (auto t : tasks) mx = max(mx, ++cnt[t - 'A']);

    ans = (n + 1) * (mx - 1);
```

```
    for (int i = 0; i < cnt.size(); ++i) {
        if (cnt[i] == mx) ans++;        // extra tail part of diagram above
    }
    return max(int(tasks.size()), ans); // in case distance n = 0
}
```

Ref: #4 #75 #148 #215 #347 #355 #373

217/219/220. Contains duplicate/II/III (review)

```
// 217 - contains dup
bool containsDuplicate(vector<int>& nums) {
    return nums.size() > unordered_set<int>(nums.begin(), nums.end()).size();
}
```

```
// 219 - contains dup II (Ai == Aj && |i - j| <= k, absolute diff. at most k)
// soln-1: hashmap
bool containsNearbyDuplicate(vector<int>& nums, int k) {
    unordered_map<int, int> mp;
    for (int i = 0; i < nums.size(); i++) {
        auto iter = mp.find(nums[i]);
        if (iter != mp.end() && i - iter->second <= k) return true;

        mp[nums[i]] = i;
    }
    return false;
}
```

```
// 220 - contains dup III (|Ai - Aj| <= t && |i - j| <= k, abs diff. at most k and distance at most t)
// soln-1: sliding window + bst
// use bst to keep most recent k numbers, then search for x, such that |Ai - x| <= t
bool containsNearbyDuplicate(vector<int>& nums, int k, int t) {
    set<long> window;

    for (int i = 0; i < nums.size(); ++i) {
        auto iter = window.lower_bound((long)nums[i] - t);        // nums[i] - t) <= x <= (nums[i] + t
        if (iter != window.end() && *iter <= (long)nums[i] + t) return true;

        window.insert(nums[i]);
        if (i >= k) window.erase(nums[i - k]);
    }
    return false;
}

// 220 - contains dup III (|Ai - Aj| <= t && |i - j| <= k)
// soln-2: bucket sort
// 1. use k buckets, each with size of t in order for |Ai - Aj| <= t falling into cur/pre/next bucket
// 2. special case: t = 0 => Ai == Aj, same as #219
// 3. add INT_MAX to each num since it could be negative
// O(n) time, O(k) space
bool containsNearbyDuplicate(vector<int>& nums, int k, int t) {
    if (k <= 0 || t < 0) return false;
    if (t == 0) return containsNearbyDuplicate(nums, k);    // special case = #219

    unordered_map<long long, long long> mp;

    for (int i = 0; i < nums.size(); ++i) {
        long long x = (long long)nums[i] + INT_MAX, bucket = x / t;

        auto pre = mp.find(bucket - 1), cur = mp.find(bucket), nxt = mp.find(bucket + 1);
        if (cur != mp.end()                                 // each bucket should only contain 1 element.
            || (pre != mp.end() && pre->second + t >= x)
            || (nxt != mp.end() && x <= nxt->second - t)) {
            return true;
        }

        mp[bucket] = x;                                     // maintain the window size.
        if (i >= k) mp.erase(((long long)nums[i - k] + INT_MAX) / t);
    }
    return false;
}
```

Ref:

218/223/836/939. Skyline/Rectangle area/overlap … Geometry (review)

```
// 218 - skyline problem
// soln-1: sweep line using BST with enter/leave event in O(nlgn) time
// 1. we are tracking the points when height changes.
// 2. when hitting wall, add height into bst (to get highest wall).
// 3. when leaving wall, remove its height from bst.
// 4. when there is a height change, it’s the point we are looking for.
vector<pair<int, int>> getSkyline(vector<vector<int>>& bldgs) {
   vector<pair<int, int>> wall;
   for (auto& b : bldgs) {
       wall.push_back({b[0], -b[2]}),  wall.push_back({b[1], b[2]});
   }
   // nice trick to sort by pos in ascending order,
   // if pos are same, entering first, we need the height of entering wall.
   // if both entering, add the higher wall 1st (we are tracking height change, and higher should be seen 1st).
   // if both leaving, remove the shorter wall 1st.
   sort(wall.begin(), wall.end());

   multiset<int> h;                            // buildings' BST by height
   h.insert(0);                                // force to compute for last leaving case
   int curH = 0, preH = 0;                     // current/previous height

   vector<pair<int, int>> ans;
   for (auto& w : wall) {
       if (w.second < 0) h.insert(-w.second);  // for entering case
       else h.erase(h.find(w.second));         // for leaving case: erase only 1 height!

       // essentially we are chasing height changes between BST and ans.
       curH = *h.rbegin();                     // highest in BST
       if (curH != preH) {                     // tracking height change
           ans.push_back({w.first, curH}), preH = curH;
       }
   }
   return ans;
}
```

```
// 223/836 - rectangle area/overlap
// soln-1: minus the overlap
int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
   int ans = (C - A) * (D - B) + (G - E) * (H - F);
   if (!(E >= C || G <= A || H <= B || F >= D)) {    // has overlap, then minus the overlap
       int l = max(A, E), r = min(C, G), b = max(B, F), u = min(D, H);
       ans -= (r - l) * (u - b);
   }
   return ans;
}
```

```
// 836 - rectangle overlap
// soln-1: no overlap if any of the following is true
bool isRectangleOverlap(vector<int>& rec1, vector<int>& rec2) {
   return !(rec2[0] >= rec1[2] ||   // rec2's left >= rec1's right
            rec2[1] >= rec1[3] ||   // rec2's bottom >= rec1's top
            rec2[2] <= rec1[0] ||   // rec2's right <= rec1's left
            rec2[3] <= rec1[1]      // rec2's top <= rec1's bottom
           );
}
```

```
// 850 - Rectangle Area II (REVIEW)
// soln-1: sweep line
```

```
// 1. group points base on x-axis and mark each point as following:
// +(-1)------+(1)    when go right, count will be 0 if end of box
// |          |       when go up, count will be 0 too if end of box
// +(1)-------+(-1)
// 2. the key is to update Y, review it (very tricky).
int rectangleArea(vector<vector<int>>& rectangles) {
   long ans = 0, mod = 1e9+7;
   map<int, vector<pair<int, int>>> lines;     // group pts: <X, pts>
   for (auto& r : rectangles) {
       lines[r[0]].push_back({r[1], 1}), lines[r[2]].push_back({r[1], -1});
       lines[r[0]].push_back({r[3], -1}), lines[r[2]].push_back({r[3], 1});
   }
   map<int, int> h;                            // horizon lines: <Y, count>
   long preY = 0, preX = 0, count = 0, start = 0;
   for (auto& l : lines) {
       ans += (l.first - preX) * preY % mod;
       for (auto& pt : l.second) {
           h[pt.first] += pt.second;           // update start/end of a block
       }
       preY = 0, count = 0;
       for (auto& c : h) {
           if (count == 0) start = c.first;    // mark next block as a start
           count += c.second;
           if (count == 0) preY += (c.first - start);  // merge Y, and next block will be marked as start
       }
       preX = l.first;
   }
   return ans % mod;
}
```

```
// 939 - Minimum Area Rectangle
// soln-1: hashmap
// 1. fix 2 points (1 line)
// 2. check if other 2 points exist
int minAreaRect(vector<vector<int>>& points) {
   int ans = INT_MAX;
   unordered_map<int, unordered_set<int>> mp;
   for (auto& p : points) mp[p[0]].insert(p[1]);
   for (auto& p1 : points) {
       for (auto& p2 : points) {
           if (p1[0] != p2[0] && p1[1] != p2[1]) {
               auto it1 = mp.find(p1[0]), it2 = mp.find(p2[0]);
               if (it1 != mp.end() && it2 != mp.end() &&
                   it1->second.find(p2[1]) != it1->second.end() &&     // check if can form a rectangle
                   it2->second.find(p1[1]) != it1->second.end()) {
                   ans = min(ans, abs(p1[0] - p2[0]) * abs(p1[1] - p2[1]));
               }
           }
       }
   }
   return INT_MAX == ans ? 0 : ans;
}
```

```
// 963 - Minimum Area Rectangle II (given a set of points)
// soln-1: math + hashmap
// 1. brute-force will take O(n^3). we can group lines by its center point.
// 2. for lines on the same central point, then we check each pair for rectangle.
double minAreaFreeRect(vector<vector<int>>& pts) {
    unordered_map<unsigned long , vector<vector<int>>> mp;
    for (int i = 0; i < pts.size(); ++i) {
        for (int j = i + 1; j < pts.size(); ++j) {
            // instead of using x2-x1|y2-y1 as center, its same if we use x2+x1|y2+y1.
            // image a circle, and points around the edge.
            // these 2 lines must have same length.
            auto center = (((unsigned long)pts[i][0] + pts[j][0]) << 20) + (pts[i][1] + pts[j][1]);
            mp[center].push_back({pts[i][0], pts[i][1], pts[j][0], pts[j][1]});
```

```
        }
    }
    unsigned long ans = -1;
    for (auto& kv : mp) {
        for (int i = 0; i < kv.second.size(); ++i) {
            auto& l1 = kv.second[i];
            auto d1 = dist(l1[0], l1[1], l1[2], l1[3]);
            for (int j = i + 1; j < kv.second.size(); ++j) {
                auto& l2 = kv.second[j];
                auto bottom = dist(l1[0], l1[1], l2[2], l2[3]),
                     height = dist(l2[2], l2[3], l1[2], l1[3]);
                if (d1 != bottom + height) continue;
                if (-1 == ans || ans > bottom * height) ans = bottom * height;
            }
        }
    }
    return -1 == ans ? 0 : sqrt(ans);
}
unsigned long dist(int x1, int y1, int x2, int y2) {
    return (unsigned long)(x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
}
```

```
// line-segment intersection: given 2 line-segments, check if they intersect
// soln: check orientation
// 1. orientation
// angle(p, q) = (q.y - p.y) / (q.x - p.x)
// angle(q, r) = (r.y - q.y) / (r.x - q.x)
// if angle(p, q) < angle(q, r), then it's turn-left (CCW)
// 2. intersect iff one of the following case is true
//    -  (p1,q1,p2) and (p1,q1,q2) have different orientations and
//       (p2,q2,p1) and (p2,q2,q1) have different orientations
//    -  (p1,q1,p2), (p1,q1,q2), (p2,q2,p1), and (p2,q2,q1) are all collinear and
//       and (p1 is on p1q1) or (q2 is on p1q1) or (p1 is on p2q2) or (q1 is on p2q2)
//
enum {COLINEAR, CW, CCW};   // colinear, clockwise, count-clockwise
int orientation(pair<int, int>& p, pair<int, int>& q, pair<int, int>& r) {
   auto v = (q.second - p.second) * (r.first - q.first) - (q.first - p.first) * (r.second - q.second);
   if (0 == v) return COLINEAR;
   return v > 0 ? CW : CCW;
}
```

```
// simple closed path: given a set of points in plane, return the smallest dots that form the convex hall.
// soln: Graham Scan
// 1. choose the bottom dot p0, and sort the rest based on angle formed with p0 ().
// 2. pickup the most 3 pts from result stack and check orientation.
//    - if turn left, keep it, and continue
//    - if turn right, discard the top of result stack.
vector<pair<int, int>> convex(vector<pair<int, int>>& points) {
   int p0 = 0;
   for (int i = 1; i < points.size(); ++i) {
       if (points[i].second < points[p0].second) p0 = i;      // find the lowest point
   }
   swap(points[0], points[p0]);
   sort(points.begin() + 1, points.end(), [&points](pair<int, int>& a, pair<int, int>& b){
       auto d = orientation(points[0], a, b);
       if (COLINEAR == d) {
           auto d1 = (points[0].first - a.first) * (points[0].second - a.second),
                d2 = (points[0].first - b.first) * (points[0].second - b.second);
           return d1 < d2;      // shortest distance first
       }
       return d == CCW;         // smaller angle with p0 first
   });
   vector<pair<int, int>> ans;
   ans.push_back(points[0]), ans.push_back(points[1]), ans.push_back(points[2]);
   for (int i = 3; i < points.size(); ++i) {
       while (CCW != orientation(ans[ans.size() - 2], ans.back(), points[i])) ans.pop_back();
```

```
      ans.push_back(points[i]);
   }
   return ans;
}
```

```
// line-segment intersection II: given n line-segments, return intersections
// soln: sweep line
// 1. sort line by x-axis
// 2. when see line.left, insert into BST based on y
// 3. when see line.right, check intersection with above and below neighbors, and remove this line.
int lineIntersection(vector<vector<int>>& lines) {
   int ans = 0;
   // TODO
   return ans;
}
```

```
// point inclusion: given a polygon and point(s), is the point inside or outside the polygon.
// soln: even-odd rule algorithm (wiki - Point in polygon)
// 1. Draw a horizontal line to the right of each point and extend it to infinity
// 2. Count the number of times a line intersects the polygon
//    - even number ⇒ point is outside
//    - odd number ⇒ point is inside
// 3. results may be incorrect if the point lies very close to that boundary, because of rounding errors
//    This is not normally a concern, as speed is much more important than complete accuracy in most
//    applications of computer graphics. The issue is solved as follows:
// 4. If the intersection point is a vertex of a tested polygon side, then the intersection counts only if
//    the second vertex of the side lies below the ray.
```

```
// Closest Pair of Points: given a set of points, return the distance of closest pair
// soln-1: Divide-and-Conquer
// 1. sort points by x coordinate
// 2. Split points into left half and right half
// 3. Recurse on each half: find closest pair => ans
// 4. Between two halves: Only interested in pairs with distance < ans
//    - for each left point, only max 3 points needs to check, because
//    - 1. left half of circle must be empty
//    - 2. right side has at most 3 points with distance > ans
//    - 3. merge two sorted array to update ans.
// ref: http://courses.csail.mit.edu/6.006/spring11/lectures/lec24.pdf
int closetPairDistance(vector<vector<int>>& points) {
   int ans = 0;
   return ans;
}
```

Ref: #307 #308 #315

222/919. Count complete binary tree nodes/Complete binary tree inserter

```
// 222 - count complete binary tree nodes
// soln-1: binary search
// if left subtree height = right subtree height, it means left subtree is full tree, else right subtree is full
int countNodes(TreeNode* root) {
    if (root == NULL) return 0;

    int left = height(root->left), right = height(root->right);
    if (left == right) {
        return 1 + (1 << (left + 1) - 1) + countNodes(root->right);
    }

    // left must greater than right
    return 1 + (1 << (right + 1) - 1) + count(root->left);
}

int height(TreeNode* root) {
    return !root  ? 0 : 1 + height(root->left);
```

```
}
```

```
// 919 - complete binary tree inserter
class CBTInserter {
    vector<TreeNode*> _v;
public:
    CBTInserter(TreeNode* root) {
        _v.push_back(root);        // instead of using Q, simple vector should work
        for (int i = 0; i < _v.size(); ++i) {
            if (_v[i]->left)  _v.push_back(_v[i]->left);
            if (_v[i]->right) _v.push_back(_v[i]->right);
        }
    }

    int insert(int v) {
        _v.push_back(new TreeNode(v));
        int parent = (_v.size() - 2) / 2;
        _v[parent]->left ? _v[parent]->right = _v.back() : _v[parent]->left = _v.back();
        return _v[parent]->val;
    }

    TreeNode* get_root() {
        if (_v.empty()) return nullptr;
        return _v[0];
    }
};
```

Ref:

224/227/770/772. Basic calculator/II/III/IV

224/227/770 - consider +, - and (), consider +, -, *, /, consider +, -, *, / and ().

772 - consider eval variables, example:

```
input: expression = "e - 8 + temperature - pressure",evalvars = ["e", "temperature"], evalints = [1, 12]
output: ["-1*pressure","5"]
```

```
// 224/227/770 - basic calculator
// soln-1: recursion
// For example, 1 + 2 * 3 / 4
//                 ~~  ^^^
//     (preop, preval)  - pointing to '+', '2'
//     (curop, curval)  - cur-op pointing to '*', '3'
// calculate what we have when we have 2 op:
// 1. if curop > preop, curval = curval-curop-preval and reset curop
// 2. else preval = ans-preop-preval and preset preop = curop, curop = none
int eval(stack<int>& nums, stack<char>& op) {
   auto b = nums.top();    nums.pop();
   auto a = nums.top();    nums.pop();
   auto oper = op.top();   op.pop();
   switch (oper) {
       case '+':   return a + b;
       case '-':   return a - b;
       case '*':   return a * b;
       case '/':   return a / b;
   }
   return 0;
}
int prio(int op) {
   switch (op) {
       case '*':   return 4;
       case '/':   return 4;
       case '+':   return 2;
```

```
        case '-':   return 2;
    }
    return 0;
}

int calculate(string s) {
    stack<int> nums; stack<char> op;
    for (int i = 0; i < s.length(); ++i) {
        if ('+' == s[i] || '-' == s[i] || '*' == s[i] || '/' == s[i]) {
            while (!op.empty() && prio(s[i]) <= prio(op.top())) nums.push(eval(nums, op));
            if (nums.empty()) nums.push(0);         // corner case: -1 + 2
            op.push(s[i]);
        } else if ('(' == s[i]) {
            int j = i + 1, p = 1;                   // find the other pair and recursion
            while (j < s.length() && p) {
                if (s[j] == '(') ++p;
                else if (s[j] == ')') --p;
                ++j;
            }
            nums.push(calculate(s.substr(i + 1, j - (i + 1) - 1))), i = --j;
        } else if (isdigit(s[i])) {                 // fetch digits
            int j = i;
            while (j < s.length()) {
                if (!isdigit(s[j])) break;
                ++j;
            }
            nums.push(stoi(s.substr(i, j - i))), i = --j;
        }
    }
    while (!op.empty()) nums.push(eval(nums, op));
    return nums.top();
}
```

```
// 772 - basic calculator IV
// TODO
```

Ref:

225/232. Implement stack/queue using queue/stack

```
// for stack using queue, let's rotate q after pushing so we can use only 1 queue
void push(int x) {
    _q.push(x);
    // rotate q
    for (int i = 1; i < _q.size(); ++i) {
        _q.push(_q.front());
        _q.pop();
    }
}
```

Ref:

231/326/342. Power of two/three/four

Given an integer, write a function to determine if it is a power of two.

```
// 231 - only one bit set at any position
bool isPowerOfTwo(int n) {
    if (n <= 0) return false;
    return 0 == (n & (n-1));
}
```

```
// 326 - soln-1: brute-force recursion
bool isPowerofThree(int n) {
```

```
    if (0 == n) return false;
    if (1 == n) return true;
    return (n % 3 == 0) && isPowerofThree(n / 3);
}
```

```
// 342 - soln-1: brute-force with loop
bool isPowerOfFour(int num) {
   while (num && num % 4 == 0) num /= 4;
   return num == 1;
}

// 342 - soln-2: power of 2 (only 1 bit set) and must be set at odd position
bool isPowerOfFour(int n) {
    return 0 == (n & (n-1)) && 0 != (n & 0x55555555);
}
```

Ref: #342 #326

234. Palindrome linked list

#143 - re-order linked list (use n/2 stack implicitly)

```
// soln-1: recursive soln: O(n) space
bool isPalindrome(ListNode* head) {
    if (head == NULL || head->next == NULL) return true;

    ListNode* slow = head, *fast = head;
    while (fast->next && fast->next->next) slow = slow->next, fast = fast->next->next;

    return helper(head, slow->next);       // starting from head also works, just wasted half stack space.
}

bool helper(ListNode*& head, ListNode* cur) {
    if (!cur) return true;                 // case like only one node
    if (! helper(head, cur->next)) return false;

    bool ret = (cur->val == head->val);
    head = head->next;

    return ret;
}

// soln-2: iterative (list destroyed)
bool isPalindrome(ListNode* head) {
   if (head == nullptr) return true;

   // split list
   ListNode* slow = head, *fast = head->next;
   while (fast && fast->next) slow = slow->next, fast = fast->next->next;

   ListNode* h1 = head, *h2 = slow->next;
   // reverse list
   slow->next = nullptr;
   ListNode* cur = h2, *pre = nullptr;
   while (cur) {
       ListNode* next = cur->next;
       cur->next = pre, pre = cur, cur = next;
   }
   if (h2) h2->next = nullptr;

   // cmp list
   for (h2 = pre; h1 && h2; h1 = h1->next, h2 = h2->next) {
```

```
        if (h1->val != h2->val) return false;
    }
    return true;
}
```

Ref: #9 #143

235/236/450/776. LCA of bst/Delete a node in bst/Split bst (review)

```
// 235 - soln-1: p/q's LCA is n iff p/q is on different side of n.
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root || !p || !q) return nullptr;
    while (root) {
        if (root->val < q->val && root->val < p->val) root = root->right;
        else if (root->val > q->val && root->val > p->val) root = root->left; else break;
    }
    return root;
}
```

```
// 236 - soln-1: LCA for binary tree in O(n) time
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (!root || p == root || q == root) return root;   // found p or q

    // find p and q from different subtree
    auto lacFromLeft = lowestCommonAncestor(root->left, p, q);
    auto lacFromRight = lowestCommonAncestor(root->right, p, q);

    // lca found in left and right subtree respectively
    if (lacFromLeft && lacFromRight) return root;

    // lca found either in left or right subtree
    return lacFromLeft ? lacFromLeft : lacFromRight;
}
```

```
// 450 - soln-1: swap success with target, then delete target in right subtree recursively
TreeNode* deleteNode(TreeNode* root, int key) {
    if (!root) return root;

    if (root->val > key) {
        root->left = deleteNode(root->left, key);
    } else if (root->val < key) {
        root->right = deleteNode(root->right, key);
    } else {
        if (!root->left || !root->right) {
            root = root->left ? root->left : root->right;
        } else {
            // find successor, swap with target, then delete key from right subtree
            TreeNode* succ = root->right;
            while (succ->left) succ = right->left;
            swap(root->val, succ->val);
            root->right = deleteNode(root->right, key);
        }
    }
    return root;
}
```

```
// 450 - delete node in bst
// soln-2: find target and keep (parent, target), find successor and keep (parent, successor)
TreeNode* deleteNode(TreeNode* root, int key) {
     // find target node and its parent
     TreeNode* parent = nullptr, *target = nullptr;
     for(TreeNode* cur = root; cur; cur = cur->val > key : cur->left : cur->right) {
         if (cur->val == key) {
```

```
            target = cur;
            break;
        }
        parent = cur;
    }
    if (!target) return root;         // target not found
    if (!parent) return del(target);  // target is root

    parent->left == target ? parent->left = del(target) : parent->right = del(target);
    return root;
}

// 1. find successor and its parent of target.
// 2. swap target with successor
// 3. parent->right = succ->right if parent == target
//    parent->left  = succ->right if parent != target
TreeNode* del(TreeNode* target) {
    if (!target->left && !target->right) return nullptr;        // target is leaf node
    if (!target->left || !target->right) return target->left ? target->left : target->right;

    TreeNode* parent = target, *succ = target->right;
    while (succ->left) {
        parent = succ, succ = succ->left;
    }
    swap(target->val, succ->val);
    (parent == target ? parent->right : parent->left) = succ->right;

    return target;
}
```

```
// 776 - split BST
vector<TreeNode*> splitBST(TreeNode* root, int target) {
   if (nullptr == root) return vector<TreeNode*>{nullptr, nullptr};
   if (root->val >= target) {
       auto left = splitBST(root->left, target);
       root->left = left[1];       // my left child would be larger part when split left subtree
       return {left[0], root};
   } else {
       auto right = splitBST(root->right, target);
       root->right = right[0];     // my left child would be smaller part when split right subtree
       return {root, right[1]};
   }
}
```

Ref:

237. Delete node in a linked list

Trivial question

```
void deleteNode(struct ListNode* node) {
    node->val = node->next->val;
    node->next = node->next->next;
}
```

Ref:

238. Product of array except self

Given an array of n integers where $n > 1$, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

Solve it **without division** and in O(n).

For example, given [1,2,3,4], return [24,12,8,6].

Follow up:

Could you solve it with constant space complexity? (Note: The output array **does not** count as extra space for the purpose of space complexity analysis.)

Solution:
2-pass scan soln:
Very trick question for me. For position i, the answer is left_part * right_part. So the idea is to scan the array from left direction to get left part first, then from the right direction. For example,

```
       1,   2,  3,  4
       x,   1,  2,  6  <-- left part for each position
       24, 12,  4,  x  <-- right part for each position
       24, 12,  8,  6  <-- ans = left * right
```

For convenience, x should be 1.

```
vector<int> productExceptSelf(vector<int>& nums) {
    vector<int> ans(nums.size(), 1);

    int left = nums.front();
    for (int i = 1; i < nums.size(); ++i) {
        ans[i] = left;
        left *= nums[i];
    }

    int right = nums.back();
    for (int i = nums.size() - 2; i >= 0; --i) {
        ans[i] *= right;
        right *= nums[i];
    }
    return ans;
}
```

Ref: #11 #42 #152

239/480/862. Sliding window max/median/shortest subarray sum with at least k (review)

```
Window position                Max      Median (#480)
---------------               -----------------------
[1  3  -1] -3  5  3  6  7       3         1
 1 [3  -1  -3] 5  3  6  7       3        -1
 1  3 [-1  -3  5] 3  6  7       5        -1
 1  3  -1 [-3  5  3] 6  7       5         3
 1  3  -1  -3 [5  3  6] 7       6         5
 1  3  -1  -3  5 [3  6  7]      7         6
```

Monotone stack questions:
#3 - sliding window
#84/85 - largest rectangle in histogram/maximum rectangle (see #42)
#42/407 - trapping in water
#456 - 132 pattern
#496/503/556 - next greater element/II/III
#654/998 - max binary tree/II
#739 - daily temperature
#768/769 - max chunks to make sorted/II
#826 - shortest subarray sum with at least k
#901/962 - online stock planner
#907 - sum of subarray mins

```
// 239 - sliding window max
// soln-1: Monotone priority queue
// Monotone queue works as sliding window, and ensures decreasing order, so current max always on the front.
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
```

```cpp
    vector<int> ans;
    deque<int> dq;

    for (int i = 0; i < (int)nums.size(); ++i) {
        if (!dq.empty() && dq.front() + k <= i)          dq.pop_front();  // assure the window size
        while (!dq.empty() && nums[dq.back()] < nums[i]) dq.pop_back();   // remove small from back: |:.

        dq.push_back(i);
        if (i + 1 >= k) ans.push_back(nums[dq.front()]);                  // index starting from 0
    }

    return ans;
}
```

```cpp
// 456 - 132 pattern
// soln-1: monotone stack + greedy
// 1. the smallest is in front of others (132 pattern), better scan from backwards.
// 2. if cur > stk.top, then greedy take stk.top as s2.
// 3. if cur < s2, then found answer.
bool find132pattern(vector<int>& nums) {
    stack<int> stk;
    int s2 = INT_MIN;
    for (auto i = nums.rbegin(); i != nums.rend(); ++i) {
        if (*i < s2) return true;
        while (!stk.empty() && *i > stk.top()) {
            s2 = stk.top(), stk.pop();  // new s2 will always > old s2, otherwise we have 132 already.
        }
        stk.push(*i);   // .:| stack (from backwards)
    }
    return false;
}
```

```cpp
// 480 - sliding window media
// soln-1: use BST to maintain window and advance to the middle to get median in O(nk) time
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
   vector<double> ans;
   multiset<int> window(nums.begin(), nums.begin() + k - 1);
   for (int i = k - 1; i < nums.size(); ++i) {
       window.insert(nums[i]);

       auto it1 = next(window.begin(), (k - 1) / 2);
       auto it2 = next(it1, k % 2 ? 0 : 1);
       ans.push_back((*it1 + *it2) / 2.0f);
       window.erase(window.find(nums[i - k + 1]));
   }
   return ans;
}
```

```cpp
// 480 - sliding window median
// soln-2: use 2 BST to maintain window for median
// 2 BST act like heap for "median for stream data", unlike heap BST is good at removing (get out of window).
// time: O(nlgk), lgk to find/remove number
//
// improvement - use 1 BST with O(nlgk) time
// 1. maintain pointer to middle position of BST
// 2. move the middle left/right accordingly when insert new and remove old
// 3. it depends on compiler behavior
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
   vector<double> ans;

   multiset<int> window(nums.begin(), nums.begin() + k);
   auto mid = next(window.begin(), k / 2);
   for (int i = k; ; ++i) {
       ans.push_back(((double)(*mid) + *prev(mid, k % 2 ? 0 : 1)) / 2.0f);
```

```
        if (i >= nums.size()) break;

        // C++ 98: There are no guarantees on the relative order of equivalent elements.
        // C++ 11: The relative ordering of equivalent elements is preserved, and newly
        //         inserted elements follow their equivalents already in the container.
        window.insert(nums[i]);
        if (nums[i] < *mid) --mid;                          // if inserting into left side
        if (nums[i - k] <= *mid) ++mid;                     // if removing the left side number
        window.erase(window.lower_bound(nums[i - k]));  // make sure removing only 1 leftmost if have dups
    }
    return ans;
}
```

```
// 862 - shortest subarray with sum at least k
// soln-1: monotone queue
// 1. sum input at each position
// 2. keep monotone queue, .:|, because the new incoming will always
// .  ensure a shorter subarray AND a bigger subarray sum
// soln-2: priority queue (min Q based on sum, then compare cur sum with top)
int shortestSubarray(vector<int>& A, int K) {
    vector<int> B(A.size() + 1);
    for (int i = 1; i < B.size(); ++i) B[i] = B[i - 1] + A[i - 1];
    deque<int> dq;
    int ans = INT_MAX;
    for (int i = 0; i < B.size(); ++i) {
        while (!dq.empty() && (B[i] - B[dq.front()]) >= K) {
            ans = min(ans, i - dq.front()), dq.pop_front();
        }
        while (!dq.empty() && B[i] <= B[dq.back()]) dq.pop_back();
        dq.push_back(i);
    }
    return INT_MAX == ans ? -1 : ans;
}
```

Ref: #3 #76 #155 #209 #159 #295 #727

243/244/245. Shortest word distance/II/III

```
// 243 - shortest word distance
int shortestDistance(vector<string>& words, string word1, string word2) {
    int ans = INT_MAX;
    int i1 = -1, i2 = -1;

    for (int i = 0; i < words.size(); ++i) {
        if (words[i].compare(word1) == 0)    i1 = i;
        else if (words[i].compare(word2) == 0)    i2 = i;

        if (i1 != -1 && i2 != -1) ans = min(ans, abs(i1 - i2));
    }

    return ans;
}
```

```
// 244 - shortest word distance III (called multiple times)
// soln-1: hashmap to buffer index, then check distance and advance smaller index (or it creates bigger gap)
class WordDistance {
    unordered_map<string, vector<int>> m_mp;

public:
    WordDistance(vector<string>& words) {
        for (int i = 0; i < words.size(); ++i) {
            m_mp[words[i]].push_back(i);
        }
    }

    int shortest(string word1, string word2) {
        vector<int>& d1 = m_mp[word1], &d2 = m_mp[word2];
```

```
        int ans = INT_MAX;
        for (int i = 0, j = 0; i < d1.size() && j < d2.size(); d1[i] < d2[j] ? ++i : ++j) {
            ans = min(ans, abs(d1[i] - d2[j]));
        }

        return ans;
    }
};
```

```
// 245 - shortest word distance III (word1 could be same as word2)
// soln-1: always update smaller index
int shortestWordDistance(vector<string>& words, string word1, string word2) {
    int ans = INT_MAX, i1 = -1, i2 = -1;
    bool eq = (word1.compare(word2) == 0);

    for (int i = 0; i < words.size(); ++i) {
        if (eq) {
            if (word1.compare(words[i]) == 0) i1 < i2 ? i1 = i : i2 = i;
        } else {
            if (word1.compare(words[i]) == 0)  i1 = i;
            else if (word2.compare(words[i]) == 0)  i2 = i;
        }

        if (i1 != -1 && i2 != -1) ans = min(ans, abs(i1 - i2));
    }

    return ans;
}
```

Ref:

246/247/248. Strobogrammatic number/II/III

```
// 246 - strobogrammatic number (looks same when rotated 180 degrees) validation
// soln-1: hashmap with two pointers
bool isStrobogrammatic(string num) {
    unordered_map<char, char> mp{ {'0', '0'}, {'1', '1'}, {'6', '9'}, {'8', '8'}, {'9', '6'} };
    for (int i = 0; i <= num.length() / 2; ++i) {
        if (mp[num[i]] != num[num.length() - i - 1])    return false;
    }
    return true;
}
```

```
// 247 - find strobomgrammatic numbers given length
// soln-1: backtracking (brute-force enumerate)
vector<string> findStrobogrammatic(int n, bool leftmost = true) {
   if (n < 1) return vector<string>{""};
   if (n == 1) return vector<string>{"0", "1", "8"};

   vector<string> ans;
   vector<string> rest = findStrobogrammatic(n - 2, false);     // not leftmost digit now
   for (string& r : rest) {
       if (!leftmost) ans.push_back("0" + r + "0");
       ans.push_back("1" + r + "1");
       ans.push_back("6" + r + "9");
       ans.push_back("8" + r + "8");
       ans.push_back("9" + r + "6");
   }
   return ans;
}
```

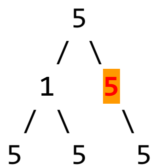
```
// 248 - find strobomgrammatic numbers within range
// soln-1: backtracking (brute-force enumerate)
int strobogrammaticInRange(string low, string high) {
   int ans = 0;
   for (int len = low.length(); len <= high.length(); ++len) {
       string str(len, '0');
       helper(low, high, str, 0, len - 1, ans);
   }
```

```
    return ans;
}

void helper(const string& lo, const string& hi, string& str, int left, int right, int& ans) {
    if (left > right) {
        if (str.length() == lo.length() && str.compare(lo) < 0) return;   // check if within range
        if (str.length() == hi.length() && str.compare(hi) > 0) return;   // check if within range
        ++ans;
        return;
    }

    const vector<pair<char, char>> mp{ {'0', '0'}, {'1', '1'}, {'6', '9'}, {'8', '8'}, {'9', '6'} };
    for (auto p : mp) {
        if (str[0] == '0' && str.size() != 1) continue;

        str[left] = p.first, str[right] = p.second;
        if (left < right || (left == right && p.first == p.second)) {     // be careful here
            helper(lo, hi, str, left + 1, right - 1, ans);
        }
    }
}
```

Ref: #39

250. Count univalue subtree

Given a binary tree, count the number subtree which has the same value in the subtree. A Uni-value subtree means all nodes of the subtree have the same value. For example: Given binary tree,



return 4. (3 leafs and one substree 5 without left subtree)

Solution:

Observation:

1. a leaf node must be uni-value subtree
2. if a left subtree is uni-value subtree and right subtree is uni-value subtree, the whole subtree is uni-value subtree if they have same value.

```
int countUnivalSubtrees(TreeNode* root) {
    int ans = 0;
    helper(root, ans);
    return ans;
}

// return true if it is uni-value subtree
bool helper(TreeNode* root, int& ans) {
    if (root == NULL) return true;  // should be treated as uni-value subtree

    // leaf node must be uni-value subtree
    if (root->left == NULL && root->right == NULL) {
        ++ans;
        return true;
    }

    bool left = helper(root->left, ans);
    bool right = helper(root->right, ans);

    if (left == false || right == false)             return false;
    if (root->left && root->left->val != root->val)     return false;
    if (root->right && root->right->val != root->val)   return false;

    ++ans;
    return true;
}
```

Ref:

251. Flatten 2d vector

Implement an iterator to flatten a 2d vector.

For example, Given 2d vector =

```
[
  [1,2],
  [3],
  [4,5,6]
]
```

By calling *next* repeatedly until *hasNext* returns false, the order of elements returned by *next* should be: `[1,2,3,4,5,6]`.

Hint:

1. How many variables do you need to keep track?
2. Two variables is all you need. Try with `x` and `y`.
3. Beware of empty rows. It could be the first few rows.
4. To write correct code, think about the invariant to maintain. What is it?
5. The invariant is `x` and `y` must always point to a valid point in the 2d vector. Should you maintain your invariant *ahead of time* or *right when you need it*?
6. Not sure? Think about how you would implement `hasNext()`. Which is more complex?
7. Common logic in two different places should be refactored into a common method.

Follow up:

As an added challenge, try to code it using only iterators in C++ or iterators in Java.

Solution:
Similar to #281. Track both current iterator and end iterator, if hitting end, remove it from list.

```
// 17 / 17 test cases passed. Runtime: 20 ms
class Vector2D {
    list<pair<vector<int>::const_iterator, vector<int>::const_iterator>> _v;

public:
    Vector2D(vector<vector<int>>& vec2d) {
        for (auto& x : vec2d) {     // must use reference, otherwise iterator will be invalid!
            if (x.size())   _v.emplace_back(x.begin(), x.end());
        }
    }

    int next() {
        int ans = *_v.front().first++;
        if (_v.front().first == _v.front().second) {
            _v.pop_front();
        }

        return ans;
    }

    bool hasNext() {
        return !_v.empty();
    }
};

// From Stephan which use constant space.
// 17 / 17 test cases passed. Runtime: 16 ms
class Vector2D_2 {
    vector<vector<int>>::const_iterator _cur, _end;
    int _idx;

public:
    Vector2D_2(vector<vector<int>>& vec2d) {
        _cur = vec2d.begin();
        _end = vec2d.end();
        _idx = 0;
    }
```

```
    int next() {
        hasNext();
        return (*_cur)[_idx++];
    }

    bool hasNext() {
        while (_cur != _end && _idx == _cur->size()) {
            _cur++;
            _idx = 0;
        }
        return _cur != _end;
    }
};
```

Ref: #173 #281 #284 #341

252/253/630. Meeting rooms/II... Greedy/Earliest finishing time (review)

435/452/646 - non-overlapping intervals/min arrows to burst balloons (greedy - earliest finishing time)
729/731/732 - my calendar I/II/III

```
// 252 - meeting room (check if possible)
// soln-1: sort - only when no overlap can a person join the meetings
bool canAttendMeetings(vector<Interval>& intervals) {
    sort(intervals.begin(), intervals.end(), [](Interval &i, Interval &j) { return i.start < j.start; });
    for (int i = 1; i < intervals.size(); ++i) {
        if (intervals[i - 1].end > intervals[i].start)  return false;
    }
    return true;
}
```

```
// 253 - meeting rooms II (min rooms required)
// soln-1: greedy + priority queue
// 1. sort meeting intervals by starting time
// 2. based on ending-time, make a priority queue, if current starting time is later than earliest ending time,
//    then it's safe to reuse the same room, which removing the old meeting from queue
// 3. finally, the size of priority queue will be the rooms needed,
//    whose meeting need to start before the ending of previous meeting
int minMeetingRooms(vector<Interval>& intervals) {
    sort(intervals.begin(), intervals.end(), [](Interval &i, Interval &j) { return i.start < j.start; });
    priority_queue<int, vector<int>, greater<int>> min_heap;
    for (auto interval : intervals) {
        if (!min_heap.empty() && min_heap.top() <= interval.start) min_heap.pop();
        min_heap.push(interval.end);
    }
    return min_heap.size();
}
```

```
// 630 - Course Schedule III (given ddl and duration, return max courses one can task)
// soln-1: greedy + priority queue (similar to meeting room II)
// 1. sort courses based on ddl (greedy pick early ddl courses)
// 2. drop the longest duration course if cur time exceeds
// 3. why not take shortest first? Ex. [4, 5], [2, 6], if take [2, 6] first, we can't take [4, 5] anymore
int scheduleCourse(vector<vector<int>>& courses) {
   sort(courses.begin(), courses.end(), [](const vector<int>& a, const vector<int>& b) {
       return a[1] < b[1];
   });
   priority_queue<int> pq;         // max heap by default, based on course duration
   for (int end_date = 0, i = 0; i < courses.size(); ++i) {
       end_date += courses[i][0], pq.push(courses[i][0]);
       if (end_date > courses[i][1]) end_date -= pq.top(), pq.pop();   // drop longest duration course
   }
   return pq.size();
}
```

```
// 1024 - Video Stitching
```

```
// soln-1: greedy (priority queue is not working)
int videoStitching(vector<vector<int>>& clips, int T) {
   sort(clips.begin(), clips.end());
   int ans = 0, n = clips.size();
   for (auto i = 0, st = 0, end = 0; st < T; st = end, ++ans) {
       for (; i < n && clips[i][0] <= st; ++i) end = max(end, clips[i][1]);
       if (st == end) return -1;       // no clip found in current loop
   }
   return ans;
}

// 1024 - Video Stitching
// soln-2: dynamic programming in O(TN) time
int videoStitching2(vector<vector<int>>& clips, int T) {
   sort(clips.begin(), clips.end());
   vector<int> dp(T + 1, clips.size() + 1);
   dp[0] = 0;
   for (auto& c : clips) {
       for (int t = c[0] + 1, st = c[0]; t <= T && t <= c[1]; ++t) {
           dp[t] = min(dp[t], dp[st] + 1);
       }
   }
   return (dp[T] == clips.size() + 1) ? -1 : dp[T];
}

// 1024 - Video Stitching
// soln-3: dynamic programming in O(TN) time
int videoStitching(vector<vector<int>>& clips, int T) {
   vector<int> dp(T + 1, clips.size() + 1);
   dp[0] = 0;
   for (int t = 1; t <= T; ++t) {
       for (auto& c : clips) {
           auto st = c[0], end = c[1];
           if (st <= t && t <= end) {
               dp[t] = min(dp[t], dp[st] + 1);
           }
       }
   }
   return (dp[T] == clips.size() + 1) ? -1 : dp[T];
}
```

```
// 1094 - Car Pooling (find out if it's possible of car pooling)
// soln-1: greedy + priority queue
// 1. sort by start time and check capacity with request each station.
// 2. use priority queue to store ending time and release capacity accordingly. (meeting room II)
bool carPooling(vector<vector<int>>& trips, int capacity) {
   sort(trips.begin(), trips.end(), [](const vector<int>& a, const vector<int>& b) {
       return a[1] < b[1];
   });
   priority_queue<pair<int, int>> pq;       // <ending time, occupants>, its max-heap by default
   for (auto& st : trips) {
       while (!pq.empty() && -pq.top().first <= st[1]) {
           capacity += pq.top().second, pq.pop();
       }
       if (capacity < st[0]) return false;
       capacity -= st[0], pq.push({-st[2], st[0]});
   }
   return true;
}
```

Ref: 435/452/646

255/331/946/971. Verify preorder/Validate stack seq/Flip binary tree to match preorder (review)

```
// 255 - verify preorder in bst
```

```cpp
// soln-1: monotone stack
bool verifyPreorder(vector<int>& preorder) {
    stack<int> s;
    int boundary = INT_MIN;

    for (int x : preorder) {
        if (x < boundary) return false;     // you can't smaller than low-boundary

        while (!s.empty() && s.top() < x) { // entering right subtree, continue pop-out left subtree
            boundary = s.top(); s.pop();
        }
        s.push(x);
    }

    return true;
}

bool verfiyPostorder(vector<int>& postorder) {
    stack<int> s;
    int boundary = INT_MAX;

    for (int i = postorder.size() - 1; i >= 0; --i) {
        if (postorder[i] > boundary) return false;      // you can't bigger than up-boundary

        while (!s.empty() && postorder[i] < s.top()) {  // continue pop-out right subtree
            boundary = s.top(); s.pop();
        }
        s.push(postorder[i]);
    }

    return true;
}
```

```cpp
// 331 - verify preorder sequence
// soln-1: stack
bool isValidSerialization(string preorder) {
    istringstream iss(preorder);
    stack<pair<string, int>> stk;
    string str;
    if (getline(iss, str, ',') && str != "#") stk.push({str, 0});
    while (!stk.empty()) {
        auto& tp = stk.top();
        if (0 == tp.second) {           // just discovered
            if (!getline(iss, str, ','))    return false;
            tp.second = 1;
            if (str != "#") stk.push({str, 0});
        } else if (1 == tp.second) {    // return from left
            if (!getline(iss, str, ','))    return false;
            tp.second = 2;
            if (str != "#") stk.push({str, 0});
        } else {                        // return from right
            stk.pop();
        }
    }
    return stk.empty() && !getline(iss, str, ',');
}
```

```cpp
// 946 - validate stack sequence
// soln-1: stack
bool validateStackSequences(vector<int>& pushed, vector<int>& popped) {
    vector<int> stk;
    for (int i = 0, j = 0; i < pushed.size(); ++i) {
        stk.push_back(pushed[i]);
        while (!stk.empty() && stk.back() == popped[j]) stk.pop_back(), ++j;
    }
    return stk.empty();
}
```

```cpp
// 971 - flip binary tree to match preorder
// soln-1: dfs (switch left/right substree accordingly)
```

```cpp
bool h(TreeNode* node, vector<int>& v, int& idx, vector<int>& ans) {
    if (node == nullptr) return true;
    if (node->val != v[idx++]) return false;
    if (node->left && node->left->val != v[idx]) {
        ans.push_back(node->val);       // then go right subtree first
        return h(node->right, v, idx, ans) && h(node->left, v, idx, ans);
    }
    return h(node->left, v, idx, ans) && h(node->right, v, idx, ans);
}

vector<int> flipMatchVoyage(TreeNode* root, vector<int>& voyage) {
    vector<int> ans;
    int idx = 0;
    return h(root, voyage, idx, ans) ? ans : vector<int>{-1};
}
```

Ref:

256/265. Paint house/II

There are a row of *n* houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a *n* x *3* cost matrix. For example, costs[0][0] is the cost of painting house 0 with color red; costs[1][2] is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

Note: All costs are positive integers.

There are a row of *n* houses, each house can be painted with one of the *k* colors. Follow up: Could you solve it in *O*(*nk*) runtime?

```cpp
// 256 - soln-1: dynamic programming
// let f(k, j) be the cost when painting house-j using color k
// f(k, j) = min{f(i, j - 1) + cost[(i + 1) % k] | i = 0~k-1}
int minCost(vector<vector<int>>& costs) {
   vector<vector<int>> dp = costs;

   for (int i = 1; i < costs.size(); ++i) {
       dp[i][0] += min(dp[i][1], dp[i][2]);
       dp[i][1] += min(dp[i][0], dp[i][2]);
       dp[i][2] += min(dp[i][0], dp[i][1]);
   }
   return min(dp.back()[0], dp.back()[1], dp.back()[2]);
}

// 265 - soln-1: dynamic programming
// similarly we want to keep k-diff. cost for each house painting, e.x.,
// if previous is painted with R, what would be the minimum cost if using rest k-1 colors to paint this one
//                              G,  \
//                              B,   > K, this will take O(nk^2) time
//                              ..  /
// actually we do not need to compare with all k-1 colors of previous house painting,
// we only need to keep the minimum 2 colors used in previous house!
int minCostII(vector<vector<int>>& costs) {
   if (costs.empty() || costs[0].empty()) return 0;

   vector<vector<int>> dp = costs;

   int col1 = -1, col2 = -1;                 // colors with minimum cost for painting current house
   for (int i = 0; i < dp.size(); ++i) {
       int precol1 = col1, precol2 = col2;     // colors with minimum cost for painting previous house
```

```
        col1 = col2 = -1;
        for (int color = 0; color < dp[i].size(); ++color) {
            if (color != precol1) {
                dp[i][color] += precol1 < 0 ? 0 : dp[i - 1][precol1];
            } else {
                dp[i][color] += precol2 < 0 ? 0 : dp[i - 1][precol2];
            }

            if (col1 < 0 || dp[i][color] < dp[i][col1]) {
                col2 = col1;    col1 = color;
            } else if (col2 < 0 || dp[i][color] < dp[i][col2]) {
                col2 = color;
            }
        }
    }
    return dp.back()[col1];
}
```

Ref:

261. Graph valid tree

Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

```
// 261: soln-1: cycle-detection of undirected graph (union-find or leaf prune)
bool validTree(int n, vector<pair<int, int> > edges) {
    UnionFind uf(n);
    for (auto& e : edges) {
        if (uf.Find(e.first) == uf.Find(e.second)) return false;
        uf.Union(e.first, e.second);
    }
    return uf.getSetCount() == 1;       // in case of forest
}
```

Ref: #305

263/264/313. Ugly number

```
// 263 - ugly number (prime factors only include 2/3/5)
// soln-1: recursion by definition
bool isUgly(int num) {
    if (num <= 0)   return false;       // should be positive numbers
    if (num == 1)   return true;

    if (num % 2 == 0)   return isUgly(num / 2);
    if (num % 3 == 0)   return isUgly(num / 3);
    if (num % 5 == 0)   return isUgly(num / 5);
    return false;
}

// 263 - ugly number (prime factors only include 2/3/5)
// soln-2: greedy
bool isUgly(int num) {
    if (num <= 0)   return false;       // should be positive numbers

    vector<int> divisors{2, 3, 5};
    for (auto d : divisors) {
        while (num % d == 0) num /= d;
    }
```

```
    return num == 1;
}
```

```
// 264 - ugly number (return nth ugly number)
// soln-1: dynamic programming
// 1. at first, we know 1 is the 1st ugly number.
// 2. 2nd ugly number is based on some previous ugly number, and should be the minimum.
// 3. for each prime factor, we need to track the position, which we based on it to create new candidate.
int nthUglyNumber(int n) {
   int t2 = 0, t3 = 0, t5 = 0; //pointers for 2, 3, 5
   vector<int> dp(n);
   dp[0] = 1;
   for(int i  = 1; i < n ; i ++) {
       dp[i] = min(dp[t2] * 2, min(dp[t3] * 3, dp[t5] * 5));
       if(dp[i] == dp[t2] * 2) t2++;
       if(dp[i] == dp[t3] * 3) t3++;
       if(dp[i] == dp[t5] * 5) t5++;
   }
   return dp[n-1];
}
```

```
// 313 - super ugly number (prime factors are given as vector instead of just 2/3/5)
// soln-1: dynamic programming (essentially it's the same as #264)
int nthSuperUglyNumber(int n, vector<int>& primes) {
    vector<int> dp(n), idx(primes.size());
    dp[0] = 1;
    for (int i = 1; i < n; ++i) {
        dp[i] = primes[0] * dp[idx[0]];
        for (int k = 1; k < primes.size(); ++k) {
            dp[i] = min(dp[i], primes[k] * dp[idx[k]]); // candidate must based on prev ugly number for this
factor
        }

        for (int k = 0; k < primes.size(); ++k) {
            if (dp[i] == primes[k] * dp[idx[k]]) idx[k]++;
        }
    }
    return dp[n - 1];
}
```

Ref: #204 #279 #313

266/267. Palindrome permutation / II

Given a string, determine if a permutation of the string could form a palindrome.

For example, "code" -> False, "aab" -> True, "carerac" -> True.

```
// soln-1: count how many chars have odd number
bool canPermutePalindrome(string s) {
    vector<int> count(256, 0);
    for (char ch : s) count[ch]++;

    int oddNum = 0;
    for (int n : count) if (n & 1) ++oddNum;

    return oddNum <= 1;
}
```

```
// soln-1: check if it can be done for given string,
//         then enumerate half of them
vector<string> generatePalindromes(string s) {
    vector<string> ans;
```

```
    // count the characters and store them into map
    unordered_map<char, int> counter;
    for (char ch : s) counter[ch]++;

    int oddNum = 0, oddChar = 0;
    for (auto &it : counter) {
        if (it.second & 1) {
            oddChar = it.first;    ++oddNum;
        }
        it.second /= 2;     // only need to enumerate half of them
    }

    if (oddNum <= 1) {
        helper(counter, "", s.length() / 2, oddChar, ans);
    }
    return ans;
}

void helper(unordered_map<char, int>& counter, string res, int n, char oddChar, vector<string>& ans) {
    if (res.length() == n) {
        string tmp = res;   reverse(tmp.begin(), tmp.end());
        oddChar ? ans.push_back(res + oddChar + tmp) : ans.push_back(res + tmp);
        return;
    }

    for (auto& it : counter) {
        if (it.second == 0) continue;

        --it.second;
        helper(counter, res + it.first, n, oddChar, ans);
        ++it.second;
    }
}
```

Ref: #5 #242

270/272/285/510. Closest binary search tree value/K-closet/inorder successor of bst/II

```
// 270 - closest bst value
// soln-1: greedy move to left side if cur > target
int closestValue(TreeNode* root, double target) {
    int ans = root->val;
    while (root) {
        ans = abs(ans - target) < abs(root->val - target) ? ans : root->val;
        root = target < root->val ? root->left : root->right;
    }
    return ans;
}
```

```
// 272 - k-closest to target
// soln-1: in-order traversal with k-size sliding window
vector<int> closestKValues(TreeNode* root, double target, int k) {
    vector<int> ans;
    helper(root, target, k, ans);
    return ans;
}

void helper(TreeNode* root, double target, int k, vector<int>& ans) {
    if (root == NULL) return;

    helper(root->left, target, k, ans);

    // idx-0 has k steps to current node, hence has big difference than idx-k
    if (ans.size() >= k && (abs(target - ans[0]) < abs(target - root->val))) return;
    ans.push_back(root->val);
    if (ans.size() >= k) ans.erase(ans.begin());
```

```cpp
    helper(root->right, target, k, ans);
}
```

```cpp
// 272 - soln-2: greedy fetch predecessor/successor
// 1. get closest first, keep a path to the closet
// 2. starting from closet, get predecessor and successor, pick from one of them
// 3. for bst, step-1 in O(lgn) time, step-2 in O(klgn)
vector<int> closestKValues(TreeNode* root, double target, int k) {
    vector<int> ans;
    TreeNode* curr = getClosest(root, target);
    stack<TreeNode*> path;
    findPath(root, curr, path);  // find path from root to current
    ans.push_back(curr->val);    // find the closest one first

    stack<TreeNode*> preStk = path, nxtStk = path;
    TreeNode* pred = getPredecessor(preStk), *succ = getSuccessor(nxtStk);

    // continue to get the left k-1 closest values
    for (int i = 0; i < k - 1 && (pred || succ); ++i) {
        if (nullptr == pred)      ans.push_back(succ->val), succ = getSuccessor(nxtStk);
        else if (nullptr == succ) ans.insert(ans.begin(), pred->val), pred = getPredecessor(preStk);
        else {
            if (abs(target - succ->val) < abs(target - pred->val)) {
                ans.push_back(succ->val), succ = getSuccessor(nxtStk);
            } else {
                ans.insert(ans.begin(), pred->val), pred = getPredecessor(preStk);
            }
        }
    }
    return ans;
}
TreeNode* getClosest(TreeNode* root, double target) {
    if (root == NULL) return NULL;

    TreeNode* cur = root, * ans = cur;
    while (cur) {
        if (abs(cur->val - target) < abs(ans->val - target)) ans = cur;
        cur = (cur->val > target) ? cur->left : cur->right;
    }
    return ans;
}

// get the path from root to target
void findPath(TreeNode* root, TreeNode* target, stack<TreeNode*>& path)
{
    while (root) {
        path.push(root);
        if (root == target) break;
        root = root->val > target->val ? root->left : root->right;
    }
}

// return successor of top element in current stack
TreeNode* getSuccessor(stack<TreeNode*>& s) {
    if (s.empty()) return NULL;

    if (s.top()->right) {
        // if having right child, get the left most of it
        for (TreeNode* cur = s.top()->right; cur; cur = cur->left) s.push(cur);
    } else {
        // no right child, continue popping out if I am the right child of top node
        TreeNode* cur = s.top();  s.pop();
        while (!s.empty() && s.top()->right == cur) cur = s.top(), s.pop();
    }
    return s.empty() ? NULL : s.top();
}
```

```
// get predecessor of top element from current stack
TreeNode* getPredecessor(stack<TreeNode*>& s) {
    if (s.empty()) return NULL;

    if (s.top()->left) {
        // if having left child, get the right most of it
        for (TreeNode* cur = s.top()->left; cur; cur = cur->right) s.push(cur);
    } else {
        // no left child, continue popping out if I am the left child of top node
        TreeNode* cur = s.top();    s.pop();
        while (!s.empty() && s.top()->left == cur) cur = s.top(), s.pop();
    }
    return s.empty() ? NULL : s.top();
}
```

```
// 285 - inorder successor of bst
// soln-1: greedy keep current as candidate if it's bigger than me
TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
    TreeNode* succ = NULL;
    while (root) {
        if (p->val < root->val) {
            succ = root;           // update the answer if current is bigger than me, then goto left subtree
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return succ;
}
```

```
// 510 - inorder successor of bst II
// TODO
```

Ref: #450

271. Encode and decode strings

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

```
// 271 - soln-1: put length ahead of each string
// "abc", "def" => 4.abcd2.ef
string encode(vector<string> strs) {
   string encoded_string;
   for (auto& str : strs) encoded_string += to_string(str.length()) + "." + str;
   return encoded_string;
}

vector<string> decode(string s) {
   vector<string> strs;
   for (int i = 0; i < s.length(); ++i) {
       int j = i;
       while (j < s.length() && s[j] != '.') ++j;
       int len = stol(s.substr(i, j - i));
       strs.push_back(s.substr(j + 1, len));
       i = j + len;
   }
   return strs;
}
```

Ref:

274/275. H-Index/II

```
// 274 - h-index (given citations for each paper, h-index: x paper have at least x citations)
// soln-1: sort
int hIndex(vector<int>& citations) {
    sort(citations.begin(), citations.end());
    for (int i = 0, n = citations.size();; i < n; ++i) {
        if (citations[i] >= n - i)  return n - i;
    }
    return 0;
}

// 274 - h-index (given citations for each paper, h-index: x paper have at least x citations)
// 1. by definition h-index: x papers have at least x citations, and <= n.
// 2. use a bucket to count: # of citations => # of paper
// 3. from backwards to count papers, return if papers >= citation #.
// soln-2: bucket sort
int hIndex(vector<int>& citations) {
    if (citations.empty()) return 0;
    vector<int> bucket(citations.size() + 1);
    for (auto c : citations) c >= citations.size() ? bucket.back()++ : bucket[c]++;

    for (int papers = 0, i = bucket.size() - 1; i >= 0; --i) {
        papers += bucket[i];
        if (papers >= i) return i;
    }
    return 0;
}
```

```
// 275 - H-Index II (given sorted citations)
// soln-1: binary search
int hIndex(vector<int>& cit) {
    if (cit.empty()) return 0;

    int n = cit.size(), lo = 0, hi = n - 1;
    while (lo + 1 < hi) {
        auto m = lo + (hi - lo) / 2;
        cit[m] < (n - m) ? lo = m : hi = m;     // to right if enough paper have higher citation
    }
    return max(min(cit[lo], n - lo), min(cit[hi], n - hi));
}
```

Ref:

276. Paint fence

Given a fence with n posts and k colors, find out the number of ways of painting the fence such that at most 2 adjacent posts have the same color. Since answer can be large return it modulo 10^9 + 7.

#376 - wiggle subsequence
#552 - student attendance record (similar DP idea, but much harder)
#600 - non-negative integers w/o consecutive 1s

```
// soln-1: dynamic programming
// let dp(i) be the ways to paint post-i, we can paint it as same color or diff:
// 1, if we paint as same color as previous one, we have same = dp(i - 1)
// 2, if we paint it using diff. color, we have diff = (pre_diff + pre_same) * (k - 1)
// 3, dp(i) = same + diff
int numsWays(int n, int k) {
   if (n == 0) return 0;

   int same = 0, diff = k;
   for (int i = 2; i <= n; ++i) {
       int pre_diff = diff;
```

```
        diff = (same + pre_diff) * (k - 1);
        same = pre_diff;
    }
    return same + diff;
}
```

Ref: #600

277/997. Find the celebrity/Find the town judge

```
// 277 - find the celebrity (everyone knows celebrity, and celebrity does not know anyone)
// soln-1: greedy
// The problem could be confusing, hence come up with O(n^2) soln.
// But it's O(n) doable. If A knows B, it means:
// 1. A is not celebrity (because celebrity doesn't know anyone)
// 2. B could be celebrity, let's assume B is the one.
// 3. After one pass, assuming X is celebrity, we're assure of others not celebrity,
// but X could be not either, because someone ahead of X may not know X.
// 4. So we need extra pass verification
int findCelebrity(int n) {
    int celebrity = 0;
    for (int i = 0; i < n; ++i) {
        if (knows(celebrity, i)) celebrity = i;
    }
    for (int i = 0; i < n; ++i) {
        if (i == celebrity) continue;
        if (!knows(i, celebrity)) return -1;     // everyone should know celebrity
        if (knows(celebrity, i))  return -1;     // and celebrity shouldn't know anyone else
    }
    return celebrity;
}
```

```
// 997 - find the town judge (everyone trust judge, and judge doesn't trust anyone)
// soln-1: graph (judge must be trusted N-1 times/degree)
// similar to #277 - find the celebrity
int findJudge(int N, vector<vector<int>>& trust) {
     vector<int> degree(N + 1);
     for (auto& t : trust) degree[t[0]]--, degree[t[1]]++;
     for (int i = 1; i <= N; ++i) {
         if (degree[i] == N - 1) return i;
     }
     return -1;
}
```

Ref:

279. Perfect squares

Given a positive integer n, find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to n.

For example, given n = 12, return 3 because 12 = 4 + 4 + 4; given n = 13, return 2 because 13 = 4 + 9.

Solution:
soln-1: dynamic programming
Minimum question often involves dynamic programming. For this question, the intuitive method is:

1) check if the number itself is a perfect square number
2) if not, try to *every possible combination* from previous result.

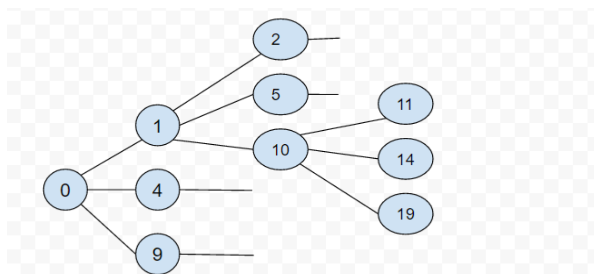
$$f(x) = \min\{ f(i) + f(x - i) \}, i = 1 \sim x\text{-}1 \text{ if x is not a perfect number.}$$

To improve the formula, i should be a perfect number to minimize f(x), in this case which is a perfect number, f(i) = 1, hence:

$$f(x) = \min\{\boldsymbol{f(i * i)} + f(x - i * i)\}, i = 1 \sim \text{sqrt}(x)$$

soln-2:

Consider a graph which consists of number from 0 to n as its nodes. Node *i* and *j* are connected **iff** *j* = *i* + *x***x* where x is {1, ..., sqrt(n)}. Start searching from node 0, breadth-first search to find the shortest path to target number.



Breadth-first search idea to solve minimum perfect number (LC279)
The first layer is perfect number that less than n.
The second layer is val + perfect number found in first layer.
e.g. 2 = 1*1 + 1*1, 19 = 3*3 + 3*3 + 1*1

BFS is slower than DP soln. There is also a math soln, never mind, just for record.

```
// soln-1: Dynamic programming. O(n^2) time, O(n) space
int helper(vector<int>& dp, int x) {
   int ans = INT_MAX;
   for (int i = 1; i * i <= x; ++i) {
       if (i * i == x) return 1;
       ans = min(ans, dp[i * i] + dp[x - i * i]);
   }
   return ans;
}
int numSquares(int n) {
   vector<int> dp(n + 1, 0);
   for (int i = 1; i <= n; ++i) {
       dp[i] = helper(dp, i);
   }
   return dp[n];
}

// soln-2: BFS. Time: O(V+E), for 1st layer, v = sqrt(n), 2nd layer?
// find all perfect_scope square num that less than for the first layer,
// find second layer according to generated perfect_scope square num.
int bfs(int n) {
    queue<int> q;
    vector<int> perfect_scope;
    vector<bool> visited(n + 1, false);
    for (int i = 1; i * i <= n; ++i) {
        perfect_scope.push_back(i * i);
        q.push(i * i); visited[i * i] = true;
    }

    int ans = 0;
    while (!q.empty()) {
        ans++;

        for (int i = q.size(); i > 0; --i) {
            int x = q.front();  q.pop();
            if (x == n) return ans;

            // neighbors: current node plus numbers within perfect_scope
            for (int k : perfect_scope) {
                if (k + x > n || visited[k + x]) continue;
                q.push(k + x); visited[k + x] = true;
            }
        }
```

```
    }

    return 0;       // should never reach here!
}
```

Ref: #204 #264

281/284. Zigzag/Peeking iterator

```
// 281 - Zigzag Iterator
// soln-1: queue
class ZigzagIterator {
    list<pair<vector<int>::const_iterator, vector<int>::const_iterator>> _v;

    void init(list<vector<int>* > & v) {
        for (auto x : v) {
            if (x->size()) {
                _v.emplace_back(x->begin(), x->end());
            }
        }
    }

public:
    ZigzagIterator(vector<int>& v1, vector<int>& v2) {
        list<vector<int> *> v;  // it's iterator, so never make temporary object here
        v.push_back(&v1), v.push_back(&v2);

        init(v);
    }

    int next() {
        int ans = *_v.front().first++;

        if (_v.front().first != _v.front().second) _v.emplace_back(_v.front().first, _v.front().second);
        _v.pop_front();

        return ans;
    }

    bool hasNext() {
        return !_v.empty();
    }
};
```

```
// 284 - Peeking Iterator
class PeekingIterator : public Iterator {
   bool _peeked;
   int _peek_tmp;
public:
    PeekingIterator(const vector<int>& nums) : _peeked(false), Iterator(nums) {
    }

    int peek() {
       if (!_peeked && Iterator::hasNext()) {
           _peeked = true, _peek_tmp = Iterator::next();
       }
       return _peek_tmp;
    }

    int next() {
          if (_peeked) {
           _peeked = false;
           return _peek_tmp;
       } else {
           return Iterator::next();
       }
    }

    bool hasNext() const {
```

```
        if (_peeked) return true;
        return Iterator::hasNext();
    }
};
```

Ref: #173 #251 #341

282/679/841. Expression add operators/24 Games/Keys and rooms (review)

#31 - next permutation
#416/494 - partition equal subset / target sum

```
// 282 - expression add operators
// soln-1: backtracking
// if we only have 2 ops, backtracking would be easier:
//     f(i, j, target) = a[p] + f(p+1, j, target - a[p]), i <= p < j
// for '*' case, there would be op priority issue, for example, 3 + 2 * 3,
// when we see 3 + 2, sum = 5, and we just added 2, then we wanna try * 3, we need to
// 5 - 2 (prev) + 2 * 3 (prev * cur)
void helper(const string& num, int pos, int target, long sum, long prev, string exp, vector<string>& ans) {
    if (pos >= num.size() && sum == target) {
        ans.push_back(exp);
    }

    for (int i = pos; i < num.length(); ++i) {
        string ss = num.substr(pos, i - pos + 1);
        if (ss.length() > 1 && ss[0] == '0') break;

        long k = stol(ss);
        helper(num, i + 1, target, sum + k, k, exp + "+" + ss, ans);
        helper(num, i + 1, target, sum - k, -k, exp + "-" + ss, ans);
        helper(num, i + 1, target, sum - prev + prev * k, prev * k, exp + "*" + ss, ans);
    }
}

vector<string> addOperators(string num, int target) {
    vector<string> ans;
    for (int i = 1; i <= num.length(); ++i) {
        string ss = num.substr(0, i);
        if (ss.length() > 1 && ss[0] == '0') break;      // starting with "0" is invalid if length > 1

        helper(num, i, target, stol(ss), stol(ss), ss, ans);
    }
    return ans;
}
```

```
// 679 - 24 game
// soln-1: backtracking
unordered_set<double> helper(vector<int>& nums, int start, int end) {
    unordered_set<double> ans;
    if (start == end) ans.insert(nums[start]);
    for (int i = start; i < end; ++i) {
        auto first = helper(nums, start, i), second = helper(nums, i + 1, end);
        for (auto a : first) {
            for (auto b : second) {
                ans.insert(a + b), ans.insert(a * b), ans.insert(a - b), ans.insert(b - a);
                if (b) ans.insert(a / b);
                if (a) ans.insert(b / a);
            }
        }
    }
    return ans;
}
```

```
bool judgePoint24(vector<int>& nums) {
   sort(nums.begin(), nums.end());
   do {
       for (auto& ans : helper(nums, 0, nums.size() - 1)) {
           if (abs(ans - 24.0f) < 0.001f) return true;
       }
   } while (next_permutation(nums.begin(), nums.end()));
   return false;
}
```

```
// 841 - keys and rooms
// soln-1: brute-force dfs
bool canVisitAllRooms(vector<vector<int>>& rooms) {
    vector<bool> visited(rooms.size());
    helper(rooms, 0, visited);
    for (auto b : visited) if (!b) return false;
    return true;
}
void helper(vector<vector<int>>& rooms, int start, vector<bool>& visited) {
    visited[start] = true;
    for (auto key : rooms[start]) {
        if (!visited[key]) helper(rooms, key, visited);
    }
}
```

Ref: #416/494

286/489. Walls and gates/Clean room robot

```
// 286 - walls and gates (fill the empty room with distance to its nearest gate)
// soln-1: brute-force dfs/bfs
vector<pair<int, int>> dirs{ {1, 0}, {-1, 0}, {0, 1}, {0, -1} };

void bfs(int x, int y, vector<vector<int>>& room) {
   int rows = room.size(), cols = room[0].size();
   queue<pair<int, int>> q;
   q.push({ x, y });

   int dist = 0;
   while (!q.empty()) {
       ++dist;
       for (int i = q.size(); i > 0; --i) {
           pair<int, int> p = q.front();   q.pop();
           for (auto& dir : dirs) {
               int nx = p.first + dir.first, ny = p.second + dir.second;
               if (nx < 0 || nx >= rows || ny < 0 || ny >= cols || room[nx][ny] <= dist) continue;
               q.push({ nx, ny }), room[nx][ny] = dist;
           }
       }
   }
}

void dfs(int dist, int x, int y, vector<vector<int>>& room) {
   room[x][y] = dist;
   for (auto& dir : dirs) {
       int nx = x + dir.first, ny = y + dir.second;
       if (nx < 0 || nx >= room.size() || ny < 0 || ny >= room[0].size() || room[nx][ny] <= dist) continue;

       dfs(dist + 1, nx, ny, room);
   }
}

void wallsAndGates(vector<vector<int>>& rooms) {
```

```
    for (int i = 0; i < (int)rooms.size(); ++i) {
        for (int j = 0; j < (int)rooms[0].size(); ++j) {
            if (rooms[i][j] == 0) bfs(i, j, rooms);
        }
    }
}
```

```
// 489 - clean room given robot API
// soln-1: brute-force dfs
void helper(Robot& r, int x, int y, int dir, unordered_set<string>& visited) {
   r.clean(), visited.insert(to_string(x) + "+" + to_string(y));

   const vector<pair<int, int>> dirs{{-1, 0}, {0, 1}, {1, 0}, {0, -1}};    // clockwise: N->E->S->W
   for (int i = dir; i < dir + 4; ++i) {
       auto& d = dirs[i % 4];
       int nx = x + d.first, ny = y + d.second;
       if (visited.find(to_string(nx) + "+" + to_string(ny)) != visited.end() || !r.move()) continue;
       helper(r, nx, ny, i % 4, visited);                                  // keep current direct for next
position
       r.turnLeft(), r.turnLeft(), r.move(), r.turnRight(), r.turnRight(); // one step back and change direction
       r.turnRight();
   }

}
void cleanRoom(Robot& robot) {
   unordered_set<string> visited;
   helper(robot, 0, 0, 0, visited);
}
```

Ref: #130 #200 #296 #317

288/544.Unique word abbreviation

#320 - generalized abbr. (backtracking)
#408 - valid word abbr.
#411 - minimum unique word abbr. (enumerate and validate)
#527 - word abbr. (hashset and hashmap)

```
// 544 - Unique word abbreviation (“dog” -> “d1g”, “inter..n” -> “i18n”)
// soln-1: hashmap
class ValidWordAbbr {
    unordered_map<string, string> m_mp;    // <abbr., original-word>

public:
    ValidWordAbbr(vector<string> &dictionary) {
        for (string& word : dictionary) {
            auto abbr = word.front() + to_string(word.length() - 2) + word.back();

            auto iter = m_mp.find(abbr);
            if (iter == m_mp.end()) {
                m_mp[abbr] = word;
            } else {
                if (iter->second != word) m_mp[abbr] = "";  // abbr. duplicated
            }
        }
    }

    bool isUnique(string word) {
        auto abbr = word.front() + to_string(word.length() - 2) + word.back();
        auto iter = m_mp.find(abbr);
        return iter == m_mp.end() || iter->second == word;
    }
};
```

```
// 544 - Output Contest Matches
// soln-1: brute-force
string findContestMatch(int n) {
   vector<string> v;
   for (int i = 1; i <= n; ++i) v.push_back(to_string(i));
   while (n > 1) {
       for (int i = 0; i < n / 2; ++i) {
           v[i] = "(" + v[i] + "," + v[n - 1 - i] + ")";
       }
       n /= 2;
   }
   return v.front();
}
```

Ref: #170

289/957. Game of life/Prison cell after N days (review)

According to the Wikipedia's article: "The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a *board* with *m* by *n* cells, each cell has an initial state *live* (1) or *dead* (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

```
// 1, compute next status based on current status, not updated status, so we need to keep both
// 2, we could use 2nd bits to keep next/updated status
// 3, by default, next status is 0 (dead), so we only care about 2 conditions which turns to alive status
//    3.1 live && with 2-3 neighbors
//    3.2 dead && with 3 neighbors
void gameOfLife(vector<vector<int>>& board) {
   if (board.empty()) return;
   int m = board.size(), n = board[0].size();
   for (int i = 0; i < m; ++i) {
       for (int j = 0; j < n; ++j) {
           int liveNeighbor = getLiveNeighbors(board, i, j);

           if (board[i][j] == 1 && (liveNeighbor == 2 || liveNeighbor == 3)) board[i][j] |= 2;
           if (board[i][j] == 0 && liveNeighbor == 3)                        board[i][j] |= 2;
       }
   }
   for (int i = 0; i < m; ++i) {
       for (int j = 0; j < n; ++j) {
           board[i][j] >>= 1;
       }
   }
}

int getLiveNeighbors(vector<vector<int>>& board, int x, int y) {
   int ans = 0;
   int m = board.size(), n = board[0].size();
   for (int i = max(0, x - 1); i < min(m, x + 2); ++i) {
```

```
        for (int j = max(0, y - 1); j < min(n, y + 2); ++j) {
            ans += (board[i][j] & 1);
        }
    }
    ans -= board[x][y];
    return ans;
}
```

```
// 957 - prison cells after N days
// soln-1: bit manipulation (see Game of life)
vector<int> prisonAfterNDays(vector<int>& cells, int N) {
    vector<int> mp(1 << 8, -1);

    int st = 0;
    for (int x : cells) st <<= 1, st = st | x;     // convert to int
    for (int i = 0; i <= (N - 1) % 14; ++i) {      // every 14 steps must be the same state
        if (-1 == mp[st]) mp[st] = nextDay(st);
        st = mp[st];
    }
    for (int i = 7; i >= 0; --i) cells[i] = (st & 1), st >>= 1; // convert back

    return cells;
}
int nextDay(int fr) {
    int to = 0;
    if (((fr >> 7) & 1) == ((fr >> 5) & 1)) to = to | (1 << 6);
    if (((fr >> 6) & 1) == ((fr >> 4) & 1)) to = to | (1 << 5);
    if (((fr >> 5) & 1) == ((fr >> 3) & 1)) to = to | (1 << 4);
    if (((fr >> 4) & 1) == ((fr >> 2) & 1)) to = to | (1 << 3);
    if (((fr >> 3) & 1) == ((fr >> 1) & 1)) to = to | (1 << 2);
    if (((fr >> 2) & 1) == ((fr >> 0) & 1)) to = to | (1 << 1);
    return to;
}
```

Ref:

205/290/291/890. Isomorphic strings/Word pattern/II

```
// 205 - Isomorphic strings
// soln-1: hashmap  (map 2 char to exactly same value)
bool isIsomorphic(string s, string t) {
    vector<int> m1(256), m2(256);
    for (int i = 0; i < s.length(); ++i) {
        if (m1[s[i]] != m2[t[i]]) return false;
        m1[s[i]] = m2[t[i]] = i + 1;     // maps are initialized to 0.
    }
    return true;
}
```

```
// 290 - word pattern (pattern = "abba", str = "dog cat cat dog")
// soln-1: hashmap (map char & str to exactly same value)
bool wordPattern(string pattern, string str) {
   istringstream iss(str);
   vector<string> words;
   for (string w; iss >> w; nullptr) words.push_back(w);
   if (words.size() != pattern.length()) return false;

   unordered_map<char, int>    ch2int;
   unordered_map<string, int>  str2int;
   for (int i = 0; i < pattern.length(); ++i) {
       if (ch2int[pattern[i]] != str2int[words[i]]) return false;
       ch2int[pattern[i]] = str2int[words[i]] = i + 1;
   }
```

```
    return true;
}
```

```
// 291 - word pattern II (pattern = “abba”, str = “dogcatcatdog”)
// soln-1: backtracking (brute-force split string)
typedef unordered_set<string> dict_t;
typedef unordered_map<char, string> pattern_map_t;

bool helper(const string& pattern, int x, const string& str, int y, pattern_map_t& mp, dict_t& dict) {
    if (x == pattern.length() && y == str.length()) return true;
    if (x >= pattern.length() || y >= str.length()) return false;

    if (mp.find(pattern[x]) != mp.end()) {
        const string& w = mp[pattern[x]];
        return w == str.substr(y, w.length()) && helper(pattern, x + 1, str, y + w.length(), mp, dict);
    }

    for (int j = y + 1; j < (int)str.length(); ++j) {
        string w = str.substr(y, j - y);
        if (dict.find(w) != dict.end()) continue;       // this word has been used before

        mp[pattern[x]] = w, dict.insert(w);
        if (helper(pattern, x + 1, str, j, mp, dict)) return true;
        mp.erase(pattern[x]), dict.erase(w);
    }
    return false;
}

bool wordPatternMatch(string pattern, string str) {
    pattern_map_t mp;
    dict_t dict;
    return helper(pattern, 0, str, 0, mp, dict);
}
```

```
// 890 - Find and Replace Pattern (same as #290)
// soln-1: hashmap (brute-force setup mappings)
vector<string> findAndReplacePattern(vector<string>& words, string pattern) {
    vector<string> ans;
    for (auto& word : words) {
        vector<int> w2p(26), p2w(26);
        bool match = true;
        for (int i = 0; match && i < word.length(); ++i) {
            auto w = word[i] - 'a', p = pattern[i] - 'a';
            if (w2p[w] != p2w[p]) match = false;
            w2p[w] = p2w[p] = i + 1;
        }
        if (match) ans.push_back(word);
    }
    return ans;
}
```

Ref: [#30](#) [#76](#) [#205](#)

292/293/294/464/486/1025. Nim game/Flip game/II/Can I win/Predict winner/Divisor game (review)

```
// 292 - nim game (remove 1~3 stone, whoever removes last one win)
// soln-1: return n % 4;
bool canWinNim(int n) {
    if (n <= 3) return true;
    return !canWinNim(n - 1) || !canWinNim(n - 2) || !canWinNim(n - 3);
}
```

```
// 293 - flip game (generate all possible next moves)
```

```cpp
vector<string> generatePossibleNextMoves(string s) {
    vector<string> ans;
    for (int i = 0; i < s.length() - 1; ++i) {
        if (s[i] == '+' && s[i+1] == '+') {
            // substr(pos, len) return empty string if pos is the length of string.
            ans.push_back(s.substr(0, i) + "--" + s.substr(i + 2));
        }
    }

    return ans;
}
```

```cpp
// 294 - flip game II (predict win or not)
// soln-1: backtracking
// time complexity: let's say the length of given string is n
// round-1: n-1 options at most
// round-2: n-1 - 2 options left
// so the total time: (n-1) * (n-3) * (n-5) * ... = O(n!!). [double factorial]
bool canWin(string& s, unordered_map<string, bool>& mp) {
    if (mp.find(s) != mp.end()) return mp[s];

    for (int i = 0; i < s.length() - 1; ++i) {
        if (s[i] != '+' || s[i + 1] != '+') continue;

        s[i] = s[i + 1] = '-';
        bool win = !canWin(s, mp);      // I win if the peer is not.
        s[i] = s[i + 1] = '+';          // flip back before keep partial result

        if (win) return mp[s] = true;   // keep the win set.
    }

    return mp[s] = false;               // keep the lose string also
}
```

```cpp
// 464 - Can I win (given max-choosable-integer, whoever sum-up >= desired-total wins)
// soln-1: bit manipulation + memo
// the key point of this question is to cache the subproblem result
// 1. as the candidates range from 1~n without dups and reuse, bitset is perfect for this!
// 2. cache what/how? we want to know <candidates-bits, desire> -> true/false
// 2.1 if we know the initial <candidate-bits, desire> pair, then when candidates-bits changed,
//     desire value would be changed accordingly and must be original-desire less used-bits,
// 2.2 so instead of make <candidate-bits, desire> pair, we can just cache <candidates-bit> -> true/false
//     with implicit desire value.
bool helper(int maxChoosableInteger, int desiredTotal, int used, vector<int>& mp) {
   if (mp[used] != -1) return mp[used];

   for (int i = maxChoosableInteger, bit = 1 << (maxChoosableInteger - 1); i > 0; --i, bit >>= 1) {
       if (used & bit) continue;

       if (i >= desiredTotal || !helper(maxChoosableInteger, desiredTotal - i, used | bit, mp)) {
           return mp[used] = 1;
       }
   }
   return mp[used] = 0;
}

bool canIWin(int maxChoosableInteger, int desiredTotal) {
   if (desiredTotal <= 0) return true;
   if (desiredTotal > (1 + maxChoosableInteger) * maxChoosableInteger / 2) return false;

   vector<int> mp(1 << maxChoosableInteger, -1);
   return helper(maxChoosableInteger, desiredTotal, 0, mp);
}
```

```cpp
// 486 - predict the winner (given positive numbers, choose from either side, whoever gets max scores win)
// soln-1: recursion / top-down-DP
// assume there are totally S score, this function return how many score p-1 can get more than p-2.
// if low == hi, current player will get it
// else max{mine - peers-score-from-left}
// O(2^n) time: 2 choice in each step, we have total n steps. O(n^2) with memorization.
int helper(const vector<int>& nums, int low, int hi, vector<vector<int>>& dp) {
   if (dp[low][hi] != INT_MAX) return dp[low][hi];

   if (low == hi) dp[low][hi] = nums[low];
   else dp[low][hi] = max(nums[low] - helper(nums, low + 1, hi, dp),
                          nums[hi] - helper(nums, low, hi - 1, dp));
   return dp[low][hi];
}

bool PredictTheWinner(vector<int>& nums) {
   int n = nums.size();
   vector<vector<int>> dp(n, vector<int>(n, INT_MAX));

   return helper(nums, 0, n - 1, dp) >= 0;
}

// 486 - soln-2: dynamic programming
// let f(i, j) be the max score I can get, for example,
// if I pick the front, then I will get the a[i] + left-over-after-my-peer, so we have
// f(i, j) = max{a[i] + sum(i+1, j) - f(i+1, j),
//               a[j] + sum(i, j+1) - f(i, j+1)}
// if i==j then I will get it.
bool PredictTheWinner(vector<int>& nums) {
   int n = nums.size();
   vector<int> sum(n);

   sum[0] = nums[0];
   for (int i = 1; i < n; ++i) sum[i] = nums[i] + sum[i - 1];

   vector<vector<int>> dp(n, vector<int>(n));
   for (int i = 0; i < n; ++i) dp[i][i] = nums[i];
   for (int i = n - 1; i >= 0; --i) {
       for (int j = i + 1; j < n; ++j) {
           dp[i][j] = max(nums[i] - dp[i + 1][j] + sum[j] - sum[i],
                          nums[j] - dp[i][j - 1] + sum[j - 1] - (i > 0 ? sum[i - 1] : 0));
       }
   }
   return dp[0][n - 1] * 2 >= sum[n - 1];
}
```

```cpp
// 1025. Divisor Game
// soln-1: recursion + memo
// soln-2: odd number wins by induction
unordered_map<int, bool> mp;
bool divisorGame(int N) {
    if (N < 2) return false;
    if (mp.find(N) != mp.end()) return mp[N];
    mp[N] = false;
    for (int i = 1; i * i <= N; ++i) {
        if (N % i == 0 && !divisorGame(N - i)) {
            mp[N] = true;
            break;
        }
    }
    return mp[N];
}
```

Ref:

295. Find median for data stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

For example:

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

Solution:

soln-1: min/max heap to get median in O(lgn) time

1. BST also can do the job and have same time complexity, but with more complicate implementation

#4 - median of two sorted arrays
#295 - median from data stream
#480 - sliding window median
#569 - median employee salary
#571 - median given frequency of numbers

```cpp
class MedianFinder {
   priority_queue<int> _maxH;          // max heap by default
   priority_queue<int, vector<int>, std::greater<int>> _minH;
   double _m;
public:
   /** initialize your data structure here. */
   MedianFinder() { }

   void addNum(int num) {
       if (_maxH.empty() && _minH.empty()) {
           _m = num;                       _minH.push(num);
       } else {
           num < _m ? _maxH.push(num) : _minH.push(num);
           if (int(_minH.size() - _maxH.size()) > 1) {
               _maxH.push(_minH.top());    _minH.pop();
           } else if (int(_maxH.size() - _minH.size()) > 1) {
               _minH.push(_maxH.top());    _maxH.pop();
           }

           if (_minH.size() == _maxH.size()) _m = (_minH.top() + _maxH.top()) / 2.0f;
           else _m = _minH.size() > _maxH.size() ? _minH.top() : _maxH.top();
       }
   }

   double findMedian() { return _m; }
};
```

Ref: #480

296. Best meeting point

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.

For example, given three people living at (0,0), (0,4), and (2,2):

```
1 - 0 - 0 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

The point (0,2) is an ideal meeting point, as the total travel distance of 2+2+2=6 is minimal. So return 6.

Solution:
Looks very similar to #317 - shortest distance from all buildings, but there is quite different soln.

The brute force soln is to calculate the distance for each possible meeting point starting from everyone. Using breadth-first search to calculate distance starting from every person, it yields O(k*n*n) time, where k is the # of person. This time complexity is not what OJ is expected, it will give TLE.

Following the hint: try to solve it in 1-D first, then apply to the 2-D case.

Let's say we have array a[1-0-0-1-0-1-0-1], we want to find some position i in a, such that the total distance d will be minimal. Two-pointers is applicable to this question. Let's consider we start walking from left and right side towards some position between. To minimize the total distance, we have to consider the price for each move. Obviously, if the number of people in one side is less than the # of other side, moving the side with less people is better. Time complexity is O(n), and space is O(1).

Now, the problem now is how to apply 1-D to 2-D. The fact is we can sum rows into just 1 row, and columns into just 1 column, 1-D soln also applicable. It looks like there are more than 1 person at a single position, which could probably a follow up question.

There is another trick soln, but the essential method is similar to above: find the median position of row and col.

```
// 2-pointers for 1-D question: O(m * n) time, O(m + n) space.
// 57 / 57 test cases passed. Runtime: 8 ms
int minTotalDistance(vector<vector<int>>& grid) {
    if (grid.empty()) return -1;

    int rows = grid.size(), cols = grid[0].size();

    vector<int> r(rows, 0), c(cols, 0);
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            c[j] += grid[i][j];
            r[i] += grid[i][j];
        }
    }
    return distance1D(r) + distance1D(c);
}

int distance1D(vector<int>& v) {
    int d = 0, left = v[0], right = v[v.size() - 1];
    for (int i = 0, j = v.size() - 1; i < j; NULL) {
        if (left > right) {
            d += right;         // add up the total cost
            right += v[--j];     // move first, then add up the people
        } else {
            d += left;
            left += v[++i];
        }
    }
```

```
    return d;
}

// soln-2: find median position of row and column respectively
int helper(vector<vector<int>>& grid) {
    if (grid.empty()) return -1;

    int rows = grid.size(), cols = grid[0].size();

    // keep the position info to get the median
    vector<int> r, c;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (grid[i][j] == 1) {
                r.push_back(i);
                c.push_back(j);
            }
        }
    }

    // now sort the position to get the median of row and column
    sort(r.begin(), r.end());       // unnecessary, sorted
    sort(c.begin(), c.end());
    int x = r[r.size() / 2], y = c[c.size() / 2];

    // calculate distance
    int ans = 0;
    for (int i : r) ans += abs(i - x);
    for (int i : c) ans += abs(i - y);

    return ans;
}
```

Ref: #130 #200 #286 #317

297/428/431/449. Serialize and deserialize binary tree/N-ary tree/N-ary to binary/BST

```
// 297 - serialize binary tree
// for fixed sized data structure, binary form has speed advantage.
void helper(TreeNode* r, ostringstream& os) {
    if (!r) {
        os << "#" << " ";
        return;
    }
    os << r->val << " ";
    helper(r->left, os), helper(r->right, os);
}

// 297 - deserialize binary tree
TreeNode* helper(istringstream& is) {
    string str;
    is >> str;
    if ("#" == str) return NULL;

    TreeNode* r = new TreeNode(stoi(str));
    r->left = helper(is), r->right = helper(is);
    return r;
}
```

```
// 428 - de/serialize N-ary tree
//   serialize: pre-order like, val #-of-children
// deserialize: stack, pop out if no child anymore
string serialize(NTreeNode* root) {
```

```
    string ans;
    stack<pair<NTreeNode*, int>> s;      // <NTreeNode*, child-idx>
    if (root) s.push({root, -1});
    while (!s.empty()) {
        auto& tp = s.top();
        if (-1 == tp.second) {          // -1 means process the node itself
            auto str = to_string(tp.first->val) + " " + to_string(tp.first->children.size());
            ans += ans.empty() ? str : (" " + str);
        }

        if (++tp.second < tp.first->children.size()) {
            s.push({tp.first->children[tp.second], -1});
        } else {
            s.pop();
        }
    }
    return ans;
}

NTreeNode* deserialize(string str) {
    NTreeNode* root = nullptr;
    stack<pair<NTreeNode*, int>> s;

    string val, child;
    istringstream iss(str);
    while (iss >> val >> child) {
        auto n = new NTreeNode(stoi(val));
        if (s.empty()) {
            root = n;   // stack empty means we are processing root.
        } else {        // pop out from stack if no more child.
            while (!s.empty() && 0 == s.top().second) s.pop();
        }
        if (!s.empty()) s.top().first->children.push_back(n), --(s.top().second);
        if (stoi(child)) s.push({n, stoi(child)});
    }
    return root;
}
```

```
// 431 - N-ary <-> binary
//   serialize: take first child as left child of binary tree and sibling as right child of previous sibling.
// deserialize: use Q to fill N-ary layer by layer
TreeNode* encode(NTreeNode* root) {
    if (nullptr == root) return nullptr;
    auto ans = new TreeNode(root->val);
    if (root->children.size()) {
        ans->left = encode(root->children[0]);

        auto child = ans->left;
        for (int i = 1; i < root->children.size(); ++i, child = child->right) {
            child->right = encode(root->children[i]);
        }
    }
    return ans;
}
NTreeNode* decode(TreeNode* root) {
    if (nullptr == root) return nullptr;
    auto ans = new NTreeNode(root->val);
    if (root->left) {
        ans->children.push_back(decode(root->left));
        for (auto child = root->left->right; child; child = child->right) {
            ans->children.push_back(decode(child));
        }
    }

    return ans;
}
```

```
// 449 - de/serialize bst
//   serialize: just do pre-order, but we can get rid of #.
// deserialize: in-order + pre-order => construct tree, see 105/106/889 for iterative solution
```

Ref: #105

298/549. Binary tree longest consecutive sequence/II (review)

549 - Given a binary tree, you need to find the length of Longest Consecutive Path in Binary Tree.

Especially, this path can be either increasing or decreasing. For example, [1,2,3,4] and [4,3,2,1] are both considered valid, but the path [1,2,4,3] is not valid. On the other hand, the path can be in the child-Parent-child order, where not necessarily be parent-child order.

Note: All the values of tree nodes are in the range of [-1e7, 1e7].

[Example 1] Input:	[Example 2] Input:
`1` / `/ \` / `2   3`	`2` / `/ \` / `1   3`
Output: 2	Output: 3
The longest consecutive path is [1, 2] or [2, 1].	The longest consecutive path is [1, 2, 3] or [3, 2, 1].

```
// soln-1: top-bottom (see better bottom-up idea)
// k - current length of consecutive path
// return longest consecutive path from top to current node.
int helper(TreeNode* r, int k) {
   int left = 0, right = 0;
   if (r->left) left = helper(r->left, r->left->val + 1 == r->val ? k + 1 : 1);
   if (r->right) right = helper(r->right, r->right->val + 1 == r->val ? k + 1 : 1);
   return max(left, right);
}

// soln-2: bottom-up
// f(x) = 1 + max{consecutive ? f(x.left) : 0, consecutive ? f(x.right) : 0}
// return the length of consecutive path from node
int helper(TreeNode* node, int& ans) {
   if (!node) return 0;

   int left = helper(node->left, ans);
   int right = helper(node->right, ans);

   int ret = (node->left && node->left->val == node->val + 1) ? left + 1 : 1;
   ret = max(ret, (node->right && node->right->val == node->val + 1) ? right + 1 : 1);

   ans = max(ans, ret);

   return ret;
}

int longestConsecutive(TreeNode* root) {
   if (!root) return 0;
   return helper(root, 1);
}
```

```
// 549: soln-1: bottom-up one pass (keep 2 values - for decreasing and increasing order)
//   2
//  /  \
// 1    3
pair<int, int> helper(TreeNode* n, int& ans) {
   if (nullptr == n) return pair<int, int>(0, 0);
   auto left = helper(n->left, ans), right = helper(n->right, ans);
```

```
    pair<int, int> ret(1, 1);               // <increasing-from-bottom, decreasing-from-bottom>
    if (n->left && n->left->val + 1 == n->val)      ret.first = max(ret.first, left.first) + 1;
    if (n->right && n->right->val + 1 == n->val)    ret.first = max(ret.first, right.first) + 1;
    if (n->left && n->left->val - 1 == n->val)      ret.second = max(ret.second, left.second) + 1;
    if (n->right && n->right->val - 1 == n->val)    ret.second = max(ret.second, right.second) + 1;

    ans = max(ret.first, ret.second);
    if (n->left && n->left->val + 1 == n->val && n->right && n->right->val == n->val + 1) {
        ans = max(ans, left.first + right.second + 1);
    }
    if (n->left && n->left->val == n->val + 1 && n->right && n->right->val + 1 == n->val) {
        ans = max(ans, left.second + right.first + 1);
    }
    return ret;
}
int longestConsecutive(TreeNode* root) {
    int ans = 0;
    helper(root, ans);
    return ans;
}
```

Ref: #128 #863

299. Bulls and cows

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example:

```
Secret number:  "1807"
Friend's guess: "7810"
```

Hint: `1` bull and `3` cows. (The bull is `8`, the cows are `0`, `1` and `7`.)
Write a function to return a hint according to the secret number and friend's guess, use `A` to indicate the bulls and `B` to indicate the cows. In the above example, your function should return `"1A3B"`.

Please note that both secret number and friend's guess may contain duplicate digits, for example:

```
Secret number:  "1123"
Friend's guess: "0111"
```

In this case, the 1st `1` in friend's guess is a bull, the 2nd or 3rd `1` is a cow, and your function should return `"1A1B"`.
You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

Solution:
bulls - match in both position and digit
cows - match in digit but not position.
There is also one pass soln here, but 2-pass is easy to understand for me.

```
// 151 / 151 test cases passed. Runtime: 8 ms
string getHint(string secret, string guess) {
    vector<int> s(10, 0), g(10, 0);
    int bull = 0, cow = 0;

    for (int i = 0; i < (int)secret.length(); ++i) {
        if (secret[i] == guess[i]) {
            ++bull;                 // match in digit and position
        } else {
            s[secret[i] - '0']++;
            g[guess[i] - '0']++;
        }
    }

    for (int i = 0; i < 10; ++i) {
```

```
        cow += min(s[i], g[i]);     // match in digit but not position
    }
    return to_string(bull) + "A" + to_string(cow) + "B";
}
```

Ref:

301. Remove invalid parentheses (review)

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

Examples:

```
"()())()" -> ["()()()", "(())()"]
"(a)())()" -> ["(a)()()", "(a())()"]
")(" -> [""]
```

Solution:

Soln-1: breadth-first search (brute force)

Generally we use **stack** to validate parentheses. However, there is a smart one, even suitable for more than one type of parentheses, such as `(), [], {}`. The answer is to use **counter**. When encounter `'('`, increase counter, `')'` decrease counter. Whenever counter is negative during this process, mismatch happened.

To fix the string, we need to remove some `'('` or `')'`. But which one? A **brute force soln** is to try to remove every character, that is, each character has too state: keep/remove, then validate the new string. In this soln, we are going to generate **2^n** substrings, and the validation is going to take O(n) time. So the total running time is quite big. Ideally, the time complexity should be O(C(n, k)), k is the # of chars needs to be removed. To improve the performance, we noticed that there will be some duplicates in the generated substring. A hashset helps eliminate the dups.

Example: given `')()('`

1st try:

[`'()('`, `'))('`, `')(('`, `')()'`] <- yield 4 substring with length -1, verify all of them

2nd try:

`'()('` -> [`')('`, `'(('`, `'()'`] <- found a valid, no 3rd try.
`'))('` -> [`')('`, `')('`, `'))'`] <- dups found
`')(('`
`')()'`

At this point, we are ready for the soln: use a task queue to store substrings, from the given one to generate substrings, and validate substring until the task queue is empty. Before putting into task queue, we check if the substring is visited or not. Once we find a result, it acts like we find the *minimum deletion for the string*. Do not look further.

Soln-2: depth-first search

If we call the above soln as breadth-first search, because of queue and generate different possible substring by deleting, there is another smart depth-first search, the recursion idea, which is much faster than the first one: 216ms -> 3ms.

Example, given `'()())()'`, let's first try to deal with `')'`. When scan to s[4], a mismatch has been found. We have to remove the extra `')'`, there are 3 options, but the last two are actually same, let's try to remove s[1] and s[3]. At this point, s[0]~s[4] has been fixed, then we fix the rest recursively.

Like always, to recursively fix the rest of string, we should continue from the last position that has been dealing with. *Two kind of such information is necessary*: the length of string that is good (this is for depth-first search the rest of the string), the index we are fixing (this is for the situation when we back from depth-first search. because there probably more than one possible positions we need to deal with, e.g. `'()())'`, at s[4] we got mismatch, possible fixings are s[1], s[3]).

After `')'` character has been fixed, the tricky thing comes out: how to fix `'('`, such as `'((()'`? The answer could be just scan from right to left. But we will not be able to reuse the code that scans from left to right. A smart idea is to reverse the string and view `')('` as a legal pair instead of `'()'`, then we are safe to reuse the code.

```
vector<string> removeInvalidParentheses(string s) {
```

```
    vector<string> ans;
    helper(0, s, 0, ans, pair<char, char>{'(', ')'});
    return ans;
}

// soln-1 brute force: generate all substrings and validate each of them
// time: O(2^n) * O(n), space: O(2^n) which is big!
// 125 / 125 test cases passed. Runtime: 216 ms
vector<string> bfs(string s) {
    vector<string> ans;

    queue<string> q;
    unordered_set<string> visited;
    q.push(s);
    visited.insert(s);
    bool found = false;

    while (!q.empty()) {
        string str = q.front(); q.pop();
        if (validParenthese(str)) {
            ans.push_back(str);
            found = true;
        }
        // found result in this layer, looking no further (act like the minimum deletion)
        if (found) continue;

        // try to delete every character to generate substring
        for (size_t i = 0; i < str.length(); ++i) {
            if (str[i] != ')' && str[i] != '(') continue;

            string ss = str.substr(0, i) + str.substr(i + 1);
            if (visited.find(ss) != visited.end())  continue;

            visited.insert(ss);
            q.push(ss);
        }
    }

    return ans;
}

// use counter to validate parentheses instead of stack
bool validParenthese(string& s) {
    int count = 0;
    for (char ch : s) {
        if (ch == '(')  ++count;
        else if (ch == ')') --count;

        if (count < 0)  return false;
    }
    return count == 0;
}

// soln-2 recursive: f(s) = fixed + f(remain)
// 125 / 125 test cases passed. Runtime: 3 ms
// good - the length of string we already fixed. Continue from this position when process the rest.
// pos - the index we are fixing. need this because of multiple fixes to bad position.
void helper(int good, string s, int pos, vector<string>& ans, pair<char, char>& p) {
    int count = 0;
    for (int i = good; i < (int)s.length(); ++i) {      // fix parentheses for one direction
        if (s[i] == p.first)    ++count;
        if (s[i] == p.second)   --count;
        if (count >= 0) continue;                       // legal parentheses
```

```
        for (int j = pos; j <= i; ++j) {                // i points to first illegal char like '())'
            if (s[j] != p.second) continue;
            // from pos, try to remove each char. if there are two consecutive dups, remove the first one only.
            if (j == pos || s[j - 1] != s[j]) helper(i, s.substr(0, j) + s.substr(j + 1), j, ans, p);
        }
        return;                                         // one direction is fixed by recursion. stop the loop.
    }

    reverse(s.begin(), s.end());
    if (p.first == '(') {                               // we just validated '()'
        helper(0, s, 0, ans, pair<char, char>{')', '('});   // let's go the other direction starting from 0.
    } else {
        ans.push_back(s);                               // return from ')(' direction, both are finished.
    }
}
```

Ref:

302. Smallest Rectangle Enclosing Black Pixels

An image is represented by a binary matrix with 0 as a white pixel and 1 as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location (x, y) of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.

For example, given the following image:

```
[
  "0010",
  "0110",
  "0100"
]
```

and x = 0, y = 2, Return 6.

Solution:

Ref:

303/304/307/308/855. Range sum query(Immutable/Mutable)

```
// 304 - Range Sum Query 2D - Immutable
class NumMatrix {
    vector<vector<int>> _s;
public:
    NumMatrix(vector<vector<int>>& matrix) {
        _s = matrix;
        if (_s.empty()) return;
        for (int j = 1; j < _s[0].size(); ++j) _s[0][j] += _s[0][j - 1];
        for (int i = 1; i < _s.size(); ++i) _s[i][0] += _s[i - 1][0];
        for (int i = 1; i < _s.size(); ++i) {
            for (int j = 1; j < _s[i].size(); ++j) {
                _s[i][j] += _s[i - 1][j] + _s[i][j - 1] - _s[i - 1][j - 1];
            }
        }
    }

    int sumRegion(int r1, int c1, int r2, int c2) {
        int ans = _s[r2][c2];
        if (c1 > 0) ans -= _s[r2][c1 - 1];
```

```
        if (r1 > 0) ans -= _s[r1 - 1][c2];
        if (c1 > 0 && r1 > 0) ans += _s[r1 - 1][c1 - 1];
        return ans;
    }
};
```

```
// 307 - Range Sum Query - Mutable
// soln-1: segment tree
// for 2d (#308), build segment tree for each row
// when querying region, query row by row and sum together
// soln-2: Fenwick/bit tree (not straightforward, but more efficient and code is shorter)
struct SegmentTree {
    int lo, hi, sum;
    SegmentTree *left, *right;
    SegmentTree(int fr = 0, int to = 0, int s = 0) : lo(fr), hi(to), sum(s), left(nullptr), right(nullptr) {}
};

class NumArray {
    SegmentTree* _root;
public:
    NumArray(vector<int>& nums) {
        _root = build(nums, 0, nums.size() - 1);
    }

    void update(int i, int val) {
        update(_root, i, val);
    }

    int sumRange(int i, int j) {
        return sumRange(_root, i, j);
    }

    // [lo, m], [m+1, hi]
    SegmentTree* build(vector<int>& nums, int lo, int hi) {
        if (lo > hi) return nullptr;
        if (lo == hi) return new SegmentTree(lo, hi, nums[lo]);

        auto node = new SegmentTree(lo, hi);
        int m = lo + (hi - lo) / 2;
        node->left = build(nums, lo, m), node->right = build(nums, m + 1, hi);
        if (node->left) node->sum += node->left->sum;
        if (node->right) node->sum += node->right->sum;
        return node;
    }

    void update(SegmentTree* node, int idx, int val) {
        if (nullptr == node) return;
        if (node->lo == idx && node->hi == idx) {
            node->sum = val;
        } else {
            auto m = node->lo + (node->hi - node->lo) / 2;
            if (idx <= m) update(node->left, idx, val);         // left part: [lo, m]
            else update(node->right, idx, val);                 // right part: [m+1, hi]
            node->sum = node->left->sum + node->right->sum;
        }
    }

    int sumRange(SegmentTree* node, int lo, int hi) {
        if (nullptr == node) return 0;
        if (node->lo == lo && node->hi == hi) return node->sum;

        auto m = node->lo + (node->hi - node->lo) / 2;
        if (lo >= m + 1) return sumRange(node->right, lo, hi);
        else if (hi <= m) return sumRange(node->left, lo, hi);
        return sumRange(node->left, lo, m) + sumRange(node->right, m + 1, hi);
    }
```

```
};
```

```
// 855 - Exam Room (maximizes the distance to the closest person)
// soln-1: segment tree
// 1. use bst to track intervals: longest first, smaller left point first if having same length
// 2. then how do we merge intervals when leaving?
//    - given p, we know some r: r + 1 = p, need to find its l. (r->l map)
//    - given p, we know some l: p + 1 = l, need to find its r. (l->r map)
// 3. line: [l, r] indicate the max available space, two corner case:
//    - if l == 0, we should take it first, else
//    - if r == N-1, we should take the right then,
//    - else return the middle of line.
int gN;
struct MySeg {
   int l, r;

   MySeg(int lo, int hi) : l(lo), r(hi) {}

   int getSeat() {
       if (0 == l)      return l;          // corner case-1
       if (gN - 1 == r) return r;          // corner case-2
       return (l + r) / 2;                 // middle of line
   }
   int length() const {                    // return segment length
       if (0 == l) return r;               // corner case-1
       if (gN - 1 == r) return r - l;      // corner case-2
       return l > r ? -1 : (r - l) / 2;    // middle of line or -1 if l <= r
   }
   bool operator<(const MySeg& that) const {
       auto l1 = length(), l2 = that.length();
       if (l1 != l2) return l1 > l2;       // longer goes first
       return l < that.l;                  // smaller left point goes first
   }
};

class ExamRoom {
   set<MySeg> _seg;
   unordered_map<int, int> _l2r, _r2l;
public:
   ExamRoom(int N) {
       gN = N;
       _seg.insert(MySeg(0, N - 1)), _l2r[0] = N - 1, _r2l[N - 1] = 0;
   }

   int seat() {
       auto cur = *(_seg.begin());     _seg.erase(_seg.begin());

       auto p = cur.getSeat();
       _seg.insert(MySeg(cur.l, p - 1)), _l2r[cur.l] = p - 1, _r2l[p - 1] = cur.l;
       _seg.insert(MySeg(p + 1, cur.r)), _l2r[p + 1] = cur.r, _r2l[cur.r] = p + 1;
       return p;
   }

   void leave(int p) {
       auto l = _r2l[p - 1], r = _l2r[p + 1];

       _seg.erase(MySeg(l, p - 1)), _seg.erase(MySeg(p + 1, r));
       _l2r.erase(p + 1), _r2l.erase(p - 1);

       _seg.insert(MySeg(l, r)), _l2r[l] = r, _r2l[r] = l;
   }
};
```

Ref: #315

306. Additive number

Additive number is a string whose digits can form additive sequence. A valid additive sequence should contain **at least** three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

Given a string containing only digits `'0'-'9'`, write a function to determine if it's an additive number.

Note: Numbers in the additive sequence **cannot** have leading zeros, so sequence `1, 2, 03` or `1, 02, 3` is invalid.

Example 1:

```
Input: "112358", Output: true
Explanation: The digits can form an additive sequence: 1, 1, 2, 3, 5, 8.
             1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
```

#842 - split array into Fibonacci sequence (almost same approach)

```
// soln-1: brute force enumerate
bool doable(const string& S, int pos, long long f0, long long f1) {
   if (pos >= S.length()) return false;

   bool ans = true;
   do {
       if (pos == S.length()) break;

       long long f2 = f0 + f1;
       string s2 = to_string(f2);
       if (s2.length() + pos > S.length()) ans = false;
       for (int i = 0; ans && i < s2.length(); ++i, ++pos) {
           if (s2[i] != S[pos]) ans = false;
       }
       f0 = f1, f1 = f2;
   } while (ans);

   return ans;
}

bool isAdditiveNumber(string num) {
   const string lmax = to_string(LONG_MAX);
   for (int i = 1; i <= min(num.length(), lmax.length()); ++i) {
       string f0 = num.substr(0, i);
       if (i > 1 && '0' == f0[0]) continue;
       if (f0.length() == lmax.length() && f0 > lmax) continue;

       for (int j = i; j < min(num.length(), lmax.length()); ++j) {
           string f1 = num.substr(i, j - i + 1);
           if (j - i + 1 > 1 && '0' == f1[0]) continue;
           if (f1.length() == lmax.length() && f1 > lmax) continue;

           if (doable(num, j + 1, stoll(f0), stoll(f1))) return true;
       }
   }
   return false;
}
```

Ref:

310. Minimum height trees (review)

Given undirected graph with tree characteristics, if we can choose any node as the root, write a function to return all the possible nodes with minimum height.

```
// 310 - soln-1: greedy prune leafs
```

```
// #261 - graph valid tree (leaf prune or union-find)
typedef vector<unordered_set<int>> t_graph;
vector<int> findMinHeightTrees(int n, vector<pair<int, int>>& edges) {
   t_graph g;
   for(auto& e : edges) {
       g[e.first].insert(e.second), g[e.second].insert(e.first);
   }

   vector<int> ans;
   for (int i = 0; i < n; ++i) if (g[i].size() == 1) ans.push_back(i);

   while(1) {
       vector<int> next;
       for (int leaf : ans) {
           for (int peer : g[leaf]) {
               g[peer].erase(leaf);
               if (g[peer].size() == 1) next.push_back(peer);
           }
       }
       if (next.size() == 0) break;
       ans = next;
   }

   return ans;
}
```

Ref: #261

311. Sparse matrix multiplication

Given two sparse matrices **A** and **B**, return the result of **AB**.

You may assume that **A**'s column number is equal to **B**'s row number.

Example:

```
A = [
  [ 1, 0, 0],
  [-1, 0, 3]
]

B = [
  [ 7, 0, 0 ],
  [ 0, 0, 0 ],
  [ 0, 0, 1 ]
]

     |  1 0 0 |   | 7 0 0 |   |  7 0 0 |
AB = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |
                  | 0 0 1 |
```

Solution:
Use map to chain non-zero #s in matrix B, the key is row # and the value is a chained pairs (col#, val). Then iterate matrix A row by row (cache friendly). Question: is it worth chaining both matrix? Probably not. Since to chain the non zero #, it needs to go through matrix.

```
// 12 / 12 test cases passed. Runtime: 40 ms
vector<vector<int>> multiply(vector<vector<int>>& A, vector<vector<int>>& B) {
    if (A.empty() || B.empty()) return vector<vector<int>>();

    unordered_map<int, vector<pair<int, int>>> mp;
    for (int i = 0; i < B.size(); ++i) {
        for (int j = 0; j < B[0].size(); ++j) {
            if (B[i][j]) mp[i].push_back({ j, B[i][j] });   // (col#, val)
        }
    }

    vector<vector<int>> C(A.size(), vector<int>(B[0].size(), 0));
```

```
    for (int i = 0; i < A.size(); ++i) {
        for (int j = 0; j < A[0].size(); ++j) {
            if (A[i][j] == 0) continue;

            auto iter = mp.find(j);
            if (iter == mp.end()) continue;
            for (pair<int, int>& b : iter->second) {
                int k = b.first;
                C[i][k] += A[i][j] * b.second;
            }
        }
    }

    return C;
}
```

Ref:

312/488. Burst balloons/Zuma game (review)

#546 - remove boxes (similar DP idea with different extra information)
#689 - max sum of 3 non-overlapping subarray
#740 - Delete and earn

```
// 312 - burst balloons to get max coins (burst one balloon will get left * cur * right)
//   nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
//   coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1    = 167
// soln-1: dynamic programming
// for A[0, n-1], to remove A[i] we need A[0, i-1] and A[i+1, n-1] to go first, then
// what we get is A[i] which is left(=1) * A[i] * right(=1).
// THIS LEFT/RIGHT BOUNDARY IS THE EXTRA INFO NEEDED TO DEFINE THE QUESTION, SIMILAR TO #546 - remove boxes
//
// let f(i, j, left, right) be the max points we could get for A[i, j] for fixed left and right.
//     ...left, ......, right, ...
//              i    j
// f(i, j, left, right) = max{ A[k] * left * right + f(i, k, left, A[k+1]) + f(k+1, j, A[k], right) }
// time: O(n^4), space: O(n^2)
int helper(vector<int>& nums, int low, int hi, vector<vector<int>>& dp) {
   if (low > hi) return 0;
   if (dp[low][hi]) return dp[low][hi];

   int ans = 0;
   for (int i = low; i <= hi; ++i) {
       int burst = nums[low - 1] * nums[i] * nums[hi + 1];     // low and hi, NOT i-1 and i+1
       ans = max(ans, burst + helper(nums, low, i - 1, dp) + helper(nums, i + 1, hi, dp));
   }
   return dp[low][hi] = ans;
}

int maxCoins(vector<int>& nums) {
   int n = nums.size();
   nums.insert(nums.begin(), 1), nums.push_back(1);
   vector<vector<int>> dp(nums.size(), vector<int>(nums.size(), 0));

   return helper(nums, 1, n, dp);
}
```

```
// 1039 - Minimum Score Triangulation of Polygon
// soln-1: recursion with memo
// f(i, j) = min{f(i, k) + f(k, j) + A[i]*A[j]*A[k]}, k = i+1 ~ j-1
int helper(vector<int>& A, int lo, int hi, vector<vector<int>>& dp) {
    if (lo + 2 > hi) return 0;
    if (dp[lo][hi] != INT_MAX) return dp[lo][hi];
    long ans = INT_MAX;
    for (int i = lo + 1; i < hi; ++i) {
```

```
        long t = A[i] * A[lo] * A[hi];
        ans = min(ans, t + helper(A, lo, i, dp) + helper(A, i, hi, dp));
    }
    return dp[lo][hi] = ans;
}
int minScoreTriangulation(vector<int>& A) {
    vector<vector<int>> dp(A.size(), vector<int>(A.size(), INT_MAX));
    return helper(A, 0, A.size() - 1, dp);
}
```

```
// 488 - zuma game
// soln-1: brute-force enumerate
int helper(string board, vector<int>& ball) {
   if (board.empty()) return 0;
   int ans = INT_MAX;
   for (int i = 0; i < board.length(); nullptr) {
       int j = i++;
       while (i < board.length() && board[j] == board[i]) ++i;
       int need = 3 - (i - j) > 0 ? 3 - (i - j) : 0;
       if (ball[board[j]] >= need) {
           ball[board[j]] -= need;
           auto tmp = helper(board.substr(0, j) + board.substr(i), ball);
           if (tmp >= 0) ans = min(ans, need + tmp);
           ball[board[j]] += need;
       }
   }
   return ans == INT_MAX ? -1 : ans;
}
int findMinStep(string board, string hand) {
   vector<int> ball(256);
   for (auto& ch : hand) ball[ch]++;
   return helper(board, ball);
}
```

Ref: #740 #689

314/987. Binary tree vertical order traversal

```
// 314 - binary tree vertical order traversal
vector<vector<int> > verticalOrder(TreeNode* root) {
    map<int, vector<int>> mp;
    h(root, 0, mp);

    vector<vector<int>> ans;
    for (auto& kv : mp) ans.push_back(kv.second);
    return ans;
}
void h(TreeNode* node, int idx, map<int, vector<int>>& mp) {
    if (nullptr == node) return;
    mp[idx].push_back(node->val);
    h(node->left, idx - 1, mp), h(node->right, idx + 1, mp);
}
```

```
// 987 - vertical order traversal of binary tree
vector<vector<int>> verticalTraversal(TreeNode* root) {
    map<int, vector<int>> mp;
    vector<vector<int>> ans;

    queue<pair<int, TreeNode*>> q;
    if (root) q.push({0, root});
    while (!q.empty()) {
        map<int, set<int>> tmp;
        for (int k = q.size(); k > 0; --k) {
            auto node = q.front();    q.pop();
            tmp[node.first].insert(node.second->val);
```

```
            if (node.second->left) q.push({node.first - 1, node.second->left});
            if (node.second->right) q.push({node.first + 1, node.second->right});
        }
        for (auto& kv : tmp) mp[kv.first].insert(mp[kv.first].end(), kv.second.begin(), kv.second.end());
    }
    for (auto& kv : mp) ans.push_back(kv.second);

    return ans;
}
```

Ref:

315/327/493/406/818. Smaller after self/Count of range sum/Reverse pairs/Q reconstruct (review)

```
// 315 - count of smaller numbers after self
// soln-1: merge-sort in T(n) = 2T(n/2) + n
// key observation:
// 1. merge-sort is divide-and-conquer method, dividing left and right part for sorting
// 2. while we merge, the left/right is known,
//    - if right goes first, number of smaller than myself wont change.
//    - if left goes first, number of smaller than myself would increase whatever the right went at this time.
// example: L:[2 5], R:[1 6] merge sort
// 1. R:1, since 1 already on right side, smaller than self not affected
// 2. L:2, processing left side, but right side already moved R:1, smaller than self +1
// 3. L:5, same as L:2
// 4. R:6, same as R:1
void mergeAndCount(vector<pair<int, int>>& nums, int beg, int end, vector<int>& ans) {
   if (beg + 1 >= end) return;   // beg + 1 == end means having only one element.
   int mid = beg + (end - beg) / 2;
   mergeAndCount(nums, beg, mid, ans), mergeAndCount(nums, mid, end, ans);

   vector<pair<int, int>> buf(end - beg);
   int j = mid, k = 0;
   for(int i = beg; i < mid; ++i) {
       while (j < end && nums[i].first > nums[j].first) buf[k++] = nums[j++];
       ans[nums[i].second] += (j - mid);
       buf[k++] = nums[i];
   }
   while (j < end) buf[k++] = nums[j++];   // not necessary, see #493

   std::copy(buf.begin(), buf.end(), nums.begin() + beg);
}

vector<int> countSmaller(vector<int>& nums) {
   vector<pair<int, int>> withIdx(nums.size());
   for (int i = 0; i < nums.size(); ++i) withIdx[i] = {nums[i], i};

   vector<int> ans(nums.size());
   mergeAndCount(withIdx, 0, nums.size(), ans);
   return ans;
}
```

```
// 327 - count of range sum
// soln-1: merge-sort
// 1. similar to #315 - count of smaller than self.
//    we check if left[i] - right[j] <= 0, then left goes first and smaller after myself will +j.
// 2. key observation for this question:
//    - each merge, we consider 1 number from left and others from right part which always form a subarray.
//    - we find [j...k) such that right[j] - left[i] >= lower and
//                                right[k] - left[i] > upper, then count of range sum +(k-j).
int mergeAndCount(vector<long>& sum, int beg, int end, pair<long, long> boundary) {
   if (beg + 1 >= end) return 0;
   int mid = (beg + end) / 2;
   int ans = mergeAndCount(sum, beg, mid, boundary) + mergeAndCount(sum, mid, end, boundary);
```

```
    vector<int> buf(end - beg);
    int j = mid, k = mid, p = 0, q = mid;
    for (int i = beg; i < mid; ++i) {
        while (j < end && sum[j] - sum[i] < boundary.first) ++j;     // first j which right[j] - left[i] >= lower
        while (k < end && sum[k] - sum[i] <= boundary.second) ++k;   // first k which right[k] - left[i] > upper
        ans += k - j;

        while (q < end && sum[q] < sum[i]) buf[p++] = sum[q++];
        buf[p++] = sum[i];
    }
    while (q < end) buf[p++] = sum[q++];

    std::copy(buf.begin(), buf.end(), sum.begin() + beg);

    return ans;
}
int countRangeSum(vector<int>& nums, int lower, int upper) {
    vector<long> sum(nums.size() + 1);
    for (int i = 1; i < sum.size(); ++i) sum[i] = sum[i - 1] + nums[i - 1];
    return mergeAndCount(sum, 0, sum.size(), {lower, upper});
}
```

```
// 493 - reverse pairs
// soln-1: merge-sort
// similar to #315 - count of smaller than self
int mergeAndCount(vector<int>& nums, int beg, int end) {
    if (beg + 1 >= end) return 0;
    int mid = (beg + end) / 2;
    int ans = mergeAndCount(nums, beg, mid) + mergeAndCount(nums, mid, end);

    vector<int> buf(end - beg);
    int j = mid, k = 0;
    for (int i = beg, t = mid; i < mid; ++i) {
        while (j < end && (long)2 * nums[j] < nums[i]) ++j;
        ans += (j - mid);

        while (t < end && nums[t] < nums[i]) buf[k++] = nums[t++];
        buf[k++] = nums[i];
    }
    // the rest of right part wont be copied to buf, then copied back to nums again.
    std::copy(buf.begin(), buf.begin() + k, nums.begin() + beg);
    return ans;
}
int reversePairs(vector<int>& nums) {
    return mergeAndCount(nums, 0, nums.size());
}
```

```
// 406 - queue reconstruction by height
// soln-1: greedy
// key observation: the shorts can be inserted into line because it won't affect tallers' result.
// follow-up: to validate the constructed queue, we would need to count of bigger number than myself (#315).
vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {
    sort(people.begin(), people.end(), [](pair<int, int>& a, pair<int, int>& b){
        return a.first > b.first || (a.first == b.first && a.second < b.second);
    });
    vector<pair<int, int>> ans;
    for (auto& p : people) {
        ans.insert(ans.begin() + p.second, p);
    }
    return ans;
}
```

```
// 881 - boats to save people
// soln-1: greedy
// greedy load heaviest person with lightest person in same boat.
int numRescueBoats(vector<int>& people, int limit) {
```

```
    int ans = 0;
    sort(people.rbegin(), people.rend());
    for (int &lo = ans, hi = people.size() - 1; lo <= hi; ++lo) {
        if (people[hi] + people[lo] <= limit) --hi;
    }
    return ans;
}
```

Ref: #218 #307 #308

317/417/934/959. Shortest distance from all buildings/Pacific atlantic water flow/Shortest bridge (review)

```
// 317 - shortest distance from all buildings
// soln-1: bfs from every building to empty land, sum up the distance and get mini. distance
int bfs(vector<vector<int>>& grid, vector<vector<int>>& dgrid, int x, int y, int landMark) {
   static vector<pair<int, int>> dirs{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
   int rows = grid.size(), cols = grid[0].size(), ans = INT_MAX, dist = 0;
   queue<pair<int, int>> q;   q.push({x, y});
   while(!q.empty()) {
       ++dist;
       for (int i = q.size(); i > 0; --i) {
           x = q.front().first, y = q.front().second; q.pop();
           for (auto dir : dirs) {
               int nx = x + dir.first, ny = y + dir.second;
               if (nx < 0 || nx > rows || ny < 0 || ny > cols || grid[nx][ny] != landMark) continue;
               q.push({nx, ny}), grid[nx][ny] = landMark - 1;    // mark it as visited
               dgrid[nx][ny] += dist, ans = min(ans, dgrid[nx][ny]);
           }
       }
   }
   return ans;
}

int shortestDistance(vector<vector<int>>& grid) {
   int rows = grid.size(), cols = grid[0].size();
   vector<vector<int>> d(rows, vector<int>(cols, 0));

   int landMark = 0, ans;
   for (int i = 0; i < rows; ++i) {
       for (int j = 0; j < cols; ++j) {
           if (grid[i][j] != 1) continue;
           ans = bfs(grid, d, i, j, landMark--);
           if (INT_MAX == ans) return -1;      // grid[i][j] is not reachable
       }
   }
   return ans;
}
```

```
// 417 - pacific atlantic water flow
// soln-1: two directions bfs
void bfs(const vector<vector<int>>& matrix, queue<pair<int, int>>& q, vector<vector<bool>>& visited) {
   const vector<pair<int, int>> dirs{{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
   while (!q.empty()) {
       auto x = q.front(); q.pop();
       for (auto dir : dirs) {
           int nx = x.first + dir.first, ny = x.second + dir.second;
           if (nx < 0 || nx >= matrix.size() || ny < 0 || ny >= matrix[0].size()) continue;
           if (visited[nx][ny] || matrix[x.first][x.second] > matrix[nx][ny]) continue;
           visited[nx][ny] = true, q.push({nx, ny});
       }
   }
}
```

```
vector<pair<int, int>> pacificAtlantic(vector<vector<int>> matrix) {
   int rows = matrix.size(), cols = matrix[0].size();
   vector<vector<bool>> pvisited(rows, vector<bool>(cols, false)), avisited(rows, vector<bool>(cols, false));
   queue<pair<int, int>> pq, aq;

   for (int i = 0; i < rows; ++i) {
       pq.push({i, 0});        pvisited[i][0] = true;
       aq.push({i, cols - 1}); avisited[i][cols - 1] = true;
   }
   for (int j = 0; j < cols; ++j) {
       pq.push({0, j});        pvisited[0][j] = true;
       aq.push({rows - 1, j}); avisited[rows - 1][j] = true;
   }
   bfs(matrix, pq, pvisited), bfs(matrix, aq, avisited);

   vector<pair<int, int>> ans;
   for (int i = 0; i < rows; ++i) {
       for (int j = 0; j < cols; ++j) {
           if (pvisited[i][j] && avisited[i][j]) ans.push_back({i, j});
       }
   }
   return ans;
}
```

```
// 934 - shortest bridge
// soln-1: dfs to get boundary then bfs for shortest path
void h(vector<vector<int>>& A, int x, int y, vector<vector<int>>& v, queue<pair<int, int>>& q) {
   if (x < 0 || x >= A.size() || y < 0 || y >= A[0].size() || v[x][y]) return;
   v[x][y] = true;
   if (0 == A[x][y]) {
       q.push({x, y});
   } else {
       h(A, x + 1, y, v, q), h(A, x - 1, y, v, q), h(A, x, y + 1, v, q), h(A, x, y - 1, v, q);
   }
}
int shortestBridge(vector<vector<int>>& A) {
   int row = A.size(), col = A[0].size(), ans = 0;
   vector<vector<int>> visited(row, vector<int>(col));
   queue<pair<int, int>> q;
   for (int i  = 0; q.empty() && i < row; ++i) {       // since only 2 islands, break if found one.
       for (int j = 0; q.empty() && j < col; ++j) {
           if (A[i][j]) h(A, i, j, visited, q);
       }
   }
   while (!q.empty()) {
       ++ans;
       for (int i = q.size(); i > 0; --i) {
           auto p = q.front(); q.pop();
           for (auto& dir : vector<pair<int, int>>{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}) {
               int x = p.first + dir.first, y = p.second + dir.second;
               if (x < 0 || x >= row || y < 0 || y >= col || visited[x][y]) continue;
               if (1 == A[x][y]) return ans;
               visited[x][y] = true, q.push({x, y});
           }
       }
   }
   return ans;
}
```

```
// 959 - regions cut by slashes
// soln-1: grid traversal (each traversal will mark a region)
// 1. dividing one cell into 4 direction, each cell will have 3 neighbors, try to access them.
//       0
//     3   1
//       2
```

```
void h(vector<string>& g, int x, int y, int z, vector<vector<int>>& v) {
   if (x < 0 || x >= g.size() || y < 0 || y >= g[0].length() || (v[x][y] & (1 << z))) return;
   v[x][y] |= (1 << z);
   switch (z) {
   case 0:
       h(g, x - 1, y, 2, v);
       if (g[x][y] != '\\') h(g, x, y, 3, v);
       if (g[x][y] != '/') h(g, x, y, 1, v);
       break;
   case 1:
       h(g, x, y + 1, 3, v);
       if (g[x][y] != '\\') h(g, x, y, 2, v);
       if (g[x][y] != '/') h(g, x, y, 0, v);
       break;
   case 2:
       h(g, x + 1, y, 0, v);
       if (g[x][y] != '\\') h(g, x, y, 1, v);
       if (g[x][y] != '/') h(g, x, y, 3, v);
       break;
   case 3:
       h(g, x, y - 1, 1, v);
       if (g[x][y] != '\\') h(g, x, y, 0, v);
       if (g[x][y] != '/') h(g, x, y, 2, v);
       break;
   }
}

int regionsBySlashes(vector<string>& grid) {
   int row = grid.size(), col = grid[0].length(), ans = 0;
   vector<vector<int>> visited(row, vector<int>(col));
   for (int i = 0; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           if (0 == (visited[i][j] & (1 << 0))) ++ans, h(grid, i, j, 0, visited);
           if (0 == (visited[i][j] & (1 << 2))) ++ans, h(grid, i, j, 2, visited);
       }
   }
   return ans;
}
```

Ref: #286 #296

318. Maximum product of word lengths

Given a string array `words`, find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lowercase letters. If no such two words exist, return 0.

Example 1:

Given `["abcw", "baz", "foo", "bar", "xtfn", "abcdef"],` Return `16`

The two words can be `"abcw", "xtfn"`.

Example 2:

Given `["a", "ab", "abc", "d", "cd", "bcd", "abcd"],` Return `4`

The two words can be `"ab", "cd"`.

Example 3:

Given `["a", "aa", "aaa", "aaaa"],` Return `0`

No such pair of words.

Solution:

soln-1: bit manipulation to mark occurrence + brute force to get answer

1. O(n^2) time to get product for those words which do not share common letters

2. how to identify if 2 strings are sharing common letters or not?
    a. use hashset for one string, then check it for each letter in other string
    b. for this question, only lowercase letters, that means only 26 bits needed, so that's bit manipulation.

```
// soln-1: use bit manipulation to get occurrence then brute force
int maxProduct(vector<string>& words) {
   vector<int> letterMask(words.size());
   for (int i = 0; i < words.size(); ++i) {
       for (char ch : words[i]) letterMask[i] |= 1 << (ch - 'a');
   }

   int ans = 0;
   for (int i = 0; i < words.size(); ++i) {
       for (int j = i + 1; j < words.size(); ++j) {
           if (0 == (letterMask[i] & letterMask[j])) {
               ans = max(ans, int(words[i].length() * words[j].length()));
           }
       }
   }
   return ans;
}
```

Ref:

319/672/777. Bulb Switcher ... Brainteaser

```
// 319 - Bulb Switcher
// 672 - Bulb Switcher II
// 777 - Swap Adjacent in LR String
// 858 - Mirror Reflection
```

Ref:

320. Generalized abbreviation

Write a function to generate the generalized abbreviations of a word.

Example:

Given word = "word", return the following list (order does not matter):

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Solution:
soln-1: each index has two options: abbreviated or not, which will generate 2^n results.
f("word") = "w" + f("ord")
            = 1 + f("ord"), add the number if there is one.
f("") return {""} which is a vector with size 1.

```
// soln-1: backtracking enumerate - do abbr. or not for each word
//         f("word") = w + f("ord")
//                   = abbr(f("ord"))
vector<string> generateAbbreviations(string word) {
   vector<string> ans;
   return helper(0, word);
}

string getAbbr(const string& str) {
   if (str[0] >= '1' && str[0] <= '9') {
       int n = stoi(str);
       return to_string(n + 1) + str.substr(to_string(n).length());
   }
```

```
    return "1" + str;
}

vector<string> helper(int start, const string& word) {
    if (start >= word.length()) return vector<string>{""};      // NOT empty vector

    vector<string> ans;
    for (auto& s : helper(start + 1, word)) {
        ans.push_back(word[start] + s); // no abbrevation
        ans.push_back(getAbbr(s));      // do abbr.
    }
    return ans;
}

// soln-2: similar to subset iterative soln
vector<string> generateAbbreviations(string word) {
    vector<string> ans;
    ans.push_back("");

    for (int i = 0; i < word.length(); ++i) {
        for (int j = ans.size() - 1; j >= 0; --j) {
            ans.push_back(ans[j] + word[i]);     // no abbr
            abbr(ans[j]);                        // in-place abbr
        }
    }
    return ans;
}
// if empty or not ending with digits, then attach "1"
// else +1 to the ending number
void abbr(string& word) {
    if (word.empty() || word.back() < '0' || word.back() > '9') {
        word.push_back('1');
    } else {
        int idx = word.length() - 1;
        for (int i = idx; i >= 0 && isdigit(word[i]); --i) idx = i;
        word = word.substr(0, idx) + to_string(atoi(word.substr(idx).c_str()) + 1);
    }
}
```

Ref: #90

321/670. Create maximum number / Maximum swap (review)

321 - Given two arrays of length `m` and `n` with digits `0-9` representing two numbers. Create the maximum number of length `k <= m + n` from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the `k` digits.

Note: You should try to optimize your time and space complexity.

Example:

`Input: nums1 = [3, 4, 6, 5], nums2 = [9, 1, 2, 5, 8, 3], k = 5, Output: [9, 8, 6, 5, 3]`

`Input: nums1 = [3, 9], nums2 = [8, 9], k = 3, Output: [9, 8, 9]`

670 - Given a non-negative integer, you could swap two digits at most once to get the maximum valued number. Return the maximum valued number you could get.

Example 1:

```
Input: 2736, Output: 7236

Explanation: Swap the number 2 and the number 7.
```

```
// soln-1: brute force
// get max number from nums1 with length-i, and nums2 with length k-i, and combine them
// then question translates to get max subsequence with given length.

// keep a window with size-k, looking from end to begin, the pattern in window
// should be w0 >= w1 >= w2 >= ... >= wk
// optimization: cache k-size window for the 1st run with O(nk) time
//               for any further subwindow with size less than k, drop one from previous
//               result, runs in O(k) time - just check the above pattern.
vector<int> maxSubsequence(const vector<int>& nums, int k) {
   if (k <= 0 || nums.size() < k) return vector<int>();

   vector<int> ans(nums.end() - k, nums.end());
   for (int i = nums.size() - k - 1; i >= 0; --i) {
       if (nums[i] >= ans[0]) ans.insert(ans.begin(), nums[i]);

       for (int j = 1; j < ans.size() && ans.size() > k; ++j) {
           if (ans[j] > ans[j - 1]) {
               ans.erase(ans.begin() + j - 1); break;
           } else if (j == ans.size() - 1) {
               ans.pop_back();                break;
           }
       }
   }
   return ans;
}

// [6, 7], [6, 5]  ==> [6, 7, 6, 5]
vector<int> merge(vector<int>& nums1, vector<int>& nums2) {
   vector<int> ans;
   while (nums1.size() + nums2.size()) {
       vector<int>& now = nums1 > nums2 ? nums1 : nums2;   // greedy pickup from bigger array
       ans.push_back(now[0]);
       now.erase(now.begin());
   }
   return ans;
}

vector<int> maxNumber(vector<int>& nums1, vector<int>& nums2, int k) {
   int n1 = nums1.size(), n2 = nums2.size();

   vector<int> ans(k);
   for (int k1 = min(k, n1); k1 >= 0; --k1) {
       vector<int> s1 = maxSubsequence(nums1, k1);
       vector<int> s2 = maxSubsequence(nums2, k - k1);
       if (s1.size() + s2.size() < k) continue;

       ans = max(ans, merge(s1, s2));
   }
   return ans;
}
```

```
// 670 - soln-1: backward scan in O(n)
int maximumSwap(int num) {
   auto ss = to_string(num);
   int mx = -1, mx_idx = -1, right = -1, left = -1;
   for (int i = ss.length() - 1; i >= 0; --i) {
       if (ss[i] > mx) {
           mx = ss[i], mx_idx = i;
       } else if (ss[i] < mx) {
           left = i, right = mx_idx;
       }
   }
```

```
    if (left == -1) return num;
    swap(ss[left], ss[right]);
    return stoi(ss);
}
```

Ref:

322/518. Coin Change / II

given infinite number of each kind of coin, and a total amount of money, compute the ***fewest number of coins*** that you need to make up that amount. Return -1 if there is no any combination of the coins.

#518 - coin change 2
You are given coins of different denominations and a total amount of money. Write a function to compute the number of combinations that make up that amount. You may assume that you have infinite number of each kind of coin.

Example 1:

```
Input: amount = 5, coins = [1, 2, 5], Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

#416/494 - partition equal sum / target sum
#474 - ones and zeros

```
// soln-1: dynamic programming
// let dp(i) be the min change for i-amount of money
// dp(i) = min{1 + dp(i - c) | for each c in coins}
int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, -1);
    dp[0] = 0;
    for (int c : coins) {
        for (int i = 1; i <= amount; ++i) {
            if (i < c || dp[i - c] == -1) continue;
            dp[i] = (dp[i] == -1 ? 1 + dp[i - c] : min(dp[i], 1 + dp[i - c]));
        }
    }
    return dp[amount];
}
```

```
// soln-1: dynamic programming
// let dp(i, j) be the way to change j-amount of money with i-coins, then we have 2 choices:
// 1, do not use ith coin - dp(i - 1, j)
// 2, using ith coin      - dp(i, j - c[i])  <=== NOT i-1 as we have infinite amount of ith-coin!
// dp(i, j) = dp(i - 1, j) + dp(i, j - c[i])
// space optimization is tricky - iterate from left to right!
int change(int amount, vector<int>& coins) {
    vector<int> dp(amount + 1);
    dp[0] = 1;                  // 0 amount and no coin
    vector<int> dp2 = dp;
    for (int coin : coins) {
        for (int i = coin; i <= amount; ++i) {  // catch: to get dp(i, j-c), we need to increase amount
            dp[i] = dp[i] + dp[i - coin];       // dp(i-1, j) + dp(i, j-c)
        }
    }
    return dp[amount];
```

```
}
```

Ref: #39 #216 #416 #474

323/990. Connected components in graph/Equality equations

```
// 323 - Undirected graph component
// soln-1: union-find (much faster than bfs/dfs)
int countComponents(int n, vector<pair<int, int>>& edges) {
    UnionFind uf(n);

    int ans = n;
    for (auto e: edges) {
        int p = uf.find(e.first), q = uf.find(e.second);
        if (p != q) {
            uf.unite(e.first, e.second);
            --ans;
        }
    }
    return ans;
}
```

```
// 990 - equality equations
// soln-1: union-find
bool equationsPossible(vector<string>& equations) {
    UnionFind uf(256);
    for (auto& eq : equations) {
        if ('=' == eq[1]) uf.u(eq.front(), eq.back());
    }
    for (auto& eq : equations) {
        if ('!' == eq[1]) {
            if (uf.f(eq.front()) == uf.f(eq.back())) return false;
        }
    }
    return true;
}
```

Ref: #200 #216 #261 #305 #547

325/523/525/560/930/974. Max size subarray sum equal K/n*K/K/divisible by K (review)

325 - Given an array of numbers and a target value k, find the maximum length of a subarray that **sums to k**. If there isn't one, return 0 instead.
E.g. `[1, -1, 5, -2, 3], k = 3.` Return 4 for the subarray [1, -1, 5, -2] sums to 3 and is the longest.
`[-2, -1, 2, 1], k = 1.` Return 2 for the subarray [-1, 2] sums to 1 and is the longest.

523/560 - Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of **k**, that is, sums up to n*k where n is also an **integer**.

Example 1:

Input: `[23, 2, 4, 6, 7], k=6,` **Output:** `True`

Explanation: `Because [2, 4] is a continuous subarray of size 2 and sums up to 6.`

```
// 325 - max length subarray that sums to k.
// soln-1: hashmap to keep sum so far
int maxSubArrayLen(vector<int>& nums, int k) {
   unordered_map<int, int> m;  // sum-index map

   // when we want to find a position that sums to 0,
   // position -1 should give us the longest subarray.
   m[0] = -1;
```

```
   int ans = 0;
   for (int sum = 0, i = 0; i < nums.size(); ++i) {
       sum += nums[i];
       if (m.find(sum) == m.end()) m[sum] = i;     // to get longest, keep index only if sum does not exist

       auto it = m.find(sum - k);                  // try to find previous sum with K difference
       if (it != m.end())  ans = max(ans, i - it->second);
   }

   return ans;
}
```

```
// 523 - subarray sums to n*k.
// soln-1: dynamic programming
// . . . . . . . . . . . . . . . .
//     | \-  k  -/
//     ^         ^
//     i         j
// if sum(0, i) % k == sum(0, j) % k, then A[i+1, j] must be sum to k.
bool checkSubarraySum(vector<int>& nums, int k) {
   unordered_map<int, int> mp;         // <sum % k, position>
   mp.insert({0, -1});

   for (int i = 0, sum = 0; i < nums.size(); ++i) {
       sum += nums[i];
       if (k) sum %= k;

       if (mp.find(sum) != mp.end()) {
           if (mp[sum] + 1 < i) return true;
       } else {
           mp[sum] = i;
       }
   }
   return false;
}
```

```
// 525 - Contiguous Array (find max length subarray with equal number of 0 and 1)
// soln-1: hashmap
// essentially it's same as #325 where 0 -> -1, k = 0.
int findMaxLength(vector<int>& nums) {
    int ans = 0;
    for (auto& x : nums) if (!x) x = -1;
    unordered_map<int, int> m;
    m[0] = -1;
    for (int i = 0, sum = 0; i < nums.size(); ++i) {
        sum += nums[i];
        if (m.find(sum) == m.end()) m[sum] = i;     // keep the 1st occurrance to get longest

        ans = max(ans, i - m[sum]);
    }
    return ans;
}
```

```
// #560 - subarray sum equals K (find all subarray sum equals K)
// soln-1: hashmap
// essentially it is same as #523 - check if there is subarray sum equals K
//                          or #1 - find 2 numbers sum to x
int subarraySum(vector<int>& nums, int k) {
   int ans = 0;
   multiset<int> seen;
   seen.insert(0);
   for (int i = 0, sum = 0; i < nums.size(); ++i) {
       sum += nums[i], ans += seen.count(sum - k);
       seen.insert(sum);
```

```
    }
    return ans;
}
```

```
// 930 - binary subarrays with sum
// soln-1: hashmap (prefix sum)
// exactly same as #560, but we could simply use vector instead of multiset for this question.
//
// soln-2: instead of considering substring, consider each 1's contribution in a substring
// for example, [1, 0, 1, 0, 1, 0, 0, 1], S = 2. consider 3rd 1s contribution:
// - find (S - 1) 1s from my left side at index j, then find out 0s aside j and cur.
// - (1 + 1) * (1 + 2) = 6, which are:
//   0, 1, 0, 1,          0, 1, 0, 1, 0         0, 1, 0, 1, 0, 0
//   1, 0, 1              1, 0, 1, 0            1, 0, 1, 0, 0
int numSubarraysWithSum(vector<int>& A, int S) {
    int ans = 0;
    vector<int> seen(A.size() + 1);
    seen[0] = 1;
    for (int i = 0, sum = 0; i < A.size(); ++i) {
        sum += A[i];
        if ((sum - S) >= 0) ans += seen[sum - S];
        ++seen[sum];
    }
    return ans;
}
```

```
// 974 - subarray sums divisible by k
// refer to #523 - subarray sums to n*k
int subarraysDivByK(vector<int>& A, int K) {
    unordered_map<int, int> seen;           // <sum%k, occurrence>
    seen[0] = 1;
    int ans = 0, sum = 0;
    for (auto x : A) {
        sum += (x > 0) ? x : (x % K + K);   // different case when x < 0
        sum %= K;

        if (seen.find(sum) != seen.end()) ans += seen[sum];
        seen[sum]++;
    }
    return ans;
}
```

```
// 1031 - Maximum Sum of Two Non-Overlapping Subarrays
// soln-1: prefix sum + sliding window
int maxSumTwoNoOverlap(vector<int>& A, int L, int M) {
    for (int i = 1; i < A.size(); ++i) A[i] += A[i - 1];
    int ans = A[L + M - 1], Lmax = A[L - 1], Mmax = A[M - 1];
    for (int i = L + M; i < A.size(); ++i) {
        Lmax = max(Lmax, A[i - M] - A[i - M - L]);
        Mmax = max(Mmax, A[i - L] - A[i - L - M]);
        ans = max(ans, max(Lmax + A[i] - A[i - M], Mmax + A[i] - A[i - L]));
    }
    return ans;
}
```

Ref: #209

328/725. Odd even linked list / Split linked list in parts

Given a singly linked list, group all odd nodes together followed by even nodes. For example, given 1->2->3->4->5->NULL, return 1->3->5->2->4->NULL.

```
ListNode* oddEvenList(ListNode* head) {
    if (head == NULL) return head;
```

```
    ListNode* odd = head, *even = head->next, *evenhead = head->next;
    helper(odd, even);
    odd->next = evenhead;

    return head;
}

// recursion - even pointer works as faster pointer
void helper(ListNode* &odd, ListNode* &even) {
    if (even && even->next) {
        odd->next = odd->next->next;
        even->next = even->next->next;

        helper(odd->next, even->next);
    }
}

// iterative version
ListNode* oddEvenList(ListNode* head) {
    if (head == NULL) return head;

    ListNode* odd = head, *even = head->next, *evenhead = head->next;
    while (even && even->next) {
        odd->next = odd->next->next;
        even->next = even->next->next;

        odd = odd->next; even = even->next;
    }
    odd->next = evenhead;
    return head;
}
```

```
// 725: split linked list in parts
ListNode* helper(ListNode* &h, int& length, int k) {
   ListNode dummy(0);
   ListNode* cur = &dummy;
   int len = length / k + (length % k == 0 ? 0 : 1);
   for (int i = 0; h && i < len; ++i) {
       cur->next = h, h = h->next, cur = cur->next, cur->next = nullptr, --length;
   }
   return dummy.next;
}

vector<ListNode*> splitListToParts(ListNode* root, int k) {
   vector<ListNode*> ans;
   int length = len(root);
   while (k) ans.push_back(helper(root, length, k--));

   return ans;
}
```

Ref:

329. Longest increasing path in matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary (i.e. wrap-around is not allowed).

Example 1:

```
nums = [
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
```

Example 2:

```
nums = [
  [3,4,5],
  [3,2,6],
  [2,2,1]
]
```

Return 4 The longest increasing path is [1, 2, 6, 9].	Return 4 The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.

Solution:

Key ideas:

1) depth-first search is obvious. Do we need visited flag matrix? No, because we will access only bigger neighbors.
2) if m[i][j] is going to access m[i-1][j] and m[i-1][j] has been accessed before. Do we have to access it again?
3) this technique is called 'memoization/cache' even Dynamic Programming?
4) similar to all depth-first search questions, such as #332

```
// O(m*n) time and space with memorization
int helper(vector<vector<int>>& m, int x, int y, vector<vector<int>>& cache) {
   if (cache[x][y]) return cache[x][y];

   static vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
   for (auto& dir : dirs) {
       int nx = dir.first + x, ny = dir.second + y;
       if (nx >= cache.size() || nx < 0 || ny >= cache[0].size() || ny < 0) continue;

       if (m[nx][ny] > m[x][y])    cache[x][y] = max(cache[x][y], 1 + helper(m, nx, ny, cache));
   }
   return ++cache[x][y];
}

int longestIncreasingPath(vector<vector<int>>& matrix) {
   if (matrix.size() == 0) return 0;
   int row = matrix.size(), col = matrix[0].size();

   int ans = 0;
   vector<vector<int>> cache(row, vector<int>(col, 0));
   for (int i = 0; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           ans = max(ans, helper(matrix, i, j, cache));
       }
   }
   return ans;
}
```

Ref: #332

330. Patching array

Solution:

Ref:

332/586/743/787. Max vacation days/Network Delay time/Cheapest flights within K stops (review)

```
// 332 - reconstruct itenarary (starting from given airport, dfs edge traversal)
// soln-1: dfs (remove each visited edge)
typedef priority_queue<string, vector<string>, greater<string>> t_minheap;
typedef unordered_map<string, t_minheap> t_graph;    // edge is sorted

void dfs(const string& departure, t_graph& g, vector<string>& ans) {
   t_minheap& neighbors = g[departure];
```

```
    while(!neighbors.empty()) {
        string next = neighbors.top();
        neighbors.pop();                              // remove this edge since visited
        dfs(next, g, ans);
    }

    ans.insert(ans.begin(), departure);
}

vector<string> findItinerary(vector<pair<string, string>> tickets) {
    t_graph g;
    for (auto& e : tickets) g[e.first].push(e.second);

    vector<string> ans;
    dfs("JFK", g, ans);
    return ans;
}
```

```
// 586 - max vacation days
// TODO
```

```
// 743 - network delay time (starting from node-k to all others)
// soln-1: BFS (Dijkstra requires heap to pick candidate)
int networkDelayTime(vector<vector<int>>& times, int N, int K) {
    typedef unordered_map<int, unordered_map<int, int>> t_graph;
    t_graph g;
    for (vector<int> e : times) g[e[0]][e[1]] = e[2];

    vector<int> delay(N + 1, INT_MAX);
    queue<int> q;   q.push(K);  delay[K] = 0;
    while (!q.empty()) {
        unordered_set<int> s;
        for (int n = q.size(); n > 0; --n) {    // This BFS, Dijkstra only picks one
            int u = q.front(); q.pop();

            for (pair<int, int> neighbor : g[u]) {
                int v = neighbor.first, w = neighbor.second;
                if (delay[v] > w + delay[u]) {
                    delay[v] = w + delay[u];
                    if (s.find(v) == s.end()) q.push(v);
                }
            }
        }
    }

    int maxDelay = 0;
    for (int i = 1; i <= N; ++i) maxDelay = max(maxDelay, delay[i]);
    return maxDelay == INT_MAX ? -1 : maxDelay;
}

// 743 - network delay time (starting from node-k to all others)
// soln-2: Bellman Ford
int networkDelayTime(vector<vector<int>>& times, int N, int K) {
    vector<int> dist(N + 1, INT_MAX);
    dist[K] = 0;
    for (int i = 0; i < N; i++) {
        for (vector<int> e : times) {
            int u = e[0], v = e[1], w = e[2];
            if (dist[u] != INT_MAX && dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
            }
        }
    }
```

```
    int maxwait = 0;
    for (int i = 1; i <= N; i++) maxwait = max(maxwait, dist[i]);
    return maxwait == INT_MAX ? -1 : maxwait;
}
```

```
// 787 - Cheapest Flights Within K Stops (fligts: <src, dst, cost>)
// soln-1: bfs
int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int K) {
    int ans = INT_MAX, stop = -1;
    vector<vector<int>> g(n, vector<int>{});
    for (int i = 0; i < flights.size(); ++i) g[flights[i][0]].push_back(i);
    queue<pair<int, int>> q;    q.push({src, 0});

    while (stop <= K && !q.empty()) {
        for (int i = q.size(); i > 0; --i) {
            auto fr = q.front();    q.pop();
            if (fr.first == dst) {
                ans = min(ans, fr.second);
            } else {
                for (auto nxt : g[fr.first]) {
                    if (flights[nxt][1] == fr.first) continue;
                    if (INT_MAX != ans &&  fr.second + flights[nxt][2] >= ans) continue;
                    q.push({flights[nxt][1], fr.second + flights[nxt][2]});
                }
            }
        }
        ++stop;
    }
    return INT_MAX == ans ? -1 : ans;
}
```

```
// 787 - Cheapest Flights Within K Stops (fligts: <src, dst, cost>)
// soln-2: dijkstra (slow than soln-1)
int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int K) {
    vector<vector<int>> g(n, vector<int>{});
    for (int i = 0; i < flights.size(); ++i) g[flights[i][0]].push_back(i);

    auto cmp = [](vector<int>& a, vector<int>& b) {
        return a[1] > b[1];     // <dst, cost, stops>
    };
    priority_queue<vector<int>, vector<vector<int>>, decltype(cmp)> pq(cmp);
    pq.push({src, 0, -1});
    while (!pq.empty()) {
        auto n = pq.top();  pq.pop();
        auto cur = n[0], cost = n[1], stop = n[2];
        if (dst == cur) return cost;

        for (auto nxt : g[cur]) {
            if (flights[nxt][1] == cur) continue;
            if (stop + 1 > K) continue;
            pq.push({flights[nxt][1], cost + flights[nxt][2], stop + 1});
        }
    }
    return -1;
}
```

Ref:

334. Increasing Triplet Subsequence (review)

given an unsorted array, return true if exist an increasing subsequence of length 3 in the array.

Solution:

At first glance, this question looks like *longest increasing subsequence* with length >= 3.

Better soln even for generalized question - *determine if there is increasing subsequence with length k*.

```
bool increasingTriplet(vector<int>& nums) {
    int small = INT_MAX, big = INT_MAX;
    for (int x : nums) {
        if (x <= small)     small = x;  // the 1st best candidate, not necessarily in the target triplet
        else if (x <= big)  big = x;    // the 2nd best candidate(x > small). consider case: [1, 2, 0, 4]
        else return true;               // we already have/had increasing pair, and this is 3rd one.
    }

    return false;
}

// soln-1: LBYL (Look before you leap) in O(nlgk)
// essentially it's same as above but generalized
bool increasingK(vector<int>& nums, int k) {
    vector<int> candidate(k - 1, INT_MAX);
    for (int x : nums) {
        auto p = lower_bound(candidate.begin(), candidate.end(), x);
        if (p == candidate.end()) return true;
        *p = x;
    }
    return false;
}
```

Ref: #300

335. Self Crossing

Solution:

Ref:

336. Palindromic Pairs

Given a list of unique words. Find all pairs of ***distinct*** indices `(i, j)` in the given list, so that the concatenation of the two words, i.e. `words[i] + words[j]` is a palindrome.

Example 1:

Given `words` = `["bat", "tab", "cat"],` Return `[[0, 1], [1, 0]]`

The palindromes are `["battab", "tabbat"]`

Example 2:

Given `words` = `["abcd", "dcba", "lls", "s", "sssll"],` Return `[[0, 1], [1, 0], [3, 2], [2, 4]]`

The palindromes are `["dcbaabcd", "abcddcba", "slls", "llssssll"]`

Solution:
soln-1: brute force
1. combine every two pairs of words, then check
2. time complexity O(k * n^2), k is the average length of palindrome.

soln-2: hashmap
1. split each word: s = s1 + s2
2. if s1 is palindrome, then find “2s” to form 2s-s1-s2
3. if s2 is palindrome, then find “1s” to form 1s-s2-s1

4. time complexity: O(k * n), k is the average length of word

```
// soln-2: partition each word then find the reverse part
vector<vector<int>> palindromePairs(vector<string>& words) {
   unordered_map<string, int> w2i;
   for (int i = 0; i < (int)words.size(); ++i)  w2i[words[i]] = i;

   vector<vector<int>> ans;
   for (int i = 0; i < (int)words.size(); ++i) {
       string& w = words[i];
       for (int j = 0; j <= (int)w.length(); ++j) {            // partition word: empty ~ fullword
           string s1 = w.substr(0, j), s2 = w.substr(j);

           if (isPal(s1)) {                                    // "", "bat"
               string rs2(s2.rbegin(), s2.rend());             // "tab"
               auto iter = w2i.find(rs2);
               if (iter != w2i.end() && iter->second != i) {
                   ans.push_back({iter->second, i});           // 2s-s1-s2
               }
           }
           if (isPal(s2) && !s2.empty()) {                     // Tricky! avoid dups s2 not empty
               string rs1(s1.rbegin(), s1.rend());
               auto iter = w2i.find(rs1);
               if (iter != w2i.end() && iter->second != i) {
                   ans.push_back({i, iter->second});          // s1-s2-1s
               }
           }
       }
   }
   return ans;
}

bool isPal(string s) {
   for (int i = 0, j = s.length() - 1; i < j; ++i, --j) {
       if (s[i] != s[j]) return false;
   }
   return true;
}
```

Ref: #5

338. Counting Bits

Given a non negative integer number **num**. For every numbers **i** in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example:

For `num = 5` you should return `[0,1,1,2,1,2]`.

Follow up:

- It is very easy to come up with a solution with runtime **O(n*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like **__builtin_popcount** in c++ or in any other language.

Solution:

dp: f(n) = f(n >> 1) + n & 1

```
vector<int> countBits(int num) {
    vector<int> ans(nums + 1);
    for (int i = 1; i <= nums; ++i) {
        ans[i] = ans[i >> 1] + (i & 1);
```

```
    }
    return ans;
}
```

Ref: #191

339/364/341. Nested List Weight Sum/II/Iterator of nested list

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.
Each element is either an integer, or a list -- whose elements may also be integers or other lists.
Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:
Given the list `[[1,1],2,[1,1]]`, return **8**. (four 1's at depth 1, one 2 at depth 2)
Example 2:
Given the list `[1,[4,[6]]]`, return **17**. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1*3 + 4*2 + 6*1 = 17)

```
struct NestedInteger {
   int _isInteger, _val;
   vector<NestedInteger> _ni;

   NestedInteger(int val) : _isInteger(1), _val(val) {}
   NestedInteger(const vector<NestedInteger>& ni) : _isInteger(0) { _ni = ni; }
   bool isInteger() const { return _isInteger; }
   int getInteger() const { return _val; }
   const vector<NestedInteger>& getList() const { return _ni; }
};

// 339 - soln-1: dfs
int depthSum(const vector<NestedInteger>& nestedList, int depth = 1) {
   int ans = 0;
   for (const auto& nl : nestedList) {
       ans += nl.isInteger() ? nl.getInteger() * depth : depthSum(nl.getList(), depth + 1);
   }
   return ans;
}
```

```
// 364 - soln-1: dfs
void dfs(const vector<NestedInteger>& nestedList, int d, vector<int>& levelSum) {
   if (levelSum.size() <= d) levelSum.push_back(0);
   for (const auto& nl : nestedList) {
       if (nl.isInteger()) {
           levelSum[d] += nl.getInteger();
       } else {
           dfs(nl.getList(), d + 1, levelSum);
       }
   }
}
int depthSumInverse(vector<NestedInteger> nestedList) {
   int ans = 0;
   vector<int> sum;
   dfs(nestedList, 1, sum);
   for (int level = sum.size(), i = 0; i < sum.size(); ++i) {
       ans += level * sum[i];
   }
   return ans;
}
```

```
// 341 - soln-1: stack
class NestedIterator {
```

```
    using iter = vector<NestedInteger>::const_iterator;
    stack<pair<iter, iter>> _stk;   // holding current and end iterator for current level
public:
    NestedIterator(vector<NestedInteger> &nestedList) {
        if (!nestedList.empty()) {
            _stk.push({nestedList.begin(), nestedList.end()});
        }
    }

    int next() {
        auto& t = _stk.top();
        int x = t.first->getInteger();
        t.first++;
        return x;
    }

    bool hasNext() {
        while (!_stk.empty()) {
            auto& t = _stk.top();
            if (t.first == t.second) {
                _stk.pop();
            } else {
                if (t.first->isInteger()) return true;
                _stk.push({t.first->getList().begin(), t.first->getList().end()});
                t.first++;
            }
        }
        return false;
    }
};
```

Ref: #332

343. Integer break

Given a positive integer *n*, break it into the sum of **at least** two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given *n* = 2, return 1 (2 = 1 + 1); given *n* = 10, return 36 (10 = 3 + 3 + 4).

Note: You may assume that *n* is not less than 2 and not larger than 58.

```
// soln-1: dynamic programming
// f(i) = max{ f(k) * (i - k) | k = 2~i-1}
int integerBreak(int n) {
    vector<int> dp(n + 1, 1);
    for (int i = 3; i <= n; ++i) {
        for (int k = 2; k < i; ++k) {
            dp[i] = max(dp[i], k * max(dp[i - k], i - k));      // dp[i-k] could be less than i-k
        }
    }
    return dp[n];
}
```

Ref:

344/345. Reverse string / reverse vowels of a string

trivial

```
string reverseVowels(string s) {
    helper(s, 0, s.length());
```

```
    return s;
}
void helper(string& s, int start, int end) {
    while (start < end && !isVowel(s[start])) ++start;
    while (start < end && !isVowel(s[end])) --end;        // reduce the depth of call stack
    if (start >= end) return;

    swap(s[start], s[end]);
    helper(s, ++start, --end);
}
```

Ref:

346. Moving average from data stream

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

For example,

MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3

Solution:
A preallocated ring-buffer should be more efficient than dynamic memory allocation, such as queue which involves pop/push.

```
class MovingAverage {
   vector<int> m_buf;        // ring-buffer
   double m_sum;
   int m_curSize, m_curIdx;

public:
   // 12 / 12 test cases passed. Runtime: 56 ms
   MovingAverage(int size) {
      m_buf.assign(size, 0);
      m_curSize = m_curIdx = 0;
      m_sum = 0.0f;
   }

   double next(int val) {
      m_sum += val - m_buf[m_curIdx];
      m_buf[m_curIdx] = val;        // minus the value before overwriting

      m_curIdx = (m_curIdx + 1) % m_buf.size();
      if (m_curSize < m_buf.size()) ++m_curSize;

      return m_sum / m_curSize;
   }
};
```

Ref:

347/692. Top-k frequent elements/words

```
// 347/692 - Top K Frequent Words
// soln-1: priority queue
vector<string> topKFrequent(vector<string>& words, int k) {
   unordered_map<string, int> cnt;
   for (auto& str : words) cnt[str]++;
```

```cpp
    auto cmp = [](const pair<string, int>& a, const pair<string, int>& b) {
        return a.second > b.second || (a.second == b.second && a.first < b.first);
    };
    priority_queue<pair<string, int>, vector<pair<string, int>>, decltype(cmp)> minh(cmp);
    for (auto& kv : cnt) {
        minh.emplace(kv.first, kv.second);
        if (minh.size() > k) minh.pop();
    }
    vector<string> ans;
    while (!minh.empty()) ans.push_back(minh.top().first), minh.pop();
    reverse(ans.begin(), ans.end());
    return ans;
}
```

Ref: #4 #220

348/353/355/362/379/622/635/641/705/706/707. Design ...

```cpp
// 348 - design tic-tac-toe
// 1. since we have only 2 players, we can add 1 or -1 for row/col
//    when row[i] == n/-n, meaning there is a winner
class TicTacToe {
public:
    TicTacToe(int n): rows(n), cols(n), N(n), diag(0), rev_diag(0) {}

    // return 0 - no one wins, else return the winner 1/2
    int move(int row, int col, int player) {
        int add = player == 1 ? 1 : -1;
        rows[row] += add, cols[col] += add;
        diag += (row == col ? add : 0), rev_diag += (row == N - col - 1 ? add : 0);
        return (abs(rows[row]) == N || abs(cols[col]) == N || abs(diag) == N || abs(rev_diag) == N) ? player : 0;
    }

private:
    vector<int> rows, cols;
    int diag, rev_diag, N;
};
```

```cpp
// 794 - Valid Tic-Tac-Toe State
// 1. As per the game rule, X goes first, so the board has at most 1 more X
// 2. Per problem statement, board is invalid if:
//     - place one more 'X' and it wins (game over)
//     - place one more 'O' and it wins (game over)
// 3. board is valid iff it has 1 more X, and place 1 more X/O and game can continue.
bool isWin(vector<string>& m, char ch) {
    if (m[0][0] == m[0][1] && m[0][0] == m[0][2] && m[0][0] == ch) return true;
    if (m[1][0] == m[1][1] && m[1][0] == m[1][2] && m[1][0] == ch) return true;
    if (m[2][0] == m[2][1] && m[2][0] == m[2][2] && m[2][0] == ch) return true;

    if (m[0][0] == m[1][0] && m[0][0] == m[2][0] && m[0][0] == ch) return true;
    if (m[0][1] == m[1][1] && m[0][1] == m[2][1] && m[0][1] == ch) return true;
    if (m[0][2] == m[1][2] && m[0][2] == m[2][2] && m[0][2] == ch) return true;

    if (m[0][0] == m[1][1] && m[0][0] == m[2][2] && m[0][0] == ch) return true;
    if (m[0][2] == m[1][1] && m[0][2] == m[2][0] && m[0][2] == ch) return true;
    return false;
}
bool validTicTacToe(vector<string>& board) {
    int countx = 0, counto = 0;
    for (auto& b : board) {
        for (auto& ch : b) {
            if ('X' == ch) countx++;
            else if ('O' == ch) counto++;
    }
    if (countx == counto) {
```

```
        return !isWin(board, 'X');  // not a valid state if put extra 'X' and it wins
    } if (countx == counto + 1) {   // X goes first, so it has at most 1 more
        return !isWin(board, 'O');  // not a valid state if put extra 'O' and it wins
    } else {
        return false;
    }
}
```

```
// 353 - Design snake game
// soln-1: deque + hashset
class SnakeGame {
   int _rows, _cols, _foodIdx;
   vector<pair<int, int>> _food;
   deque<pair<int, int>> _q;       // add a new position and remove last one if no food found
   unordered_set<string> _pos;     // snake positions used to check collision
public:
   SnakeGame(int width, int height, vector<pair<int, int>> food) {
       _rows = height, _cols = width;
       _foodIdx = 0, _food = food;
       _q.push_front({0, 0}), _pos.insert("0,0");
   }

   int move(string direction) {    // return total food eaten
       auto cur = _q.front();
       if ("U" == direction)       --cur.first;
       else if ("L" == direction)  --cur.second;
       else if ("R" == direction)  ++cur.second;
       else if ("D" == direction)  ++cur.first;

       auto poshash = to_string(cur.first) + "," + to_string(cur.second);
       if (_pos.count(poshash) || cur.first >= _rows || cur.first < 0 || cur.second >= _cols || cur.second < 0)
return -1;

       if (_foodIdx < _food.size() && cur == _food[_foodIdx]) {
           ++_foodIdx;                             // eat food, don't pop_back
       } else {
           _pos.erase(poshash), _q.pop_back();     // eat nothing, remove tail
       }
       _q.push_front(cur), _pos.insert(poshash);

       return _q.size() - 1;
   }
};
```

```
// 355 - Design Twitter
// soln-1: hashmap + list
class Twitter {
    int _ts;
    unordered_map<int, unordered_set<int>> _f;  // <id, following>
    typedef list<pair<int, int>> tlist;         // <ts, tweet-id>
    unordered_map<int, tlist> _post;            // <uid, most-recent-10-tweets>
public:
    Twitter() : _ts(0) {
    }

    void postTweet(int userId, int tweetId) {
        _post[userId].push_front({++_ts, tweetId});
        if (_post[userId].size() > 10) _post[userId].pop_back();
    }

    vector<int> getNewsFeed(int userId) {       // return 10 most recent tweets
        typedef pair<int, tlist::iterator> pp;  // <ts, tweet-iterator>
        auto cmp = [](pp& a, pp& b) {
            return a.second->first < b.second->first;
        };
        priority_queue<pp, vector<pp>, decltype(cmp)> mh(cmp);
```

```
        if (!_post[userId].empty()) {
            mh.push({userId, _post[userId].begin()});
        }
        for (auto& follow: _f[userId]) {
            if (!_post[follow].empty()) {
                mh.push({follow, _post[follow].begin()});
            }
        }
        vector<int> ans;
        while (ans.size() < 10 && !mh.empty()) {
            auto x = mh.top();  mh.pop();
            auto it = x.second;                  // tweet-iterator
            ans.push_back(it->second);
            if (++it != _post[x.first].end()) mh.push({x.first, it});
        }
        return ans;
    }

    void follow(int followerId, int followeeId) {
        if (followerId != followeeId) _f[followerId].insert(followeeId);
    }

    void unfollow(int followerId, int followeeId) {
        _f[followerId].erase(followeeId);
    }
};
```

```
// 362 - Design Hit Counter
// soln-1: queue
// keep <timestamp, hit-count> in queue, if the timestamp is same, then increase hit-count
// improvement: since the size of Q is fixed (<300), use ring buffer to impl. Q
// soln-2: array
// 1. define <timestamp, hit-count> array with size of 300 (5-minutes)
// 2. when hit, update A[timestamp % 300], if the timestamp is same, ++count, else reset it.
class HitCounter {
   unsigned long m_counter;
   queue<pair<int, unsigned long>> m_hit;      // <timestamp, counter>
public:
   HitCounter() : m_counter(0) { }

   void hit(int timestamp) {
       if (!m_hit.empty() && m_hit.back().first == timestamp) {
           ++m_hit.back().second;               // in case hits per second could be very large
       } else {
           m_hit.push({timestamp, 1});
       }
       ++m_counter;
   }

   int getHits(int timestamp) {
       while (!m_hit.empty() && m_hit.front().first <= timestamp - 300) {
           m_counter -= m_hit.front().second, m_hit.pop();
       }
       return m_counter;
   }
};
```

```
// 379 - Design Phone Directory (support get/check/release)
// soln-1: array
// use 2 array to mark used/recycled
// 1. if there are recycled numbers available, return it first
// 2. when release a phone number, append it to recycled array
// TODO
```

```
// 622/641 - circular queue/deque
```

```cpp
class MyCircularQueue {
   // x x x x ? ? ?
   // ^       ^
   // front   rear                                // rear pointing to undefined space
   vector<int> _buff;
   int _front, _rear;
public:
   MyCircularQueue(int k) {
       _buff.assign(k + 1, 0), _front = _rear = -1;        // one more extra space
   }
   bool enQueue(int value) {
       if (isFull()) return false;
       if (isEmpty()) {                                    // starting from 0 if empty
           _buff[0] = value, _front = 0, _rear = 1;
       } else {
           _buff[_rear] = value, _rear = (_rear + 1) % _buff.size();
       }
       return true;
   }
   bool deQueue() {
       if (isEmpty()) return false;
       _front = (_front + 1) % _buff.size();
       return true;
   }
   int Front() {
       if (isEmpty()) return -1;
       return _buff[_front];
   }
   int Rear() {
       if (isEmpty()) return -1;
       return _buff[(_rear - 1 + _buff.size()) % _buff.size()];
   }
   bool isEmpty() {
       return _front == _rear;
   }
   bool isFull() {
       return (_rear + 1) % _buff.size() == _front;
   }
};
```

```cpp
// 535 - Design TinyURL TODO
```

```cpp
// 635 - Design Log Storage System
// support 2 methods:
//   put(1, "2017:01:01:23:59:59");
//   retrieve("2016:01:01:01:01:01","2017:01:01:23:00:00","Year");
//                 from_ts               to_ts       compare_granularity
class LogSystem {
   vector<pair<int, string>> _ts;
   vector<string> units;
   vector<int> indices;
public:
   LogSystem() {
       units = {"Year", "Month", "Day", "Hour", "Minute", "Second"};
       indices = {4, 7, 10, 13, 16, 19};
   }

   void put(int id, string timestamp) {
       _ts.push_back({id, timestamp});
   }

   vector<int> retrieve(string s, string e, string gra) {
       vector<int> ans;
       int idx = indices[find(units.begin(), units.end(), gra) - units.begin()];
       for (auto& p : _ts) {
           string& t = p.second;
```

```
            if (t.substr(0, idx).compare(s.substr(0, idx)) >= 0 &&
                t.substr(0, idx).compare(e.substr(0, idx)) <= 0) {
                ans.push_back(p.first);
            }
        }
        return ans;
    }
};
```

```
// 705/706 Design hashset/hashmap TODO
```

```
// 707 - Design linked list
template<class T>
class MyLinkedList {
   struct ListNode {
       ListNode* next;
       T val;
       ListNode(T val, ListNode* next=nullptr) : val(val), next(next){}
   };
   ListNode *_head, *_tail;

public:
   MyLinkedList() : _head(nullptr), _tail(nullptr) {}

   int get(int index) {
       ListNode* n = _head;
       for (int i = 0; n && i < index; ++i) {
           if (n) n = n->next;
       }
       return n ? n->val : -1;
   }

   void addAtHead(T val) {
       _head = new ListNode(val, _head);
       if (_tail == nullptr) _tail = _head;
   }

   void addAtTail(T val) {
       ListNode* n = new ListNode(val);
       if (_head == nullptr) _head = n;
       if (_tail) _tail->next = n;
       _tail = n;
   }

   void addAtIndex(int index, T val) {
       ListNode* cur = _head, *pre = nullptr;
       for (int i = 0; i < index; ++i) {
           if (cur == nullptr) return;     // out of index
           pre = cur, cur = cur->next;
       }
       ListNode* n = new ListNode(val, cur);
       if (pre) pre->next = n;
       if (cur == nullptr) _tail = n;
       if (pre == nullptr) _head = n;
   }

   void deleteAtIndex(int index) {
       ListNode* cur = _head, *pre = nullptr;;
       for (int i = 0; i < index; ++i) {
           if (cur == nullptr) return;
           pre = cur, cur = cur->next;
       }
       if (cur == _head) _head = _head->next;
       if (cur == _tail) _tail = pre;
       if (pre) pre->next = cur ? cur->next : nullptr;
       if (cur) delete cur;
```

```
    }
};
```

Ref: #51 #52

349/350. Intersection of two array / II

Given two arrays, write a function to compute their intersection.

Example: Given *nums1* = [1, 2, 2, 1], *nums2* = [2, 2], return [2].

```
// 350: soln-1: easy hashmap
vector<int> intersect(vector<int>& nums1, vector<int>& nums2) {
    unordered_map<int, int> mp;     // number -> count
    for (auto x : nums1) mp[x]++;

    vector<int> ans;
    for (auto x : nums2) {
        if (mp.find(x) == mp.end()) continue;
        ans.push_back(x);
        if (--mp[x] == 0) mp.erase(x);
    }
    return ans;
}
```

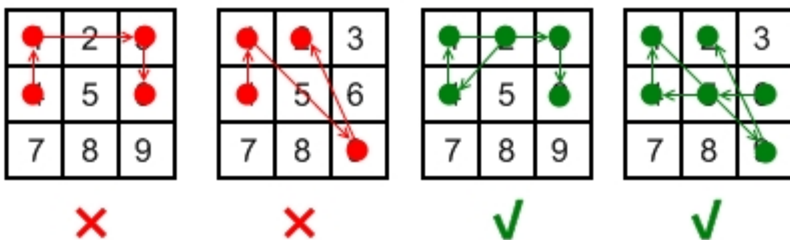
Ref: #350

351. Android unlock patterns

Given an Android **3x3** key lock screen and two integers **m** and **n**, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of **m** keys and maximum **n** keys.

Rules for a valid pattern:

1. Each pattern must connect at least **m** keys and at most **n** keys.
2. All the keys must be distinct.
3. If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
4. The order of keys used matters.



Explanation:

1	2	3
4	5	6
7	8	9

Invalid move: 4 - 1 - 3 - 6 Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: 4 - 1 - 9 - 2 Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: 2 - 4 - 1 - 3 - 6 Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: 6 - 5 - 4 - 1 - 9 - 2 Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example: Given **m** = 1, **n** = 1, return 9.

Solution:
This question asks the total number of unlock patterns from minimum m keys to maximum n keys. My first thought is to enumerate all possible combinations, if the number of key in the combinations is between m and n, then we find 1 result. So the time complexity is the sum of Permutation(nchoosek(9, i)), i = m~n.

The nasty thing in this question is how to prepare candidates. For example, if we start from key-1, the next possible key is {2, 6, 5, 8, 4} in clockwise direction, but if key-5 is used, then key-9 should be in the candidate set. In other words, the keys in candidate set are dynamically changing depending on current state, which keys are used. In this sense, it has the same attribute as N-Queen question (#51 #52), which requires to update the candidates during computation.

There is a better way to check if it is possible from A to B. By carefully looking at the keyboard, we find for most jumps from A to B is doable, only a few of them are blocked by other keys. The trick is when the blocked key is accessed, it will not block anything. To sum up, we have m[from][to] = block-key, e.g., m[1][3] = 2, when 2 is accessed, 1->3 is doable.

Improvement: keys-1/3/7/9 are symmetric, keys-2/4/6/8 too, we only need to compute 3 set, 1, 2, 5, instead of 9.

```
// 24 / 24 test cases passed. Runtime: 512 ms. (SLOW! But code is clean.)
int numberOfPatterns(int m, int n) {
   vector<vector<int>> bm(10, vector<int>(10, 0));
   blockMatrix(bm);

   int ans = 0;
   for (int i = 1; i <= 9; ++i) {
      vector<bool> used(10, false);
      used[i] = true;
      helper(i, m, n, 1, used, ans, bm);
   }

   return ans;
}

void blockMatrix(vector<vector<int>>& bm) {
   bm[1][3] = bm[3][1] = 2;
   bm[1][7] = bm[7][1] = 4;
   bm[1][9] = bm[9][1] = 5;
   bm[2][8] = bm[8][2] = 5;
   bm[3][7] = bm[7][3] = 5;
   bm[3][9] = bm[9][3] = 6;
   bm[4][6] = bm[6][4] = 5;
   bm[7][9] = bm[9][7] = 8;
}

void helper(int from, int m, int n, int cur, vector<bool>& used, int& ans, vector<vector<int>>& bm) {
   if (cur > n)    return;    // path is too long
   if (cur >= m)   ++ans;

   for (size_t to = 1; to <= 9; ++to) {
      if (used[to])             continue;  // skip used key
      int block = bm[from][to];            // A -> B has a block?
      if (block && !used[block])   continue;  // has block && not used

      used[to] = true;
      helper(to, m, n, cur + 1, used, ans, bm);
      used[to] = false;
   }
}
```

Ref: #51 #52

352/436/715. Intervals/Range add/query/Find right intervals (review)

```
// 352 - soln-1: segment tree merge/split
class SummaryRanges {
   map<int, int> _seg;
public:
   void addNum(int val) {
       auto next = _seg.upper_bound(val);
       if (next == _seg.begin()) {         // insert at beginning
           if (next->first == val + 1)     _seg[val] = next->second, _seg.erase(next);
           else _seg[val] = val;
       } else if (next == _seg.end()) {    // insert at ending
           auto pre = std::prev(next);
           if (pre->second + 1 == val)     pre->second++;
           else if (pre->second + 1 < val) _seg[val] = val;
       } else {                            // insert in between
           auto pre = std::prev(next);
           if (pre->second + 1 < val)      _seg[val] = val, ++pre;
           else if (pre->second + 1 == val) pre->second++;

           if (pre->second + 1 == next->first) {
               pre->second = next->second, _seg.erase(next);
           }
       }
   }

   vector<Interval> getIntervals() {
       vector<Interval> ans;
       for (auto& p : _seg) ans.push_back(Interval(p.first, p.second));
       return ans;
   }
};
```

```
// 436 - soln-1: bst to keep <left-boundary, index>
vector<int> findRightInterval(vector<Interval>& intervals) {
    vector<int> ans(intervals.size(), -1);
    vector<pair<int, int>> mp;              // <left-boundary, index>
    for (int i = 0; i < intervals.size(); ++i) mp.push_back({intervals[i].start, i});
    sort(mp.begin(), mp.end());

    for (int i = 0; i < intervals.size(); ++i) {
        int t = intervals[i].end, lo = 0, hi = intervals.size() - 1;
        while (lo + 1 < hi) {
            int m = lo + (hi - lo) / 2;
            mp[m].first < t ? lo = m : hi = m;
        }
        if (mp[lo].first >= t) ans[i] = mp[lo].second;
        else if (mp[hi].first >= t) ans[i] = mp[hi].second;
    }
    return ans;
}
```

```
// 715 - range module
// soln-1: segment tree merge/split with binary search
// 1. keep range in bst, merge it when add range, split/remove it when del range
// 2. binary search when query
class RangeModule {
    map<int, int> _seg;     // <left, right> of segment
public:
    void addRange(int left, int right) {
        auto it1 = _seg.upper_bound(left), it2 = _seg.upper_bound(right);
        if (it1 != _seg.begin()) {
            if ((--it1)->second < left) ++it1;    // move to the one needs to be removed
        }
        if (it1 != it2) {
            left = min(left, it1->first), right = max(right, (--it2)->second);
```

```
            _seg.erase(it1, ++it2);
        }
        _seg[left] = right;
    }

    bool queryRange(int left, int right) {
        auto it = _seg.upper_bound(left);
        return it != _seg.begin() && (--it)->second >= right;
    }

    void removeRange(int left, int right) {
        auto it1 = _seg.upper_bound(left), it2 = _seg.upper_bound(right);
        if (it1 != _seg.begin()) {
            if ((--it1)->second < left) ++it1;    // move to the one needs to be removed
        }
        if (it1 == it2) return;
        int l1 = min(left, it1->first), r1 = max(right, (--it2)->second);
        _seg.erase(it1, ++it2);
        if (l1 < left) _seg[l1] = left;
        if (right < r1) _seg[right] = r1;
    }
};
```

Ref:

354. Russian doll envelopes

You have a number of envelopes with widths and heights given as a pair of integers `(w, h)`. One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

Example:

Given envelopes = `[[5,4],[6,4],[6,7],[2,3]]`, the maximum number of envelopes you can Russian doll is `3` ([2,3] => [5,4] => [6,7]).

```
// soln-1: dynamic programming
// sort the envelopes by (w + h), then it's same as LIS.
// f(i) = max{1 + f(j) | j = 0~i-1, if envelope can be contained. }
int maxEnvelopes(vector<pair<int, int>>& envelopes) {
   sort(envelopes.begin(), envelopes.end());

   int ans = 0;
   vector<int> dp(envelopes.size(), 1);
   for (int i = 0; i < envelopes.size(); ++i) {
       for (int j = 0; j < i; ++j) {
           if (envelopes[j].first < envelopes[i].first && envelopes[j].second < envelopes[i].second) {
               dp[i] = max(dp[i], dp[j] + 1);
           }
       }
       ans = max(ans, dp[i]);
   }
   return ans;
}
```

Ref: #300

356. Line reflection

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflect the given points.

Example 1:

Given *points* = `[[1,1],[-1,1]]`, return `true`.

Example 2:

Given *points* = `[[1,1],[-1,-1]]`, return `false`.

Follow up:

Could you do better than O(n^2)?

Solution:

Line reflection is sometimes called the mirror line. This question asks if there is a mirror line that parallel to y-axis.

1. midX = (minX + maxX) / 2
2. if this is a mirror line, then every point should have its mirror point {(minX + maxX) - x, y}.

```
struct PairHash {
    size_t operator() (const pair<int, int>& p) const {
        return p.first ^ p.second;
    }
};

bool isReflected(vector<pair<int, int>>& points) {
    unordered_set<pair<int, int>, PairHash> s;
    int minX = INT_MAX, maxX = INT_MIN;

    for (auto pt : points) {
        minX = min(minX, pt.first);
        maxX = max(maxX, pt.first);
        s.insert(pt);
    }

    for (auto pt : points) {
        if (s.find({ minX + maxX - pt.first, pt.second }) == s.end()) return false;    // not able to find
mirror
    }
    return true;
}
```

Ref: #149

357. Count numbers with unique digits

Given a non-negative integer n, count all numbers with unique digits, x, where 0 ≤ x < 10^n.

Example:

Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding `[11, 22, 33, 44,55, 66,77, 88, 99]`)

Solution:

soln-1: enumerate all possible options

1. for 3 digits number: xyz, for digit x, we have 1~9, total 9 options, for digit y, we have (0~9) - 1 = 9 options, for z, we have (0 ~ 9) - 1 - 1 = 8 options.
2. f(1) = 10, f(2) = 9 * 9 + f(1), f(3) = 9 * 9 * 8 + f2 + f1
3. f(11) = 0

```
// soln-1: f(3) = 9 * 9 * 8 + f(2)
//         f(2) = 9 * 9 + f(1)
//         f(1) = 10
//         f(11+) = f(10)
int countNumbersWithUniqueDigits(int n) {
   if (n < 1) return 1;
   if (n > 10) n = 10;

   int ans = 10;
   for (int i = 1; i < n; ++i) {
```

```
        int cur = 9;
        for (int j = 0, start = 9; j < i; ++j, --start) {
            cur *= start;
        }
        ans += cur;
    }
    return ans;
}
```

Ref:

359. Logger rate limiter

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is **not printed in the last 10 seconds**.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```
Logger logger = new Logger();

// logging string "foo" at timestamp 1
logger.shouldPrintMessage(1, "foo"); returns true;

// logging string "bar" at timestamp 2
logger.shouldPrintMessage(2,"bar"); returns true;

// logging string "foo" at timestamp 3
logger.shouldPrintMessage(3,"foo"); returns false;

// logging string "bar" at timestamp 8
logger.shouldPrintMessage(8,"bar"); returns false;

// logging string "foo" at timestamp 10
logger.shouldPrintMessage(10,"foo"); returns false;

// logging string "foo" at timestamp 11
logger.shouldPrintMessage(11,"foo"); returns true;
```

Solution:
use a hashmap to keep <message, timestamp> info.

```
class Logger {
    unordered_map<string, int> m_mp;    // <message, timestamp> map

public:
    Logger() { }

    bool shouldPrintMessage(int timestamp, string message) {
        // message does not exist before, or message exists earlier.
        if (m_mp.find(message) == m_mp.end() || m_mp[message] + 10 <= timestamp) {
            m_mp[message] = timestamp;
            return true;
        }

        return false;   // message exists within 10s (just leave it there)
    }
};
```

Ref: #362

361/764. Bomb enemy/Largest plus sign (review)

```
// 361 - bomb enemy (hits all enemies in the same row and col until hits the wall)
// soln-1: count E in between 2 walls if necessary
int maxKilledEnemies(vector<vector<char>>& grid) {
   if (grid.empty() || grid[0].empty()) return 0;

   int row = grid.size(), col = grid[0].size(), rCount = 0, cCount[col], ans = 0;
   for (int i = 0; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           if (j == 0 || grid[i][j - 1] == 'W') {      // need recount E for current row
               rCount = 0;
               for (int k = j; k < col && grid[i][k] != 'W'; ++k) rCount += grid[i][k] == 'E';
           }
           if (i == 0 || grid[i - 1][j] == 'W') {      // need recount E for current col
               cCount[j] = 0;
               for (int k = i; k < row && grid[k][j] != 'W'; ++k) cCount[j] += grid[k][j] == 'E';
           }

           if (grid[i][j] == '0') ans = max(ans, rCount + cCount[j]);
       }
   }
   return ans;
}
```

```
// 764. Largest Plus Sign
// soln-1: brute-force (expandable)
// soln-2: 4-pointers trick
int orderOfLargestPlusSign(int N, vector<vector<int>>& mines) {
    vector<vector<int>> grid(N, vector<int>(N, N));
    for (auto& m : mines) grid[m[0]][m[1]] = 0;
    for (int i = 0; i < N; ++i) {
        int U = 0, D = 0, L = 0, R = 0;
        for (int l = 0, r = N - 1; l < N; ++l, --r) {
            grid[i][l] = min(grid[i][l], L = (grid[i][l] ? L + 1 : 0));
            grid[i][r] = min(grid[i][r], R = (grid[i][r] ? R + 1 : 0));
            grid[l][i] = min(grid[l][i], U = (grid[l][i] ? U + 1: 0));
            grid[r][i] = min(grid[r][i], D = (grid[r][i] ? D + 1 : 0));
        }
    }
    int ans = 0;
    for (auto& row : grid) {
        for (auto& v : row) ans = max(ans, v);
    }
    return ans;
}
```

Ref:

366/1026/1080. Find leaves of binary tree/Max diff. between node and ancestor

```
// 366 - find leafs of binary tree
// soln-1: dfs (count height bottom-up)
vector<vector<int>> findLeaves(TreeNode* root) {
    vector<vector<int>> ans;
    helper(root, ans);
    return ans;
}

int helper(TreeNode* root, vector<vector<int>>& ans) {
    if (root == NULL) return -1;    // let leaf has height of 0.

    int h = 1 + max(helper(root->left, ans), helper(root->right, ans));
    if (h == ans.size())    ans.push_back(vector<int>{});
    ans[h].push_back(root->val);
```

```
    return h;
}
```

```
// 1026 - Maximum Difference Between Node and Ancestor
// soln-1: dfs (carry with min/max ancestor val)
int helper(TreeNode* node, int mi, int mx) {
    if (nullptr == node) return INT_MIN;
    auto diff = max(abs(node->val - mi), abs(node->val - mx));

    mi = min(mi, node->val), mx = max(mx, node->val);
    auto l = helper(node->left, mi, mx), r = helper(node->right, mi, mx);
    return max(diff, max(l, r));
}
int maxAncestorDiff(TreeNode* root) {
    return helper(root, root->val, root->val);
}
```

```
// 1080 - Insufficient Nodes in Root to Leaf Paths
// soln-1: dfs
TreeNode* sufficientSubset(TreeNode* root, int limit) {
    if (nullptr == root) return nullptr;
    if (nullptr == root->left && nullptr == root->right) {
        return (root->val < limit) ? nullptr : root;
    }
    root->left = sufficientSubset(root->left, limit - root->val);
    root->right = sufficientSubset(root->right, limit - root->val);
    if (nullptr == root->left && nullptr == root->right) return nullptr;    // this is not leaf

    return root;
}
```

Ref: #310

367. Valid perfect square

Given a positive integer *num*, write a function which returns True if *num* is a perfect square else False.

```
// soln-1: binary search
bool isPerfectSquare(int num) {
    long long low = 1, hi = num / 2;         // avoid overflow
    while (low + 1 < hi) {
        long long m = low + (hi - low) / 2;
        if (m * m == num) return true;

        m * m > num ? hi = m : low = m;
    }
    return low * low == num || hi * hi == num;
}
```

Ref: #69

368. Largest divisible subset

Given a set of **distinct** positive integers, find the largest subset such that every pair (S_i, S_j) of elements in this subset satisfies: S_i % S_j = 0 or S_j % S_i = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

```
nums: [1,2,3], Result: [1,2] (of course, [1,3] will also be ok)
```

```
// soln-1: dynamic programming
```

```
// let Si = {x, y} be the subset at index-i, where x < y, now consider z = nums[i+1]:
// given y % x = 0, if z % y = 0, then we have z % x = 0
// proof: z % x = (k.y) % x = (k.k.x) % x = 0
// let dp(i) be the size of subset that a[i] belongs to, check how i would contribute to [0, i)
//     dp(i) = max{ dp(k) + 1 | 0 <= k < i, if nums[i] % nums[k] }
vector<int> largestDivisibleSubset(vector<int>& nums) {
   if (nums.empty()) return vector<int>();
   sort(nums.begin(), nums.end());

   int n = nums.size(), maxsize = 0, ans_idx = 0;
   vector<int> dp(n), parent(n);
   for (int i = 0; i < n; ++i) {
       parent[i] = i, dp[i] = 1;
       for (int j = 0; j < i; ++j) {
           if (nums[i] % nums[j] == 0) {      // means i can be fit into set at index-j
               if (dp[j] >= dp[i]) dp[i] = dp[j] + 1, parent[i] = j;
           }
       }
       if (maxsize < dp[i]) maxsize = dp[i], ans_idx = i;
   }

   // build result set from ans_idx
   vector<int> ans;
   do {
       ans.push_back(nums[ans_idx]);
       if (ans_idx == parent[ans_idx]) break;
       ans_idx = parent[ans_idx];
   } while (1);
   return ans;
}
```

Ref:

370/598/1052. Range addition/Grumpy bookstore owner (review)

Assume you have an array of length ***n*** initialized with all **0**'s and are given ***k*** update operations.

Each operation is represented as a triplet: **[startIndex, endIndex, inc]** which increments each element of subarray **A[startIndex ... endIndex]** (startIndex and endIndex inclusive) with **inc**.

Return the modified array after all ***k*** operations were executed.

Example:	**Explanation:**

`Given: length = 5,`
`    updates = [`
`        [1,  3,  2],`
`        [2,  4,  3],`
`        [0,  2, -2]`
`    ]`
 	
`Output: [-2, 0, 3, 5, 3]`
	`Initial state: [ 0, 0, 0, 0, 0 ]`
	
	`After applying operation [1, 3, 2]: [ 0, 2, 2, 2, 0 ]`
	
	`After applying operation [2, 4, 3]: [ 0, 2, 5, 5, 3 ]`
	
	`After applying operation [0, 2, -2]: [-2, 0, 3, 5, 3 ]`

```
// soln-1: tricky one: mark only left-index and sum from left to right (A[i+1] += A[i])
// the key is to add offset starting at A[right-idx + 1]
// variation: if the given array is not initialized 0, then we could use extra O(n) space by
// applying this approach
vector<int> getModifiedArray(int length, vector<vector<int>>& updates) {
    vector<int> ans(length, 0);

    for (auto& triplet : updates) {
        int start = triplet[0], end = triplet[1], inc = triplet[2];
        ans[start] += inc;
```

```
        if (end + 1 < length) ans[end + 1] -= inc;
    }
    for (int i = 1; i < length; ++i) ans[i] += ans[i - 1];

    return ans;
}
```

```
// 589 - range addition II
int maxCount(int m, int n, vector<vector<int>>& ops) {
   for (auto& op : ops) {
       m = min(m, op[0]), n = min(n, op[1]);
   }
   return m * n;
}
```

```
// 1052 - Grumpy Bookstore Owner
// soln-1: sliding window (prefix sum)
// ..........i.......i+x..........
//       \--/\---------/\--/
// sum of 3 parts:
// 1. sum-of-window-size
// 2. customers ahead of window when not grumpy
// 3. customers after the window when not grumpy
int maxSatisfied(vector<int>& customers, vector<int>& grumpy, int X) {
   int ans = 0, n = customers.size();
   vector<int> sum(n), l2r(n), r2l(n);
   for (int i = 0; i < n; ++i) {
       sum[i] = customers[i] + (i - 1 >= 0 ? sum[i - 1] : 0);
       l2r[i] = (i - 1 >= 0 ? l2r[i - 1] : 0) + (grumpy[i] ? 0 : customers[i]);
       auto t = n - 1 - i;
       r2l[t] = (t + 1 < n ? r2l[t + 1] : 0) + (grumpy[t] ? 0 : customers[t]);
   }
   for (int i = X - 1; i < n; ++i) {
       auto tmp = sum[i];
       if (i - X >= 0) tmp = tmp - sum[i - X] + l2r[i - X];
       if (i + 1 < n) tmp += r2l[i + 1];
       ans = max(ans, tmp);
   }
   return ans;
}
```

Ref:

373. Find k pairs with smallest sums

You are given two integer arrays **nums1** and **nums2** sorted in ascending order and an integer **k**. Define a pair **(u,v)** which consists of one element from the first array and one element from the second array.

Find the k pairs **$(u_1,v_1),(u_2,v_2)$ ...(u_k,v_k)** with the smallest sums.

Example 1:

```
Given nums1 = [1,7,11], nums2 = [2,4,6],  k = 3, Return: [1,2],[1,4],[1,6]
```

#378 - Kth smallest in sorted array (soln-1 same with #373, soln-2 is same as #668)
#668 - Kth smallest in multiple table

```
// soln-1: keep first row + each # in 2nd row in heap
vector<pair<int, int>> kSmallestPairs(vector<int>& nums1, vector<int>& nums2, int k) {
   auto cmp = [&nums1, &nums2](const pair<int, int>& a, const pair<int, int>& b) {
       return nums1[a.first] + nums2[a.second] > nums1[b.first] + nums2[b.second];
   };
   priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp)> q(cmp);
```

```
    vector<pair<int, int>> ans;
    for (int j = 0; j < nums2.size(); ++j) q.push({0, j});
    for (int i = 0; i < k && !q.empty(); ++i) {
        auto p = q.top();       q.pop();
        if (p.first + 1 < nums1.size()) q.push({p.first + 1, p.second});
        ans.push_back({nums1[p.first], nums2[p.second]});
    }
    return ans;
}
```

Ref: #347 #355 #358

374/375. Guess number higher or lower / II

We are playing the Guess Game. The game is as follows:

I pick a number from **1** to ***n***. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API `guess(int num)` which returns 3 possible results (`-1`, `1`, or `0`): `lower/higher/hit`

Follow-up:
However, when you guess a particular number x, and you guess wrong, you pay **$x**. You win the game when you guess the number I picked.

Example:

```
n = 10, I pick 8.
First round:  You guess 5, I tell you that it's higher. You pay $5.
Second round: You guess 7, I tell you that it's higher. You pay $7.
Third round:  You guess 9, I tell you that it's lower. You pay $9.
Game over. 8 is the number I picked. You end up paying $5 + $7 + $9 = $21.
```

Given a particular **n ≥ 1**, find out how much money you need to have to guarantee a **win**.

```
// return -1/0/1 if lower/equal/higher
int guess(int x);

// soln-1: binary search without dups
int guessNumber(int n) {
    int low = 1, hi = n;
    while (low + 1 < hi) {
        int m = low + (hi - low) / 2;
        int test = guess(m);
        if (0 == test) return m;
        test > 0 ? low = m : hi = m;
    }
    return guess(low) ? hi : low;
}

// soln-1: dynamic programming
// this is MinMax question for minimizing the possible loss for worst case scenario.
// f(i, j) = min{k + max{f(i, k-1), f(k+1, j)} | k = i~j}
//               |      peer will max its gains
//            minimize my loss
int helper(int lo, int hi, vector<vector<int>>& dp) {
    if (lo >= hi) return 0;
    if (dp[lo][hi]) return dp[lo][hi];

    int ans = INT_MAX;
    for (int i = lo; i <= hi; ++i) {
        ans = min(ans, i + max(helper(lo, i - 1, dp), helper(i + 1, hi, dp)));
    }
```

```
    return dp[lo][hi] = ans;
}

int getMoneyAmount(int n) {
    vector<vector<int>> dp(n + 1, vector<int>(n + 1));
    return helper(1, n, dp);
}
```

Ref:

376. Wiggle subsequence

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:

Input: [1,17,5,10,13,15,10,5,16,8], **Output:** 7

There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8].

```
// soln-1: keep the max/min so far if ascending / descending
int wiggleMaxLength(vector<int>& nums) {
    if (nums.empty()) return 0;

    int ans = 1, start = 1;
    for (start = 1; start < nums.size(); ++start) {
        if (nums[start] != nums[start - 1]) break;
    }
    if (start == nums.size()) return ans;

    bool asc = nums[start] > nums[start - 1];
    for (int i = start, pre = nums[start]; i < nums.size(); ++i) {
        if ((asc && pre > nums[i]) || (!asc && pre < nums[i])) {
            asc = !asc;
            pre = nums[i];
            ++ans;
        }
        asc ? pre = max(pre, nums[i]) : pre = min(pre, nums[i]);
    }
    return ans + 1;
}

// soln-2: dynamic programming
// let's consider at index-i, it will contribute if:
// a[i-1] < a[i] and a[i-1] is downward
// a[i-1] > a[i] and a[i-1] is upward
// let up/down(i) be the max length when (i-1, i) is up/down direction
// 1, down(i) = up(i - 1) + 1, if a[i-1] > a[i]
// 2, up(i) = down(i - 1) + 1, if a[i-1] < a[i]
// 3, a[i-1] == a[i], up/down(i) = up/down(i-1)
int wiggleMaxLength(vector<int>& nums) {
    if (nums.empty()) return 0;

    vector<int> up(nums.size(), 1);
    vector<int> down(nums.size(), 1);
    for (int i = 1; i < nums.size(); ++i) {
        down[i] = nums[i - 1] > nums[i] ? up[i - 1] + 1 : down[i - 1];
        up[i] = nums[i - 1] < nums[i] ? down[i - 1] + 1 : up[i - 1];
    }
    return max(up.back(), down.back());
}
```

Ref: #276 #280

378/668. Kth smallest in sorted matrix/in multiple table (review)

Given *n* x *n* matrix where each of the rows and columns are sorted in ascending order, find the kth smallest element in the matrix. Note that it is the kth smallest element in the sorted order, not the kth distinct element.

#410 - split array largest sum (another value-range based binary search / dynamic programming)

```
// 378 - soln-1: keep first row/col into heap, then add from next next row/col in O(klgn) time
int kthSmallest(vector<vector<int>>& matrix, int k) {
   auto cmp = [&matrix](const pair<int, int>& a, pair<int, int>& b) {
       return matrix[a.first][a.second] > matrix[b.first][b.second];
   };
   priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp)> q(cmp);

   for (int j = 0; j < matrix[0].size(); ++j) q.push({0, j});
   for (int i = 1; i < k; ++i) {
       auto t = q.top();   q.pop();
       if (t.first + 1 < matrix.size()) q.push({t.first + 1, t.second});
   }
   return matrix[q.top().first][q.top().second];
}

// 378 - soln-2: value-range based binary search, then count from matrix[0][0] till mid
// O(nlgn) time - log(n) times search, O(n) counting for each search.
// FASTER THAN SOLN-1 - not dealing with memory allocation.
int kthSmallest(vector<vector<int>>& matrix, int k) {
   int row = matrix.size(), col = matrix[0].size();
   int low = matrix[0][0], hi = matrix[row - 1][col - 1] + 1;  // [low, hi)

   while (low < hi) {
       int mid = low + (hi - low) / 2, count = 0;
       for (int i = 0, j = col - 1; i < row; ++i) {
           while (j >= 0 && matrix[i][j] > mid) --j;          // count all <= mid #s
           count += j + 1;
       }
       count < k ? low = mid + 1 : hi = mid;                  // [low, hi)
   }
   return low;
}
```

```
// 668 - soln-1: value-range based binary search (same as #378)
int findKthNumber(int m, int n, int k) {
   int low = 1, hi = m * n + 1;                // [low, hi)
   while (low < hi) {
       int mid = low + (hi - low) / 2;
       // count all #s <= mid
       int count = 0;
       for (int i = 1; i <= m; ++i) {
           count += min(mid / i, n);
       }
       count < k ? low = mid + 1 : hi = mid;   // keep [low, hi)
   }
   return low;
}
```

Ref: #4 #373

380/381. Insert/Delete/GetRandom O(1)

Design a data structure that supports all following operations in *average* **O(1)** time.

1. `insert(val)`: Inserts an item val to the set if not already present.
2. `remove(val)`: Removes an item val from the set if present.
3. `getRandom`: Returns a random element from current set of elements. Each element must have the **same probability** of being returned.

```
// soln-1: hashmap + vector for direct access
class RandomizedCollection {
   unordered_map<int, unordered_set<int>> _mp; // val -> index set
   vector<int> _val;
public:
   /** Returns true if the collection did not already contain the specified element. */
   bool insert(int val) {
       _val.push_back(val), _mp[val].insert(_val.size() - 1);
       return _mp[val].size() == 1;
   }

   /** Returns true if the collection contained the specified element. */
   bool remove(int val) {
       if (_mp.find(val) == _mp.end()) return false;

       int last = _val.back();
       if (last == val) {
           _val.pop_back(), _mp[val].erase(_val.size());
       } else {
           int val_idx = *(_mp[val].begin()), last_idx = _val.size() - 1;
           _val[val_idx] = last, _mp[last].insert(val_idx), _mp[last].erase(last_idx);

           _val.pop_back(), _mp[val].erase(val_idx);
       }
       if (_mp[val].empty()) _mp.erase(val);

       return true;
   }

   /** Get a random element from the collection. */
   int getRandom() {
       return _val[rand() % _val.size()];
   }
};
```

Ref:

382/384/398/528/710/470/478/497. Random pick ... Random/Rejection Sampling (review)

```
// 382 - Random pick from linked list
// soln-1: Reservoir sampling (math, see Programming Pearls)
class Solution382 {
   vector<int> _nums;
public:
   Solution382(ListNode* head) {
       while (head) _nums.push_back(head->val), head = head->next;
   }

   // reservoir sampling - pick any one out of N, so the probability = 1/N.
   // 1. ith = 1, p(1) = 1/1
   // 2. ith = 2, p(2) = 1/2, p(1) = 1*(1-1/2) = 1/2
   // 3. ith = N, P(n) = 1/n, p(1..n-1) = (n-1)/n*(1-1/n) = 1/n
   // 4. so when ith == n, each number would have p=1/n to be selected.
   int getRandom() {
```

```
        int ans = 0;
        for (int i = 0; i < _nums.size(); ++i) {
            if (rand() % (i + 1) == 0) ans = _nums[i];
        }
        return ans;
    }
    // random pick k out of N.
    // 1. p(1..k) = 1       <== pick the first k numbers
    // 2. p(k+1) = k/(k+1)  <== keep this number with k/(k+1) probability
    //    p(1..k) = 1 * (1-k/(k+1)) + k/(k+1) * (k-1)/k = k/(k+1)
    //                   ~~~~~~~~~~~            ~~~~~~~
    //                   k+1 not picked         k+1 picked, but I am not removed
    // 3. p(k+2) = k/(k+2)
    //    p(1..k) = k/(k+1) * (2/(k+2) + k/(k+2) * (k-1)/k) = k/(k+2)
    // 4. p(i) = k/i
    //    p(1..k) = k/(i-1) * ((i-k)/i + k/i * (k-1)/k) = k/i
    vector<int> getRandom(int k) {
        vector<int> ans(_nums.begin(), _nums.begin() + k);
        for (int i = k; i < _nums.size(); ++i) {
            int r = rand() % i;
            if (r < k) ans[r] = _nums[i];
        }
        return ans;
    }
};
```

```
// 398 - Random pick index
// soln-1: Reservoir sampling
class Solution398 {
    vector<int> _nums;
public:
    Solution398(vector<int> nums) : _nums(nums) {
    }

    int pick(int target) {
        int ans = 0;
        for (int i = 0, tc = 0; i < _nums.size(); ++i) {
            if (_nums[i] != target) continue;
            ++tc;   // target count
            if (rand() % tc == 0) ans = i;
        }
        return ans;
    }
};
```

```
// 528 - Random pick with weight
class Solution528 {
    vector<int> _w;
public:
    Solution528(vector<int> w) : _w(w) {
    }

    int pickIndex() {
        int ans = 0, total = _w.front();
        for (int i = 1; i < _w.size(); ++i) {
            total += _w[i];
            if (_w[i] < rand() % total) ans = i;
        }
        return ans;
    }
};
```

```
// 710 - Random pick with blacklist TODO
```

```
// 384 - Shuffle an Array
class Solution {
```

```
    vector<int> _nums;
public:
    Solution(vector<int>& nums) {
        _nums = nums;
    }

    vector<int> reset() {
        return _nums;
    }

    vector<int> shuffle() {
        vector<int> ans(_nums);
        for (int i = 0; i < _nums.size(); ++i) {
            int r = rand() % (_nums.size() - i);
            swap(ans[i], ans[r + i]);
        }
        return ans;
    }
};
```

```
// 470 - Implement Rand10() Using Rand7()  TODO
// 478 - Generate Random Point in a Circle  TODO
// 497 - Random Point in Non-overlapping Rectangles  TODO
// 519 - Random Flip Matrix TODO
```

Ref:

383/691. Ransom notes / Stickers to spell word

383 - check if ransom note can be constructed from the magazines
691 - We are given N different types of stickers. Each sticker has a lowercase English word on it.
You would like to spell out the given `target` string by cutting individual letters from your collection of stickers and rearranging them.

You can use each sticker more than once if you want, and you have infinite quantities of each sticker.

What is the minimum number of stickers that you need to spell out the `target`? If the task is impossible, return -1.

Example 1:

Input:`["with", "example", "science"], "thehat",` Output: `3`

Explanation:

```
We can use 2 "with" stickers, and 1 "example" sticker.
After cutting and rearrange the letters of those stickers, we can form the target "thehat".
Also, this is the minimum number of stickers necessary to form the target string.
```

```
// 383: soln-1: easy hashmap
bool canConstruct(string ransomNote, string magazine) {
   vector<int> letters(256);
   for (auto ch : magazine) letters[ch]++;
   for (auto ch : ransomNote) {
       if (--letters[ch] < 0) return false;
   }
   return true;
}
```

Ref:

387/451. First unique character in a string / Sort characters by frequency

451 - sort characters by frequency. e.g. “tree” -> “eert”

```
// 387: soln-1: easy hashmap
int firstUniqChar(string s) {
   vector<int> letters(256);
   for (auto ch : s) letters[ch]++;
   for (int i = 0; i < s.length(); ++i) {
       if (letters[s[i]] == 1) return i;
   }
   return -1;
}
```

```
// 451: soln-1: heap in O(nlgn) time
string frequencySort(string s) {
   vector<pair<int, char>> h(256);
   for (auto ch : s) h[ch].first++, h[ch].second = ch;
   make_heap(h.rbegin(), h.rend());
   sort_heap(h.rbegin(), h.rend());

   int idx = 0;
   for (auto& p : h) {
       if (p.first == 0) break;
       for (int i = 0; i < p.first; ++i) s[idx++] = p.second;
   }
   return s;
}
```

Ref:

391. TODO

Solution:

Ref:

392. Is subsequence

Given a string **s** and a string **t**, check if **s** is subsequence of **t**.

You may assume that there is only lower case English letters in both **s** and **t**. **t** is potentially a very long (length ~= 500,000) string, and **s** is a short string (<=100).

Example 1:

s = `"abc"`, **t** = `"ahbgdc"`

Return `true`.

Follow up:

If there are lots of incoming S, say S1, S2, ... , Sk where k >= 1B, and you want to check one by one to see if T has its subsequence. In this scenario, how would you change your code?

```
// soln-1: dynamic programming in O(n * m) time and space
// let dp(i, j) be the result
//     dp(i, j) = S[i] == T[j] ? dp(i - 1, j - 1) : dp(i, j - 1)
//
// soln-2: brute force scan in O(n + m) time
// follow-up: keep accurrance and do binary search, see #792
```

```
bool isSubsequence(string s, string t) {
   for (int i = 0, j = 0; i < s.length(); ++i) {
       for (; j < t.length(); ++j) if (s[i] == t[j]) break;
       if (j == t.length()) return false;
       ++j;
   }
   return true;
}
```

Ref: #792

393. UTF-8 validation

A character in UTF8 can be from **1 to 4 bytes** long, subjected to the following rules:

1. For 1-byte character, the first bit is a 0, followed by its unicode code.
2. For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10.

This is how the UTF-8 encoding would work:

```
   Char. number range  |        UTF-8 octet sequence
      (hexadecimal)    |              (binary)
   --------------------+---------------------------------------------
   0000 0000-0000 007F | 0xxxxxxx
   0000 0080-0000 07FF | 110xxxxx 10xxxxxx
   0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
   0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.

```
bool validUtf8(vector<int>& data) {
   uint8_t l2 = (1 << 3) - 2, l3 = (1 << 4) - 2, l4 = (1 << 5) - 2;

   for (int i = 0; i < data.size(); ++i) {
       if ((data[i] >> 7) == 0) {                // 1-byte
       } else if ((data[i] >> 5) == l2) {       // 2-bytes
           if (++i >= data.size() || (data[i] >> 6) != 2)      return false;
       } else if ((data[i] >> 4) == l3) {       // 3-bytes
           for (int j = 0; j < 2; ++j) {
               if (++i >= data.size() || (data[i] >> 6) != 2)  return false;
           }
       } else if ((data[i] >> 3) == l4) {       // 4-bytes
           for (int j = 0; j < 3; ++j) {
               if (++i >= data.size() || (data[i] >> 6) != 2)  return false;
           }
       } else {
           return false;
       }
   }
   return true;
}
```

Ref:

396. TODO

Solution:

Ref:

397. Integer replacement

Given a positive integer *n* and you can do operations as follow:

1. If *n* is even, replace *n* with `n/2`.
2. If *n* is odd, you can replace *n* with either `n + 1` or `n - 1`.

What is the minimum number of replacements needed for *n* to become 1?

Solution:

What is not so obvious is what to do with odd numbers. One may think that you just need to remove as many 1's as possible to increase the evenness of the number. Wrong! Look at this example:

```
111011 -> 111010 -> 11101 -> 11100 -> 1110 -> 111 -> 1000 -> 100 -> 10 -> 1
```

And yet, this is not the best way because

```
111011 -> 111100 -> 11110 -> 1111 -> 10000 -> 1000 -> 100 -> 10 -> 1
```

See? Both `111011 -> 111010` and `111011 -> 111100` remove the same number of 1's, but the second way is better.

So, we just need to remove as many 1's as possible, doing +1 in case of a tie? Not quite. The infamous test with n=3 fails for that strategy because `11 -> 10 -> 1` is better than `11 -> 100 -> 10 -> 1`. Fortunately, that's the only exception (or at least I can't think of any other, and there are none in the tests).

So the logic is:

1. If `n` is even, halve it.
2. If `n=3` or `n-1` has less 1's than `n+1`, decrement `n`.
3. Otherwise, increment `n`.

Indeed, if a number ends with 01, then certainly decrementing is the way to go. Otherwise, if it ends with 11, then certainly incrementing is at least as good as decrementing

```
// soln-1: brute force
int integerReplacement(int n) {
   if (n <= 1) return 0;
   return 1 + ((n % 2) ? min(integerReplacement(n - 1), integerReplacement(n + 1))
                       : integerReplacement(n >> 1));
}

// soln-2: -1 if ending with 01 with exception 3
int integerReplacement(int m) {
   unsigned int n = (unsigned int) m;      // get 1 more bit to deal with corner case

   int ans = 0;
   while (n != 1) {
       if ((n & 1) == 0) n >>= 1;
       // -1 if ending with 01 with exception 3
       else if (n == 3 || ((n >> 1) & 1) == 0) --n;
       else ++n;

       ++ans;
   }
   return ans;
}
```

Ref:

399. Evaluate Division

Equations are given in the format `A / B = k`, where `A` and `B` are variables represented as strings, and `k` is a real number (floating point number). Given some queries, return the answers. If the answer does not exist, return `-1.0`.

Example:

Given `a / b = 2.0, b / c = 3.0.`

queries are: `a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .`

return `[6.0, 0.5, -1.0, 1.0, -1.0 ].`

Solution:
build directed graph then apply path find algorithm.

```
struct pair_hash_t {
   size_t operator() (const pair<string, double>& p) const {
       return std::hash<std::string>{}(p.first);
   }
};

typedef unordered_map<string, unordered_set<pair<string, double>, pair_hash_t>> t_graph;

bool helper(const string& start, const string& target, t_graph& g, unordered_set<string>& visited, double& res)
{
   if (visited.find(start) != visited.end()) return false;
   visited.insert(start);

   if (start == target) return true;
   for (auto& next : g[start]) {
       res *= next.second;
       if (helper(next.first, target, g, visited, res)) return true;
       res /= next.second;
   }
   return false;
}

vector<double> calcEquation(vector<pair<string, string>> equations, vector<double> values, vector<pair<string,
string>> queries) {
   t_graph g;
   unordered_set<string> nodes;
   for (int i = 0; i < equations.size(); ++i) {
       g[equations[i].first].insert(make_pair(equations[i].second, values[i]));
       g[equations[i].second].insert(make_pair(equations[i].first, 1.0f / values[i]));
       nodes.insert(equations[i].first);
       nodes.insert(equations[i].second);
   }

   vector<double> ans;
   for (auto& q : queries) {
       double res = 1.0f;

       unordered_set<string> visited;
       if (nodes.find(q.first) == nodes.end() || nodes.find(q.second) == nodes.end()) res = -1.0f;
       else if(!helper(q.first, q.second, g, visited, res)) res = -1.0f;

       ans.push_back(res);
   }
   return ans;
}
```

Ref:

400. Nth digit

Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

```
// soln-1: recursion (brute-force, no optimize)
// f(k, startNum) = if k <= digit(startNum), the fetch kth from startNum
//                = f(k - digit(startNum), startNum + 1)
// digit(x) - return # of digits for x
int findNthDigit(int k, int start = 1) {
   int d = 0;
   for (int t = start; t; t /= 10) d++;
   if (d >= k) return to_string(start).at(k - 1) - '0';

   return findNthDigit(k - d, start + 1);
}
// soln-2: brute-force recursion with optimize
// In above, each recursion will reduce k to (k - digit(startNum))
// we can greedy reduce more, for example, startNum = 1
// if k > 9, then f(k) = f(k - 1 * 9, 10)
// if k > 2 * 9 * 10, then f(k) = f(k - 2 * 9 * 10, 100)
// if k > 3 * 9 * 100, then f(k) = f(k - 3 * 9 * 100, 1000)
//
// if k < digits * 9 * 10^(digit - 1), startNum = x
// f(k, startNum) = f (k - k / digits, startNum + k / digits)
// e.g, f(15, 1) = f (15 - 9, 1 + 9)
//               = f(6, 10) = f(6 - 10 / 2, 10 + 10 / 2) = f(1, 15)
int findNthDigit(int k) {
   long digits = 1, base = 9, start = 1;
   while (k > digits * base) {
       k -= digits * base, digits++, base *= 10, start *= 10;
   }
   start += k / digits, k -= k / digits * digits;

   if (0 == k) return to_string(start - 1).back() - '0';
   return to_string(start).at(k - 1) - '0';
}
```

Ref:

401. Binary watch (review)

A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6 LEDs on the bottom represent the minutes (0-59).

Each LED represents a zero or one, with the least significant bit on the right.

Given a non-negative integer *n* which represents the number of LEDs that are currently on, return all possible times the watch could represent.

Example:

```
Input: n = 1

Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]
```

Solution:

soln-1: backtracking to enumerate at most 10 bits
soln-2: brute force enumerate for h (0-12) and m (0-60)

#17 - letter comb of a phone number

```
// soln-1: backtracking enumerate at most 10 bits
vector<string> readBinaryWatch(int num) {
```

```
    vector<string> ans;
    bitset<10> bs;
    helper(0, num, bs, ans);
    return ans;
}
void helper(int start, int num, bitset<10>& bs, vector<string>& ans) {
    if (0 == num) {
        int h = int(bs.to_ulong()) >> 6, m = int(bs.to_ulong()) & 0x3f;
        if (h < 12 && m < 60) ans.push_back(to_string(h) + (m >= 10 ? ":" : ":0") + to_string(m));
        return;
    }
    for (int i = start; i < 10; ++i) {
        if (bs.test(i)) continue;

        bs.set(i);
        helper(i + 1, num - 1, bs, ans);
        bs.reset(i);
    }
}

// soln-2: brute force enumerate for h (0-12) and m (0-60)
vector<string> readBinaryWatch(int num) {
    vector<string> ans;
    for (int h = 0; h < 12; ++h) {
        for (int m = 0; m < 60; ++m) {
            if (bitset<10>(h << 6 | m).count() == m) {
                ans.push_back(to_string(h) + (m < 10 ? ":0" : ":") + to_string(m));
            }
        }
    }
    return ans;
}
```

Ref: #17

403. Frog jump

A frog is crossing a river. The river is divided into x units and at each unit there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted ascending order, determine if the frog is able to cross the river by landing on the last stone. Initially, the frog is on the first stone and assume the first jump must be 1 unit.

If the frog's last jump was *k* units, then its next jump must be either *k* - 1, *k*, or *k* + 1 units. Note that the frog can only jump in the forward direction.

Note:

- The number of stones is ≥ 2 and is < 1,100.
- Each stone's position will be a non-negative integer < 2^{31}.
- The first stone's position is always 0.

Example 1:

```
[0,1,3,5,6,8,12,17]
There are a total of 8 stones.
The first stone at the 0th unit, second stone at the 1st unit,
third stone at the 3rd unit, and so on...
The last stone at the 17th unit.


Return true. The frog can jump to the last stone by jumping
1 unit to the 2nd stone, then 2 units to the 3rd stone, then
```

```
2 units to the 4th stone, then 3 units to the 6th stone,
4 units to the 7th stone, and 5 units to the 8th stone.
```

```
// soln-1: recursion with memorization
// try each stone from start, use memorization to optimize search
// memorization: <position, steps> -> true/false
unordered_map<int, bool> dp;
bool canCross(vector<int>& stones, int start = 0, int k = 0) {
   int key = start | k << 11;              // given the number of stones are less than 1100
   if (dp.find(key) != dp.end()) return dp[key];

   for (int i = start + 1; i < stones.size(); ++i) {
       int gap = stones[i] - stones[start];
       if (gap < k - 1)              continue;
       if (gap > k + 1)              return dp[key] = false;
       if (canCross(stones, i, gap)) return dp[key] = true;
   }
   return dp[key] = (start == stones.size() - 1);
}

// soln-2: jump from start pos, then keep all possible steps for each position
bool canCross(vector<int>& stones) {
   unordered_map<int, unordered_set<int>> mp;      // <stone #, steps can be used>
   for (int stone : stones) mp[stone] = unordered_set<int>();  // placeholder

   mp[stones.front()].insert(1);
   for (int stone : stones) {
       for (int step : mp[stone]) {
           int reach = step + stone;
           if (stones.back() == reach) return true;

           auto it = mp.find(reach);
           if (it == mp.end()) continue;             // reached stone does not exist
           if (step - 1 > 0) it->second.insert(step - 1);
           it->second.insert({step, step + 1});
       }
   }
   return false;
}
```

Ref:

[404. Sum of left leaves]

Sum of all left leaves node.

```
void helper(TreeNode* r, int& ans) {
   if (!r) return;

   if (r->left && !r->left->left && !r->left->right) ans += r->left->val;
   helper(r->left, ans);
   helper(r->right, ans);
}
```

Ref:

405. Convert a number to Hex

```
string toHex(int num) {
   if (num == 0) return "0";

   const char* hex = "0123456789abcdef";
   string ans;
   for (int i = sizeof(int) * 8 / 4; num && i > 0; --i, num >>= 4) {
       ans.push_back(hex[num & 0xf]);
   }

   return string(ans.rbegin(), ans.rend());
}
```

Ref:

408. Valid word abbreviation

Given a non-empty string s and an abbreviation abbr, return whether the string matches with the given abbreviation.

A string such as "word" contains only the following valid abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Note:

Assume s contains only lowercase letters and abbr contains only lowercase letters and digits.

Example 1:

Given s = "internationalization", abbr = "i12iz4n", Return true.

```
// soln-1: brute force scan
bool validWordAbbreviation(string word, string abbr) {
   int iw = 0, ia = 0;          // index of word and abbr.
   while (iw < word.length() && ia < abbr.length()) {
       if (isalpha(abbr[ia])) {
           if (word[iw] != abbr[ia]) break;
           ++iw, ++ia;
       } else {
           int n = stoi(abbr.substr(ia));
           iw += n, ia += to_string(n).length();
       }
   }
   return word.length() == iw && abbr.length() == ia;
}
```

Ref:

409. Longest palindrome

Given a string which consists of lowercase or uppercase letters, find the length of the longest palindromes that can be built with those letters. This is case sensitive, for example `"Aa"` is not considered a palindrome here.

```
int longestPalindrome(string s) {
   vector<int> mp(256);
   int ans = 0;
   for (char ch : s) if (++mp[ch] == 2) ans += 2, mp[ch] = 0;     // consume asap, consider "ccc"

   for (int i : mp) if (i % 2) return 1 + ans;
   return ans;
```

```
}
```

Ref: #226

410/548. Split array largest sum/with equal sum (review)

#375 - guess number hi/low II (MinMax question - with only 2-players)
#378 - kth smallest in sorted matrix (value-range based binary search)
#416/494/805 - partition equal subset / target sum / split array with same average

```
// 410 - split array into k subarrays but minimize the largest sum among them (given non-negative)
// soln-1: dynamic programming (not good, see soln-2: value-based binary search)
// Similar to minMax question #379 but with more than 2-players.
// Explained in TADMe2 ch8 the partition problem - CPU load balance problem.
// let dp(n, k) be the min sum with n-numbers and k-splitters. for 1-more splitter, we want to try each place
// to find out the mim sum of segment, each of which is the max{dp(i, k - 1), sum(A[i+1, n))}
//     dp(n, k) = min{max{dp(i, k - 1), sum-of-A[i+1...n)} | 1 <= i <= n}, i - insert divider before position-i
// boundary case:
//     dp(n, 1) = sum of A[0, n)    - no divider (one big block)
//     dp(1, k) = A[0]              - one number only
int splitArray(vector<int>& nums, int K) {
   if (nums.empty() || 0 == K) return 0;

   int N = nums.size();
   vector<vector<long>> dp(N, vector<long>(K));

   vector<long> s(N + 1);
   for (int i = 1; i <= N; ++i) s[i] = s[i - 1] + nums[i - 1];

   for (int j = 0; j < K; ++j) dp[0][j] = nums[0];
   for (int i = 1; i < N; ++i) dp[i][0] = dp[i - 1][0] + nums[i];

   for (int n = 1; n < N; ++n) {
       for (int k = 1; k < K; ++k) {
           dp[n][k] = s[n + 1];                // at most this value
           for (int i = 1; i <= n; ++i) {      // 1 <= i <= n
               long sum = s[n + 1] - s[i];     // sum of A[i+1...n)
               dp[n][k] = min(dp[n][k], max(dp[i - 1][k - 1], sum));
           }
       }
   }
   return (int)dp.back().back();
}
// 410 - split array into k subarrays but minimize the largest sum among them (given non-negative)
// soln-2: value-range based binary search + trial and error
// 1, if the array can be segmented into K parts and sum of each part < S, then
// we can use O(n) time to check if this is doable.
// 2, to split the array into K parts, sum of each part must within this range: [S/K, S]
// 3, hence the binary search soln: O(nlgS)
bool cutable(const vector<int>& nums, int K, int sum) {
   long long cur = 0;
   for (auto& num : nums) {
       if (cur + num <= sum) {
           cur += num;
       } else {
           --K, cur = num;      // cur alone could be larger than sum
       }
       if (K < 0 || cur > sum) return false;
   }
   return true;
}
```

```
int splitArray(vector<int>& nums, int K) {
   long long sum = 0;
   for (auto& num : nums) sum += num;

   long long low = sum / K, hi = sum;
   while (low < hi) {
       long long m = low + (hi - low) / 2;
       cutable(nums, K - 1, m) ? hi = m : low = m + 1;
   }
   return low;
}
```

```
// 548 - split array with equal sum (cut into 4 segments with eqal sum)
// soln-1: divide-and-conquer with trick
// 1. if the question is to cut array into 2-subarray, it's O(n) question.
// 2. now there is 3-cutter making 4-subarray, we can fix the middle cutter within [3, n - 3)
//    then solve the left and right side respectively.
bool splitArray(vector<int>& nums) {
   vector<int> sum(nums);
   for (int i = 1; i < sum.size(); ++i) sum[i] += sum[i - 1];

   for (int j = 3; j < nums.size() - 3; ++j) {
       unordered_set<int> s;                              // keep all the possible sum of subarray
       for (int i = 1; i < j - 1; ++i) {
           if (sum[i - 1] == (sum[j - 1] - sum[i])) s.insert(sum[i - 1]);
       }

       for (int k = j + 2; k < nums.size() - 1; ++k) {     // do brute-force verification
           if (sum[k - 1] - sum[j] == sum[nums.size() - 1] - sum[k]) {
               if (s.find(sum[k - 1] - sum[j]) != s.end()) return true;
           }
       }
   }
   return false;
}
```

```
// 932 - Beautiful Array
// no k with i < k < j such that A[k] * 2 = A[i] + A[j].
// soln-1: divide-and-conquer (tricky)
// 1. [beautiful-odd] + [beautiful-even] => [beautiful-odd-even]
// 2. [beautiful-array] * 2 => [still-beautiful]
// (A[k] * x) * 2 = A[k] * 2 * x != (A[i] + A[j]) * x = (A[i] * x) + (A[j] * x)
// .  [1, 3, 2] => [2, 6, 4]
// 3. this soln starts from 1 and left half is odd number beautiful array,
// .                               right half is even beautiful array.
vector<int> beautifulArray(int N) {
    vector<int> ans{1};
    while (ans.size() < N) {
        vector<int> tmp;
        for (int x : ans) if (x * 2 - 1 <= N) tmp.push_back(2 * x - 1);
        for (int x : ans) if (x * 2 <= N) tmp.push_back(2 * x);
        ans = tmp;
    }
    return ans;
}
```

Ref: [#375](#375) [#416](#416)

411. Minimum unique word abbreviation

A string such as "word" contains the following abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Given a target string and a set of strings in a dictionary, find an abbreviation of this target string with the ***smallest possible*** length such that it does not conflict with abbreviations of the strings in the dictionary.

Each number or letter in the abbreviation is considered length = 1. For example, the abbreviation "a32bc" has length = 4.

Note:

- In the case of multiple answers as shown in the second example below, you may return any one of them.
- Assume length of target string = m, and dictionary size = n. You may assume that m ≤ 21, n ≤ 1000, and log2(n) + m ≤ 20.

Examples:

"apple", ["blade"] -> "a4" (because "5" or "4e" conflicts with "blade")

"apple", ["plain", "amber", "blade"] -> "1p3" (other valid answers include "ap3", "a3e", "2p2", "3le", "3l1").

```
// soln-1: brute force enumerate all abbrs for target word, sort them by length, then valid against dict.
vector<string> helper(int idx, const string& target) {
   if (idx >= target.length()) return vector<string> {""};

   vector<string> ans;
   for (auto& s : helper(idx + 1, target)) {
       ans.push_back(target[idx] + s);          // no abbr.
       if (!s.empty() && isdigit(s[0])) {      // do abbr.
           int n = stoi(s);
           ans.push_back(to_string(n + 1) + s.substr(to_string(n).length()));
       } else {
           ans.push_back("1" + s);
       }
   }
   return ans;
}

// conflict if any of abbr. is valid
bool isConflict(const string& abbr, const vector<string>& dict) {
   for (const string& word : dict) {
       int iw = 0, ia = 0;
       while (iw < word.length() && ia < abbr.length()) {
           if (!isdigit(abbr[ia])) {
               if (word[iw] != abbr[ia]) break;
               else ++iw, ++ia;
           } else {
               int n = stoi(abbr.substr(ia));
               iw += n, ia += to_string(n).length();
           }
       }
       if (word.length() == iw && abbr.length() == ia) return true;
   }
   return false;
}

string minAbbreviation(string target, vector<string>& dictionary) {
   vector<string> abbrs = helper(0, target);
   sort(abbrs.begin(), abbrs.end(), [](const string& a, const string& b) { return a.length() < b.length(); });

   for (auto abbr : abbrs) {
       if (!isConflict(abbr, dictionary)) return abbr;     // first non-conflict abbr.
   }
   return "";
}
```

Ref: #288 #527

413/446. Arithmetic slices / II (subsequence) (review)

A zero-indexed array A consisting of N numbers is given. A slice of that array is any pair of integers (P, Q) such that 0 <= P < Q < N.

A slice (P, Q) of array A is called arithmetic if the sequence: A[P], A[p + 1], ..., A[Q - 1], A[Q] is arithmetic. In particular, this means that P + 1 < Q.

The function should return the number of arithmetic slices in the array A.

Example:

```
A = [1, 2, 3, 4] return: 3, for 3 arithmetic slices in A: [1, 2, 3], [2, 3, 4] and [1, 2, 3, 4] itself.
```

#446 - Arithmetic slices II (subsequence) much harder

Example:

```
Input: [2, 4, 6, 8, 10], Output: 7
Explanation: All arithmetic subsequence slices are:

[2,4,6], [4,6,8], [6,8,10], [2,4,6,8], [4,6,8,10], [2,4,6,8,10], [2,6,10]
```

```
// soln-1: dynamic programming
// let dp(i) be the number of slices at position i, if the distance between
// [i-2, i-1, i] are same, the index-i will contribute 1 to dp(i-1), hence
//     dp(i) = a[i-1] - a[i-2] == a[i] -a[i-1] ? 1 + dp(i-1): 0
//     where: dp(i-1) - the existing slices ending at i-1
//            1 - new slice: (i-2, i-1, i)
// dp(0) = dp(1) = 0 as it requires at least 3 numbers to form a slice
int numberOfArithmeticSlices(vector<int>& A) {
   int ans = 0;
   vector<int> dp(A.size());
   for (int i = 2; i < A.size(); ++i) {
       if ((A[i - 1] - A[i - 2]) == (A[i] - A[i - 1])) dp[i] = dp[i - 1] + 1;
       ans += dp[i];
   }
   return ans;
}
```

```
// soln-1: dynamic programming
// this question asks subsequence not only subarray, so we need to look further
// for example, at index i, we will not only looking for i-1, but also [0, i)
// d0 = a[i] - a[i-1]
// d1 = a[i] - a[i-2]
// ...
// di = a[i] - a[0]    <--- sequence [0, i] incase it will be used after i
//
// let dp(i, d) be the number of weak subsequence (length >= 2, not >= 3), ending at i with d-length
//     dp(i, d) = dp(k, d) + 1 | d = a[i] - a[k], 0 <= j < i
//     where: dp(k, d) - the existing number of weak subsequence
//            1 - the new weak subsequence: (a[k], a[i])
// if dp(i, d) is the number of weak subsequence, how do we calculate non-weak subsequence?
// notice that dp(k, d) is the existing number of weak subsequence, with i, it must be non-weak subsequence
//
// since d is not fixed, we can use hashmap(i, hashmap(d, val)) or vector<hashmap(d, val)>
int numberOfArithmeticSlicesII(vector<int>& A) {
   int ans = 0;

   vector<unordered_map<int, long>> dp(A.size());
   for (int i = 1; i < A.size(); ++i) {
```

```
        for (int j = 0; j < i; ++j) {
            long d = (long)A[i] - (long)A[j];
            if (d >= INT_MAX || d <= INT_MIN) continue;

            int nonweak = 0;
            if (dp[j].find(d) != dp[j].end()) nonweak = dp[j][d];   // existing weak must be nonweak for me!
            dp[i][d] = dp[i][d] + nonweak + 1;

            ans += nonweak;
        }
    }
    return ans;
}
```

Ref:

414/747. Third max number / Largest number at least twice of others

747 - Largest number at least twice of others
In a given integer array `nums`, there is always exactly one largest element.
Find whether the largest element in the array is at least twice as much as every other number in the array.
If it is, return the **index** of the largest element, otherwise return -1.

```
// 414 - 3rd max number
// keep (max1, max2, max3) and update
int thirdMax(vector<int>& nums) {
    long m1 = LONG_MIN, m2 = LONG_MIN, m3 = LONG_MIN;
    for (int x : nums) {
        if (x > m1) {
            m3 = m2, m2 = m1, m1 = x;
        } else if (x > m2 && x < m1) {
            m3 = m2, m2 = x;
        } else if (x > m3 && x < m2) {
            m3 = x;
        }
    }
    return (m3 != LONG_MIN) ? m3 : m1;
}
```

```
// 747 - dominant index
int dominantIndex(vector<int>& nums) {
   int max1 = INT_MIN, max2 = INT_MIN, ans = -1;
   for (int i = 0; i < nums.size(); ++i) {
       if (nums[i] > max1) max2 = max1, max1 = nums[i], ans = i;
       else if (nums[i] > max2) max2 = nums[i];
   }
   return (max1 >= 2 * max2) ? ans : -1;
}
```

Ref:

416/473/494/698/805. Partition equal subset/Target sum/Split array with average sum (review)

416 - Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

473 - partition into 4 equal subsets (brute-force recursion NP-complete in O(4^n) time)

698 - partition into k equal subsets

494 - You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols `+` and `-`. For each integer, you should choose one from `+` and `-` as its new symbol. Find out how many ways to assign symbols to make sum of integers equal to target S. (variation of 0/1 knapsack)

805 - In a given non-negative integer array A, we must move every element of A to either list B or list C. (B and C initially start empty.) Return true if and only if after such a move, it is possible that the average value of B is equal to the average value of C, and B and C are both non-empty.

#123 - buy/sell stock III
#410/548 - split array largest sum/with equal sum (binary search, divide-and-conquer)
#494 - target sum (how many subset that sums to S)

背包问题九讲

```
// 416 - containing only positives, check if we can partition into 2 subsets
// soln-1: 0/1 knapsack - knapsack capacity is half of sum
// let f(i, v) be the max value for i items with capacity v.
// f(i, v) = max{f(i-1, v), f(i-1, v - w[i]) + c[i]}
// optimization:
// O(v) space - for i-th item, try every capacity from the opposite direction (right to left)
// time optimization - only need to check only when the knap capacity is bigger enough.
bool canPartition(vector<int>& nums) {
   int sum = accumulate(nums.begin(), nums.end(), 0);
   if (sum & 1) return false;

   int cap = sum / 2;
   vector<int> dp(sum + 1);
   for (int x : nums) {
       for (int i = cap; i >= x; --i) {
           dp[i] = max(dp[i], dp[i - x] + x);
       }
   }
   return dp.back() == sum;
}

// 416 - soln-2: brute force in O(2^n)
bool helper(vector<int>& nums, int idx, int sum) {
   if (nums[idx] >= sum) return nums[idx] == sum;
   return helper(nums, idx + 1, sum) || helper(nums, idx + 1, sum - nums[idx]);
}
bool canPartition(vector<int>& nums) {
   int sum = accumulate(nums.begin(), nums.end(), 0);
   if (sum & 1) return false;
   sort(nums.rbegin(), nums.rend());
   return helper(nums, 0, sum/2);
}
```

```
// 473 - matchsticks to square (partition into 4 subsets with equal sum)
// soln-1: brute-force recursion in O(4^n), NP-complete question
// question similar to #416 partition equal subsets
//                       #698 partition to k equal subsets
bool makesquare(vector<int>& nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if (nums.empty() || sum % 4) return false;

    sort(nums.rbegin(), nums.rend());
    vector<int> sums(4);
    return helper(nums, sums, 0, sum / 4);
}
bool helper(vector<int>& nums, vector<int>& sums, int start, int target) {
    if (start >= nums.size()) return sums[0] == sums[1] && sums[1] == sums[2] && sums[2] == target;

    for (int i = 0; i < 4; ++i) {
```

```
        if (sums[i] + nums[start] > target) continue;
        sums[i] += nums[start];
        if (helper(nums, sums, start + 1, target)) return true;
        sums[i] -= nums[start];
    }
    return false;
}
```

```
// 494 - add +/- symbol to make target (0/1 knapsack)
// soln-1: dynamic programming
// assuming we split the numbers into two groups:
// 1. group-1: we do +, let is be P
// 2. group-2: we do -, let it be N.
// facts:
// 1. P + N = Sum (sum-of-given-array)
// 2. P - N = S (the given target)
// then we have P = (Sum + S) / 2
// So we want to pick some non-negative numbers which sum to P, which is 0/1 knapsack.
// Code is almost same as #416 - partition equal subset.
int findTargetSumWays(vector<int>& nums, int S) {
   int sum = accumulate(nums.begin(), nums.end(), 0);
   if (S > sum || (S + sum) & 1) return 0;

   int cap = (S + sum) >> 1;
   vector<int> dp(cap + 1);
   dp[0] = 1;                          // no item and no capacity
   for (int n : nums) {
       for (int i = cap; i >= n; --i) {
           dp[i] += dp[i - n];
       }
   }
   return dp.back();
}
```

```
// 1049 - Last Stone Weight (smallest possible weight after hitting rocks)
// soln-1: dynamic programming (knapsack)
// divide the rocks into two group, what is the min diff. of weight?
// knapsack capacity = sum / 2
int lastStoneWeightII(vector<int>& stones) {
   int sum = accumulate(stones.begin(), stones.end(), 0), cap = sum / 2;
   vector<int> dp(cap + 1);
   for (auto x : stones) {
       for (int i = cap; i >= x; --i) {
           dp[i] = max(dp[i], max(dp[i - 1], dp[i - x] + x));
       }
   }
   return sum - dp.back() * 2;
}
```

```
// 698 - partition to k equal subset
// soln-1: brute force recursion in O(k^n) NP-complete problem
bool canPartitionKSubsets(vector<int>& nums, int k) {
   int sum = 0;
   for (int x : nums) sum += x;
   if (k > nums.size() || k < 1 || sum % k) return false;

   vector<bool> taken(nums.size());
   return helper(nums, sum / k, 0, k, 0, taken);
}

bool helper(vector<int>& nums, int target, int cur, int k, int idx, vector<bool>& taken) {
   if (k <= 0) return true;
   if (cur >= target) return cur == target ? helper(nums, target, 0, k - 1, 0, taken) : false;
```

```
    for (int i = idx; i < nums.size(); ++i) {
        if (taken[i]) continue;
        taken[i] = true;
        if (helper(nums, target, cur + nums[i], k, i + 1, taken)) return true;
        taken[i] = false;
    }
    return false;
}
```

```
// 805 - split array with average sum
// soln-1: brute-force backtracking (TLE)
// the question asks to split into 2 subarray with average sum(A)/A.size() each.
// knapsack does not work here because we can't guarantee to pickup x-items given v-capacity.
// this needs to enumerate systematically, hence it's back tracking, with following optimization:
// 1, decide how many numbers in one subarray.
//    assume B and C are the two subarray, k is B.size() and k <= C.size().
//    sum(A) / n = sum(B) / k, 1 <= k <= n/2
//    sum(B) = sum(A) * k / n ===> sum(A) * k % n = 0, 1 <= k <= n/2
//    hence only k satisfying this condition worth a try.
// 2, sort the given input helps terminate search too.
bool helper(const vector<int>& A, int idx, int k, int s) {
    if (idx >= A.size()) return false;
    if (A[idx] >= k) return A[idx] == s && 1 == k;
    return helper(A, idx + 1, k, s) || helper(A, idx + 1, k - 1, s - A[idx]);
}

bool splitArraySameAverage(vector<int>& A) {
    int s = accumulate(A.begin(), A.end(), 0), n = A.size();
    sort(A.begin(), A.end());
    for (int i = 1; i <= n / 2; ++i) {
        if (s * i % n == 0 && helper(A, 0, i, i * s / n)) return true;
    }
    return false;
}
```

```
// 805 - split array with average sum
// soln-2: (TODO)
```

Ref: #410 #698

418/779/880. Sentence screen fitting/K-th symbol/Decoded string at index (review)

Given a (rows x cols) screen and a sentence represented by a list of words, find how many times the given sentence can be fitted on the screen.

Example 1:	Example 2:

Input: rows = 3, cols = 6, sentence = ["a", "bcd", "e"]
Output: 2
Explanation:
`a-bcd-`
`e-a---`
`bcd-e-`
	Input: rows = 4, cols = 5, sentence = ["I", "had", "apple", "pie"]
	Output: 1
	Explanation:
	`I-had`
	`apple`
	`pie-I`
	`had--`

```
// soln-1: dynamic programming
// The brute force soln would be go-thru each word in the sentence and put
// into screen row by row. In this case, the steps in each row would be the
// average length of words in sentence.
//
// A better way is to image the sentence has infinite loop and fill in the whole row with sentences
```

```
// 1, compose the whole sentence as a string
// 2, fit as much words in sentence as possible into a row
//  - if right boundary is '_', then no waste, fit + 1 ⇒ proceed next word
//  - otherwise go back a few letters until hit '_',   ⇒ restart from current word
// 3, loop until we used all row
//
int wordsTyping(vector<string>& sentence, int rows, int cols) {
   string all;
   for (auto& w : sentence) all += w + " ";

   int fit = 0, len = all.length();
   for (int i = 0; i < rows; ++i) {
       fit += cols;
       if (all[fit % len] == ' ') fit++;                          // 'apple_apple_' hit '_'
       else while (fit > 0 && all[(fit - 1) % len] != ' ') --fit;  //           ^ - hit 'p', go back
   }
   return fit / len;
}
```

```
// 779 - K-th Symbol in Grammar (starting from 0, rule: 0->01, 1->10 in next row)
// soln-1: recursion
// 1. image we have a binary tree, when node is 0, it has {0, 1} children, and 1 -> {1, 0}
// 2. Kth will be determined by its parent:
//    - 0 if parent = 0 and k is odd, or, parent = 1 and k is even
//    - 1 if parent = 0 and k is even, or, parent = 1 and k is odd
int kthGrammar(int N, int K) {
    if (N <= 1) return 0;
    if (K % 2) {
        return kthGrammar(N - 1, (K + 1) / 2) == 0 ? 0 : 1;
    } else {
        return kthGrammar(N - 1, (K + 1) / 2) == 0 ? 1 : 0;
    }
}
```

```
// 880 - decoded string at index (a2b3 => aab+aab+aab)
// soln-1: recursion
// decoding until total length >= K, then recursion
string decodeAtIndex(string S, int K) {
   long total = 0;
   for (auto ch : S) {
       if (isalpha(ch)) {
           if (++total == K) return string(1, ch);
       } else {
           if (total * (ch - '0') >= K) return decodeAtIndex(S, (K % total) ? (K % total) : total);
           total *= (ch - '0');
       }
   }
   return "0";     // should never reach here
}

// 880 - decoded string at index
// soln-2: math (backward decoding)
string decodeAtIndex(string S, int K) {
   long total = 0, idx = 0;
   for (idx = 0; total < K; ++idx) {
       isalpha(S[idx]) ? ++total : total *= (S[idx] - '0');
   }
   for (--idx; idx >= 0; --idx) {
       if (isalpha(S[idx])) {
           if (total-- == K) return string(1, S[idx]);
       } else {
           total /= (S[idx] - '0'), K % total ? K %= total : K = total;
       }
   }
   return "0";     // should never reach here
```

```
}
```

Ref:

420. TODO

Solution:

Ref:

421. Max XOR of 2 numbers in an array

Given a non-empty array of numbers, $a_0, a_1, a_2, \ldots, a_{n-1}$, where $0 \le a_i < 2^{31}$. Find the maximum result of a_i XOR a_j, where $0 \le i, j < n$.

Could you do this in O(*n*) runtime?

Example:

```
Input: [3, 10, 5, 25, 2, 8], Output: 28
Explanation: The maximum result is 5 ^ 25 = 28.
```

Solution:
soln-1: bit-by-bit consideration in O(n) time

soln-2: build a trie tree, then brute force search in O(n * lg32) time (faster than soln-1)

1. trie tree is pretty good for searching

#210 - Max AND of range array

```
// soln-1: consider the anwer bit-by-bit
// let a, b be any number from given input, we have a ^ b = c
// consider bit-31, it must be from bit-31 of a ^ b, so we generate a set of {a, b} as candidates
// now, we have c and a, if b exists in generated set, that means c has bit-31.
// no compare to get max? we are trying from bit-31 to bit-0 for possible 1, it's max to min process.
//
// findMinimumXOR ? N/A
int findMaximumXOR(vector<int>& nums) {
   int ans = 0;
   for (int i = 31, mask = 0; i >= 0; --i) {
       mask |= 1 << i;
       unordered_set<int> ab;
       for (int x : nums) ab.insert(x & mask);

       int c = ans | (1 << i);                  // assuming we have 1 set in our answer (max out it)
       for (int b : ab) {                       // check if a ^ b = c
           if (ab.find(b ^ c) != ab.end()) {    // a ^ b = c <===> a = b ^ c
               ans = c;    break;
           }
       }
   }
   return ans;
}

// soln-2: build trie tree using given nums (h = 32)
//         do XOR for each number by looking at trie
int findMaximumXOR(vector<int>& nums) {
   TrieNode root;
   root.buildTrie(nums);
```

```
    int ans = 0;
    for (int x : nums) ans = max(ans, root.getMaxXOR(x));
    return ans;
}

struct TrieNode {
    TrieNode* next[2];           // bit can be either 0 or 1
    TrieNode() { next[0] = next[1] = nullptr; }

    // binary tree with h = 32
    void buildTrie(vector<int>& nums) {
        for (int x : nums) {
            TrieNode* cur = this;
            for (int i = 31; i >= 0; --i) {
                if (!cur->next[(x >> i) & 1]) cur->next[(x >> i) & 1] = new TrieNode;
                cur = cur->next[(x >> i) & 1];
            }
        }
    }

    // get max possible XOR value given x
    int getMaxXOR(int x) {
        int ans = 0;
        TrieNode* cur = this;
        for (int i = 31; i >= 0; --i) {
            int b = (x >> i) & 1;
            if (cur->next[b ^ 1]) {             // go with 1 if exist
                ans |= 1 << i;    cur = cur->next[b ^ 1];
            } else cur = cur->next[b];
        }
        return ans;
    }
};
```

Ref:

422. Valid word square

#425 - word squares

```
bool validWordSquare(vector<string>& words) {
    for (int i = 0; i < words.size(); ++i) {
        for (int j = 0; j < (int)words[i].length(); ++j) {
            if (j >= words.size() || i >= words[j].size()) return false;
            if (words[i][j] != words[j][i]) return false;
        }
    }
    return true;
}
```

Ref:

423/492/789. Reconstruct digits from English/Construct the rectangle ... math

```
// 423 - Reconstruct Original Digits from English
// soln-1: math + greedy
// 1. greedy construct those having unique digits (0->z, 2->w, 4->u, 6->x, 8->g)
// 2. then construct those left:
//    3->r - 0 - 4  (0 and 4 are constructed because they have unique char)
//    5->f - 4
//    7 -v - 5
```

```cpp
//    9->i - 5 - 6 - 8
//    1->o - 4 - 2 - 0
string originalDigits(string s) {
   vector<int> m(256), cnt(10);
   for (auto& ch : s) m[ch]++;

   cnt[0] = m['z'], cnt[2] = m['w'], cnt[4] = m['u'], cnt[6] = m['x'], cnt[8] = m['g'];
   cnt[3] = m['r'] - cnt[0] - cnt[4], cnt[5] = m['f'] - cnt[4], cnt[7] = m['v'] - cnt[5];
   cnt[9] = m['i'] - cnt[5] - cnt[6] - cnt[8];
   cnt[1] = m['o'] - cnt[4] - cnt[2] - cnt[0];

   string ans;
   for (int i = 0; i < cnt.size(); ++i) ans += string(cnt[i], i + '0');
   return ans;
}
```

```cpp
// 492 - construct the rectangle (given area, L >= W, abs(L - W) as small as possible)
// soln-1: math
vector<int> constructRectangle(int area) {
   int h = sqrt(area);     // max possible height
   while (area % h) --h;
   return {area / h, h};
}
```

```cpp
// 789 - Escape The Ghosts
// soln-1: math
// if any of ghost can reach to target than you, then you lose.
bool escapeGhosts(vector<vector<int>>& ghosts, vector<int>& target) {
    int d = abs(target[0]) + abs(target[1]);
    for (auto& g : ghosts) {
        int x = abs(g[0] - target[0]) + abs(g[1] - target[1]);
        if (x <= d) return false;
    }
    return true;
}
```

```cpp
// 1006 - Clumsy Factorial
// soln-1: brute-force
// a*b/c+d-e*f/g+... = a*b/c+(d-e*f/g)+(...)
// f(n) = n - (n-1) * (n-2) / (n-3)
int h(int N) {
    switch (N) {
        case 1: return 1;
        case 2: return 1;    // 2 - 1
        case 3: return 1;    // 3 - 2 * 1
        case 4: return -2;   // 4 - 3 * 2 / 1
        default: return N - (N - 1) * (N - 2) / (N - 3) + h(N - 4);
    }
}
int clumsy(int N) {
    switch (N) {
        case 1: return 1;
        case 2: return 2;    // 2 * 1
        case 3: return 6;    // 3 * 2 / 1
        default: return N * (N - 1) / (N - 2) + h(N - 3);
    }
}
// soln-2: math
// i * (i-1) / (i-2) = i+1 when i > 4
// proof:
// i*(i-1)/(i-2) = i*(1+1/(i-2)) = i+1+2/(i-2), 2/(i-2) < 1 ==> i > 4
//
// i * (i-1) / (i-2) + (i-3) - (i-4) * (i-5) / (i-6) + (i-7) - (i-8) * .... + rest
// ~~~~~~~~~~~~~~~~~            ~~~~~~~~~~~~~~~~~~~~
// i+1                 + (i-3) - (i-3)                                    + rest
// i+1 + rest
```

```
// so the rest could only be 4 possibilities for the rest part:
// 0 - (i+1) + ... + 5 - (4*3/2) + 1 - (nothing)     ==> i+1
// 1 - (i+1) + ... + 6 - (5*4/3) + 2 - 1             ==> i+2
// 2 - (i+1) + ... + 7 - (6*5/4) + 3 - 2 * 1         ==> i+2
// 3 - (i+1) + ... + 8 - (7*6/5) + 4 - 3 * 2 / 1     ==> i-1
```

Ref:

425. Word squares

Given a set of words (without duplicates), find all word squares you can build from them.

A sequence of words forms a valid word square if the *k*th row and column read the exact same string, where 0 ≤ *k* < max(numRows, numColumns).

For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

Note:

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet a-z.

```
// soln-1: build a trie tree then fill the matrix (backtracking enumerate)
vector<vector<string>> wordSquares(vector<string>& words) {
   Trie trie;
   trie.build(words);

   vector<vector<string>> ans;
   int n = words[0].length();
   vector<string> square(n, string(n, 0));
   for (const string& w : words) {
       helper(0, w, &trie, square, ans);
   }

   return ans;
}

void helper(int row, const string& word, const Trie* trie, vector<string>& square, vector<vector<string>>& ans)
{
   if (row >= square.size()) { ans.push_back(square); return; }

   square[row] = word;                                              // fill row
   for (int i = 0; i < square.size(); ++i) square[i][row] = word[i];   // fill col

   string prefix = square[row + 1].substr(0, row + 1);
   for (auto& next : trie->startsWith(prefix)) {
       helper(row + 1, next, trie, square, ans);
   }
}
```

Ref: #422

427/558. Quad tree construction / intersection

```
A:                   B:                   C (A or B):
+-------+-------+    +-------+---+---+    +-------+-------+
|       |       |    |       | F | F |    |       |       |
|   T   |   T   |    |   T   +---+---+    |   T   |   T   |
|       |       |    |       | T | T |    |       |       |
+-------+-------+    +-------+---+---+    +-------+-------+
|       |       |    |       |       |    |       |       |
|   F   |   F   |    |   T   |   F   |    |   T   |   F   |
|       |       |    |       |       |    |       |       |
+-------+-------+    +-------+-------+    +-------+-------+
```

```
// 427 - construct quad tree
Node* helper(vector<vector<int>>& grid, int x, int y, int d) {
   if (d <= 0) return nullptr;

   bool needPartition = false;
   for (int i = x; !needPartition && i < x + d; ++i) {
       for (int j = y; !needPartition && j < y + d; ++j) {
           if (grid[x][y] != grid[i][j]) needPartition = true;
       }
   }
   if (!needPartition) return new Node(grid[x][y], true, 0, 0, 0, 0);

   d /= 2;
   return new Node(grid[x][y], false,                         // value is any actually
                       helper(grid, x, y, d),                 // top-left
                       helper(grid, x, y + d, d),             // top-right
                       helper(grid, x + d, y, d),             // bottom-left
                       helper(grid, x + d, y + d, d));        // bottom-right
}

Node* construct(vector<vector<int>>& grid) {
   return helper(grid, 0, 0, grid.size());
}
```

```
// 558 - quad tree intersection
Node* intersect(Node* qt1, Node* qt2) {
   // return if anyone is leaf
   if (qt1->isLeaf && qt1->val) return qt1;    // should return copy of t1
   if (qt2->isLeaf && qt2->val) return qt2;
   if (qt1->isLeaf && !qt1->val) return qt2;   // depend on qt2 in this case
   if (qt2->isLeaf && !qt2->val) return qt1;

   // both are not leafs
   auto tl = intersect(qt1->topLeft, qt2->topLeft);
   auto tr = intersect(qt1->topRight, qt2->topRight);
   auto bl = intersect(qt1->bottomLeft, qt2->bottomLeft);
   auto br = intersect(qt1->bottomRight, qt2->bottomRight);

   // 4-dirs are all leafs and having same value
   if (tl->isLeaf && tr->isLeaf && bl->isLeaf && br->isLeaf &&
       tl->val == tr->val && tl->val == bl->val && tl->val == br->val) {
       return new Node(tl->val, true, 0, 0, 0, 0);
   }

   // not all 4-dirs are leafs OR having diff. val
   return new Node(false, false, tl, tr, bl, br);
}
```

Ref:

429/589/590/559. N-ary tree level/pre/post order traversal/maxdepth

```
vector<vector<int>> levelOrder(Node* root) {
   vector<vector<int>> ans;
   queue<Node*> q;
   if (root) q.push(root);
   while (!q.empty()) {
       ans.push_back({});
       vector<int>& this_level = ans.back();
       for (int i = q.size(); i > 0; --i) {
           this_level.push_back(q.front()->val);
           auto& children = q.front()->children;
           for (auto it = children.begin(); it != children.end(); ++it) {
               q.push(*it);
           }
           q.pop();
       }
   }
   return ans;
}

vector<int> preorder(Node* root) {
   vector<int> ans;
   stack<Node*> stk;
   if (root) stk.push(root);
   while (!stk.empty()) {
       ans.push_back(stk.top()->val);
       auto node = stk.top();  stk.pop();
       for (auto it = node->children.rbegin(); it != node->children.rend(); ++it) {
           stk.push(*it);
       }
   }
   return ans;
}

vector<int> postorder(Node* root) {
   stack<pair<Node*, int>> stk;        // <Node*, op>
   if (root) stk.push({root, 0});      // {node, discover/visit}
   vector<int> ans;
   while (!stk.empty()) {
       auto& p = stk.top();
       if (p.second == 0) {    // discovery -> visit
           p.second = 1;       // change state
           auto& children = p.first->children;
           for (auto it = children.rbegin(); it != children.rend(); ++it) {
               stk.push({*it, 0});
           }
       } else {
           ans.push_back(p.first->val), stk.pop();
       }
   }
   return ans;
}
```

Ref:

432. TODO

Solution:

Ref:

434. Number of segments in a string

```
// 434 - number of segments in a string
int countSegments(string s) {
   int ans = 0;
   bool in_new_seg = false;
   for (auto ch : s) {
       if (ch == ' ') {
           if (in_new_seg) ++ans;
           in_new_seg = false;
       } else {
           in_new_seg = true;
       }
   }
   return in_new_seg ? ans + 1 : ans;
}
```

Ref:

435/452/646. Non-overlapping intervals/Min arrow to burst balloons/Max length of pair chain

435 - Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

`Input: [ [1,2], [2,3], [3,4], [1,3] ], Output: 1`

`Explanation: [1,3] can be removed and the rest of intervals are non-overlapping.`

646 - You are given `n` pairs of numbers. In every pair, the first number is always smaller than the second number. Now, we define a pair `(c, d)` can follow another pair `(a, b)` if and only if `b < c`. Chain of pairs can be formed in this fashion.
Given a set of pairs, find the length longest chain which can be formed. You needn't use up all the given pairs. You can select pairs in any order.

```
// 435 - non-overlapping intervals
// soln-1: greedy - sort by earliest finishing time
// greedy remove overlapping intervals
int eraseOverlapIntervals(vector<Interval>& intervals) {
   if (intervals.size() == 0) return 0;
   sort(intervals.begin(), intervals.end(), [](const Interval& a, const Interval& b){
       return a.end <= b.end;
   });

   int ans = 0;
   Interval& pre = intervals[0];
   for (int i = 1; i < intervals.size(); ++i) {
       if (pre.end > intervals[i].start) ++ans;
       else pre = intervals[i];
   }
   return ans;
}
```

```
// 452 - min arrows to burst balloons
// soln-1: greedy - sort by earliest finishing time
// if there is overlap between current and next ballon, 1 arrow is fine.
// otherwise, we need a new arrow.
int findMinArrowShots(vector<pair<int, int>>& points) {
   if (points.size() == 0) return 0;
```

```
    sort(points.begin(), points.end(), [](const pair<int, int>& a, const pair<int, int>& b) {
        return a.second <= b.second;
    });

    int ans = 1;
    auto cur = points[0];
    for (int i = 1; i < points.size(); ++i) {
        if (cur.second >= points[i].first) continue;
        cur = points[i], ++ans;
    }
    return ans;
}
```

```
// 646 - max length of pair chain
// soln-1: greedy - sort by earliest finishing time
// the earliest finish pair must be better worth keeping than 2nd earliest finish pair
int findLongestChain(vector<vector<int>>& pairs) {
    sort(pairs.begin(), pairs.end(), [](vector<int>& a, vector<int>& b) { return a[1] < b[1]; });
    int ans = 0;
    for (auto i = 0; i < pairs.size(); NULL) {
        ++ans;
        int j = i + 1;
        for (int finished = pairs[i][1]; j < pairs.size(); ++j) {
            if (pairs[j][0] > finished) break;      // found next pair available
        }
        i = j;
    }
    return ans;
}
```

Ref: #252

440. TODO

Solution:

Ref:

441. Arranging coins

You have a total of *n* coins that you want to form in a staircase shape, where every *k*-th row must have exactly *k*coins.

Given *n*, find the total number of **full** staircase rows that can be formed.

n is a non-negative integer and fits within the range of a 32-bit signed integer.

Example 1:

```
n = 5
The coins can form the following rows:
¤
¤ ¤
¤ ¤
Because the 3rd row is incomplete, we return 2.
```

```
// 441 - soln-1: binary search
int arrangeCoins(int n) {
    long lo = 1, hi = n, ans = 0;
```

```
    while (lo <= hi) {
        long m = lo + (hi - lo) / 2;
        if (n >= ((1 + m) * m / 2)) ans = m, lo = m + 1;
        else hi = m - 1;
    }
    return ans;
}
```

Ref:

442. Find all dups in an array

Given an array of integers, 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements that appear twice in this array.

Could you do it without extra space and in O(*n*) runtime?

Example: `Input: [4,3,2,7,8,2,3,1], Output: [2,3]`

Solution:
mark negative of a position, if it already marked, then appeared more than once.

```
vector<int> findDuplicates(vector<int>& nums) {
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        int pos = abs(nums[i]) - 1;
        if (nums[pos] < 0) ans.push_back(abs(nums[i]));
        if (nums[pos] > 0) nums[pos] = -nums[pos];
    }
    return ans;
}
```

Ref: #448 #645

443. String compression

```
// soln-1: two-pointers
int compress(vector<char>& chars) {
    int w = 0, ch = 0, ch_num = 0;      // writer, current char
    for (int r = 0; r < chars.size(); ++r) {
        if (ch != chars[r]) {
            if (ch_num) {
                chars[w++] = ch;
                if (ch_num > 1) {
                    for (auto x : to_string(ch_num)) chars[w++] = x;
                }
            }
            ch = chars[r], ch_num = 1;
        } else {
            ch_num++;
        }
    }
    if (ch_num) {
        chars[w++] = ch;
        if (ch_num > 1) {
            for (auto x : to_string(ch_num)) chars[w++] = x;
        }
    }
    return w;
}
```

Ref:

446. TODO

Solution:

Ref:

447. Number of boomeranges

Given *n* points in the plane that are all pairwise distinct, a "boomerang" is a tuple of points `(i, j, k)` such that the distance between `i` and `j` equals the distance between `i` and `k` (**the order of the tuple matters**). Find the number of boomerangs.

```
// soln-1: hashmap (question is a little confusing)
// <distance, n = # of points w/ same distance>
// # of tuple = n-choose-2 (2 plus starting point form one tuple)
int numberOfBoomerangs(vector<pair<int, int>>& points) {
   int ans = 0;
   unordered_map<int, int> m;
   for (int i = 0; i < points.size(); ++i) {
       m.clear();
       for (int j = 0; j < points.size(); ++j) {
           if (i == j) continue;

           int x = points[i].first - points[j].first;
           int y = points[i].second - points[j].second;
           m[x * x + y * y]++;
       }
       for (auto& p : m) ans += p.second * (p.second - 1); // n-choose-2
   }
   return ans;
}
```

Ref:

448. Find all numbers disappeared in an array

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

Example: `Input: [4,3,2,7,8,2,3,1], Output: [5,6]`

Solution:
soln-1: put all the number to the right place, if the place does not have the right number, then missing.
soln-2: touch the right place of a number by **mark negative**, if the place hasn't been touched, then missing.

- *appear twice* seems useless.

```
// soln-1: swap the number to the right place
vector<int> findDisappearedNumbers(vector<int>& nums) {
   for (int i = 0; i < nums.size(); ++i) {
       while (nums[i] != i + 1 && nums[i] != nums[nums[i] - 1]) swap(nums[i], nums[nums[i] - 1]);
   }
```

```
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] != i + 1) ans.push_back(i + 1);
    }
    return ans;
}

// soln-2: touch the place for a number
vector<int> findDisappearedNumbers(vector<int>& nums) {
    for (int i = 0; i < nums.size(); ++i) {
        int pos = abs(nums[i]) - 1;
        if (nums[pos] > 0) nums[pos] = -nums[pos];
    }

    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] > 0) ans.push_back(i + 1);
    }
    return ans;
}
```

Ref: #442

453/462. Min moves to equal array elements / II

453 - Given a non-empty integer array of size *n*, find the minimum number of moves required to make all array elements equal, where a move is incrementing *n* - 1 elements by 1.

Example:

```
Input: [1,2,3], Output: 3
Explanation: Only three moves are needed (remember each move increments two elements):
[1,2,3]  =>  [2,3,3]  =>  [3,4,3]  =>  [4,4,4]
```

462 - where a move is incrementing a selected element by 1 or decrementing a selected element by 1. The above example will be 2.

```
// 453 - soln-1: math
// assume m moves to reach to equal value x, define sum as the total value before any moves:
// (1) sum + m * (n - 1) = x * n
// (2) x = minNum + m
// For equation (2), because: minNum has to increase in each move to reach to final x.
int minMoves(vector<int>& nums) {
    long sum = accumulate(nums.begin(), nums.end(), 0L);
    int minNum = *min_element(nums.begin(), nums.end());
    return sum - minNum * nums.size();
}
```

```
// 462 - soln-1: fix the target point then increase min, decrease max
// To find the min moves, we need to increase/decrease min/max to target middle point (if sorted)
// let middle point be m, then the moves will be:
// m - a[i] if a[i] < m, otherwise a[i] - m.
int minMoves2(vector<int>& nums) {
    sort(nums.begin(), nums.end());

    int ans = 0;
    for (int i = 0, m = nums[nums.size() / 2]; i < nums.size(); ++i) {
        ans += m > nums[i] ? m - nums[i] : nums[i] - m;
    }
    return ans;
}
```

Ref:

455. Assign cookies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie. Each child i has a greed factor g_i, which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s_j. If $s_j >= g_i$, we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Input: [1,2], [1,2,3], **Output:** 2

Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2. You have 3 cookies and their sizes are big enough to gratify all of the children, You need to output 2.

```
// 455 - soln-1: greedy (multiset/upbound also does the job)
// sort cookies size and greed value accordingly, and then match
int findContentChildren(vector<int>& g, vector<int>& cookies) {
   sort(g.begin(), g.end()), sort(cookies.begin(), cookies.end());
   int ans = 0;
   for (int i = 0; i < cookies.size() && ans < g.size(); ++i) {
       if (g[ans] <= cookies[i]) ++ans;  // satisfied this child's greedy
   }
   return ans;
}
```

Ref:

458. Poor pigs (review)

There are 1000 buckets, one and only one of them contains poison, the rest are filled with water. They all look the same. If a pig drinks that poison it will die within 15 minutes. What is the minimum amount of pigs you need to figure out which bucket contains the poison within one hour.

Answer this question, and write an algorithm for the follow-up general case.

Follow-up:

If there are n buckets and a pig drinking poison will die within m minutes, how many pigs (x) you need to figure out the "poison" bucket within p minutes? There is exact one bucket with poison.

```
// 458 - soln-1: 2-pigs can verify upto 125 buckets within 1-hour
// given it will die within 15-mins. Array buckets as following
//  1  2  3  4  5
//  ...
// 21 22 23 24 25
// Let pig-1 drinks 1st row and wait for 15-mins, then 2nd row
// Let pig-2 drinks 1st col and wait for 15-mins, then 2nd col
// 3-pigs can verify upto 5*5*5 buckets.
// generic formula:
// (ceil(mintestToTest/minutesToDie) + 1)^pigs >= buckets
int poorPigs(int buckets, int minutesToDie, int minutesToTest) {
   int ans = 0;
   while (pow(minutesToTest / minutesToDie + 1, ans) < buckets) ++ans;
   return ans;
}
```

Ref:

459. Repeated substring pattern

Given a non-empty string check if it can be constructed by taking a substring of it and appending multiple copies of the substring together. You may assume the given string consists of lowercase English letters only and its length will not exceed 10000.

Example 1:

`**Input:** "abab", **Output:** True`

`**Explanation:** It's the substring "ab" twice.`

```
// soln-1: brute-force to slice string and cmp each slice
bool repeatedSubstringPattern(string s) {
   for (int segs = 2, len = s.length(); segs <= len; ++segs) {
       if (len % segs) continue;
       // do verify for each segment
       bool mismatch = false;
       for (int seg_idx = 1, seg_len = len / segs; !mismatch && seg_idx < segs; ++seg_idx) {
           for (int k = 0; !mismatch && k < seg_len; ++k) {
               if (s[k] != s[seg_len * seg_idx + k]) mismatch = true;
           }
       }
       if (!mismatch) return true;
   }
   return false;
}
```

Ref:

461. Hamming distance

The Hamming distance between two integers is the number of positions at which the **corresponding bits are different.**

Given two integers `x` and `y`, calculate the Hamming distance.

Solution:
do XOR and count 1s.

#477 - total hamming distance (Hamming distance of all pairs of given numbers)

```
int hammingDistance(int x, int y) {
   int ans = 0;
   for (int n = x ^ y; n; n &= n - 1) ans++;
   return ans;

   // use gcc builtin to count 1s.
   // return __builtin_popcount(x ^ y);
}
```

Ref:

463/1030/1034. Island perimeter/Cells in distance order/Color boarder

```
// 463 - Island perimeter
// soln-1: bfs
int islandPerimeter(vector<vector<int>>& grid) {
   if (grid.empty()) return 0;
   int ans = 0, row = grid.size(), col = grid[0].size();
   for (int i = 0; i < row; ++i) {
       for (int j = 0; j < col; ++j) {
           if (0 == grid[i][j]) continue;
           if (0 == i || !grid[i-1][j])        ans++;
```

```
            if (row - 1 == i || !grid[i+1][j])  ans++;
            if (0 == j || !grid[i][j-1])        ans++;
            if (col - 1 == j || !grid[i][j+1])  ans++;
        }
    }
    return ans;
}
```

```
// 1030 - Matrix Cells in Distance Order
// soln-1: bfs
vector<vector<int>> allCellsDistOrder(int R, int C, int r0, int c0) {
    vector<vector<int>> ans;
    vector<bool> visited(R * C);
    queue<pair<int, int>> q;
    q.push({r0, c0}), visited[r0 * C + c0] = true;
    while (!q.empty()) {
        for (int i = q.size(); i > 0; --i) {
            auto x = q.front().first, y = q.front().second;
            ans.push_back({x, y}), q.pop();
            for (auto& d : vector<pair<int, int>>{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}) {
                auto nx = x + d.first, ny = y + d.second;
                if (nx < 0 || nx >= R || ny < 0 || ny >= C || visited[nx * C + ny]) continue;
                q.push({nx, ny}), visited[nx * C + ny] = true;
            }
        }
    }
    return ans;
}
```

```
// 1034 - Coloring A Border
// soln-1: dfs (restore if neighbors are colored)
// not a board if all other 4-directions are colored.
void dfs(vector<vector<int>>& g, int x, int y, int color) {
   int m = g.size(), n = g[0].size();
   if (x < 0 || x >= m || y < 0 || y >= n || g[x][y] != color) return;
   g[x][y] = -g[x][y];                              // color it, then restore if not a boarder
   dfs(g, x + 1, y, color), dfs(g, x - 1, y, color), dfs(g, x, y + 1, color), dfs(g, x, y - 1, color);
   if ((x - 1 >= 0 && abs(g[x - 1][y]) == color) &&    // restore color if not a boarder
       (x + 1 < m  && abs(g[x + 1][y]) == color) &&
       (y - 1 >= 0 && abs(g[x][y - 1]) == color) &&
       (y + 1 < n  && abs(g[x][y + 1]) == color)) {
       g[x][y] = abs(g[x][y]);
   }
}
vector<vector<int>> colorBorder(vector<vector<int>>& grid, int r0, int c0, int color) {
   dfs(grid, r0, c0, grid[r0][c0]);
   for (auto& r : grid) {
       for (auto& x : r) {
           if (x < 0) x = color;
       }
   }
   return grid;
}
```

Ref:

465. TODO

Solution:

Ref:

466. Count the repetitions

Define `S = [s,n]` as the string S which consists of n connected strings s. For example, `["abc", 3]`="abcabcabc".

On the other hand, we define that string s1 can be obtained from string s2 if we can remove some characters from s2 such that it becomes s1. For example, “abc” can be obtained from “abdbec” based on our definition, but it can not be obtained from “acbbe”.

You are given two non-empty strings s1 and s2 (each at most 100 characters long) and two integers $0 \le n1 \le 10^6$ and $1 \le n2 \le 10^6$. Now consider the strings S1 and S2, where `S1=[s1,n1]` and `S2=[s2,n2]`. Find the maximum integer M such that `[S2,M]` can be obtained from `S1`.

Example:

`Input: s1="acb", n1=4, s2="ab", n2=2, Return: 2`

```
// soln-1: brute force - passed with optimization
// if s2 is repeated R times in S1 (expanded s1), then S2 (expanded s2) must be repeated R / n2 times in S1.
// there are other solns but quite hard to understand ...
int getMaxRepetitions(string s1, int n1, string s2, int n2) {
   const char* p1 = s1.c_str(), *p2 = s2.c_str();
   int c1 = 0, c2 = 0, i = 0, j = 0;
   while (c1 < n1) {
       if (p1[i] == p2[j]) {
           if (++j == s2.length()) {   // move s2-index only if matched with s1
               ++c2, j = 0;            // s2 is matched once
           }
       }
       if (++i == s1.length()) {       // move s1-index
           ++c1, i = 0;                // s1 is used once
       }
       if (0 == i && 0 == j) {         // optimization
           int skip = (n1 - c1) / c1;
           if (skip) {
               c2 *= (1 + skip);
               c1 += skip * c1;
           }
       }
   }
   return c2 / n2;
}
```

Ref:

467. Number of unique substrings in wraparound string (review)

Consider the string `s` to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so `s` will look like this: "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

Now we have another string `p`. Your job is to find out how many unique non-empty substrings of `p` are present in `s`. In particular, your input is the string `p` and you need to output the number of different non-empty substrings of `p` in the string `s`.

```
// soln-1: keep longest substring ending with each letter
// let dp(i) be the longest unique substring ending with p[i]
// dp(i) = max{dp(i), p[i-1] + 1 == p[i] ? len + 1 : 1}, len - previous substring length
// the unique substring is the sum of all longest substring ending with each letter
// for example, "z a b"       "a b a b c"
//               1 2 3 => 6    1 2 1 2 3 => 6 (a, b, c, ab, abc, bc)
int findSubstringInWraproundString(string p) {
   vector<int> dp(26);
```

```
    for (int i = 0, len = 0; i < p.length(); ++i) {
        int cur = p[i] - 'a';
        if (i > 0) {
            int pre = p[i - 1] - 'a';
            (pre + 1) % 26 == cur ? len++ : len = 1;
        } else {
            len = 1;
        }
        dp[cur] = max(dp[cur], len);    // max length substring ending with current char
    }

    return accumulate(dp.begin(), dp.end(), 0);
}
```

Ref:

469. TODO

Solution:

Ref:

471. TODO

Solution:

Ref:

472. Concatenated words

Given a list of words (**without duplicates**), please write a program that returns all concatenated words in the given list of words. A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example:

```
Input: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]
Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]


Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";

"dogcatsdog" can be concatenated by "dog", "cats" and "dog";

"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".
```

```
// soln-1: dynamic programming
// essentially this is a word break problem, see #139
// put the words into dictionary, then check each one against others
// O(len^2) time to check if breakable.
bool breakable(const string& word, unordered_set<string>& dict, int maxl, int minl) {
    if (word.length() < minl * 2) return false;     // defensive code

    vector<bool> dp(word.length());
    for (int i = minl - 1; i < word.length(); ++i) {
```

```
        for (int j = max(0, i - maxl); j <= i; ++j) {
            if (j > 0 && dp[j - 1] == false) continue;  // j-1 must be breakable first

            string str = word.substr(j, i - j + 1);
            if (dict.find(str) != dict.end()) {
                dp[i] = true;   break;
            }
        }

        if (dp[i] && dict.find(word.substr(i + 1)) != dict.end()) return true;  // optimize
    }
    return dp.back();
}

vector<string> findAllConcatenatedWordsInADict(vector<string>& words) {
    // words are distinct, shorter words must not breakable, so sort it first
    sort(words.begin(), words.end(), [](const string& a, const string& b){
        return a.length() < b.length();
    });
    unordered_set<string> dict;
    int maxl = max(1, (int)words.back().length()), minl = max(1, (int)words.front().length());

    vector<string> ans;
    for (string& word : words) {
        if (breakable(word, dict, maxl, minl)) ans.push_back(word);
        dict.insert(word);      // longer words can only be broke by shorter ones.
    }
    return ans;
}
```

Ref: #139 #694

473. TODO

Solution:

Ref:

474. Ones and zeroes (review)

find the maximum number of strings that you can form with given **m** `0s` and **n** `1s`. Each `0` and `1` can be used at most **once**.

Note:

1. The given numbers of `0s` and `1s` will both not exceed `100`
2. The size of given string array won't exceed `600`.

Example 1:

```
Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3, Output: 4
Explanation: This are totally 4 strings can be formed by the using of 5 0s and 3 1s, which are
“10,”0001”,”1”,”0”
```

#322 - coin change (0/1 knapsack problem)
#741 - cherry pickup (3D to 2D dynamic programming)

```
// soln-1: 0/1 knapsack dynamic programming
// to convert this problem to 0/1 knapsack, consider given m/n as capacity
// and each string has some values as 0/1 as their unit. the question asks
```

```
// how many strings we can put into knapsack. now we have each string as item.
//
// another difficulty is to view m/n as capacity expressed as 2D matrix.
// let dp(i, j, k) be the max number of strings we can take, as usual we can take i or not.
// dp(i, j, k) = max{dp(i - 1, j, k), dp(i - 1, j - ones, k - zeros) + 1}
//
// space optimization - as ith result depends on previous item,
// we can do it from right-bottom to top-left, it's same as "cherry pickup"
int findMaxForm(vector<string>& strs, int m, int n) {
   vector<vector<int>> dp(m + 1, vector<int>(n + 1));
   for (string& str : strs) {
       int zeros = count(str.begin(), str.end(), '0');
       int ones = str.length() - zeros;

       for (int i = m; i >= zeros; --i) {
           for (int j = n; j >= ones; --j) {
               dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);
           }
       }
   }
   return dp[m][n];
}
```

Ref: #322 #741

475. Heaters

Winter is coming! Your first job during the contest is to design a standard heater with fixed warm radius to warm all the houses.

Now, you are given positions of houses and heaters on a horizontal line, find out minimum radius of heaters so that all houses could be covered by those heaters.

So, your input will be the positions of houses and heaters separately, and your expected output will be the minimum radius standard of heaters.

```
// 475 - soln-1: binary search
// for each house, find its nearest heater, which is the min(leftSide, rightSide)
// the min distance is the max distance among house
// Example:
//         1   2   3   4   5   6   7   8   9    house index
//         *           *       *                heaters
//
//         0   2   1   0   1   0   -   -   -    (distance to nearest right heater)
//         0   1   2   0   1   0   1   2   3    (distance to nearest left heater)
//
//         0   1   1   0   1   0   1   2   3    (res = minimum of above two)
// Result is maximum value in res, which is 3.
int getCloset(vector<int>& heaters, int house) {
    if (house > heaters.back()) return house - heaters.back();
    if (heaters.front() > house) return heaters.front() - house;

    int lo = 0, hi = heaters.size();
    while (lo + 1 < hi) {
        int m = lo + (hi - lo) / 2;
        if (heaters[m] == house) return 0;
        heaters[m] > house ? hi = m : lo = m;
    }
    return min(heaters[hi] - house, house - heaters[lo]);
```

```
}

int findRadius(vector<int>& houses, vector<int>& heaters) {
    sort(heaters.begin(), heaters.end());

    int ans = INT_MIN;
    for (auto h : houses) {
        ans = max(ans, getCloset(heaters, h));
    }
    return ans;
}
```

Ref:

477. Total hamming distance

find the total Hamming distance between all pairs of the given numbers.

Example:

```
Input: 4, 14, 2, Output: 6


Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2 is 0010 (just showing the four bits relevant in this case). So the answer will be: HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 + 2 + 2 = 6.
```

Solution:

soln-1: bit-by-bit consider the distance

```
    0100

    1110

    0010

------------, for highest bits {0, 0, 1}, we know the total distance is number-of-0s * number-of-1s
```

```
int totalHammingDistance(vector<int>& nums) {
   int ans = 0;
   for (int i = 0; i < 32; ++i) {
       int ones = 0, zeros = 0;
       for (int x : nums) (x & (1 << i)) ? ++ones : ++zeros;
       ans += ones * zeros;
   }
   return ans;
}
```

Ref:

479. TODO

Solution:

Ref:

481. TODO

Solution:

Ref:

482. License key formatting

```
// 482 - soln-1: brute-force from backward
string licenseKeyFormatting(string S, int K) {
   string ans;
   for (int i = S.length() - 1, j = 0; i >= 0; --i) {
       if (S[i] == '-') continue;

       if (j && j % K == 0) ans += '-';
       ans += toupper(S[i]);
       j++;
   }
   reverse(ans.begin(), ans.end());
   return ans;
}
```

Ref:

490/499/505. The Maze/II/III

```
// 490 - the maze (ball won't stop until hitting wall. determine if the ball could stop at the destination)
// soln-1: BFS
bool hasPath(vector<vector<int>>& m, pair<int, int> start, pair<int, int> end) {
   if (start == end) return true;
   const vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

   queue<pair<int, int>> q;
   q.push(start), m[start.first][start.second] = 2;       // mark it as visited
   while (!q.empty()) {
       start = q.front(); q.pop();
       for (auto dir : dirs) {
           int x = start.first, y = start.second;

           // rolling the ball along with current direction until hitting boundary
           while (x >= 0 && x < m.size() && y >= 0 && y < m[0].size() && m[x][y] != 1) {
               x += dir.first, y += dir.second;
           }
           x -= dir.first, y -= dir.second;
           if (x < 0 || x >= m.size() || y < 0 || y >= m[0].size() || m[x][y] != 0) continue;     // visited

           if (x == end.first && y == end.second) return true;
           q.push({x, y}); m[x][y] = 2;    // put (x, y) into q and mark it as visited.
       }
   }
   return false;
}

// 490 - the maze (ball won't stop until hitting wall. determine if the ball could stop at the destination)
// soln-2: DFS
bool hasPath(vector<vector<int>>& m, pair<int, int> start, pair<int, int> end) {
   if (start == end) return true;
   const vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};

   m[start.first][start.second] = 2;        // mark as visited
   for (auto dir : dirs) {
```

```
        int x = start.first, y = start.second;
        while (x >= 0 && x < m.size() && y >= 0 && y < m[0].size() && m[x][y] != 1) {
            x += dir.first, y += dir.second;
        }
        x -= dir.first, y -= dir.second;
        if (x < 0 || x >= m.size() || y < 0 || y >= m[0].size() || m[x][y] != 0) continue;     // visited

        if (hasPath(m, pair<int, int>{x, y}, end)) return true;
    }
    return false;
}
```

```
// 499 - the maze III (find out how the ball could drop into the hole by moving the shortest distance)
// soln-1: BFS
string findShortestWay(vector<vector<int>>& maze, vector<int> ball, vector<int> hole) {
    const vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
    const vector<char> dirs_ch{'d', 'r', 'u', 'l'};

    int rows = maze.size(), cols = maze[0].size();
    vector<vector<int>> dist(rows, vector<int>(cols, INT_MAX));
    unordered_map<int, string> path_map;
    queue<pair<int, int>> q;

    q.push({ball[0], ball[1]});
    dist[ball[0]][ball[1]] = 0;
    path_map[ball[0] * cols + ball[1]] = string("");

    while(!q.empty()) {
        auto start = q.front();  q.pop();
        int walkedSteps = dist[start.first][start.second];
        for (int i = 0; i < dirs.size(); ++i) {
            int x = start.first, y = start.second, rollingSteps = 0;
            while (x >= 0 && x < rows && y >= 0 && y < cols && maze[x][y] != 1) {
                x += dirs[i].first, y += dirs[i].second, ++rollingSteps;
            }
            x -= dirs[i].first, y -= dirs[i].second, --rollingSteps;
            if (x < 0 || x >= rows || y < 0 || y >= cols || maze[x][y] != 0) continue;
            if (x == ball[0] && y == ball[1]) continue; // hitting back

            int newDist = walkedSteps + rollingSteps;
            if (dist[x][y] < newDist) continue;

            string newPath = path_map[start.first * cols + start.second] + dirs_ch[i];
            string& oldPath = path_map[x * cols + y];
            if (dist[x][y] == newDist && !oldPath.empty() && oldPath.compare(newPath) < 0) continue;

            oldPath = newPath;
            dist[x][y] = newDist;
            if (x != hole[0] || y != hole[1]) q.push({x, y});
        }
    }
    return path_map[hole[0] * cols + hole[1]];
}
```

```
// 505 - the maze II (find out the shortest distance from start to dest)
// soln-1: DFS
void helper(vector<vector<int>>& maze, vector<vector<int>>& dist, pair<int, int> start, pair<int, int> end, int
walkedStep) {
    const vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
    if (start == end) return;

    for (auto dir : dirs) {
        int x = start.first, y = start.second, rollingStep = 0;
```

```
        while (x >= 0 && x < maze.size() && y >= 0 && y < maze[0].size() && maze[x][y] != 1) {
            x += dir.first, y += dir.second, ++rollingStep;
        }
        x -= dir.first, y -= dir.second, --rollingStep;
        if (x < 0 || x >= maze.size() || y < 0 || y >= maze[0].size() || maze[x][y] != 0) continue;
        if (dist[x][y] > rollingStep + walkedStep) {
            dist[x][y] = rollingStep + walkedStep;
            helper(maze, dist, pair<int, int>{x, y}, end, dist[x][y]);
        }
    }
}

int shortestDistance(vector<vector<int>>& maze, vector<int> start, vector<int> dest) {
    int rows = maze.size(), cols = maze[0].size();
    vector<vector<int>> dist(rows, vector<int>(cols, INT_MAX));
    helper(maze, dist, pair<int, int>{start[0], start[1]}, pair<int, int>{dest[0], dest[1]}, 0);
    return dist[dest[0]][dest[1]] == INT_MAX ? -1 : dist[dest[0]][dest[1]];
}

// 505 - the maze II (find out the shortest distance from start to dest)
// soln-2: BFS
int shortestDistance(vector<vector<int>>& maze, vector<int> start, vector<int> dest) {
    const vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
    vector<vector<int>> dist(maze.size(), vector<int>(maze[0].size(), INT_MAX));
    queue<pair<int, int>> q;
    q.push({start[0], start[1]});   dist[start[0]][start[1]] = 0;
    while (!q.empty()) {
        auto s = q.front(); q.pop();
        int walkedSteps = dist[s.first][s.second];
        for (auto dir : dirs) {
            int x = s.first, y = s.second, rollingSteps = 0;
            while (x >= 0 && x < maze.size() && y >= 0 && y < maze[0].size() && maze[x][y] != 1) {
                x += dir.first, y += dir.second, rollingSteps++;
            }
            x -= dir.first, y -= dir.second, rollingSteps--;
            if (x < 0 || x >= maze.size() || y < 0 || y >= maze[0].size()) continue;
            if (dist[x][y] <= walkedSteps + rollingSteps) continue;

            dist[x][y] = walkedSteps + rollingSteps;
            if(x != dest[0] || y != dest[1]) q.push({x, y});
        }
    }
    return dist[dest[0]][dest[1]] ==  INT_MAX ? -1 : dist[dest[0]][dest[1]];
}
```

```
// 1036 - Escape a Large Maze
// soln-1: bfs/dfs
// 1. the key to terminate bfs/dfs search is len(block) <= 200,
//    in a 45-degree case, this could block (1+..+199)=19900 cells.
// 2. then use 2-way search: either hit target/source, or BOTH from direction can visit more than 19900 cells.
const int MAX_CELLS = 19900;
bool dfs(int x, int y, pair<int, int> target, unordered_set<long long>& blocks, unordered_set<long long>& seen)
{
    if (x < 0 || x > 1e6 - 1 || y < 0 || y > 1e6 - 1) return false;
    if (blocks.count((long long)x << 32 | y) || seen.count((long long)x << 32 | y)) return false;
    seen.insert((long long)x << 32 | y);
    if ((x == target.first && y == target.second) || seen.size() > MAX_CELLS) return true;
    return dfs(x + 1, y, target, blocks, seen) ||
           dfs(x - 1, y, target, blocks, seen) ||
           dfs(x, y + 1, target, blocks, seen) ||
           dfs(x, y - 1, target, blocks, seen);
}
bool isEscapePossible(vector<vector<int>>& blocked, vector<int>& source, vector<int>& target) {
    unordered_set<long long> seen1, seen2, blocks;
```

```
    for (auto& b : blocked) blocks.insert((long long)b[0] << 32 | b[1]);
    return dfs(source[0], source[1], {target[0], target[1]}, blocks, seen1) &&
           dfs(target[0], target[1], {source[0], source[1]}, blocks, seen2);
}
```

Ref: #286

491. Increasing subsequences

Given an integer array, your task is to find all the different possible increasing subsequences of the given array, and the length of an increasing subsequence should be at least 2 .

Example:

`Input: [4, 6, 7, 7]`

`Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7], [4,7,7]]`

```
// soln-1: backtracking
void helper(int start, const vector<int>& nums, vector<int>& seq, vector<vector<int>>& ans) {
   if (seq.size() > 1) ans.push_back(seq);

   unordered_set<int> dup;
   for (int i = start; i < nums.size(); ++i) {
       if (dup.find(nums[i]) != dup.end()) continue;
       if (!seq.empty() && seq.back() > nums[i]) continue;

       seq.push_back(nums[i]);
       helper(i + 1, nums, seq, ans);
       seq.pop_back();
       dup.insert(nums[i]);
   }
}
vector<vector<int>> findSubsequences(vector<int>& nums) {
   vector<vector<int>>  ans;
   vector<int> seq;
   helper(0, nums, seq, ans);
   return ans;
}
```

Ref:

495/649. Teemo attacking/Dota2 senate

```
// 495 - soln-1: brute-force
int findPoisonedDuration(vector<int>& timeSeries, int duration) {
   if (timeSeries.empty()) return 0;
   int ans = 0;
   for (int i = 1; i < timeSeries.size(); ++i) {
       ans += min(duration, timeSeries[i] - timeSeries[i - 1]);
   }
   ans += duration;    // plus the last attack
   return ans;
}
```

```
// 649 - soln-1: greedy kill next closest other senate will give the max benefit to our party
// there is a smart way to kill next closest, which use Q.
string predictPartyVictory(string senate) {
   queue<int> rq, dq;
   for (int i = 0; i < senate.length(); ++i) {
```

```
        senate[i] == 'R' ? rq.push(i) : dq.push(i);
    }

    while (!rq.empty() && !dq.empty()) {
        if (rq.front() < dq.front()) {                  // smaller is the winner
            rq.push(rq.front() + senate.length());      // +n for next round comparison
        } else {
            dq.push(dq.front() + senate.length());
        }
        rq.pop(), dq.pop();
    }
    return !rq.empty() ? "Radiant" : "Dire";
}
```

Ref:

496/503/556/739/901/1019. Next greater element/Daily temperatures/Online stock (review)

monotone stack + math:

907 - sum of subarray mins (monotone stack + substring consider each letter)

```
// 496 - next greater element (given unique numbers, find each next greater number)
// soln-1: monotone stack + hashmap
vector<int> nextGreaterElement(vector<int>& findNums, vector<int>& nums) {
    stack<int> stk;                 // .:|
    unordered_map<int, int> mp;     // <num, next-greater-num>
    for (auto n : nums) {
        while (!stk.empty() && stk.top() < n) mp[stk.top()] = n, stk.pop();
        stk.push(n);
    }

    vector<int> ans;
    for (auto n : findNums) ans.push_back(mp.find(n) != mp.end() ? mp[n] : -1);
    return ans;
}
```

```
// 503 - next greater element (given a circular array)
// soln-1: monotone stack .:|
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> ans(n, -1);             // populate to this directly instead of using map
    vector<int> stk;                    // holding index this time

    for (int i = 0; i < 2 * n; ++i) {   // NICE TRICK to deal with circular array
        while (!stk.empty() && nums[stk.back()] < nums[i % n]) ans[stk.back()] = nums[i % n], stk.pop_back();
        if (i < n) stk.push_back(i);
    }

    return ans;
}
```

```
// 556 - next greater element (smallest greater than the given number)
// soln-1: brute-force NextPermutation
// Next permutation - next value that bigger than current
// Steps as follows, Ex. 536974 => 534679
// 1. from right to left scan till a[i] < a[i+1]: 53[69]74
// 2. use the smallest from right part to replace current digit: 53[7]964
// 3. sort right part: 537[469]
int nextGreaterElement(int n) {
    string sn = to_string(n);
    if (sn.length() < 2) return -1;
```

```
    int i = sn.length() - 2;
    for (; i >= 0; --i) {
        if (sn[i] < sn[i + 1]) break;
    }
    if (i < 0) return -1;

    // find smallest but bigger than sn[i] from right part
    int smallest = i + 1;
    for (int j = i + 1; j < sn.length(); ++j) {
        if (sn[j] < sn[smallest] && sn[j] > sn[i]) smallest = j;
    }
    swap(sn[i], sn[smallest]);
    sort(sn.begin() + i + 1, sn.end());
    return stoll(sn) > INT_MAX ? -1 : stoll(sn);
}
```

```
// 739 - Daily temperatures
// soln-1: monotone stack (similar to #503)
vector<int> dailyTemperatures(vector<int>& T) {
    vector<int> ans(T.size(), 0);
    stack<int> stk;
    for (int i = 0; i < T.size(); ++i) {
        while (!stk.empty() && T[stk.top()] < T[i]) {
            int idx = stk.top();    stk.pop();
            ans[idx] = i - idx;
        }
        stk.push(i);
    }
    return ans;
}
```

```
// 901 - online stock planner (backwards find the stock price great than current, return span)
// soln-1: monotone stack
// Ex. [100, 80, 60, 70, 60, 75, 85] => [1, 1, 1, 2, 1, 4, 6]
class StockSpanner {
    stack<pair<int, int>> _stk;    // <price, index>
    int _idx;
public:
    StockSpanner() : _idx(0) {}

    int next(int price) {
        ++_idx;
        while (!_stk.empty() && _stk.top().first <= price) _stk.pop();
        int ans = _stk.empty() ? _idx : _idx - _stk.top().second;
        _stk.emplace(price, _idx);
        return ans;
    }
};
```

```
// 962 - soln-1: monotone stack + binary search
// 1. keep a stack like |:. when going through from left
// 2. check against this stack if cur >= last-item-in-stack, binary search to find first item <= cur.
// 3. keep increasing monotone stack if cur < last-item-in-stack
int maxWidthRamp(vector<int>& A) {
    int ans = 0;
    vector<int> stk;
    for (int i = 0; i < A.size(); ++i) {
        if (!stk.empty() && A[i] >= A[stk.back()]) {
            int l = 0, h = stk.size() - 1;
            while (l + 1 < h) {
                int m = l + (h - l) / 2;
                A[stk[m]] <= A[i] ? h = m : l = m;
            }
            if (A[stk[h]] <= A[i]) ans = max(ans, i - stk[h]);
            if (A[stk[l]] <= A[i]) ans = max(ans, i - stk[l]);
```

```cpp
        }
        if (stk.empty() || A[stk.back()] > A[i]) stk.push_back(i);
    }
    return ans;
}
```

```cpp
// 962 - soln-2: monotone stack + greedy
int maxWidthRamp(vector<int>& A) {
    int ans = 0;
    vector<int> stk;
    for (int i = 0; i < A.size(); ++i) {
        if (stk.empty() || A[stk.back()] > A[i]) stk.push_back(i);
    }
    for (int i = A.size() - 1; i > ans; --i) {
        while (!stk.empty() && A[stk.back()] <= A[i]) {
            ans = max(ans, i - stk.back()), stk.pop_back();
        }
    }
    return ans;
}
```

```cpp
// 1019. Next Greater Node In Linked List
// soln-1: monotone stack
vector<int> nextLargerNodes(ListNode* head) {
    vector<pair<int, int>> stk;  // <val, idx>
    int idx = 0;
    vector<int> ans;
    for (auto cur = head; cur != nullptr; cur = cur->next, ++idx) {
        ans.push_back(0);
        while (!stk.empty() && stk.back().first < cur->val) {
            ans[stk.back().second] = cur->val, stk.pop_back();
        }
        stk.push_back({cur->val, idx});
    }
    return ans;
}
```

Ref: #31

501. Find *mode* in bst

Modes defined as the most frequently occurred element. BST in this question is defined as *greater or equal* increasing for in-order sequence.

Solution:
in-order traversal and compare the previous node value and current value. Keep 2 important info: frequency for current val, max frequency so far.

```cpp
void helper(TreeNode* root, TreeNode*& pre, int& curCount, int& maxCountSoFar, vector<int>& ansSoFar) {
   if (!root) return;

   helper(root->left, pre, curCount, maxCountSoFar, ansSoFar);

   (pre && pre->val == root->val) ? ++curCount : curCount = 1;
   if (curCount >= maxCountSoFar) {
       if (curCount > maxCountSoFar) ansSoFar.clear();
       ansSoFar.push_back(root->val);
       maxCountSoFar = max(maxCountSoFar, curCount);
   }
   pre = root;

   helper(root->right, pre, curCount, maxCountSoFar, ansSoFar);
}
```

```
vector<int> findMode(TreeNode* root) {
   vector<int> ans;
   TreeNode* pre = nullptr;
   int curCount = 0, maxCountSoFar = 0;
   helper(root, pre, curCount, maxCountSoFar, ans);

   return ans;
}
```

Ref:

502/1029. IPO/Two city scheduling

```
// 502 - IPO
// TODO
```

```
// 1029 - Two City Scheduling (fly to city A/B with min costs and split persons equally)
// soln-1: greedy (sort by savings)
// improvement: sort half of them is good ==> quick selection.
int twoCitySchedCost(vector<vector<int>>& costs) {
    sort(costs.begin(), costs.end(), [](vector<int>& a, vector<int>& b) {
        return (a[0] - a[1]) < (b[0] - b[1]);
    });
    int ans = 0;
    for (int i = 0; i < costs.size() / 2; ++i) {
        ans += costs[i][0] + costs[costs.size() - i - 1][1];
    }
    return ans;
}
```

Ref:

503. TODO

Solution:

Ref:

506. Relative ranks

```
// soln-1: easy sorting / ordered-map
vector<string> findRelativeRanks(vector<int>& nums) {
   vector<int> sorted(nums.size());
   for (int i = 0; i < nums.size(); ++i) sorted[i] = i;
   sort(sorted.begin(), sorted.end(), [&](int a, int b) {
       return nums[a] > nums[b];
   });

   vector<string> ans(nums.size());
   if (nums.size() > 0) ans[sorted[0]] = "Gold Medal";
   if (nums.size() > 1) ans[sorted[1]] = "Silver Medal";
   if (nums.size() > 2) ans[sorted[2]] = "Bronze Medal";
   for (int i = 3; i < nums.size(); ++i) ans[sorted[i]] = to_string(i + 1);
   return ans;
}
```

Ref:

508. Most frequent subtree sum

1. this question asks to compute every subtree sum. obviously f(x) = f(x.left) + f(x.right) + x
2. when we get a sum of subtree, we need to track the occurrence of this sum - map<sum of subtree, occurrence>.
3. to avoid repeat computation, need to compute from bottom-up

```
int helper(TreeNode* root, int& maxOccurSoFar, unordered_map<int, int>& occurrenceMap) {
   if (!root) return 0;

   int left = helper(root->left, maxOccurSoFar, occurrenceMap);
   int right = helper(root->right, maxOccurSoFar, occurrenceMap);

   int sum = left + right + root->val;
   maxOccurSoFar = max(maxOccurSoFar, ++occurrenceMap[sum]);
   return sum;
}

vector<int> findFrequentTreeSum(TreeNode* root) {
   int maxOccurSoFar = 0;
   unordered_map<int, int> occurrenceMap;    // <sum of subtree, occurrence>
   helper(root, maxOccurSoFar, occurrenceMap);

   vector<int> ans;
   for (auto it : occurrenceMap) {
       if (it.second == maxOccurSoFar) ans.push_back(it.first);
   }
   return ans;
}
```

Ref:

511. TODO

Solution:

Ref:

512. TODO

Solution:

Ref:

513. Find bottom left tree value

1. level order (with accessing right node first) does the job.
2. pre-order also works (as it always hits left deepest node first).

```
// soln-1: DFS and keep max height so far and update accordingly
void helper(TreeNode* n, int& maxH, int curH, int& ans) {
   if (NULL == n) return;
```

```
    if (curH > maxH) {
        ans = n->val; maxH = curH;
    }
    helper(n->left, maxH, curH + 1, ans);
    helper(n->right, maxH, curH + 1, ans);
}

int findBottomLeftValue(TreeNode* root) {
    int maxH = -1, ans = 0;
    helper(root, maxH, 0, ans);
    return ans;
}

// soln-2: level order with right subtree first, then left subtree which ensure to touch leftmost leaf lastly.
int findBottomLeftValue(TreeNode* root) {
    queue<TreeNode*> q; q.push(root);
    TreeNode* leftmostLeaf = root;

    while (!q.empty()) {
        for (int i = q.size(); i > 0; --i) {
            TreeNode* n = q.front(); q.pop();
            leftmostLeaf = n;
            if (n->right) q.push(n->right);
            if (n->left) q.push(n->left);
        }
    }
    return leftmostLeaf->val;
}
```

Ref: #515

514. Freedom trail

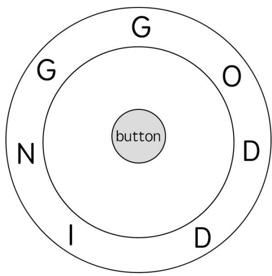
Given a string **ring**, which represents the code engraved on the outer ring and another string **key**, which represents the keyword needs to be spelled. You need to find the **minimum** number of steps in order to spell all the characters in the keyword.

Initially, the first character of the **ring** is aligned at 12:00 direction. You need to spell all the characters in the string **key** one by one by rotating the ring clockwise or anticlockwise to make each character of the string **key** aligned at 12:00 direction and then by pressing the center button.

At the stage of rotating the ring to spell the key character **key[i]**:

1. You can rotate the **ring** clockwise or anticlockwise **one place**, which counts as 1 step. The final purpose of the rotation is to align one of the string **ring's** characters at the 12:00 direction, where this character must equal to the character **key[i]**.
2. If the character **key[i]** has been aligned at the 12:00 direction, you need to press the center button to spell, which also counts as 1 step. After the pressing, you could begin to spell the next character in the key (next stage), otherwise, you've finished all the spelling.

Example:



Input: ring = "godding", key = "gd", **Output:** 4

Explanation:

For the first key character 'g', since it is already in place, we just need 1 step to spell this character.

For the second key character 'd', we need to rotate the ring "godding" anticlockwise by two steps to make it become "ddinggo".

Also, we need 1 more step for spelling.

So the final output is 4.

Note:

1. Length of both ring and **key** will be in range 1 to 100.
2. There are only lowercase letters in both strings and might be some duplicate characters in both strings.
3. It's guaranteed that string **key** could always be spelled by rotating the string **ring**.

```
// soln-1: brute force recursion with memorization
// for current position in key, need to enumerate systematically in ring string
// when we find the occurrence in ring string, we have 2 options:
// - clockwise from current position to target position
// - anti-clockwise from current position to target position
// we will greedy take whoever requires mini steps.
//
// let dp(i, j) be the mini steps to match key[j] when ring[i] is treated as starting point (ie. on 12:00
position)
// dp(i, j) = min {dp(k, j + 1) + min-steps-from-k-to-j, 0 <= k < length-of-ring}
//                          ~~~~~
//                            to make global optimization, current j has to consider the contribution for the
future.
// as current j depends on j+1, we need to iterate from right to left if using dynamic programming.
// time: O(k * r * r), k - length of key, r - length of ring, space: O(kr)
typedef unordered_map<int, unordered_map<int, int>> cache_t;
int helper(int ringpos, int keypos, const string& ring, const string& key, cache_t& dp) {
   if (keypos >= key.length()) return 0;
   if (dp.find(ringpos) != dp.end() &&
       dp[ringpos].find(keypos) != dp[ringpos].end()) return dp[ringpos][keypos];

   int ans = INT_MAX;
   for (int i = 0; i < ring.length(); ++i) {   // try out all matching chars
       if (ring[i] != key[keypos]) continue;

       int dist = abs(ringpos - i);            // one direction
       int rdist = ring.length() - dist;       // the other direction
       ans = min(ans, min(dist, rdist) + helper(i, keypos + 1, ring, key, dp));
   }
   return dp[ringpos][keypos] = ans + 1;       // +1 because of pushing the button
}

int findRotateSteps(string ring, string key) {
   cache_t dp;
   return helper(0, 0, ring, key, dp);
}

// soln-2: dynamic programming
// space could be done in O(r)
int findRotateSteps(string ring, string key) {
   int R = ring.length(), K = key.length();
   vector<vector<int>> dp(R, vector<int>(K));
```

```
    for (int k = K - 1; k >= 0; --k) {
        for (int r = 0; r < R; ++r) {
            dp[r][k] = INT_MAX;
            for (int i = 0; i < R; ++i) {
                if (ring[i] != key[k]) continue;

                int dist = abs(r - i);
                int rdist = ring.length() - dist;   // reverse distance
                dp[r][k] = min(dp[r][k], (k + 1 < K ? dp[i][k + 1] : 0) + min(dist, rdist));
            }
            dp[r][k] += 1;                          // +1 because of pushing the button
        }
    }
    return dp[0][0];
}
```

Ref:

515. Find largest value in each tree row

level order (could be done by dfs too)

```
vector<int> largestValues(TreeNode* root) {
    vector<int> ans;
    queue<TreeNode*> q;

    if (root) q.push(root);
    while (!q.empty()) {
        ans.push_back(q.front()->val);

        for (int i = q.size(); i > 0; --i) {
            TreeNode* n = q.front(); q.pop();
            ans.back() = max(ans.back(), n->val);
            if (n->left) q.push(n->left);
            if (n->right) q.push(n->right);
        }
    }

    return ans;
}
```

Ref: #513

516. Longest palindrome subsequence

Given a string s, find the longest palindromic subsequence length in s. You may assume that the maximum length of s is 1000.

Example 1:

Input: `"bbbab"`, Output: `4`

One possible longest palindromic subsequence is "bbbb".

```
// soln-1: recursion
//        if s[i] == s[j], the f(i..j) = 2 + f(i+1..j-1)
//        else f(i..j) = max{f(i+1..j), f(i..j-1)}
int longestPalindromeSubseq(string s) {
    if (s.length() <= 1) return s.length();
    if (s.front() == s.back()) return 2 + longestPalindromeSubseq(s.substr(1, s.length() - 2));
    return max(longestPalindromeSubseq(s.substr(1)), longestPalindromeSubseq(s.substr(0, s.length() - 1)));
}
```

```
// soln-2: dynamic programming
// let dp(i, j) be longest if s[i] == s[j], then 2 + dp(i+1, j-1)
//                                           or dp(i, j-1) || dp(i+1, j)
// because of the dependency, i has to go down and j has to go up. see #5
int longestPalindromeSubseq(string s) {
   if (s.empty()) return 0;
   int n = s.length();
   vector<vector<int>> dp(n, vector<int>(n));

   for (int i = n - 1; i >= 0; --i) {
       dp[i][i] = 1;                            // dp(i, i) must be palindrome with length 1.
       for (int j = i + 1; j < n; ++j) {     // subsequence has at least 2 elements, j = i + 1
           if (s[i] == s[j]) {
               dp[i][j] = 2 + dp[i + 1][j - 1];
           } else {
               dp[i][j] = max(dp[i][j - 1], dp[i + 1][j]);
           }
       }
   }
   return dp[0][n - 1];
}
```

Ref: #5

517. Super washing machines

You have **n** super washing machines on a line. Initially, each washing machine has some dresses or is empty.

For each **move**, you could choose **any m** (1 ≤ m ≤ n) washing machines, and pass **one dress** of each washing machine to one of its adjacent washing machines **at the same time** .

Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the **minimum number of moves** to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.

Example1

```
Input: [1,0,5], Output: 3
Explanation:
1st move:    1     0 <-- 5    =>    1     1     4
2nd move:    1 <-- 1 <-- 4    =>    2     1     3
3rd move:    2     1 <-- 3    =>    2     2     2
```

```
// soln-1: dynamic programming
// 1, get the average load for each machine
// 2, track running balance for current machine, since we can pick any number of machines for each move,
// the max move is the highest running balance.
// 3, the max move could also be (dresses - avg), for example, [1, 1, 8, 2], avg = 3, max offload = 8 - 3 = 5
// running balance: [-2, -4, 1, 0]
//                           ^
// This number may be higher than the running balance if dresses are passed both ways.
int findMinMoves(vector<int>& machines) {
   int ans = 0, sum = 0;
   for (int x : machines) sum += x;
   if (sum % machines.size()) return -1;

   int avg = sum / machines.size(), balance = 0;
   for (int dresses : machines) {
       balance += (dresses - avg);
       ans = max(ans, max(dresses - avg, abs(balance)));
   }
```

```
    return ans;
}
```

Ref:

520. Detect capital

Given a word, you need to judge whether the usage of capitals in it is right or not.

We define the usage of capitals in a word to be right when one of the following cases holds:

1. All letters in this word are capitals, like "USA".
2. All letters in this word are not capitals, like "leetcode".
3. Only the first letter in this word is capital if it has more than one letter, like "Google".

Otherwise, we define that this word doesn't use capitals in a right way.

```
// 520: soln-1: by lookuping 1st 2 chars, we could determine up/low from 3rd and after
bool detectCapitalUse(string s) {
    for (int i = 0; i < s.length(); ++i) {
        if ((i + 1 == s.length()) || s[i + 1] == ' ') continue;

        bool lo = islower(s[i]) || (isupper(s[i]) && islower(s[i + 1]));
        ++i;
        for (int j = i; j < s.length() && s[j] != ' '; ++j, ++i) {
            if (lo && isupper(s[j])) return false;
            if (!lo && islower(s[j])) return false;
        }

    }
    return true;
}
```

Ref:

521/522. Longest uncommom subsequence / II

Input: "aba", "cdc", **Output:** 3

Explanation: The longest uncommon subsequence is "aba" (or "cdc"), because "aba" is a subsequence of "aba", but not a subsequence of any other strings in the group of two strings.

```
// 521 - longest uncommon string
int findLUSlength(string a, string b) {
        return a == b ? -1 : max(a.length(), b.length());
}
```

```
// 522 - longest uncommon string
// return true if b is subsquence of a
bool isSubsequence(const string& a, const string& b) {
    int j = 0;
    for (int i = 0; i < a.length() && j < b.length(); ++i) {
        if (a[i] == b[j]) ++j;
    }
    return j == b.length();
}

// 1. sort strings by length (descending order), then alphabetic if same
// 2. if strs[i] is not subsequence of [0..i), then strs[i] might be the answer if
//    strs[i] != strs[i + 1] if i + 1 w/ scope
int findLUSlength(vector<string>& strs) {
    sort(strs.begin(), strs.end(), [](const string& a, const string& b) {
```

```
        if (a.length() != b.length()) return a.length() > b.length();
        return a > b;
    });

    for (int i = 0; i < strs.size(); ++i) {
        string& candidate = strs[i];
        bool subsequence = false;
        for (int j = 0; !subsequence && j < i; ++j) {
            if (isSubsequence(strs[j], candidate)) subsequence = true;
        }
        if (!subsequence) {
            if (i == strs.size() - 1 || strs[i] != strs[i + 1]) return candidate.length();
        }
    }
    return -1;
}
```

Ref:

524/720. Longest word in dict through deleting/Longest word in dict

```
// 524 - Longest Word in Dictionary through Deleting
// soln-1: sort + greedy
bool isSubsequence(string& w, string& seq) {
    int i = 0, j = 0;
    while (i < w.length() && j < seq.length()) {
        if (w[i] == seq[j]) ++i;
        ++j;
    }
    return i == w.length();
}
string findLongestWord(string s, vector<string>& d) {
    sort(d.begin(), d.end(), [](const string& a, const string& b) {
        if (a.length() != b.length()) return a.length() > b.length();
        return a < b;
    });
    for (auto& w : d) {
        if (isSubsequence(w, s)) return w;
    }
    return "";
}
```

```
// 720 - Longest Word in Dict (buildable one char at each time)
// soln-1: sort + hashset (if “world” buildable, “worl” must be too.)
string longestWord(vector<string>& words) {
    sort(words.begin(), words.end());

    string ans;
    unordered_set<string> buildable;
    for (string& w : words) {
        if (w.length() == 1 ||
            // w[0...n-1] must be appeared before to make w buildable
            buildable.find(w.substr(0, w.length() - 1)) != buildable.end()) {
                buildable.insert(w);
                if (w.length() > ans.length()) ans = w;
            }
    }
    return ans;
}
```

Ref:

526/667. Beautiful arrangement/II

526 - Suppose you have **N** integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these **N** numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:

1. The number at the ith position is divisible by **i**.
2. **i** is divisible by the number at the ith position.

Now given N, how many beautiful arrangements can you construct?

Example 1:

```
Input: 2, Output: 2
Explanation:

The first beautiful arrangement is [1, 2]:
Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).
Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).
The second beautiful arrangement is [2, 1]:
Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).
Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.
```

667 - Given two integers n and k, you need to construct a list which contains n different positive integers ranging from 1 to n and obeys the following requirement:

Suppose this list is [a1, a2, a3, ... , an], then the list [|a1 - a2|, |a2 - a3|, |a3 - a4|, ... , |an-1 - an|] has exactly k distinct integers.

If there are multiple answers, print any of them.

```
// soln-1: brute force backtracking enumerate
// this is permutation problem, time complexity: O(n!)
void helper(int idx, vector<bool>& visited, int& ans) {
   if (idx > visited.size()) { ++ans; return; }

   for (int i = visited.size(); i >= 1; --i) {
       if ((idx % i) && (i % idx)) continue;
       if (visited[i - 1])         continue;    // number (i) has been verified in current permutation
       visited[i - 1] = true;
       helper(idx + 1, visited, ans);
       visited[i - 1] = false;
   }
}

int countArrangement(int N) {
   int ans = 0;
   vector<bool> visited(N);
   helper(1, visited, ans);
   return ans;
}
```

```
// 667 - soln-1: brute-force (observation from examples)
// n numbers have at most n-1 distinct k, for example, n = 9, k = 3
// i  j
// v  v
// 1  9  2    3  4  5  6  7  8        n = 9, k = 3 (if k is odd, append last number + 1)
// 1  9  2  8    7  6  5  4  3        n = 9, k = 4 (if k is odd, append last number - 1)
vector<int> constructArray(int n, int k) {
```

```
    vector<int> ans;
    for (int lo = 1, hi = n; ans.size() < k; nullptr) {
        ans.push_back(hi--);
        if (ans.size() < k) ans.push_back(lo++);
    }
    // if not enough, +1 (if k is even)/-1 (k is odd) for the last number.
    while (ans.size() < n) ans.push_back(k % 2 ? ans.back() - 1 : ans.back() + 1);

    return ans;
}
```

Ref:

527. Word abbreviation

Given an array of n distinct non-empty strings, you need to generate minimal possible abbreviations for every word following rules below.

1. Begin with the first character and then the number of characters abbreviated, which followed by the last character.
2. If there are any conflict, that is more than one words share the same abbreviation, a longer prefix is used instead of only the first character until making the map from word to abbreviation become unique. In other words, a final abbreviation cannot map to more than one original words.
3. If the abbreviation doesn't make the word shorter, then keep it as original.

Example:

Input: ["like", "god", "internal", "me", "internet", "interval", "intension", "face", "intrusion"]
Output: ["l2e","god","internal","me","i6t","interval","inte4n","f2e","intr4n"]

Note:
1. Both n and the length of each word will not exceed 400.
2. The length of each word is greater than 1.
3. The words consist of lowercase English letters only.
4. The return answers should be in the same order as the original array.

```
// soln-1: do abbr. from 0, then put dups into set, then do abbr. starting from next index
vector<string> wordsAbbreviation(vector<string>& dict) {
    vector<string> ans(dict.size());

    unordered_set<int> s;
    for (int i = 0; i < dict.size(); ++i) s.insert(i);

    int k = 1;
    do {
        unordered_map<string, unordered_set<int>> mp;       // if dups, then size of set > 1
        for (int i : s) {
            ans[i] = getAbbr(dict[i], k);
            mp[ans[i]].insert(i);
        }

        s.clear();
        for (auto it = mp.begin(); it != mp.end(); ++it) {
            if (it->second.size() > 1) s.insert(it->second.begin(), it->second.end());
        }
        ++k;
    } while (!s.empty());

    return ans;
}

// do abbr. starting from k (1...)
string getAbbr(const string& str, int k) {
```

```
    if (k + 2 >= str.length()) return str;
    return str.substr(0, k) + to_string(str.length() - k - 1) + str.back();
}
```

Ref: #288

529. Minesweeper

You are given a 2D char matrix representing the game board. 'M' represents an unrevealed mine, 'E' represents an unrevealed empty square, 'B' represents a revealed blank square that has no adjacent (above, below, left, right, and all 4 diagonals) mines, digit ('1' to '8') represents how many mines are adjacent to this revealed square, and finally 'X' represents a revealed mine.

Now given the next click position (row and column indices) among all the unrevealed squares ('M' or 'E'), return the board after revealing this position according to the following rules:

1. If a mine ('M') is revealed, then the game is over - change it to 'X'.
2. If an empty square ('E') with no adjacent mines is revealed, then change it to revealed blank ('B') and all of its adjacent unrevealed squares should be revealed recursively.
3. If an empty square ('E') with at least one adjacent mine is revealed, then change it to a digit ('1' to '8') representing the number of adjacent mines.
4. Return the board when no more squares will be revealed.

Solution:
When click happens, two cases:
1. mine, then mark cell with 'X' and end
2. not mine. check if there is mine for 8 directions:
    a. has mines, then mark # and no further traversal.
    b. no mine, which means 'E', then start searching from these neighbor points, just like clicking these cell.

```
// soln-1: BFS with 29ms. start with clicking point, then traversal all neighbors if
// there is no 'M' around, mark the number of 'M' if found (do not traversal in this case)
vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {
    int x = click[0], y = click[1];
    if (board[x][y] == 'M') {
        board[x][y] = 'X';
        return board;
    }

    vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}, {1, 1}, {-1, -1}, {1, -1}, {-1, 1}};
    vector<vector<int>> visited(board.size(), vector<int>(board[0].size(), false));

    queue<pair<int, int>> q;
    q.push({x, y}); visited[x][y] = true;
    while(!q.empty()) {
        int x = q.front().first, y = q.front().second; q.pop();

        int neighborMines = 0;              // mines around me
        vector<pair<int, int>> neighbors;   // unrevealed neighbors
        for (auto dir : dirs) {
            int nx = x + dir.first, ny = y + dir.second;
            if (nx < 0 || nx >= board.size() || ny < 0 || ny >= board[0].size() || visited[nx][ny]) continue;
            switch (board[nx][ny]) {
            case 'M':
                ++neighborMines;    break;
            case 'E':
                neighbors.push_back({nx, ny});  break;
            case 'X':
                assert(0);
            }
        }

        if (neighborMines) board[x][y] = neighborMines + '0';
```

```
        else {
            board[x][y] = 'B';
            for (auto n : neighbors) {
                visited[n.first][n.second] = true;  // mark it only when put into q.
                q.push(n);
            }
        }
    }

    return board;
}

// soln-2: DFS with 29ms. Continue traversal only if current is 'E'.
void helper(vector<vector<char>>& board, pair<int, int> click) {
    int x = click.first, y = click.second;
    if (board[x][y] != 'E') {
        if (board[x][y] == 'M') board[x][y] = 'X';
        return;
    }

    const vector<pair<int, int>> dirs{{1, 0}, {0, 1}, {-1, 0}, {0, -1}, {1, 1}, {-1, -1}, {1, -1}, {-1, 1}};

    board[x][y] = 'B';  // mark it as Blank first, then update if necessary
    int mines = 0;
    vector<pair<int, int>> neighbors;
    for (auto dir : dirs) {
        int nx = x + dir.first, ny = y + dir.second;
        if (nx < 0 || nx >= board.size() || ny < 0 || ny >= board[0].size()) continue;

        if (board[nx][ny] == 'E') neighbors.push_back({nx, ny});
        else if (board[nx][ny] == 'M') ++mines;
    }
    if (mines) board[x][y] = '0' + mines;
    else for (auto n : neighbors) helper(board, pair<int, int>{n.first, n.second});
}

vector<vector<char>> updateBoard(vector<vector<char>>& board, vector<int>& click) {
    helper(board, pair<int, int>{click[0], click[1]});
    return board;
}
```

Ref:

531/533. Lonely pixel /II

531 - Given a picture consisting of black and white pixels, find the number of black lonely pixels. A black lonely pixel is character 'B' that located at a specific position where the same row and same column don't have any other black pixels.

533 - Given a picture consisting of black and white pixels, and a positive integer N, find the number of black pixels located at some specific row **R** and column **C** that align with all the following rules:

1. Row R and column C both contain exactly N black pixels.
2. For all rows that have a black pixel at column C, they should be exactly the same as row R

Example:

Input:
`[['W', 'B', 'W', 'B', 'B', 'W'],`
` ['W', 'B', 'W', 'B', 'B', 'W'],`
` ['W', 'B', 'W', 'B', 'B', 'W'],`
` ['W', 'W', 'B', 'W', 'B', 'W']]`
	Explanation: All the bold 'B' are the black pixels we need (all 'B's at column 1 and 3).
	`        0    1    2    3    4    5         column index`
	`0    [['W', 'B', 'W', 'B', 'B', 'W'],`
	`1     ['W', 'B', 'W', 'B', 'B', 'W'],`
	`2     ['W', 'B', 'W', 'B', 'B', 'W'],`

N = 3
Output: 6
	`3     ['W', 'W', 'B', 'W', 'B', 'W']]`
	`row index`
	
	Take 'B' at row R = 0 and column C = 1 as an example:
	Rule 1, row R = 0 and column C = 1 both have exactly N = 3 black pixels.
	Rule 2, the rows have black pixel at column C = 1 are row 0, row 1 and row 2.
	They are exactly the same as row R = 0.

```
// 531/532 - soln-1: count black-pixels for each row and col, use hashmap to aggregate good rows
int findBlackPixel(vector<vector<char>>& picture, int N) {
   if (picture.empty() || picture[0].empty()) return 0;

   int res = 0;
   vector<int> colCount(picture[0].size(), 0);
   unordered_map<string, int> mp;                 // rows having N-black pixels are aggregated
   for (int i = 0; i < picture.size(); ++i) {
       int rolCount = 0;
       for (int j = 0; j < picture[i].size(); ++j) {
           if (picture[i][j] == 'B') {
               ++colCount[j], ++rolCount;
           }
       }
       if (rolCount == N) ++mp[string(picture[i].begin(), picture[i].end())];
   }
   for (auto& p : mp) {
       if (p.second != N) continue;
       for (int i = 0; i < p.first.length(); ++i) {
           res += (p.first[i] == 'B' && colCount[i] == N) ? N : 0;
       }
   }
   return res;
}
```

Ref:

534. TODO

Solution:

Ref:

536/606. Construct binary tree <-> string (review)

```
// 606 - construct binary tree from string like “4( 2(3)(1) ) ( 6(5) )”
TreeNode* str2tree(string s) {
   if (s.empty()) return nullptr;

   // extract root value
   size_t found = s.find('(');
   int pos = (found != string::npos ? found : s.length());
   TreeNode* root = new TreeNode(stoi(s.substr(0, pos)));

   // find left/right subtree
   int start = pos, cnt = 0;
   while (pos < s.length()) {       // find the boundary for left/right subtree
       if (s[pos] == '(') ++cnt;    // good trick to match '()'
       else if (s[pos] == ')') --cnt;
```

```
        if (cnt) {
            ++pos;                    // continue check next char
            continue;
        }

        // cnt == 0, found match substring for left/right subtree
        // start points to '(', pos points to ')'
        if (start == found) {         // process left subtree
            ++start;                  // skip '(' to get empty string for '()'
            root->left = str2tree(s.substr(start, pos - start));
        } else {                      // process right subtree
            ++start;                  // skip '(' to get empty string for '()'
            root->right = str2tree(s.substr(start, pos - start));
        }
        start = ++pos;                // start points to '(' if exist
    }
    return root;
}
```

```
// 606 - construct string from binary tree
string tree2str(TreeNode* node) {
    if (!node) return "";
    if (!node->left && !node->right) return to_string(node->val);
    if (node->left && !node->right) return to_string(node->val) + "(" + tree2str(node->left) + ")";

    return to_string(node->val) + "(" + tree2str(node->left) + ")(" + tree2str(node->right) + ")";
}
```

Ref: #606

539/681. Min time difference/Next closest time

```
int diff(const string& t1, const string& t2) {
    const int offset = 23 * 60 + 60;
    int i1 = stoi(t1.substr(0, 2)) * 60 + stoi(t1.substr(3));
    int i2 = stoi(t2.substr(0, 2)) * 60 + stoi(t2.substr(3));
    if (i1 > i2) {
        return min(i1 - i2, i2 + offset - i1);
    } else {
        return min(i2 - i1, i1 + offset - i2);
    }
}
// 539 - min time diff.
// soln-1: brute-force
// sort and get the diff. for each neighbors plus, the diff. between head and tail
int findMinDifference(vector<string>& timePoints) {
    std::sort(timePoints.begin(), timePoints.end());

    int ans = INT_MAX;
    for (int i = 0; i < timePoints.size() - 1; ++i) {
        ans = min(ans, diff(timePoints[i], timePoints[i + 1]));
    }
    ans = min(ans, diff(timePoints[0], timePoints.back()));
    return ans;
}
```

```
// 681 - Next Closest Time (permutation numbers in the given time)
// soln-1: backtracking or brute-force
// 1. if numbers are allowed multiple times, adding 1 min each time and check every digit.
// 2. if asking number permutation, then it is same as permutation with dups in O(n!) time.
string nextClosestTime(string time) {
    string ans;
    int cur = stoi(time) * 60 + stoi(time.substr(3));
```

```
    for (int max_mins = 24 * 60, i = 1; i <= max_mins; ++i) {
        int next = (cur + i) % max_mins;
        for (auto& x : vector<int>{600, 60, 10, 1}) {
            char d = '0' + next / x;
            if (time.find(d) == string::npos) break;

            ans.push_back(d), next = next % x;
        }
        if (4 == ans.length()) break;
        else                   ans.clear();
    }
    return ans.substr(0, 2) + ":" + ans.substr(3);
}
```

Ref:

542. 01 Matrix (review)

Given a matrix consists of 0 and 1, find the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.

Note:

1. The number of elements of the given matrix will not exceed 10,000.
2. There are at least one 0 in the given matrix.
3. The cells are adjacent in only four directions: up, down, left and right.

Example 1:

Input:

```
0 0 0
0 1 0
1 1 1
```

Output:

```
0 0 0
0 1 0
1 2 1
```

Solution:

```
soln-1: brute-force BFS (bad)
   Starting with 0 to use BFS to update all none-0 cells, stop updating if cell is visited and distance is smaller.
soln-2: multiple points BFS with little trick
   Starting with 0 to use BFS to update all none-0 cells. Key: collect all 0-cells as starting point and mark others as INT_MAX.
soln-3: 2-direction asking around
   ask 4-neighbor states and update myself: 1) from top-left to bottom-right to ask above and left neighbors; 2) from bottom-right to top-left to ask below and right neighbors.
```

```
// soln-1: brute-force (bad, TLE)
void update(vector<vector<int>>& m, vector<vector<bool>>& updated, int x, int y) {
    int d = 0, rows = m.size(), cols = m[0].size();
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));
    queue<pair<int, int>> q;    q.push({x, y}); visited[x][y] = true;
    vector<pair<int, int>> dirs{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    while(!q.empty()) {
        ++d;

        for (int i = q.size(); i > 0; --i) {
            x = q.front().first, y = q.front().second;  q.pop();
            for (auto dir : dirs) {
```

```
                int nx = x + dir.first, ny = y + dir.second;
                if (nx < 0 || nx >= rows || ny < 0 || ny >= cols) continue;
                if (visited[nx][ny] || 0 == m[nx][ny]) continue;

                if (m[nx][ny] >= d || (m[nx][ny] < d && !updated[nx][ny])) {
                    updated[nx][ny] = true;
                    m[nx][ny] = d;
                    q.push({nx, ny});
                    visited[nx][ny] = true;
                }
            }
        }
    }

}
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    if (matrix.empty()) return matrix;
    int rows = matrix.size(), cols = matrix[0].size();
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (0 == matrix[i][j]) update(matrix, visited, i, j);
        }
    }
    return matrix;
}

// soln-2: multiple starting point BFS with initialized distance with 219ms
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    if (matrix.empty()) return matrix;
    int rows = matrix.size(), cols = matrix[0].size();

    queue<pair<int, int>> q;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            if (1 == matrix[i][j]) matrix[i][j] = INT_MAX;
            else q.push({i, j});
        }
    }

    vector<pair<int, int>> dirs{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    while(!q.empty()) {
        for (int i = q.size(); i > 0; --i) {
            int x = q.front().first, y = q.front().second;  q.pop();
            for (auto& dir : dirs) {
                int nx = x + dir.first, ny = y + dir.second;
                if (nx < 0 || nx >= rows || ny < 0 || ny >= cols) continue;

                if (matrix[nx][ny] > matrix[x][y] + 1) {
                    matrix[nx][ny] = matrix[x][y] + 1;
                    q.push({nx, ny});
                }
            }
        }
    }

    return matrix;
}

// soln-3: 2-direction asking around with 222ms
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    int rows = matrix.size(), cols = matrix[0].size();
    vector<vector<int>> ans(rows, vector<int>(cols, rows + cols));  // max distance is rows + cols
```

```
    for (int i = 0; i < rows; ++i) {
        for(int j = 0; j < cols; ++j) {
            if (0 == matrix[i][j]) ans[i][j] = 0;
            else {
                if (i > 0) ans[i][j] = min(ans[i][j], 1 + ans[i - 1][j]);
                if (j > 0) ans[i][j] = min(ans[i][j], 1 + ans[i][j - 1]);
            }
        }
    }
    for (int i = rows - 1; i >= 0; --i) {
        for (int j = cols - 1; j >= 0; --j) {
            if (i < rows - 1) ans[i][j] = min(ans[i][j], 1 + ans[i + 1][j]);
            if (j < cols - 1) ans[i][j] = min(ans[i][j], 1 + ans[i][j + 1]);
        }
    }
    return ans;
}
```

Ref:

543. Diameter of binary tree

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

Example:

Given a binary tree

Return 3, which is the length of the path [4,2,1,3] or [5,2,1,3].

Note: The length of path between two nodes is represented by the number of edges between them.

Solution:

Diameter of tree d_tree(x) = max(d_node(i), for each node i). So we need to compute diameter for each node first. d_node(x) = h(x.left) + h(x.right) + 1

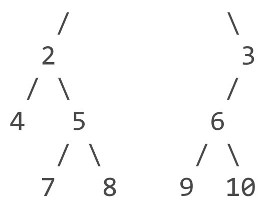
```
// return height of a node
// ans carries diameter of current subtree
int helper(TreeNode* r, int& ans) {
    if (!r) return 0;

    int lh = helper(r->left, ans);
    int rh = helper(r->right, ans);
    ans = max(lh + rh + 1, ans);

    return max(lh, rh) + 1;
}
```

Ref:

545. Boundary of binary tree

[Example 2] Input:

____1_____




```
Output: [1,2,4,7,8,9,10,6,3]
```

Explanation:The left boundary are node 1,2,4. (4 is the left-most node according to definition). The leaves are node 4,7,8,9,10. The right boundary are node 1,3,6,10. (10 is the right-most node).

```
// 545 - soln-1: dfs
// 0. keep root to result if it has child.
// 1. get left boundary (not include last one if it also is a leaf)
// 2. get all leaves
// 3. get right boundary (not include last one if it also is a leaf)
void lb(TreeNode* node, vector<int>& ans) {
    if (!node) return;
    if (node->left || node->right) ans.push_back(node->val);
    lb(node->left, ans);
}
void leaf(TreeNode* node, vector<int>& ans) {
    if (!node) return;
    leaf(node->left, ans);
    if (!node->left && !node->right) ans.push_back(node->val);
    leaf(node->right, ans);
}
void rb(TreeNode* node, vector<int>& ans) {
    if (!node) return;
    rb(node->right, ans);       // count-clockwise
    if (node->left || node->right) ans.push_back(node->val);
}

void helper(TreeNode* root, vector<int>& ans) {
    if (!root) return;

    // keep root if it has child, otherwise let leaf() to get it.
    if (root->left || root->right) ans.push_back(root->val);
    lb(root->left, ans);
    leaf(root, ans);
    rb(root->right, ans);
}
```

Ref:

546. Remove boxes

Given several boxes with different colors represented by different positive numbers.

You may experience several rounds to remove boxes until there is no box left. Each time you can choose some continuous boxes with the same color (composed of k boxes, k >= 1), remove them and get `k*k` points.

Find the maximum points you can get.

Example 1:

Input: `[1, 3, 2, 2, 2, 3, 4, 3, 1]`
Output: 23
Explanation: `[1, 3, 2, 2, 2, 3, 4, 3, 1]`

```
----> [1, 3, 3, 4, 3, 1] (3*3=9 points)
----> [1, 3, 3, 3, 1] (1*1=1 points)
----> [1, 1] (3*3=9 points)
----> [] (2*2=4 points)
```

```
// 546 - soln-1: dynamic programming
```

```
// if we define f(i, j) be the max points we can get for A[i, j], then remove any one in them
// the two parts are f(i, k - 1) and f(k + 1, j), but the two parts is not independ question!
// because the rightmost of 1st part could be same as the leftmost of 2nd part.
//
// what if we have the box number on my left side that have same color as me? then we can either remove me
// immediately or waiting for possible bigger gain somewhere on my right side having same color as mine.
//
// let f(i, j, c) be the max points we can get for A[i, j] AND the left boxes which are same A[i] is c.
// 1, the question is f(0, n-1, 0)  - no boxes has same color with A[0] yet.
// 2, f(i, i-1, any) = 0            - no boxes no points
// 3, f(i, i, k) = (k+1) * (k+1)    - k boxes with same color as A[i] is attached to left already.
//
// now we can either remove A[i] or waiting for potential bigger gain:
// - remove immediately: (k + 1) * (k + 1) + f(i + 1, j, 0)
// - wait for potential bigger gain - only if there are boxes having same color with me
//    f(i+1, m-1, 0) + f(m, j, c+1), m is the potential bigger position which boxes[i] == boxes[m]
//
typedef unordered_map<int, unordered_map<int, unordered_map<int, int>>> cache_t;
int helper(vector<int>& boxes, int low, int hi, int k, cache_t& dp) {
   if (low > hi) return 0;
   if (low == hi) return (k + 1) * (k + 1);
   if (dp.find(low) != dp.end() &&
       dp[low].find(hi) != dp[low].end() &&
       dp[low][hi].find(k) != dp[low][hi].end()) return dp[low][hi][k];

   while (low + 1 <= hi && boxes[low] == boxes[low + 1])   low++, k++; // optimization - skip same color boxes

   int ans = (k + 1) * (k + 1) + helper(boxes, low + 1, hi, 0, dp);    // remove low immediately
   for (int i = low + 1; i <= hi; ++i) {                                // check for potential bigger gain
       if (boxes[i] == boxes[low]) {
           ans = max(ans, helper(boxes, low + 1, i - 1, 0, dp) + helper(boxes, i, hi, k + 1, dp));
       }
   }
   dp[low][hi][k] = ans;
   return ans;
}

int removeBoxes(vector<int>& boxes) {
   cache_t dp;
   return helper(boxes, 0, boxes.size() - 1, 0, dp);
}
```

Ref: #312

547. Friend Circles

There are N students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a direct friend of B, and B is a direct friend of C, then A is an indirect friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

Given a N*N matrix M representing the friend relationship between students in the class. If M[i][j] = 1, then the ith and jth students are direct friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

Example 1:

```
Input:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
Output: 2
Explanation:The 0th and 1st students are direct friends, so they are in a friend circle.
```

```
The 2nd student himself is in a friend circle. So return 2.
```

Note:

1. N is in range [1,200].
2. M[i][i] = 1 for all students.
3. If M[i][j] = 1, then M[j][i] = 1.

```
// soln-1: union-find (faster than dfs)
int findCircleNum(vector<vector<int>>& M) {
   UnionFind uf(M.size());

   for (int i = 0; i < M.size(); ++i) {
       for (int j = i; j < M[i].size(); ++j) {
           if (M[i][j]) uf.Union(i, j);
       }
   }

   unordered_set<int> ans;
   for (int i = 0; i < M.size(); ++i) {
       ans.insert(uf.Find(i));
   }
   return ans.size();
}

// soln-2: dfs and mark visited
void helper(int k, vector<vector<int>>& m, vector<bool>& visited) {
   if (visited[k]) return;

   visited[k] = true;
   for (int i = 0; i < m[k].size(); ++i) {
       if (m[k][i]) helper(i, m, visited);
   }
}

int findCircleNum(vector<vector<int>>& M) {
   int ans = 0;
   vector<bool> visited(M.size(), false);
   for (int j = 0; j < M.size(); ++j) {
       if (!visited[j]) {
           helper(j, M, visited);
           ++ans;
       }
   }
   return ans;
}
```

Ref: #323

550. TODO

Solution:

Ref:

551/552. Student attendance record I / II

Given a positive integer **n**, return the number of all possible attendance records with length n, which will be regarded as rewardable. The answer may be very large, return it after mod $10^9 + 7$.

A student attendance record is a string that only contains the following three characters:

1. **'A'** : Absent.
2. **'L'** : Late.
3. **'P'** : Present.

A record is regarded as rewardable if it doesn't contain **more than one 'A' (absent)** or **more than two continuous 'L' (late)**.

Example 1:

```
Input: n = 2, Output: 8
Explanation:
There are 8 records with length 2 will be regarded as rewardable:
"PP" , "AP", "PA", "LP", "PL", "AL", "LA", "LL"
Only "AA" won't be regarded as rewardable owing to more than one absent times.
```

```
// soln-1: dynamic programming
// similar to "fence painting" question, consider 3 cases: ending with P/L/A respectively
// let P(n) be the number of ways of attendance ending with P, then
// T(n) = P(n) + L(n) + A(n)
// P(n) = P(n-1) + L(n-1) + A(n-1)
// L(n) = P(n-1) + A(n-1) + P(n-2) + A(n-2)
//        "PL"      "AL"    "PLL"    "ALL"
// let noAP(n) be the number of ways ending with "P" but without "A" record,
//     noAL(n) be the number of ways ending with "L" but without "A" record
// A(n) = noAP(n-1) + noAL(n-1)         (1) \                    \ => A(n) = A(n-1) + noAL(n-1)
// noAP(n) = noAP(n-1) + noAL(n-1)      (2) / => A(n) = noAP(n)  /
// noAL(n) = noAP(n-1) + noAP(n-2)      (3)              / => noAL(n) = A(n-1) + A(n-2)
// combine (1)(2)(3), we have A(n) = A(n-1) + A(n-2) + A(n-3)
int checkRecord(int n) {
   long M = 1e9 + 7;
   vector<int> P(n), L(n), A(n);
   P[0] = A[0] = L[0] = 1;         // one attendance ending with P/L/A respectively
   L[1] = 3;                       // PL/LL/AL
   A[1] = 2;                       // PA/LA
   A[2] = 4;                       // PPA/LLA/PLA/LPA
   for (int i = 1; i < n; ++i) {
       P[i] = (P[i - 1] % M + L[i - 1] % M + A[i - 1] % M) % M;
       if (i > 1) L[i] = (P[i - 1] % M + A[i - 1] % M + P[i - 2] % M + A[i - 2] % M) % M;
       if (i > 2) A[i] = (A[i - 1] % M + A[i - 2] % M + A[i - 3] % M) % M;
   }
   return (P.back() % M + L.back() % M + A.back() % M) % M;
}
```

Ref: #276

554. TODO

Solution:

Ref:

555. Split concatenated strings

```
// 555 - Split Concatenated Strings
// soln-1: brute-force
// 1. concatenate str or reverse-str whichever is bigger to form bigger string
```

```
// 2. then for each str, strip it from current long string and try to re-concatenate it.
//    example: ["lc", "lov", "ycd"] => -lclovycd- => -ylclovdc- (answer)
string splitLoopedString(vector<string>& strs) {
   string s, ans;
   for (string str : strs) {
       string rstr = string(str.rbegin(), str.rend());
       s += str > rstr ? str : rstr;
   }
   for (int i = 0, pos = 0; i < strs.size(); ++i) {
       string t1 = strs[i], t2 = string(t1.rbegin(), t1.rend());
       string mid = s.substr(pos + t1.size()) + s.substr(0, pos);    // get p-1 and p-2 from <p-1>.cur.<p-2>
       for (int j = 0; j < t1.size(); ++j) {                         // split cur and concat. with existing str
           if (ans.empty() || t1[j] >= ans[0]) ans = max(ans, t1.substr(j) + mid + t1.substr(0, j));
           if (ans.empty() || t2[j] >= ans[0]) ans = max(ans, t2.substr(j) + mid + t2.substr(0, j));
       }
       pos += t1.size();
   }
   return ans;
}
```

Ref:

556. TODO

Solution:

Ref:

557. TODO

Solution:

Ref:

561. Array partition I

Given an array of **2n** integers, your task is to group these integers into **n** pairs of integer, say (a_1, b_1), (a_2, b_2), ..., (a_n, b_n) which makes sum of min(a_i, b_i) for all i from 1 to n as large as possible.

Example 1:

Input: `[1,4,3,2]`, **Output:** `4`

Explanation: `n is 2, and the maximum sum of pairs is 4 = min(1, 2) + min(3, 4).`

```
// soln-1: sort input to minimize the loss of each pair
int arrayPairSum(vector<int>& nums) {
   sort(nums.begin(), nums.end());
   int ans = 0;
   for (int i = 0; i < nums.size(); i += 2) ans += min(nums[i], nums[i + 1]);
   return ans;
}
```

Ref:

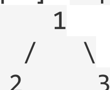
563. Binary tree tilt

Given a binary tree, return the tilt of the whole tree.

The tilt of a tree node is defined as the absolute difference between the sum of all left subtree node values and the sum of all right subtree node values. Null node has tilt 0.

The tilt of the whole tree is defined as the sum of all nodes' tilt.

[Example] - `Input:`




```
Output: 1
Explanation:
Tilt of node 2 : 0
Tilt of node 3 : 0
Tilt of node 1 : |2-3| = 1
Tilt of binary tree : 0 + 0 + 1 = 1
```

Solution:
Tilt(x) = |sum(x.left) - sum(x.right)|, so bottom-up we can get the tilt value for the whole tree.

```
int helper(TreeNode* node, int& ans) {
   if (!node) return 0;

   int left = helper(node->left, ans);
   int right = helper(node->right, ans);
   ans += abs(left - right);
   return node->val + left + right;
}
```

Ref:

564/866. Find the closest palindrome/Prime palindrome

```
// 564 - find the closest palindrome given an integer (closet is defined as absolute difference)
// soln-1: build a list of candidate answers, then choose from
// If the final answer has the same number of digits as the input string S,
// then the answer must be the middle digits + (-1, 0, or 1) flipped into a palindrome. For example,
//      23456 had middle part 234, then yields 23332, (23432), 23532.
//      88800 had middle part 888, then yields (88788), 88888, 88988.
//      10899 had middle part 108, then yields 10701, 10801, (10901).
// If the final answer has a different number of digits, it must be of the form 99..99 or 100..001. For example,
//      1000 -> 999, and 999 -> 1001.
string nearestPalindromic(string n) {
   int len = n.length();
   function<long(long, int)> mirror = [&n](long mid, int d) {
       string prefix = to_string(mid + d);
       return stol(prefix + string(prefix.rbegin() + n.length() % 2, prefix.rend()));
   };
   long m = stol(n.substr(0, (len + 1) / 2));
   set<long> c{mirror(m, -1), mirror(m, 0), mirror(m, 1)};// set is ordered, min_element will take advantage of
it.
   c.insert({pow(10, len - 1) - 1, pow(10, len) + 1});    // two more candidates with diff. length
   c.erase(stol(n));                                      // not including itself
   return to_string(*min_element(c.begin(), c.end(), [&num](long a, long b){return abs(num - a) < abs(num -
b);}));
}
```

```
// 866 - prime palindrome
// soln-1: math
// 1. palindrome numbers with even length digits is a multiple of 11, so it must not be prime, except 11 itself.
// proof: "abba" = a * 1001 + b * 11 = a * (100 * 11 - 10 * 11) + b * 11
```

```
// 2. so only generate palindrome numbers with odd length and check if it is prime.
bool prime(int n) {
    if (n < 2)          return false;
    if (0 == n % 2)     return 2 == n;
    for (int i = 3; i * i <= n; i += 2) {
        if (0 == n % i) return false;
    }
    return true;
}

int primePalindrome(int N) {
    if (8 <= N && N <= 11) return 11;   // so do not bother generate even length palindrome
    for (int i = 1; i < 100000; ++i) {
        string s(to_string(i)), rs(s.rbegin(), s.rend());
        int p = stoi(s + rs.substr(1));          // generate odd length palindrome
        if (p >= N && prime(p)) return p;
    }
    return -1;
}
```

Ref: #5

565. TODO

Solution:

Ref:

566. Reshape matrix

trivial

```
// soln-1: 2-d to 1-d conversion
vector<vector<int>> matrixReshape(vector<vector<int>>& nums, int r, int c) {
   if (nums.empty()) return nums;
   if (nums.size() * nums[0].size() != r * c) return nums;

   vector<vector<int>> ans(r, vector<int>(c));
   for (int i = 0, k = 0; i < nums.size(); ++i) {
       for (int j = 0; j < nums[0].size(); ++j, ++k) {
           ans[k / c][k % c] = nums[i][j];
       }
   }
   return ans;
}
```

Ref:

568. TODO

Solution:

Ref:

572/652. Subtree of another tree/Find dup subtrees (review)

572 - Given two non-empty binary trees s and t, check whether tree t has exactly the same structure and node values with a subtree of s.

652 - Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of any one of them. Two trees are duplicate if they have the same structure with same node values.

```
// 572 - subtree of another tree
// soln-1: brute-force (bottom up in O(mn) time)
// soln-2: serialize s and t, do find substring (same as 652), in O(m+n) time for serialize and string compare.
bool isSame(TreeNode* p, TreeNode* q) {
   if (!p && !q) return true;
   if (!p || !q) return false;
   if (p && q && p->val != q->val) return false;

   return isSame(p->left, q->left) && isSame(p->right, q->right);
}

bool isSubtree(TreeNode* s, TreeNode* t) {
   if (s && isSubtree(s->left, t)) return true;
   if (s && isSubtree(s->right, t)) return true;
   if (isSame(s, t)) return true;
   return false;
}
```

```
// 652 - find duplicate subtrees
// soln-1: serialize subtree then hashmap find dups
// pre-order serialize will not work, in-order/post-order will work
string helper(TreeNode* node, unordered_map<string, int>& mp,vector<TreeNode*>& ans) {
    if (!node) return "";

    string l = helper(node->left, mp, ans), r = helper(node->right, mp, ans);
    string ss = "(" + l + to_string(node->val) + r + ")";
    if (1 == mp[ss]) ans.push_back(node);
    ++mp[ss];
    return ss;
}

vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
    vector<TreeNode*> ans;
    unordered_map<string, int> mp;
    helper(root, mp, ans);

    return ans;
}
```

Ref: #652

574. TODO

Ref:

575. Distribute candies

Given an integer array with **even** length, where different numbers in this array represent different **kinds** of candies. Each number means one candy of the corresponding kind. You need to distribute these candies **equally** in number to brother and sister. Return the maximum number of **kinds** of candies the sister could gain.

Example 1:

```
Input: candies = [1,1,2,2,3,3], Output: 3
Explanation:
There are three different kinds of candies (1, 2 and 3), and two candies for each kind.
Optimal distribution: The sister has candies [1,2,3] and the brother has candies [1,2,3], too.
The sister has three different kinds of candies.
```

```
// soln-1: max different kinds is min(kinds, candies / 2)
int distributeCandies(vector<int>& candies) {
   unordered_set<int> kinds;
   for (int x : candies) kinds.insert(x);
   return min(kinds.size(), candies.size() / 2);
}
```

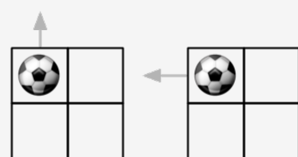
Ref:

576. Out of boundary paths (review)

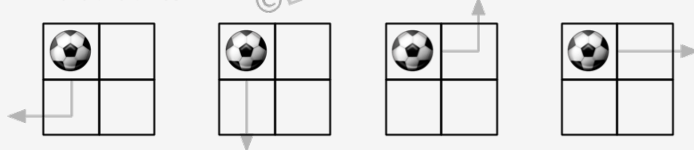
There is an **m** by **n** grid with a ball. Given the start coordinate **(i,j)** of the ball, you can move the ball to **adjacent** cell or cross the grid boundary in four directions (up, down, left, right). However, you can **at most** move **N** times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod $10^9 + 7$.

```
Input:m = 2, n = 2, N = 2, i = 0, j = 0
Output: 6
Explanation:
```

Move one time:



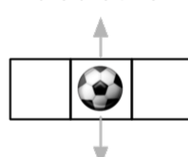
Move two times:



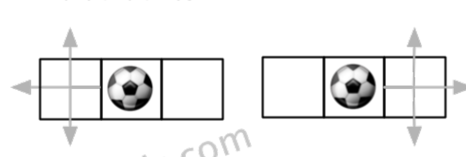
©LeetCode.com

```
Input:m = 1, n = 3, N = 3, i = 0, j = 1
Output: 12
Explanation:
```

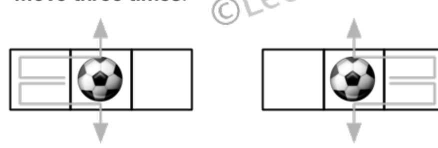
Move one time:



Move two times:



Move three times:



©LeetCode.com

#688 - knight probability in chessboard (very similar DP)

```
// soln-1: dynamic programming
// key points:
// 1, compute from outside then flood into grid, as outside has only 1 way to enter
// 2, 3-D dp to simulate the back-and-forth move between cells: dp[N][i][j] = sum of 4 directions for N - 1
steps
// 3, boundary cases - 4 boundary is 1, except 4 points:
//                       dp[0][0][0] = dp[0][0][n - 1] = dp[0][m - 1][0] = dp[0][m - 1][n - 1] = 2
// space could be O(n * m) instead of O(n * m * N)
int findPaths(int m, int n, int N, int i, int j) {
   const long M = 1e9+7;
   vector<vector<vector<long>>> dp(N + 1, vector<vector<long>>(m, vector<long>(n)));
```

```
    for (int iN = 1; iN <= N; ++iN) {
        for (int im = 0; im < m; ++im) {
            for (int in = 0; in < n; ++in) {
                dp[iN][im][in] = ((im == 0     ? 1 : dp[iN - 1][im - 1][in]) +   // effectively make 4 corners be
2.
                                 (im == m - 1 ? 1 : dp[iN - 1][im + 1][in]) +
                                 (in == 0     ? 1 : dp[iN - 1][im][in - 1]) +
                                 (in == n - 1 ? 1 : dp[iN - 1][im][in + 1])) % M;
            }
        }
    }
    return dp[N][i][j];
}
```

Ref: #688

577. TODO

Solution:

Ref:

578. TODO

Solution:

Ref:

579. TODO

Solution:

Ref:

580. TODO

Solution:

Ref:

581. Shortest unsorted continuous subarray (review)

Given an integer array, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order, too.

You need to find the **shortest** such subarray and output its length.

Example 1:

Input: [2, 6, 4, 8, 10, 9, 15], **Output:** 5

Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.

```
// soln-1: 2-pass O(n) time
// we need to find 2 index i and j which a[0..i) and a(j..n-1] are sorted.
// notice max(a[0..i)) <= min(a[i, j]) and min(a(j..n-1]) >= max(a[i..j])
int findUnsortedSubarray(vector<int>& nums) {
   int l = 0, r = nums.size() - 1, mx = INT_MIN, mn = INT_MAX;

   while (l < r && nums[l + 1] >= nums[l]) l++;
   if (l >= r) return 0;
   while (nums[r - 1] <= nums[r]) r--;

   // find min and max in a[i..j]
   for (int i = l; i <= r; ++i) mx = max(mx, nums[i]), mn = min(mn, nums[i]);

   // back l and r boundary
   while (l >= 0 && mn < nums[l]) --l;
   while (r < nums.size() && mx > nums[r]) ++r;

   return r - l - 1;
}

// soln-2: 1-pass O(n) time (tricky)
int findUnsortedSubarray(vector<int>& nums) {
   int i = 0, j = - 1;
   for (int l = 0, r = j, mx = INT_MIN, mn = INT_MAX; r >= 0; --r, ++l) {
       mx = max(mx, nums[l]);          // max of left parts
       if (nums[l] != mx) j = l;       // max marks right boundary

       mn = min(mn, nums[r]);          // min of right parts
       if (nums[r] != mn) i = r;       // min marks left boundary
   }
   return j - i + 1;
}
```

Ref:

582. Kill process

Given n processes, each process has a unique PID (process id) and its PPID (parent process id).

Each process only has one parent process, but may have one or more children processes. This is just like a tree structure. Only one process has PPID that is 0, which means this process has no parent process. All the PIDs will be distinct positive integers.

We use two list of integers to represent a list of processes, where the first list contains PID for each process and the second list contains the corresponding PPID.

Now given the two lists, and a PID representing a process you want to kill, return a list of PIDs of processes that will be killed in the end. You should assume that when a process is killed, all its children processes will be killed. No order is required for the final answer.

```
[Example 1] - Input:
pid =  [1, 3, 10, 5]
ppid = [3, 0, 5, 3]
kill = 5
Output: [5,10]
Explanation:
           3
         /   \
```

```
        1     5
             /
           10
```
Kill 5 will also kill 10.

```
void helper(unordered_map<int, unordered_set<int> >& pt, int kill, vector<int>& ans) {
   ans.push_back(kill);
   for (int i : pt[kill]) {
       helper(pt, i, ans);
   }
}

vector<int> killProcess(vector<int>& pid, vector<int>& ppid, int kill) {
   vector<int> ans;
   unordered_map<int, unordered_set<int> > processTree;
   for (int i = 0; i < pid.size(); ++i) {
       processTree[ppid[i]].insert(pid[i]);
   }

   helper(processTree, kill, ans);
   return ans;
}
```

Ref:

583/712. Delete operation / Minimum ASCII delete sum for 2 strings

Given two words *word1* and *word2*, find the minimum number of steps required to make *word1* and *word2* the same, where in each step you can delete one character in either string.

Example 1:

`Input: "sea", "eat", Output: 2`

`Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".`

`#712 -` Given two strings `s1, s2`, find the lowest ASCII sum of deleted characters to make two strings equal.

```
// soln-1: dynamic programming
// let dp(i, j) be the min distance by deletion operation
// if s[i] == t[j] then dp(i, j) = dp(i - 1, j - 1)
// else dp(i, j) = min{dp(i - 1, j), dp(i, j - 1)} + 1
int minDistance(string word1, string word2) {
   vector<vector<int>> dp(word1.length() + 1, vector<int>(word2.length() + 1));

   for (int i = 0; i <= word1.length(); ++i) dp[i][0] = i;
   for (int j = 0; j <= word2.length(); ++j) dp[0][j] = j;

   for (int i = 1; i <= word1.length(); ++i) {
       for (int j = 1; j <= word2.length(); ++j) {
           if (word1[i - 1] == word2[j - 1]) dp[i][j] = dp[i - 1][j - 1];
           else dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1;
       }
   }
   return dp[word1.length()][word2.length()];
}
```

```
// soln-1: dynamic programming
// let dp(i, j) be the min delete sum
// if s[i] == t[j] then dp(i, j) = dp(i - 1, j - 1)
// else dp(i, j) = min{dp(i - 1, j) + s[i], dp(i, j - 1) + s[j]}
```

```
int minimumDeleteSum(string s1, string s2) {
    vector<vector<int>> dp(s1.length() + 1, vector<int>(s2.length() + 1));

   for (int i = 1; i <= s1.length(); ++i) dp[i][0] = s1[i - 1] + dp[i - 1][0];
   for (int j = 1; j <= s2.length(); ++j) dp[0][j] = s2[j - 1] + dp[0][j - 1];

   for (int i = 1; i <= s1.length(); ++i) {
       for (int j = 1; j <= s2.length(); ++j) {
           if (s1[i - 1] == s2[j - 1]) dp[i][j] = dp[i - 1][j - 1];
           else dp[i][j] = min(dp[i - 1][j] + s1[i - 1], dp[i][j - 1] + s2[j - 1]);
       }
   }
   return dp[s1.length()][s2.length()];
}
```

Ref: #5

584. TODO

Solution:

Ref:

585. TODO

Solution:

Ref:

587. TODO

Solution:

Ref:

591. TODO

Solution:

Ref:

593/611/812/976. Square/triangle ... math

```
// 593 - Valid Square
// soln-1: math
// 1. to form square, 4 edges should have equal length and so to 2 diagonal
// 2. similar for rectangle: must have 3 unique length and a*a + b*b = c*c
int d(vector<int>& p1, vector<int>& p2) {
```

```
    return (p1[0]-p2[0]) * (p1[0]-p2[0]) + (p1[1]-p2[1]) * (p1[1]-p2[1]);
}
bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {
    unordered_set s{d(p1, p2), d(p1, p3), d(p1, p4), d(p2, p3), d(p2, p4), d(p3, p4)};
    return s.size() == 2 && 0 == s.count(0);
}
```

```
// 611 - valid triangle number (how many valid combinations to form triangle)
// soln-1: math + two pointers in O(n^2)
int triangleNumber(vector<int>& nums) {
   sort(nums.begin(), nums.end());

   int ans = 0;
   for (int c = nums.size() - 1; c >= 2; --c) {
       int a = 0, b = c - 1;
       while (a < b) {
           (nums[a] + nums[b] > nums[c]) ? ans += b - a, --b : ++a;
       }
   }
   return ans;
}
```

```
// 812 - largest triangle area
// soln-1: math (brute-force)
// "three points triangle area formula" => S(ABC) = [S(AOB) + S(BOC) + S(COA)] / 2
double largestTriangleArea(vector<vector<int>>& points) {
   double ans = 0.0f;
   for (auto& A : points) {
       for (auto& B : points) {
           for (auto& C : points) {
               ans = max(ans, (A[0]*B[1] + B[0]*C[1] + C[0]*A[1] - A[0]*C[1] - C[0]*B[1] - B[0]*A[1]) / 2.0);
           }
       }
   }
   return ans;
}
```

```
// 976 - largest perimeter triangle (return largest possible perimeter of a triangle)
// soln-1: sorting
// sort the input then fetch from right, whenever triangle is able to form, it has the largest perimeter.
int largestPerimeter(vector<int>& A) {
    sort(A.begin(), A.end());
    for (int i = A.size() - 3; i >= 0; --i) {
        if (A[i] + A[i + 1] > A[i + 2]) return A[i] + A[i + 1] + A[i + 2];
    }
    return 0;
}
```

Ref:

594. Longest harmonious subsequence

We define a harmonious array is an array where the difference between its maximum value and its minimum value is **exactly** 1. Now, given an integer array, you need to find the length of its longest harmonious subsequence among all its possible subsequences.

Example 1:

Input: [1,3,2,2,5,2,3,7], **Output:** 5

Explanation: The longest harmonious subsequence is [3,2,2,2,3].

```
// soln-1: hashmap to count the numbers then check its neighbor
// soln-2: sort the given string and count
```

```
int findLHS(vector<int>& nums) {
   unordered_map<int, int> mp; // val -> count
   for (auto x : nums) mp[x]++;

   int ans = 0;
   for (auto& it : mp) {
       if (mp.find(it.first + 1) != mp.end()) {
           ans = max(ans, it.second + mp[it.first + 1]);
       }
   }
   return ans;
}
```

Ref:

597. TODO

Solution:

Ref:

599. Min index sum of 2 lists

trivial hashmap question

Ref:

600. Non-negative integers without consecutive 1s

Given a positive integer n, find the number of **non-negative** integers less than or equal to n, whose binary representations do NOT contain **consecutive ones**.

Example 1:

```
Input: 5, Output: 5
Explanation:
Here are the non-negative integers <= 5 with their corresponding binary representations:
0 : 0
1 : 1
2 : 10
3 : 11
4 : 100
5 : 101
Among them, only integer 3 disobeys the rule (two consecutive ones) and the other 5 satisfy the rule.
```

```
// soln-1: dynamic programming (memory exceeded)
// write down a few, then found what's happening, similar to "count bits"
// let dp(i) be if i contains consecutive 1s
// dp(i) = i & 3 == 3 || dp(i >> 1)
int findIntegers(int num) {
   vector<int> dp(num + 1);
   for (int i = 3; i <= num; ++i) dp[i] = dp[i >> 1] || ((i & 3) == 3);
   return count(dp.begin(), dp.end(), 0);
```

```
}

// soln-2: dynamic programming
// the previous one requires too much space, this soln will use const space
// https://www.geeksforgeeks.org/count-number-binary-strings-without-consecutive-1s/
// consider the given number as binary string, at index-i,
// let a[i] be the number of integers w/o consecutive 1s ending with 0
// let b[i] be the number of integers w/o consecutive 1s ending with 1
//
// similar to "#276 paint fence", the total number is a[i] + b[i]
// a[i] = a[i - 1] + b[i - 1]
// b[i] = a[i - 1]
//
// trick: the above answer is for 2^len - 1, but the question asks till num
// so we need to remove all over-counted numbers in [num + 1, 2^len - 1)
// observation:
// 1. "00" is causing over-counting, for example, num = "1000",
//    the 1st "00" over-counted "1010", the 2nd over-counted "1001"
//    *** why minus one[len - i -1] not one[i]?
// 2. "01" / "10" will not, as "01" -> "00" (lesser), "10" -> "11" (excluded because unqualified)
// 3. "11" - xx11xx00xx itself is not qualified
//           and all further qualified (xx10xx10xx / xx01xx10xx) would be less than num, so break.
int findIntegers(int num) {
   string str = bitset<sizeof(int) * 8>(num).to_string();
   str.erase(0, str.find_first_not_of('0'));
   int len = str.length();

   vector<int> zero(len), one(len);
   zero[0] = one[0] = 1;                     // have 1-bit
   for (int i = 1; i < len; ++i) {
       zero[i] = zero[i - 1] + one[i - 1];
       one[i] = zero[i - 1];
   }
   int ans = zero.back() + one.back();     // need to remove over-counted numbers (w/o consecutive 1s)

   for (int i = 1; i < len; ++i) {
       if (str[i - 1] == '0' && str[i] == '0') ans -= one[len - i - 1];    // HARD !!!
       else if ('1' == str[i - 1] && '1' == str[i]) break;
   }
   return ans;
}
```

Ref: #276

601. TODO

Solution:

Ref:

602. TODO

Solution:

Ref:

603. TODO

Solution:

Ref:

605. Can place flowers

Suppose you have a long flowerbed in which some of the plots are planted and some are not. However, flowers cannot be planted in adjacent plots - they would compete for water and both would die.

Given a flowerbed (represented as an array containing 0 and 1, where 0 means empty and 1 means not empty), and a number **n**, return if **n** new flowers can be planted in it without violating the no-adjacent-flowers rule.

Example 1:

`Input: flowerbed = [1,0,0,0,1], n = 1, Output: True`

```
// soln-1: greedy plant flowers by checking its neighbors
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
   for (int i = 0; i < flowerbed.size(); ++i) {
       if (flowerbed[i]) continue;
       if (i - 1 >= 0 && flowerbed[i - 1]) continue;
       if (i + 1 < flowerbed.size() && flowerbed[i + 1]) continue;

       flowerbed[i] = 1;
       if (--n <= 0) return true;
   }
   return n <= 0;
}
```

Ref: #495

607. TODO

Solution:

Ref:

608. TODO

Solution:

Ref:

610. TODO

Solution:

Ref:

612. TODO

Solution:

Ref:

613. TODO

Solution:

Ref:

614. TODO

Solution:

Ref:

615. TODO

Solution:

Ref:

617. Merge two binary trees

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

```
TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
   if (!t1 && !t2) return nullptr;

   TreeNode* root = new TreeNode((t1 ? t1->val : 0) + (t2 ? t2->val : 0));
   root->left = mergeTrees(t1 ? t1->left : nullptr, t2 ? t2->left : nullptr);
   root->right = mergeTrees(t1 ? t1->right : nullptr, t2 ? t2->right : nullptr);

   return root;
}
```

Ref:

618. TODO

Solution:

Ref:

619. TODO

Solution:

Ref:

623. Add one row to tree

Given the root of a binary tree, then value `v` and depth `d`, you need to add a row of nodes with value `v` at the given depth `d`. The root node is at depth 1.

The adding rule is: given a positive integer depth `d`, for each NOT null tree nodes `N` in depth `d-1`, create two tree nodes with value `v` as `N's` left subtree root and right subtree root. And `N's` original left subtree should be the left subtree of the new left subtree root, its original right subtree should be the right subtree of the new right subtree root. If depth `d` is 1 that means there is no depth d-1 at all, then create a tree node with value v as the new root of the whole original tree, and the original tree is the new root's left subtree.

```
void helper(TreeNode* node, int v, int d, int current_d) {
   if (!node || current_d >= d) return;

   if (current_d == d - 1) {
       node->left = new TreeNode(v, node->left, nullptr);
       node->right = new TreeNode(v, nullptr, node->right);
       return;
   }
   helper(node->left, v, d, current_d + 1);
   helper(node->right, v, d, current_d + 1);
}

TreeNode* addOneRow(TreeNode* root, int v, int d) {
   if (d == 1) return new TreeNode(v, root);   // special case
   helper(root, v, d, 1);
   return root;
}
```

Ref:

624. Max distance in arrays

Given m arrays, and each array is sorted in ascending order. Now you can pick up two integers from two different arrays (each array picks one) and calculate the distance. We define the distance between two integers a and b to be their absolute difference |a-b|. Your task is to find the maximum distance.

Example 1:

Input:
`[[1,2,3],`
` [4,5],`
` [1,2,3]]`
	Output: 4
	Explanation: One way to reach the maximum distance 4 is to pick 1 in the first or third array and pick 5 in the second array.

```
// 624 - soln-1: heap
int maxDistance(vector<vector<int>>& arrays) {
   priority_queue<pair<int, int>> mi, mx;
   for (int i = 0; i < arrays.size(); ++i) {
       mi.push({-arrays[i].front(), i}), mx.push({arrays[i].back(), i});
   }
   auto minx = mi.top(), maxx = mx.top();
   if (minx.second != maxx.second) return minx.first + maxx.first;
   mi.pop(), mx.pop();
   return max(minx.first + mx.top().first, mi.top().first + maxx.first);
}

// 624 - soln-2: array
int maxDistance(vector<vector<int>>& arrays) {
   int ans = 0, mi = arrays[0].front(), mx = arrays[0].back();
   for (int i = 1; i < arrays.size(); ++i) {
       // this always compare with new array front and end, so dont bother check if it's same line.
       ans = max(ans, max(abs(mi - arrays[i].back()), abs(mx - arrays[i].front())));
       mi = min(mi, arrays[i].front()), mx = max(mx, arrays[i].back());
   }
   return ans;
}
```

Ref:

626. TODO

Solution:

Ref:

628. Max product of 3 numbers

Given an integer array, find three numbers whose product is maximum and output the maximum product.

Example 1: `Input: [1,2,3], Output: 6`

Solution:
find out 3 max numbers and 2 min number

```
int maximumProduct(vector<int>& nums) {
   int b1 = INT_MIN, b2 = INT_MIN, b3 = INT_MIN;
   int s1 = INT_MAX, s2 = INT_MAX;
   for (int x : nums) {
       if (x > b1) {
```

```
            b3 = b2; b2 = b1; b1 = x;
        } else if (x > b2) {
            b3 = b2; b2 = x;
        } else if (x > b3) {
            b3 = x;
        }

        if (x < s1) {
            s2 = s1; s1 = x;
        } else if (x < s2) {
            s2 = x;
        }
    }
    return max(b1 * b2 * b3, b1 * s1 * s2);
}
```

Ref:

629. K inverse pairs array

Given two integers `n` and `k`, find how many different arrays consist of numbers from `1` to `n` such that there are exactly `k` inverse pairs.

We define an inverse pair as following: For `ith` and `jth` element in the array, if `i` < `j` and `a[i]` > `a[j]` then it's an inverse pair; Otherwise, it's not.

Since the answer may be very large, the answer should be modulo $10^9 + 7$.

Example 1:

```
Input: n = 3, k = 0, Output: 1
Explanation: Only the array [1,2,3] which consists of numbers from 1 to 3 has exactly 0 inverse pair.
```

```
// soln-1: dynamic programming
// let f(n, k) be the k-number of inverse pairs with n numbers
// - if n was put at the end of array, then we have f(n - 1, k) inverse pairs
// - if n was put at the last 2nd of array, then we have f(n - 1, k - 1) inverse pairs
// - ...
// - if n was put at the head of array, then we have f(n - 1, k - (n - 1))
//   since n is able to form n-1 inverse pairs with rest numbers
//
// f(n, k) = f(n - 1, k) + f(n - 1, k - 1) + ... + f(n - 1, k - n + 1)      (1)
// f(n, k + 1) = f(n - 1, k + 1) + f(n - 1, k) + ... + f(n - 1, k - n)      (2)
// combine (1) and (2):
// f(n, k + 1) = f(n - 1, k + 1) + f(n , k) - f(n - 1, k - n + 1)
//
// boundary cases:
// f(n, 0) = 1      ==> without inverse pairs means only one increasing sequence
// f(1, k) = 0      ==> 1 number would never be able for form pairs
int kInversePairs(int n, int k) {
   const long M = 1e9+7;
   vector<vector<long>> dp(n + 1, vector<long>(k + 1));

   dp[1][0] = 1;
   for (int i = 2; i <= n; ++i) {
       dp[i][0] = 1;
       for (int j = 1; j <= k; ++j) {
           dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
           if (j >= i) dp[i][j] -= dp[i - 1][j - i];

           dp[i][j] = (dp[i][j] + M) % M;  // +M because of above one, this could be negative
       }
   }
```

```
    return dp[n][k];
}
```

Ref:

631. TODO

Solution:

Ref:

632/908/910. Smallest range / I / II (review)

632 - You have `k` lists of sorted integers in ascending order. Find the smallest range that includes at least one number from each of the `k` lists. We define the range [a,b] is smaller than range [c,d] if `b-a < d-c` or `a < c` if `b-a == d-c`.

```
Input:[[4,10,15,24,26], [0,9,12,20], [5,18,22,30]], Output: [20,24]
Explanation:
List 1: [4, 10, 15, 24,26], 24 is in range [20,24].
List 2: [0, 9, 12, 20], 20 is in range [20,24].
List 3: [5, 18, 22, 30], 22 is in range [20,24].
```

908 - Given an array `A` of integers, for each integer `A[i]` we may choose any `x`with `-K <= x <= K`, and add `x` to `A[i]`. After this process, we have some array `B`. Return the smallest possible difference between the maximum value of `B` and the minimum value of `B`.

910 - Given an array `A` of integers, for each integer `A[i]` we need to choose either `x = -K` or `x = K`, and add `x` to `A[i] (only once)`. After this process, we have some array `B`. Return the smallest possible difference between the maximum value of `B` and the minimum value of `B`.

```
// 632 - smallest range
// soln-1: sliding window
// assuming each vector has its own color, if we merge the given vectors,
// the question is to find a smallest window containing all colors.
// same as other sliding window question, like #3, #76, #239
// 1. brute-force is to merge the input and then use slide window idea.
// 2. better way is to use priority-queue as slide window naturally.
vector<int> smallestRange(vector<vector<int>>& nums) {
    auto cmp = [](const pair<int, int>& a, const pair<int, int>& b) { return a.first > b.first; };
    priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp)> pq(cmp);

    int max_in_pq = INT_MIN;
    vector<int> idx(nums.size());
    for (int i = 0; i < nums.size(); ++i) pq.push({nums[i][0], i}), idx[i]++, max_in_pq = max(max_in_pq,
nums[i][0]);

    vector<int> ans{pq.top().first, max_in_pq};
    while (!pq.empty()) {
        auto x = pq.top();  pq.pop();
        if (ans[1] - ans[0] > max_in_pq - x.first) {
            ans[0] = x.first, ans[1] = max_in_pq;
        }
        if (idx[x.second] < nums[x.second].size()) {
```

```
            int n = nums[x.second][idx[x.second]];
            pq.push({n, x.second}), idx[x.second]++, max_in_pq = max(max_in_pq, n);
        } else {
            // one color is out-of-range
            break;
        }
    }
    return ans;
}
```

```
// 908 - soln-1: math
// max-diff = (max - min) - 2 * k < 0 ? 0 : (max - min) - 2 * k
int smallestRangeI(vector<int>& A, int K) {
    int maxx = INT_MIN, minx = INT_MAX;
    for (auto x : A) maxx = max(maxx, x), minx = min(minx, x);

    return maxx - minx > 2 * K ? (maxx - minx) - 2 * K : 0;
}
```

```
// 910 - soln-1: greedy
// observation:
// 1. For each a[i], we may choose +k/-k to get array B.
// 2. If we +k to each element in B, the result would not change.
// 3. Hence the question is to add 0/2k to A.
// 4. If we sort A, then +2k for each a[i] in hoping reduce diff., the min/max of B could potentially change:
//          min-of-B = min(A.front() + 2k, A[i + 1])
//          max-of-B = max(A.back() + 0, A[i] + 2 * k)
// corner case:
// 1. if max(A) - min(A) <= k, return max(A) - min(A) (we can not -k for any number to reduce diff.)
// 2. if max(A) - min(A) >= 4k, return max(A) - min(A) - 2k. (why 4k? not so obvious)
int smallestRangeII(vector<int>& A, int k) {
    sort(A.begin(), A.end());

    int ans = A.back() - A.front();
    for (int i = 0; i < A.size() - 1; ++i) {
        int mx = max(A.back(), A[i] + 2 * k);        // in hoping reduce diff., +2k from left
        int mn = min(A.front() + 2 * k, A[i + 1]);
        ans = min(ans, mx - mn);
    }
    return ans;
}
```

Ref:

633/507. Sum of square numbers/Perfect number

Given a non-negative integer `c`, your task is to decide whether there're two integers `a` and `b` such that $a^2 + b^2 = c$.

Example 1:

```
Input: 5, Output: True
Explanation: 1 * 1 + 2 * 2 = 5
```

```
// 507: soln-1: math
bool checkPerfectNumber(int num) {
    if (num < 2) return false;

    int sum = 1;
    for (int i = 2; i <= sqrt(num); ++i) {
        if (num % i) continue;
        sum += num / i + i;
    }
```

```
    return sum == num;
}
```

```
// 633: soln-1: two-pointers
// 2-sum by hashset also applies
bool judgeSquareSum(int c) {
    for (int left = 0, right = sqrt(c); left < right; nullptr) {
        int tmp = left * left + right * right;
        if (tmp == c) return true;
        tmp > c ? --right : ++left;
    }
    return false;
}
```

Ref: #1 #367

634. TODO

Solution:

Ref:

637. Average of levels in binary tree

Given a non-empty binary tree, return the average value of the nodes on each level in the form of an array.

Example 1: `Input:`

```
    3
   / \
  9  20
    /  \
   15   7
Output: [3, 14.5, 11]
Explanation:
The average value of nodes on level 0 is 3,  on level 1 is 14.5, and on level 2 is 11. Hence return
[3, 14.5, 11].
```

```
void helper(TreeNode* node, int h, vector<pair<long, int> >& ans) {
    if (!node) return;

    if (h >= ans.size()) ans.push_back({0, 0});
    ans[h].first += node->val;      // sum the value
    ++ans[h].second;                // keep the number

    helper(node->left, h + 1, ans);
    helper(node->right, h + 1, ans);
}

vector<double> averageOfLevels(TreeNode* root) {
    vector<pair<long, int> > ans;
    helper(root, 0, ans);

    vector<double> ret;
    for (auto i : ans) {
        ret.push_back((double)i.first / i.second);
    }
    return ret;
}
```

Ref:

638/983. Shopping offers/Min costs for tickets (review)

You are given the each item's price, a set of special offers, and the number we need to buy for each item. The job is to output the lowest price you have to pay for **exactly** certain items as given, where you could make optimal use of the special offers.

Each special offer is represented in the form of an array, the last number represents the price you need to pay for this special offer, other numbers represents how many specific items you could get if you buy this offer.

You could use any of special offers as many times as you want.

Example 1:

```
Input: [2,5], [[3,0,5],[1,2,10]], [3,2], Output: 14
Explanation:
There are two kinds of items, A and B. Their prices are $2 and $5 respectively.
In special offer 1, you can pay $5 for 3A and 0B. In special offer 2, you can pay $10 for 1A and 2B.
Offers may be used many times.
You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer #2), and $4 for 2A.
```

```
// soln-1: backtracking
// let f(i) be the minimal cost given special offers, we have to systematically enumerate
// all combinations. Greedy is not going to work.
//
// this question is also marked as Dynamic Programming, somehow we should cache the value
// for each needs.
bool operator<(const vector<int>& needs, int target) {
   for (int x : needs) if (x < target) return true;
   return false;
}
int operator*(const vector<int>& needs, const vector<int>& price) {
   int res = 0;
   for (int i = 0; i < needs.size(); ++i) res += needs[i] * price[i];
   return res;
}
void operator-=(vector<int>& needs, const vector<int>& offer) {
   for (int i = 0; i < needs.size(); ++i) needs[i] -= offer[i];
}
void operator+=(vector<int>& needs, const vector<int>& offer) {
   for (int i = 0; i < needs.size(); ++i) needs[i] += offer[i];
}

int shoppingOffers(vector<int>& price, vector<vector<int>>& special, vector<int>& needs, int cost = 0) {
   if (needs < 0) return INT_MAX;

   int m = cost + needs * price;
   for (auto& offer : special) {                // enumerate systematically, greedy is not working
       if (cost + offer.back() >= m) continue; // pruning search

       needs -= offer;
       m = min(m, shoppingOffers(price, special, needs, cost + offer.back()));
       needs += offer;
   }
   return m;
}
```

```
// 983 - min cost for tickets
// soln-1: dynamic programming
// f(x) = min{f(x-1) + c[0], f(x-7) + c[1], f(x-30) + c[2]}
int mincostTickets(vector<int> &days, vector<int> &costs) {
```

```
    vector<int> dp(days.size());
    dp[0] = costs[0];
    for (int i = 1; i < days.size(); ++i) {
        dp[i] = dp[i - 1] + costs[0];

        auto d7 = upper_bound(days.begin(), days.end(), days[i] - 7);
        dp[i] = min(dp[i],  (d7 == days.begin() ? 0 : dp[d7 - days.begin() - 1]) + costs[1]);

        auto d30 = upper_bound(days.begin(), days.end(), days[i] - 30);
        dp[i] = min(dp[i], (d30 == days.begin() ? 0 : dp[d30 - days.begin() - 1]) + costs[2]);
    }
    return dp.back();
}
```

Ref:

643/644. Max average subarray I/II (review)

643 - Given an array consisting of n integers, find the contiguous subarray of given length k that has the maximum average value. And you need to output the maximum average value.

644 - output the maximum average value if the size of subarray >= k.

Example:

Input: [1,12,-5,-6,50,3], k = 4, Output: 12.75

Explanation:

when length is 5, maximum average value is 10.8, when length is 6, maximum average value is 9.16667. Thus return 12.75.

```
// 643 - soln-1: sliding window with size-k
double findMaxAverage(vector<int>& nums, int k) {
   double sum = accumulate(nums.begin(), nums.begin() + k, 0);

   double ans = sum;
   for (int i = k; i < nums.size(); ++i) {
       sum += (nums[i] - nums[i - k]);
       ans = max(ans, sum);
   }
   return ans / k;
}
```

```
// 644 - soln-1: value-range based binary search + sliding window
// brute-force will take O(n^2) time as we have n^2 subarrays.
// binary search: lo = min(A), hi = max(A), then verify if we have subarray with average >= target
double findMaxAverageII(vector<int>& nums, int k) {
   double lo = *min_element(nums.begin(), nums.end()), hi = *max_element(nums.begin(), nums.end());
   while (lo + 1e-5 < hi) {
       double m = lo + (hi - lo) / 2;
       maxAverageSubarrayK(nums, k, m) ? lo = m : hi = m;
   }
   return lo;
}

// return true if there is a subarray with size >= k, and average >= given target
// if we have a subarray such that: A[i] + .. + A[i + k] >= target * k
//                            then: (A[i] - target) + .. + (A[i + k] - target) >= 0
// the beauty of this is, we can ignore the position where the minimum sum was, just update minPreSum so far.
//
// .....................\............./........
```

```
//       ^              ^       k       ^
//     minPreSum      preSum          sum
// minPreSum is somewhere before sliding window which has min sum of [0...x]
// if (sum - minSum) >= 0, then we found the subarray [x..i] which is size >= k and average >= target
//
bool maxAverageSubarrayK(vector<int>& nums, int k, double target) {
   double sum = 0, minPreSum = 0, preSum = 0;
   for (int i = 0; i < k; ++i) sum += nums[i] - target;
   if (sum >= 0) return true;

   for (int i = k; i < nums.size(); ++i) {
       sum += nums[i] - target;
       preSum += nums[i - k] - target, minPreSum = min(minPreSum, preSum);
       if (sum >= minPreSum) return true;    // sum - minPreSum >= 0  ==> the average for this subarray >=
target
   }
   return false;
}
```

Ref:

645. Set mismatch

The set `S` originally contains numbers from 1 to `n`. But unfortunately, due to the data error, one of the numbers in the set got duplicated to another number in the set, which results in repetition of one number and loss of another number.

Given an array `nums` representing the data status of this set after the error. Your task is to firstly find the number occurs twice and then find the number that is missing. Return them in the form of an array.

Example 1:

`Input: nums = [1,2,2,4], Output: [2,3]`

Solution:

soln-1: bit manipulation

1. use XOR to find out missing^dup = md
2. divide given number (2 sets: given and 1..n) into 2 groups to find out missing number and dup number
3. then distinguish which one is dup, which one is missing

soln-2: negate the number corresponding to its position to mark dups. For those not as negative, it's missing.

#260 - single number III (2 appeared once)
#442 - find all dups (negative the number corresponding to it position as visited/duped mark)

```
// soln-1: bit manipulation
vector<int> findErrorNums(vector<int>& nums) {
   // find out missing ^ dup = md
   int md = 0;
   for (int i = 0; i < nums.size(); ++i) md ^= (i + 1) ^ nums[i];

   // divide number into 2 groups by last digit
   int mask = md & ~(md - 1), n1 = 0, n2 = 0;
   for (int i = 0; i < nums.size(); ++i) {
       mask & nums[i] ? n1 ^= nums[i] : n2 ^= nums[i];
       mask & (i + 1) ? n1 ^= (i + 1) : n2 ^= (i + 1);
   }

   // determine which one is not missing
   for (int x : nums) if (x == n1) return vector<int>{n1, n2};

   return vector<int>{n2, n1};
}
```

Ref: #260

647. Palindromic substrings

Given a string, your task is to count how many palindromic substrings in this string.

The substrings with different start indexes or end indexes are counted as different substrings even they consist of same characters.

Example 1:

```
Input: "abc", Output: 3
Explanation: Three palindromic strings: "a", "b", "c".
```

```
// soln-1: extend substring and count
int countSubstrings(string s) {
   int ans = 0, n = s.length();
   for (int i = 0; i < n; ++i) {
       for (int l = i, r = i; l >= 0 && r < n && s[l] == s[r]; --l, ++r)       ++ans; // odd palindromic
substring
       for (int l = i, r = i + 1; l >= 0 && r < n && s[l] == s[r]; --l, ++r)   ++ans; // even palindromic
substring
   }
   return ans;
}

// soln-2: dynamic programming
// let dp(i, j) be true if s[i] == s[j] && dp(i+1, j-1), count if dp(i, j) is true
// similar to #5, instead of keeping longest, here we keep true/false
int countSubstrings(string s) {
   int ans = 0, n = s.length();
   vector<vector<bool>> dp(n, vector<bool>(n));

   for (int i = n - 1; i >= 0; --i) {
       dp[i][i] = true;
       for (int j = i; j < n; ++j) {
           dp[i][j] = s[i] == s[j];
           if (i + 1 < j - 1 && dp[i][j]) dp[i][j] = dp[i][j] && dp[i + 1][j - 1];

           if (dp[i][j]) ++ans;
       }
   }
   return ans;
}
```

Ref: #5

649. TODO

Solution:

Ref:

650/651. 2-keys/4-keys keyboard

Initially on a notepad only one character 'A' is present. You can perform two operations on this notepad for each step:

1. `Copy All`: You can copy all the characters present on the notepad (partial copy is not allowed).
2. `Paste`: You can paste the characters which are copied **last time**.

Given a number n. You have to get **exactly** n 'A' on the notepad by performing the minimum number of steps permitted. Output the minimum number of steps to get n 'A'.

#651 - 4-keys keyboard

Imagine you have a special keyboard with the following keys:

1. Key 1: (A): Print one 'A' on screen.
2. Key 2: (Ctrl-A): Select the whole screen.
3. Key 3: (Ctrl-C): Copy selection to buffer.
4. Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

Now, you can only press the keyboard for N times (with the above four keys), find out the maximum numbers of 'A' you can print on screen.

Example 1:

Input: N = 7, Output: 9
Explanation: We can at most get 9 A's on screen by pressing following key sequence: A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

```
// soln-1: dynamic programming
// we want to reuse previous block as possible, for example,
// if target = 20, we have 10, then copy and paste, dp(20) = dp(10) + 2
// let dp(i) be the min steps for i
//     dp(i) = dp(k) + i/k, 1 <= k < i
// dp(2) = 2
int minSteps(int n) {
   vector<int> dp(n + 1);
   for (int i = 2; i <= n; ++i) {
       dp[i] = i;
       for (int j = i - 1; j >= 2; --j) {
           if (i % j == 0) {
               dp[i] = dp[j] + i / j;
               break;
           }
       }
   }
   return dp.back();
}
```

```
// soln-1: dynamic programming
// at step i, we have 3 choices:
// 1, at least we can print i-length string
// 2, do multiple paste action from j = [1, i-2) with (i - j - 1) times
// let dp(i) be the max length with i-steps
//     dp(i) = max{dp(i), dp(j) + dp(j) * (i - j - 2) | 1 <= j <= i - 3}
//                         ~~~     ~~~~~~~~~~~~~~~~~~
//                        orig      by pasting
int maxA(int N) {
   vector<int> dp(N + 1);
   for (int i = 1; i <= N; ++i) {
       dp[i] = i;
       for (int j = 1; j <= i - 3; ++j) {
           dp[i] = max(dp[i], dp[j] + dp[j] * (i - j - 2));
       }
   }
   return dp.back();
}
```

Ref:

654/998. Maximum binary tree (review)

```
// 654 - max binary tree (root is the max, left and right child is the max divided by root)
// soln-1: monotone stack (decreasing)
// 0. brute-force will be in O(nlgn) time if tree is balanced, worst case is O(n^2).
// 1. monotone decreasing stack in O(n) time,
//    1) if cur < top, cur must be right child of top.
//    2) else pop stack until cur < top, cur will be the right of top,
//       those popped out will be left subtree of cur.
TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
    vector<TreeNode*> s;
    for (auto x : nums) {
        auto n = new TreeNode(x);
        while (!s.empty() && s.back()->val < x) n->left = s.back(), s.pop_back();

        if (!s.empty()) s.back()->right = n;
        s.push_back(n);
    }
    return s.front();
}
```

```
// 998 - max binary tree II
// soln-1: recursion
TreeNode* insertIntoMaxTree(TreeNode* root, int val) {
    if (nullptr == root || root->val < val) {
        auto r = new TreeNode(val);
        r->left = root;
        return r;
    }
    root->right = insertIntoMaxTree(root->right, val);
    return root;
}
```

Ref: #105

655. Print binary tree

Print a binary tree in an m*n 2D string array following these rules:

1. The row number `m` should be equal to the height of the given binary tree.
2. The column number `n` should always be an odd number.
3. The root node's value (in string format) should be put in the exactly middle of the first row it can be put. **The column and the row where the root node belongs will separate the rest space into two parts (left-bottom part and right-bottom part)**. You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree. However, if two subtrees are none, then you don't need to leave space for both of them.
4. Each unused space should contain an empty string `""`.
5. Print the subtrees following the same rules.

```
int height(TreeNode* root) {
   if (!root) return 0;
   return 1 + max(height(root->left), height(root->right));
}

void helper(TreeNode* node, int row, int col_start, int col_end, vector<vector<string> >& ans) {
   if (!node) return;

   int col = col_start + (col_end - col_start)/2;
   ans[row][col] = to_string(node->val);
   helper(node->left, row + 1, col_start, col - 1 , ans);
   helper(node->right, row + 1, col + 1, col_end, ans);
}
```

```
vector<vector<string> > printTree(TreeNode* root) {
   int h = height(root);
   int col = (h == 1 ? 1: (2 << (h - 2)) * 2 - 1);
   vector<vector<string> > ans(h, vector<string>(col, ""));

   helper(root, 0, 0, col, ans);
   return ans;
}
```

Ref:

656. TODO

Solution:

Ref:

658. Find K closest elements

Given a sorted array, two integers `k` and `x`, find the `k` closest elements to `x` in the array. The result should also be sorted in ascending order. If there is a tie, the smaller elements are always preferred.

Example 1:

`Input: [1,2,3,4,5], k=4, x=3, Output: [1,2,3,4]`

```
// soln-1: binary search with sliding window
// Assuming our initial window is on [n-k, n), then we need to move this
// window to proper position between [0, n-k). We compare 2 ends with x,
// whichever is closer to x, move the other side. (move low first as the result expect smaller set)
vector<int> findClosestElements(vector<int>& arr, int k, int x) {
   int lo = 0, hi = arr.size() - k;
   while (lo < hi) {
       int m = lo + (hi - lo) / 2;                    // hopefully x is in the middle, then check
       if (x - arr[m] > arr[m + k] - x) lo = m + 1;    // hi end is closer (move low pointer first)
       else hi = m;
   }
   return vector<int>(arr.begin() + lo, arr.begin() + lo + k);
}
```

Ref:

660. TODO

Solution:

Ref:

661. Image smoother

trivial question

```
// soln-1: brute-force (in O(1) space similar to "Game of Life")
```

```
vector<vector<int>> imageSmoother(vector<vector<int>>& M) {
   const vector<pair<int, int>> dirs {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 0}, {0, 1}, {1, -1}, {1, 0}, {1,
1}};
   for (int i = 0; i < M.size(); ++i) {
       for (int j = 0; j < M[0].size(); ++j) {
           int cells = 0, gray = 0;
           // sum grayscale from each directory
           for (auto& d : dirs) {
               int ni = i + d.first, nj = j + d.second;
               if (ni < 0 || ni >= M.size() || nj < 0 || nj >= M[0].size()) continue;
               ++cells, gray += (M[ni][nj] & 0xff);
           }
           M[i][j] |= ((gray / cells) << 8);
       }
   }

   for (auto& row : M) {
       for (auto& x : row) x = (x >> 8);
   }
   return M;
}
```

Ref:

662. Max width of binary tree

The width of one level is defined as the length between the end-nodes (the leftmost and right most non-null nodes in the level, where the null nodes between the end-nodes are also counted into the length calculation.

```
// soln-2: add index to each node
// traversal to find out the leftmost node, and keep it.
// for each node, we know the distance from itself to leftmost node in the same level
void helper(TreeNode* node, int idx, int depth, vector<int>& leftMostIdx, int& ans) {
   if (!node) return;

   if (depth >= leftMostIdx.size()) leftMostIdx.push_back(idx);
   helper(node->left, 2 * idx, depth + 1, leftMostIdx, ans);
   helper(node->right, 2 * idx + 1, depth + 1, leftMostIdx, ans);

   ans = max(ans, idx - leftMostIdx[depth] + 1);
}

int widthOfBinaryTree(TreeNode* root) {
   int ans = 0;
   vector<int> idx;
   helper(root, 1, 0, idx, ans);
   return ans;
}
```

Ref:

663. Equal tree partition

Given a binary tree with n nodes, your task is to check if it's possible to partition the tree to two trees which have the equal sum of values after removing exactly one edge on the original tree.

Solution:

```
1, count the total sum of tree, then check if any node including its children has half of
sum.
2, target node has to have parent node to cut off. (every node has parent except root.)
```

```
int countTree(TreeNode* node) {
```

```
    if (!node) return 0;
    return (countTree(node->left) + countTree(node->right) + node->val);
}

// return sum of subtree
int helper(TreeNode* node, TreeNode* root, bool& dividable, int half) {
    if (!node) return 0;
    if (dividable) return 0;

    int ret = helper(node->left, root, dividable, half)+ helper(node->right, root, dividable, half) + node->val;
    if (ret == half && node != root) dividable = true;
    return ret;
}

bool checkEqualTree(TreeNode* root) {
    int total = countTree(root);
    if (total % 2) return false;

    bool dividable = false;
    helper(root, root, dividable, total / 2);
    return dividable;
}
```

Ref:

664. Strange printer

There is a strange printer with the following two special requirements:

1. The printer can only print a sequence of the same character each time.
2. At each turn, the printer can print new characters starting from and ending at any places, and will cover the original existing characters.

Given a string consists of lower English letters only, your job is to count the minimum number of turns the printer needed in order to print it.

Example 1:

`**Input:** "aaabbb", **Output:** 2`

`**Explanation:** Print "aaa" first and then print "bbb".`

```
// soln-1: dynamic programming
// let f(i, j) be the minimum turns needed to print from i to j.
// f(i, j) = min{ f(i, k) + f(k + 1, j) - (a[k] == a[j] ? 1 : 0) | i <= k < j}
//
// corner case:
// - f(i, i) = 1
// - f(i, j) is at most (j - i + 1)
int strangePrinter(string s) {
    if (s.empty()) return 0;
    vector<vector<int>> dp(s.length(), vector<int>(s.length(), s.length()));

    for (int i = s.length() - 1; i >= 0; --i) {      // from bottom to up in the matrix
        dp[i][i] = 1;
        for (int j = i + 1; j < s.length(); ++j) {
            for (int k = i; k < j; ++k) {             // k starting from i, not i + 1
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] - (s[k] == s[j]));
            }
        }
    }
    return dp[0][s.length() - 1];
}
```

Ref: #5

665. Non-decreasing array

Given an array with `n` integers, your task is to check if it could become non-decreasing by modifying **at most** `1`element.

We define an array is non-decreasing if `array[i] <= array[i + 1]` holds for every `i` (1 <= i < n).

Example 1:

```
Input: [4,2,3], Output: True
Explanation: You could modify the first 4 to 1 to get a non-decreasing array.
```

```
// soln-1: greedy
// if a[i] < a[i-1], we could either increase a[i] or decrease a[i-1].
// 1, increase a[i] would have higher risk, so this is a 2nd choice
// 2, so try to decrease a[i-1] if possible
bool checkPossibility(vector<int>& nums) {
   int modify = 0;
   for (int i = 1; modify < 2 && i < nums.size(); ++i) {
       if (nums[i] < nums[i - 1]) {
           if (i - 2 < 0 || nums[i - 2] <= nums[i]) {
               ++modify, nums[i - 1] = nums[i];    // decrease a[i-1]
           } else {
               ++modify, nums[i] = nums[i - 1];    // increase a[i]
           }
       }
   }
   return modify < 2;
}
```

Ref:

669/700/701. Trim a BST/Search in a BST/Insert into a BST

669 - Given a binary search tree and the lowest and highest boundaries as `L` and `R`, trim the tree so that all its elements lies in `[L, R]` (R >= L). You might need to change the root of the tree, so the result should return the new root of the trimmed binary search tree.

```
// 669 - trim a binary search tree
TreeNode* trimBST(TreeNode* root, int L, int R) {
   if (!root) return nullptr;
   if (root->val < L) return trimBST(root->right, L, R);
   if (root->val > R) return trimBST(root->left, L, R);
   root->left = trimBST(root->left, L, R);
   root->right = trimBST(root->right, L, R);
   return root;
}
```

```
// 701 - insert into binary search tree
TreeNode* insertIntoBST(TreeNode* root, int val) {
    TreeNode* parent = nullptr;
    for (TreeNode* cur = root; cur; cur = (cur->val < val) ? cur->right : cur->left) {
        parent = cur;
    }
    auto n = new TreeNode(val);
    if (parent) parent->val > val ? parent->left = n : parent->right = n;
    return root;
}
```

Ref:

675. Cut off trees for golf event

You are asked to cut off trees in a forest for a golf event. The forest is represented as a non-negative 2D map, in this map:

1. `0` represents the `obstacle` can't be reached.
2. `1` represents the `ground` can be walked through.
3. `The place with number bigger than 1` represents a `tree` can be walked through, and this positive number represents the tree's height.

You are asked to cut off all the trees in this forest in the order of tree's height - always cut off the tree with lowest height first. And after cutting, the original place has the tree will become a grass (value 1).

You will start from the point (0, 0) and you should output the minimum steps you need to walk to cut off all the trees. If you can't cut off all the trees, output -1 in that situation.

You are guaranteed that no two `trees` have the same height and there is at least one tree needs to be cut off.

Example 1:
```
Input:
[
 [1,2,3],
 [0,0,4],
 [7,6,5]
]
Output: 6
```

Solution:

```
soln-1:
1) collect all tree position (either use priority queue or array then simply sort it)
2) get shortest distance from x to y (either use all-pairs-shortest-path to compute it first or use bidirectional BFS)
```

```
struct pair_hash_t {
   size_t operator() (const pair<int, int>& p) const {
       return std::hash<int>{}(p.first ^ p.second);
   }
};
typedef unordered_set<pair<int, int>, pair_hash_t> tree_set_t;

bool getNeighbors(const vector<vector<int>>& forest, pair<int, int> from, const tree_set_t& other,
                  tree_set_t& ans, vector<vector<bool>>& visited) {
   const vector<pair<int, int>> dirs{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
   for (auto dir : dirs) {
       int x = from.first + dir.first, y = from.second + dir.second;
       if (x < 0 || x >= forest.size() || y < 0 || y >= forest[0].size() || forest[x][y] < 1) continue;
       if (other.find({x, y}) != other.end()) return true;
       if (visited[x][y]) continue;

       visited[x][y] = true;
       ans.insert({x, y});
   }
   return false;
}

// optimization-2: check if there is a shortest path between from and to.
int shortestPath(pair<int, int> from, pair<int, int> to, const vector<vector<int>>& forest) {
   // TODO
   return -1;
}

// BFS bidirectional search to get distance between two nodes.
int BBFS(pair<int, int> from, pair<int, int> to, const vector<vector<int>>& forest) {
   int dist = 0;
   bool hit = false;
   vector<vector<bool>> visited(forest.size(), vector<bool>(forest[0].size(), false));
   tree_set_t s[2];
```

```cpp
    s[0].insert(from), s[1].insert(to);
    visited[from.first][from.second] = visited[to.first][to.second] = true;

    while (!hit && !s[0].empty() && !s[1].empty()) {
        ++dist;

        tree_set_t neighbor;
        int i = s[0].size() <= s[1].size() ? 0 : 1;
        for (auto p : s[i]) {
            if (getNeighbors(forest, p, s[(i + 1) % 2], neighbor, visited)) {
                hit = true;
                break;
            }
        }
        if (!hit) s[i] = neighbor;
    }
    return hit ? dist : -1;
}

// try shortest path first, then use bidirectional BFS search if needed.
int getDistance(pair<int, int> from, pair<int, int> to, const vector<vector<int>>& forest) {
    if (from == to) return 0;

    int ans = shortestPath(from, to, forest);
    if (-1 == ans) ans = BBFS(from, to, forest);
    return ans;
}

int DFS(const vector<vector<int>>& forest, vector<vector<bool>>& visited, int x, int y) {
    const vector<pair<int, int>> dirs{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

    visited[x][y] = true;
    int count = (forest[x][y] > 1) ? 1 : 0;

    for (auto dir : dirs) {
        int nx = x + dir.first, ny = y + dir.second;
        if (nx < 0 || nx >= forest.size() || ny < 0 || ny >= forest[0].size()) continue;
        if (visited[nx][ny] || forest[nx][ny] < 1) continue;

        count += DFS(forest, visited, nx, ny);
    }
    return count;
}

int cutOffTree(vector<vector<int> >& forest) {
    vector<vector<int>> trees;
    for (int i = 0; i < forest.size(); ++i) {
        for (int j = 0; j < forest[i].size(); ++j) {
            if (forest[i][j] > 1) trees.push_back({i, j, forest[i][j]});
        }
    }
    std::sort(trees.begin(), trees.end(), [](const vector<int>& a, const vector<int>& b) {
        return a[2] < b[2];
    });

    // optimization-1: terminate search earlier if possible (little help)
    vector<vector<bool>> visited(forest.size(), vector<bool>(forest[0].size(), false));
    if (DFS(forest, visited, 0, 0) != trees.size()) return -1;

    int ans = 0;
    pair<int, int> lastPos{0, 0};
    for (vector<int> tree: trees) {
        int dist = getDistance(lastPos, {tree[0], tree[1]}, forest);
```

```
        if (-1 == dist) return -1;

        ans += dist;
        lastPos = {tree[0], tree[1]};
    }
    return ans;
}
```

Ref: #126

682. Baseball game

You're now a baseball game point recorder.

Given a list of strings, each string can be one of the 4 following types:

1. `Integer` (one round's score): Directly represents the number of points you get in this round.
2. `"+"` (one round's score): Represents that the points you get in this round are the sum of the last two `valid`round's points.
3. `"D"` (one round's score): Represents that the points you get in this round are the doubled data of the last `valid` round's points.
4. `"C"` (an operation, which isn't a round's score): Represents the last `valid` round's points you get were invalid and should be removed.

Each round's operation is permanent and could have an impact on the round before and the round after.

You need to return the sum of the points you could get in all the rounds.

Example 1:

```
Input: ["5","2","C","D","+"], Output: 30
Explanation:
Round 1: You could get 5 points. The sum is: 5.
Round 2: You could get 2 points. The sum is: 7.
Operation 1: The round 2's data was invalid. The sum is: 5.
Round 3: You could get 10 points (the round 2's data has been removed). The sum is: 15.
Round 4: You could get 5 + 10 = 15 points. The sum is: 30.
```

```
// soln-1: easy stack application
int calPoints(vector<string>& ops) {
    int ans = 0;
    stack<pair<int, int>> last;
    for (auto& op : ops) {
        if ("+" == op) {
            int t = last.top().first + last.top().second;
            ans += t;
            last.push({t, last.top().first});
        } else if ("C" == op) {
            ans -= last.top().first;
            last.pop();
        } else if ("D" == op) {
            ans += last.top().first * 2;
            last.push({last.top().first * 2, last.top().first});
        } else {
            ans += stoi(op);
            last.push({stoi(op), last.empty() ? 0 : last.top().first});
        }
    }
    return ans;
}
```

Ref:

683. TODO

Solution:

Ref:

684/685. Redundant connection/II

684 - undirected graph, 685 - directed.

The given input is a graph that started as a tree with N nodes (with distinct values 1, 2, ..., N), with one additional edge added. The added edge has two different vertices chosen from 1 to N, and was not an edge that already existed.

Return an edge that can be removed so that the resulting graph is a tree of N nodes.

```
Input: [[1,2], [1,3], [2,3]]
Output: [2,3]
Explanation: graph will be like this:
  1
 / \
2 - 3
```

```
Input: [[1,2], [2,3], [3,4], [1,4], [1,5]]
Output: [1,4]
Explanation: graph will be like this:
5 - 1 - 2
    |   |
    4 - 3
```

```
// 684 - soln-1: union-find cycle detection
vector<int> findRedundantConnection(vector<vector<int>>& edges) {
   UnionFind uf(1001);     // N is between 3 to 1000.
   for (auto& e : edges) {
       if (uf.Find(e[0]) == uf.Find(e[1])) return e;

       uf.Union(e[0], e[1]);
   }
   return vector<int>{-1, -1};
}
```

```
// 685 - soln-1: cycle detection with 3 cases
// For this question, there are 3 cases:
// 1) if a node has 2 in-degrees, one edge must be removed, as example-1.
// 2) if there is a cycle, one edge must be removed, as example-2.
// 3) if case-1 and case-2, for exmaple: [[1, 2], [2, 3], [4, 2], [3, 4]].
// so the soln is to find the 2 in-degree edges, and cut the 2nd edge,
// then detect cycle, if still found cycle, it means we need to remove 1st edge.
vector<int> findRedundantDirectedConnection(vector<vector<int>>& edges) {
    pair<int, int> A, B;            // keep 2 edges that makes a node 2 in-degree.
    unordered_map<int, int> parentMap;
    for (auto& e : edges) {
        if (parentMap.find(e[1]) != parentMap.end()) {
            A = {parentMap[e[1]], e[1]}, B = {e[0], e[1]}, e[0] = -1;  // path B was cut
            break;
        }
        parentMap[e[1]] = e[0];
    }

    UnionFind uf(edges.size() + 1);
    for (auto& e : edges) {
        if (e[0] == -1) continue;              // skip this special edge
        if (uf.Find(e[0]) == uf.Find(e[1])) {
            // cycle found even if path B is cut, could be A or e.
            return A.first ? vector<int>{A.first, A.second} : e;
        }
        uf.Union(e[0], e[1]);
```

```
    }
    return vector<int>{B.first, B.second};    // no cycle has been found
}
```

Ref: #685

686. Repeated string match (review)

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it. If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times (“abcdabcdabcd”), B is a substring of it; and B is not a substring of A repeated two times ("abcdabcd").

```
// soln-1: brute force in O(mn) time
// 1, view A as infinite length
// 2, for each position in A, try to match B from start
// 3, if B is matched, we found repeated A matching B.
int repeatedStringMatch(string A, string B) {
   for (int i = 0, j = 0; i < A.length(); ++i) {
       for (j = 0; j < B.length(); ++j) {
           if (A[(i + j) % A.length()] != B[j]) break;
       }

       if (B.length() == j) {
           int ans = (i + j) / A.length();
           int offset = (i + j) % A.length() == 0 ? 0 : 1;
           return ans + offset;
       }
   }
   return -1;
}
```

Ref: #459

687. Longest univalue path of binary tree

Given a binary tree, find the length of the longest path where each node in the path has the same value. This path may or may not pass through the root.

Note: The length of path between two nodes is represented by the number of edges between them.

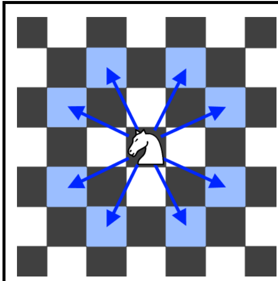
```
// dfs - post-order bottom up search
// return max possible contribution to its parent (either left or right plus me)
int helper(TreeNode* node, int& ans) {
   if (!node) return 0;

   int left = helper(node->left, ans);
   int right = helper(node->right, ans);

   int ret = 0, combinePath = 0;
   if (node->left && node->val == node->left->val) {
       combinePath = left + 1;
       ret = left + 1;
   }
   if (node->right && node->val == node->right->val) {
       combinePath += right + 1;
       ret = max(ret, right + 1);      // either left OR right as contributor
   }
   ans = max(ans, combinePath);
```

```
    return ret;
}

int longestUnivaluePath(TreeNode* root) {
    int ans = 0;
    helper(root, ans);
    return ans;
}
```

Ref:

688. Knight probability in chessboard

A chess knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.



Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.

The knight continues moving until it has made exactly `K` moves or has moved off the chessboard. Return the probability that the knight remains on the board after it has stopped moving.

Example:

```
Input: 3, 2, 0, 0, Output: 0.0625
Explanation: There are two moves (to (1,2), (2,1)) that will keep the knight on the board.
From each of those positions, there are also two moves that will keep the knight on the board.
The total probability the knight stays on the board is 0.0625.
```

```
// soln-1: dynamic programming
// let f(x, y, step) be the probability on chessboard, it is combined from 8 directions:
// f(x, y, step) = sum{ f(x + dx, y + dy, step - 1) / 8.0f }, where (dx, dy) = {(2, 1), (2, -1), (-2, 1), (-2, -1), ...}
//
// space could be in O(N^2)
double knightProbability(int N, int K, int r, int c) {
    const vector<pair<int, int>> dirs{{2, 1}, {2, -1}, {-2, 1}, {-2, -1}, {1, 2}, {-1, 2}, {1, -2}, {-1, -2}};
    vector<vector<vector<double>>> dp(K + 1, vector<vector<double>>(N, vector<double>(N)));

    dp[0][r][c] = 1.0f;             // 0-move yet
    for (int k = 1; k <= K; ++k) {
        for (int x = 0; x < N; ++x) {
            for (int y = 0; y < N; ++y) {
                for (auto& d : dirs) {
                    int nx = x + d.first, ny = y + d.second;
                    if (nx >= 0 && nx < N && ny >= 0 && ny < N) dp[k][nx][ny] += dp[k - 1][x][y] / 8.0f;
                }
            }
        }
    }

    // sum up all the probability from the chessboard
    double ans = 0.0f;
    for (auto& row : dp[K]) {
        for (auto& p : row) ans += p;
    }
    return ans;
```

```
}
```

Ref: #576

690. Employee Importance

You are given a data structure of employee information, which includes the employee's unique id, his importance value and his direct subordinates' id.

For example, employee 1 is the leader of employee 2, and employee 2 is the leader of employee 3. They have importance value 15, 10 and 5, respectively. Then employee 1 has a data structure like [1, 15, [2]], and employee 2 has [2, 10, [3]], and employee 3 has [3, 5, []]. Note that although employee 3 is also a subordinate of employee 1, the relationship is not direct.

Now given the employee information of a company, and an employee id, you need to return the total importance value of this employee and all his subordinates.

Example 1:

```
Input: [[1, 5, [2, 3]], [2, 3, []], [3, 3, []]], 1
Output: 11
Explanation:
Employee 1 has importance value 5, and he has two direct subordinates: employee 2 and employee 3.
They both have importance value 3. So the total importance value of employee 1 is 5 + 3 + 3 = 11.
```

```
int helper(unordered_map<int, Employee*>& m, int id) {
   int ans = m[id]->importance;
   for (auto i : m[id]->subordinates) ans += helper(m, i);
   return ans;
}
int getImportance(vector<Employee*> employees, int id) {
   unordered_map<int, Employee*> m;
   for (auto e : employees) m[e->id] = e;
   return helper(m, id);
}
```

Ref:

693. Binary number with alternating bits

Given a positive integer, check whether it has alternating bits: namely, if two adjacent bits will always have different values.

```
bool hasAlternatingBits(int n) {
   n ^= (n >> 1);          // all 1s now for the right part
   return (n & (n + 1)) == 0;
}
```

Ref:

694/711. Number of distinct islands/II (review)

Given a non-empty 2D array grid of 0's and 1's, an island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Count the number of distinct islands. An island is considered to be the same as another if and only if one island can be translated (and not rotated or reflected) to equal the other.

Example 1:	Example 2:
11000 11000 00011 00011 Given the above grid map, return 1.	11011 10000 00001 11011 Given the above grid map, return 3.

711 - An island is considered to be the same as another if they have the same shape, or have the same shape after rotation (90, 180, or 270 degrees only) or reflection (left/right direction or up/down direction).

```
// 694/711 - soln-1: dfs
// the key to this question is how to identify the two island are same or not.
// 1. keep RELATIVE position works.
// 2. if asking identify rotation and reflection, we need to normalize the positions of island.
//    1) when find an island, keep all 8 forms (rotation and reflection)
//       rotation - (x, y) -> (y, -x) -> (-x, -y) -> (-y, x)
//       reflection - (-x, y)  <== left/right
//                    (x, -y)  <== up/down
//                    (y, x)   <== rotate 90-degree then upside-down
//                    (-y, -x) <== rotate 270-degree then upside-down
//    2) sort 8 forms and use min or max as representation of island
vector<pair<int, int>> normalize(vector<pair<int, int>>& island) {
   vector<vector<pair<int, int>>> ans(8);
   for (auto& x : island) {                         // push all 8 shapes
       ans[0].push_back(x);
       ans[1].push_back({x.second, -x.first});
       ans[2].push_back({-x.first, -x.second});
       ans[3].push_back({-x.second, x.first});
       ans[4].push_back({-x.first, x.second});
       ans[5].push_back({x.first, -x.second});
       ans[6].push_back({x.second, x.first});
       ans[7].push_back({-x.first, -x.second});
   }
   for (auto& i : ans) {
       sort(i.begin(), i.end());
       for (int j = 0; j < i.size(); ++j) {         // normalize the shape
           i[j] = {i[j].first - i[0].first, i[j].second - i[0].second};
       }
   }
   sort(ans.begin(), ans.end());
   return ans.front();
}

void dfs(vector<vector<int>>& m, int x0, int y0, int x, int y, vector<pair<int, int>>& island) {
   const vector<pair<int, int>> dirs {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
   island.push_back({x - x0, y - y0}), m[x][y] = -1;
   for (const auto& d : dirs) {
       int nx = x + d.first, ny = y + d.second;
       if (nx < 0 || nx >= m[x].size() || ny < 0 || ny >= m.size() || 1 != m[nx][ny]) continue;
       dfs(m, x0, y0, nx, ny, island);
   }
}

int numDistinctIslands(vector<vector<int>>& grid) {
   set<vector<pair<int, int>>> ans;                 // unordered_set is better, but need to customized hash
function
   for (int i = 0; i < grid.size(); ++i) {
       for (int j = 0; j < grid[i].size(); ++j) {
           if (1 == grid[i][j]) {
               vector<pair<int, int>> island;
               dfs(grid, i, j, i, j, island);
               ans.insert(normalize(island));      // normailize is required only for follow-up
```

```
            }
        }
    }
    return ans.size();
}
```

Ref: #200

695. Max area of Island

trivial DFS question

Ref:

696. Count binary substrings (review)

Give a string s, count the number of non-empty (contiguous) substrings that have the same number of 0's and 1's, and all the 0's and all the 1's in these substrings are grouped consecutively.

Substrings that occur multiple times are counted the number of times they occur.

Example 1:

```
Input: "00110011", Output: 6
Explanation: There are 6 substrings that have equal number of consecutive 1's and 0's: "0011", "01", "1100", "10", "0011", and "01".

Notice that some of these substrings repeat and are counted the number of times they occur.

Also, "00110011" is not a valid substring because all the 0's (and 1's) are not grouped together.
```

```
// 696 - soln-1: group consecutive 0/1s
// 1, group consecutive 0/1s. "0110001111" => [1, 2, 3, 4]
// 2, contributions happen only in between shift (0->1 or 1->0)
//    and the number of contribution is min. between the two group.
//    [1, 2, 3, 4] => min(1, 2) + min(2, 3) + min(3, 4)
int countBinarySubstrings(string s) {
   int pre = 0, cur = 1, ans = 0;
   for (int i = 1; i < s.length(); ++i) {
       if (s[i - 1] == s[i]) {
           cur++;
       } else {
           ans += min(pre, cur);
           pre = cur, cur = 1;
       }
   }
   return ans + min(pre, cur);
}
```

```
# Python soln
def countBinarySubstrings(self, s):
   s = map(len, s.replace('10', '1 0').replace('01', '0 1').split())
   return sum(min(a, b) for a, b in zip(s, s[1:]))
```

Ref:

697. TODO

Solution:

Ref:

699. TODO

Solution:

Ref:

702. TODO

Solution:

Ref:

703. Kth largest element in a stream

Design a class to find the **k**th largest element in a stream. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Your `KthLargest` class will have a constructor which accepts an integer `k` and an integer array `nums`, which contains initial elements from the stream. For each call to the method `KthLargest.add`, return the element representing the kth largest element in the stream.

```
class KthLargest {
   int _k;
   priority_queue<int, vector<int>, std::greater<int>> _minH;
public:
   KthLargest(int k, vector<int> nums) {
       _k = k;
       priority_queue<int> maxH;
       for (int x : nums) maxH.push(x);
       while (k > 0 && !maxH.empty()) _minH.push(maxH.top()), maxH.pop(), --k;
   }

   int add(int val) {
       if (_minH.empty() || _minH.size() < _k) _minH.push(val);
       else if (_minH.top() < val)  _minH.pop(), _minH.push(val);

       return _minH.top();
   }
};
```

Ref:

704. Binary search

trivial question

Ref:

711. TODO

Solution:

Ref:

717. 1-bit / 2-bit chars

We have two special characters. The first character can be represented by one bit `0`. The second character can be represented by two bits (`10` or `11`).

Now given a string represented by several bits. Return whether the last character must be a one-bit character or not. The given string will always end with a zero.

```
// soln-1: dynamic programming (same as climbing stairs)
// if cur == 1, move 2 steps (10/11)
// else move 1 step
// since the last step is reserved, we move n - 1 steps in total.
// finally steps should reach at 2nd last idx.
bool isOneBitCharacter(vector<int>& bits) {
   int steps = 0;
   while (steps < bits.size() - 1) {
       bits[steps] ? steps += 2: steps += 1;
   }
   return steps == bits.size() - 1;
}
```

Ref:

718. Max length of repeated subarray

Given two integer arrays `A` and `B`, return the maximum length of an subarray that appears in both arrays.

Example 1:

```
Input: A: [1,2,3,2,1], B: [3,2,1,4,7]
Output: 3
Explanation:  The repeated subarray with maximum length is [3, 2, 1].
```

```
// soln-1: dynamic programming (longest common substring)
// let dp(i, j) be the longest subarray,
// dp(i, j) = (Ai == Bi) ? dp(i-1, j-1) + 1 ? 0                          <--- longest common substring
// dp(i, j) = (Ai == Bi) ? dp(i-1, j-1) + 1 ? max{dp(i-1, j), dp(i, j-1)}  <--- longest common subsequence
// space complexity could be done in O(n) not O(nm) because of the dependency
int findLength(vector<int>& A, vector<int>& B) {
   int m = A.size(), n = B.size(), ans = 0;
   vector<vector<int>> dp(m + 1, vector<int>(n + 1));

   for (int i = 1; i <= m; ++i) {
       for (int j = 1; j <= n; ++j) {
           if (A[i - 1] == B[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;
           ans = max(ans, dp[i][j]);
       }
```

```
    }
    return ans;
}
```

Ref: #5

719/786. Kth smallest pair distance / Kth smallest prime fraction (review)

719 - Given an integer array, return the k-th smallest distance among all the pairs. The distance of a pair (A, B) is defined as the absolute difference between A and B.

786 - A sorted list A contains 1, plus some number of primes. Then, for every p < q in the list, we consider the fraction p/q.
What is the K-th smallest fraction considered? Return your answer as an array of ints, where answer[0] = p and answer[1] = q.

```
// 719 - soln-1: heap (O(nlgn) but TLE)
int smallestDistancePair(vector<int>& nums, int k) {
    sort(nums.begin(), nums.end());
    auto cmp = [&nums](pair<int, int>& a, pair<int, int>& b) {
        return (nums[a.second] - nums[a.first]) > (nums[b.second] - nums[b.first]);
    };
    priority_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp)> minh(cmp);
    for (int i = 0; i < nums.size() - 1; ++i) minh.push({i, i + 1});
    while (--k) {
        auto& t = minh.top();
        if (t.second + 1 < nums.size())minh.push({t.first, t.second + 1});
        minh.pop();
    }
    return nums[minh.top().second] - nums[minh.top().first];
}
```

```
// 719 - soln-2: binary search (trial and error)
// 1. sort the input, Kth smallest pair distance is within [0, d], d = a.back() - a[0]
// 2. now given distance-d, can we count how many pairs within this distance in O(n)?
// 3. if step-2 is yes, then we can solve this question in O(nlgd + nlgn)
//
// trial and error (guess and check) similars:
// 004 - median of two sorted arrays
// 278 - find dups number
// 410 - split array largest sum (dynamic programing, binary search)
// 378 - kth smallest in sorted matrix (priorityQ, binary search, zigzag search)
// 668 - kth smallest in multiple table (priorityQ, binary search, zigzag search)
// 644 - maximum average subarray II (challenge)
// 774 - min/max distance to gas station
// 786 - kth smallest prime fraction
int smallestDistancePair(vector<int>& nums, int k) {
    sort(nums.begin(), nums.end());
    int l = 0, h = nums.back() - nums[0];
    while (l < h) {
        int m = l + (h - l) / 2, count = 0;
        // within distance m, how many pairs do we have?
        // count it in O(n) as each number would be visited at most twice - tricky part
        for (int i = 0, j = 0; i < nums.size() - 1; ++i) {
            while (j < nums.size() && nums[j] <= nums[i] + m) ++j;  // faster than next line, cache-friendly?
            // j = int(upper_bound(nums.begin() + j, nums.end(), nums[i] + m) - nums.begin());
            count += j - i - 1;
        }
        count < k ? l = m + 1 : h = m;
    }
    return l;
```

```
}
```

```
// 786 - soln-1: value-range based binary search (heap would TLE)
// Another "trial and error" type of question - fix value range and check.
// 1. value range: [a[0]/a[n-1], a[n-2]/a[n-1]]
// 2. check: for each number, use binary search to detect how many peers are within range.

// binary search count in O(nlgn) time
tuple<int, int, int> countSmaller(vector<int>& A, double target) {
   int ans = 0, p = A[0], q = A.back();
   for (int i = A.size() - 1; i > 0; --i) {
       int lo = 0, hi = i;
       while (lo < hi) {
           int m = lo + (hi - lo) / 2;
           (double)A[m] / A[i] < target ? lo = m + 1 : hi = m; // fraction of A[i] not A[lo/hi]
       }
       if (lo && p * A[i] < A[lo - 1] * q) p = A[lo - 1], q = A[i];

       ans += lo;
   }
   return {ans, p, q};
}

// zigzag count in O(n) time
tuple<int, int, int> countSmaller2(vector<int>& A, double target) {
    int ans = 0, p = A[0], q = A.back();
    for (int i = 0, j = 1; i < A.size(); ++i) {
        while (j < A.size() && A[i] > target * A[j]) ++j;

        int cnt = A.size() - j;
        if (cnt && (A[i] * q > p * A[j])) p = A[i], q = A[j];
        ans += cnt;
    }
    return {ans, p, q};
}

vector<int> kthSmallestPrimeFraction(vector<int>& A, int K) {
    double lo = .0f, hi = 1.0f;     // instead of finding hi/lo, use [0.0 .. 1.0]
    while (lo < hi) {
        double m = lo + (hi - lo) / 2;

        int count, p, q;
        tie(count, p, q) = countSmaller2(A, m);
        if (count == K) return vector<int>{p, q};
        count < K ? lo = m : hi = m;
    }
    return vector<int>{0, 0};
}
```

```
// 774 - min gas station distance (insert K more gas stations to min gas station distance)
// soln-1: value-range based binary search (trial and error)
// assuming a distance, try if we need to add gas stations
// NOTE: greedy cut max distance into half will not work.
bool feasible(vector<int>& stations, double dist, int k) {
   int need = 0;
   for (int i = 1; i < stations.size(); ++i) {
       need += (stations[i] - stations[i - 1]) / dist;
   }
   return need <= k;
}
double minmaxGasDist(vector<int>& stations, int K) {
   double left = 0, right = 1e8;
   while (right - left > 1e-6) {
       double mid = left + (right - left) / 2.0;
       feasible(stations, mid, K) ? right = mid : left = mid;
   }
```

```
    return left;
}
```

```
// 875 - eating bananas
// soln-1: value-range based binary search
bool eatable(vector<int>& piles, int amount, int H) {
    int need = 0;
    for (int i = 0; i < piles.size(); ++i) {
        need += piles[i] / amount;
        if (piles[i] % amount) ++need;
    }
    return need <= H;
}
int minEatingSpeed(vector<int>& piles, int H) {
    int left = 1, right = 100, ans = 0;
    while (left < right) {
        auto m = left + (right - left) / 2;
        eatable(piles, m, H) ? right = m : left = m + 1;
    }
    return left;
}
```

```
// 1011 - capacity to ship packages within d-days
// soln-1: value-range based binary search (trial and error)
bool ship(vector<int>& weights, int D, int capacity) {
    int d = 0, cur = 0;
    for (auto w : weights) {
        (cur + w > capacity) ? ++d, cur = w : cur += w;
        if (cur > capacity) return false;
    }
    if (cur) ++d;   // need one more ship
    return d <= D;
}
int shipWithinDays(vector<int>& weights, int D) {
    int lo = 1, hi = 500 * 50000;
    while (lo < hi) {
        int m = lo + (hi - lo) / 2;
        ship(weights, D, m) ? hi = m : lo = m + 1;
    }
    return lo;
}
```

Ref:

723/755. Candy crush/Pour water

723 - need to restore the board to a *stable state* by crushing candies according to the following rules:

1. If three or more candies of the same type are adjacent vertically or horizontally, "crush" them all at the same time - these positions become empty.
2. After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies will drop until they hit a candy or bottom at the same time. (No new candies will drop outside the top boundary.)
3. After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.
4. If there does not exist more candies that can be crushed (ie. the board is *stable*), then return the current board.

755 - flows according to the following rules:

- If the droplet would eventually fall by moving left, then move left.
- Otherwise, if the droplet would eventually fall by moving right, then move right.
- Otherwise, rise at its current position.

```
// 723 - soln-1: brute-force
// 1. for each position, extend horizontally and vertically at most 3 position.
```

```
// 2. if the gap between extended position >= 3 horizontally or vertically, keep it.
// 3. mark crushed candies, then drop for each column (two pointers).
vector<vector<int>> candyCrush(vector<vector<int>>& board) {
   for (int m = board.size(), n = board[0].size(); nullptr; nullptr) {
       vector<pair<int, int>> del;
       for (int i = 0; i < m; ++i) {
           for (int j = 0; j < n; ++j) {
               if (board[i][j]) {
                   int x0 = i, x1 = i, y0 = j, y1 = j;
                   while (x0 >= 0 && x0 > i - 3 && board[x0][j] == board[i][j]) --x0;  // extend up at most 3
                   while (x1 < m  && x1 < i + 3 && board[x1][j] == board[i][j]) ++x1;
                   while (y0 >= 0 && y0 > j - 3 && board[i][y0] == board[i][j]) --y0;  // extend left at most 3
                   while (y1 < n  && y1 < j + 3 && board[i][y1] == board[i][j]) ++y1;
                   if (x1 - x0 > 3 || y1 - y0 > 3) del.push_back({i, j});
               }
           }
       }
       if (del.empty()) break;
       for (auto a : del) board[a.first][a.second] = 0;        // mark as crushed
       for (int j = 0; j < n; ++j) {
           for (int i = m - 1, zero = m - 1; i >= 0; --i) {    // drop candy for each column
               if (board[i][j]) swap(board[zero--][j], board[i][j]);
           }
       }
   }
   return board;
}
```

```
// 755 - soln-1: brute-force
// a montone stack might be helpful, but depends on input.
vector<int> pourWater(vector<int>& heights, int V, int K) {
   for (int i = 0; i < V; ++i) {
       int l = K, r = K, n = heights.size();
       while (l > 0 && heights[l] >= heights[l - 1]) --l;
       while (l < K && heights[l] == heights[l + 1]) ++l;
       while (r < n - 1 && heights[r] >= heights[r + 1]) ++r;
       while (r > K && heights[r] == heights[r - 1]) --r;
       (heights[l] < heights[K]) ? ++heights[l] : ++heights[r];
   }
   return heights;
}
```

Ref:

724. Find pivot index

Given an array of integers `nums`, write a method that returns the "pivot" index of this array.

We define the pivot index as the index where the sum of the numbers to the left of the index is equal to the sum of the numbers to the right of the index.

If no such index exists, we should return -1. If there are multiple pivot indexes, you should return the left-most pivot index.

```
// soln-1: find the sum which is half of total minus current value
int pivotIndex(vector<int>& nums) {
   int total = accumulate(nums.begin(), nums.end(), 0);
   for (int i = 0, cur = 0; i < nums.size(); ++i) {
       if (cur * 2 == total - nums[i]) return i;
       cur += nums[i];
   }
   return -1;
}
```

Ref:

726. TODO

Solution:

Ref:

728. Self dividing numbers

A *self-dividing number* is a number that is divisible by every digit it contains.

For example, 128 is a self-dividing number because `128 % 1 == 0`, `128 % 2 == 0`, and `128 % 8 == 0`.

Also, a self-dividing number is not allowed to contain the digit zero.

Given a lower and upper number bound, output a list of every possible self dividing number, including the bounds if possible.

Example 1:

```
Input: left = 1, right = 22, Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 22]
```

```
// soln-1: brute-force
vector<int> selfDividingNumbers(int left, int right) {
   vector<int> ans;
   for (int i = left; i <= right; ++i) {
       int j = i;
       while (j) {
           int k = j % 10;
           if (!k || i % k) break;
           j /= 10;
       }
       if (!j) ans.push_back(i);
   }
   return ans;
}
```

Ref:

729/731/732. My calendar I/II/III

929 - Implement a `MyCalendar` class to store your events. A new event can be added if adding the event will not cause a double booking.
731 - Implement a `MyCalendarTwo` class to store your events. A new event can be added if adding the event will not cause a triple booking.
732 - Implement a `MyCalendarThree` class to store your events. A new event can **always** be added.

252/253 - meeting room

```
// 729 - hashset
class MyCalendar {
   set<pair<int, int>> _booked;
public:
   // soln-1: segment in hashset
   // brute force check overlapping: max(s1.start, s2.start) < min(s1.end, s2.end)
```

```
    bool book(int start, int end) {
        auto r = _booked.lower_bound({start, end});                // everything equal or bigger than me
        if (r != _booked.end() && r->first < end) return false;
        if (r != _booked.begin() && (--r)->second > start) return false;
        _booked.insert({start, end});
        return true;
    }
};
```

```
// 731 - brute-force
class MyCalendarTwo {
    vector<pair<int, int>> _booked;
public:
    // soln-1: brute-force with trick
    // overlap only twice means the overlapped interval can not be booked again.
    bool book(int start, int end) {
        MyCalendar overlapped;

        for (auto& p : _booked) {
            pair<int, int> overlap{max(p.first, start), min(p.second, end)};
            if (overlap.first > overlap.second) continue;
            if (!overlapped.book(overlap.first, overlap.second)) return false;
        }
        _booked.push_back({start, end});
        return true;
    }
};
```

```
// 732 - return the largest intersection count
// soln-1: greedy + priority queue
// 1. sort based on start time
// 2. if cur start time > some finish time, means no intersection
// essentially it's same as 253 - meeting room II
class MyCalendarThree {
    multiset<pair<int, int>> _booked;
public:
    int book(int start, int end) {
        _booked.insert({start, end});

        int ans = 0;
        priority_queue<int, vector<int>, greater<int>> h;
        for (auto& p : _booked) {
            if (!h.empty() && h.top() <= p.first) h.pop();
            h.push(p.second);
            ans = max(ans, (int)h.size());
        }
        return ans;
    }

    // soln-2: smarter way
    int book2(int start, int end) {
        static map<int, int> time_line;
        time_line[start]++, time_line[end]--;
        int ans = 0, ongoing = 0;
        for (auto& p : time_line) {
            ans = max(ans, ongoing += p.second);
        }
        return ans;
    }
};
```

Ref: 252/253

730. Count diff. palindromic subsequences

Given a string S, find the number of different non-empty palindromic subsequences in S, and return that number modulo `10^9 + 7`.

Example 1:

```
Input: S = 'bccb', Output: 6
Explanation:
The 6 different non-empty palindromic subsequences are 'b', 'c', 'bb', 'cc', 'bcb', 'bccb'.
Note that 'bcb' is counted only once, even though it occurs twice.
```

Example 2:

```
Input: S = 'abcdabcdabcdabcdabcdabcdabcdabcddcbadcbadcbadcbadcbadcbadcbadcba', Output: 104860361
Explanation:
There are 3104860382 different non-empty palindromic subsequences, which is 104860361 modulo 10^9 + 7.
```

Solution:

Ref: #5

736. TODO

Solution:

Ref:

738. Monotone increasing digits

Given a non-negative integer `N`, find the largest number that is less than or equal to `N` with monotone increasing digits.

(Recall that an integer has *monotone increasing digits* if and only if each pair of adjacent digits `x` and `y` satisfy `x <= y`.)

Example 1: `Input: N = 332, Output: 299`

```
// soln-1: brute force
// monotone increasing digits iff each pair of adjacent digits x and y satisfy x <= y
int monotoneIncreasingDigits(int N) {
   string str = to_string(N);

   int pos = str.length();
   for (int i = str.length() - 1; i > 0; --i) {
       if (str[i] < str[i - 1]) {
           pos = i;
           --str[i - 1];
       }
   }
   for (int i = pos; i < str.length(); ++i) str[i] = '9';

   return stoi(str);
}
```

Ref: #402

740. Delete and earn (review)

Given an array `nums` of integers, you can perform operations on the array.

In each operation, you pick any `nums[i]` and delete it to earn `nums[i]` points. After, you must delete **every** element equal to `nums[i] - 1` or `nums[i] + 1`.

You start with 0 points. Return the maximum number of points you can earn by applying such operations.

Example 2:

```
Input: nums = [2, 2, 3, 3, 3, 4]
Output: 9
Explanation:
Delete 3 to earn 3 points, deleting both 2's and the 4. Then, delete 3 again to earn 3 points, and 3
again to earn 3 points. 9 total points are earned.
```

Note: The length of nums is at most 20000. Each element nums[i] is an integer in the range [1, 10000].

#337 - house robber III

```
// soln-1: dynamic programming
// Great bucket sort + house rob idea.
// 1. accumulate all numbers (value) into each bucket which is 10000 at most
// 2. at position i, we could either take or skip
//      - take[i] = a[i] + skip[i - 1]
//      - skip[i] = max(take[i - 1], skip[i - 1])
//      - max value we can get at i is max(take[i], skip[i])
int deleteAndEarn(vector<int>& nums) {
   int max_buckets = 10001;
   vector<int> bucket(max_buckets, 0);
   for (int x : nums) bucket[x] += x;

   int take = 0, skip = 0;     // the value of take/skip at pos i-1
   for (int i = 1; i < max_buckets; ++i) {
       int takei = skip + bucket[i];
       int skipi = max(take, skip);
       take = takei, skip = skipi;
   }
   return max(take, skip);
}
```

Ref: #312 #337

741. Cherry pickup

In a N x N `grid` representing a field of cherries, each cell is one of three possible integers.

- 0 means the cell is empty, so you can pass through;
- 1 means the cell contains a cherry, that you can pick up and pass through;
- -1 means the cell contains a thorn that blocks your way.

Your task is to collect maximum number of cherries possible by following the rules below:

- Starting at the position (0, 0) and reaching (N-1, N-1) by moving right or down through valid path cells (cells with value 0 or 1);
- After reaching (N-1, N-1), returning to (0, 0) by moving left or up through valid path cells;
- When passing through a path cell containing a cherry, you pick it up and the cell becomes an empty cell (0);
- If there is no valid path between (0, 0) and (N-1, N-1), then no cherries can be collected.

Example 1:

```
Input: grid =
[[0, 1, -1],
 [1, 0, -1],
 [1, 1,  1]]
```

```
Output: 5
Explanation:
The player started at (0, 0) and went down, down, right right to reach (2, 2).
4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-
1],[0,0,0]].
Then, the player went left, up, up, left to return home, picking up one more cherry.
The total number of cherries picked up is 5, and this is the maximum possible.
```

```cpp
// soln-1: dynamic programming
// If we apply greedy from top-left to bottom-right, then return, it will not work
// because local optimum does not necessary equal to global optimum.
//
// 2-leg DP - sending out 2 ppl at same time to pick up
// T(i, j, p, q) = max{T(i-1, j, p-1, q), T(i, j-1, p-1, q),
//                     T(i-1, j, p, q-1), T(i, j-1, p, q-1)} + a[i][j] + a[p][q]
// as i+j = p+q = n-steps, we could reduce 1-dimension
// T(n, i, p) = max{T(n-1, i-1, p-1), T(n-1, i, p-1), T(n-1, i-1, p), T(n-1, i, p)} + a[i][n-i] + a[p][n-p]
// we have 3 variables to enumerate, so time complexity: O(n^3)
// space could be O(n^2) as we only depend on previous(n-1) step
int cherryPickup(vector<vector<int>>& grid) {
   int n = grid[0].size();
   vector<vector<int>> dp(n, vector<int>(n, -1));

   dp[0][0] = grid[0][0];
   for (int step = 1; step < 2 * n - 1; ++step) {
       for (int i = min(n - 1, step); i >= 0; --i) {
           for (int p = min(n - 1, step); p >= 0; --p) {
               int j = step - i, q = step - p;
               if (j < 0 || j >= n || q < 0 || q >= n || grid[i][j] < 0 || grid[p][q] < 0) {
                   dp[i][p] = -1;      // dp keeps the combined result, both need to accessible
                   continue;
               }

               if (i > 0) dp[i][p] = max(dp[i][p], dp[i - 1][p]);
               if (p > 0) dp[i][p] = max(dp[i][p], dp[i][p - 1]);
               if (i > 0 && p > 0) dp[i][p] = max(dp[i][p], dp[i - 1][p - 1]);

               if (dp[i][p] >= 0) dp[i][p] += grid[i][j] + (i != p ? grid[p][q] : 0);
           }
       }
   }
   return max(0, dp[n - 1][n - 1]);
}
```

Ref: #62 #474

742/863. Nearest leaf from target node/Distance K in binary tree

742 - Given a binary tree where every node has a unique value, and a target key k, find the value of the nearest leaf node to target k in the tree.

863 - We are given a binary tree, a `target` node, and an integer value `K`. Return a list of the values of all nodes that have a distance `K` from the `target` node. The answer can be returned in any order.

```
// 742 - soln-1: dfs
// essentially its same as #863, the steps are almost same too.
// 1. find the path to target.
// 2. for each ancestor of target, find the nearest leaf since we know:
```

```
//     a) the distance from target to ancestor and
//     b) which substree we should search for
// 3. other soln could be rebuilding the binary tree as graph, then bfs.
```

```
// 863 - soln-1: dfs
// find K distance children from given node
void findChild(TreeNode* node, int K, vector<int>& ans) {
   if (nullptr == node || K < 0) return;
   if (0 == K) {
       ans.push_back(node->val);
   } else {
       findChild(node->left, K - 1, ans), findChild(node->right, K - 1, ans);
   }
}

// dfs to find path from root to target
bool findPath(TreeNode* n, TreeNode* target, vector<TreeNode*>& ans) {
   if (nullptr == n) return false;

   ans.push_back(n);
   if (n == target || findPath(n->left, target, ans) || findPath(n->right, target, ans)) return true;
   ans.pop_back();
   return false;
}
vector<int> distanceK(TreeNode* root, TreeNode* target, int K) {
   vector<TreeNode*> path;
   findPath(root, target, path);

   vector<int> ans;
   findChild(target, K, ans);                              // k distance from target
   for (int i = path.size() - 2; i >= 0; --i, --K) {   // k' distance from my ancestors
       if (K > 1)  findChild(path[i]->left == path[i + 1] ? path[i]->right : path[i]->left, K - 2, ans);
       else findChild(path[i], K - 1, ans);
   }
   return ans;
}
```

Ref:

744. Find smallest letter greater than target

Given a list of sorted characters `letters` containing only lowercase letters, and given a target letter `target`, find the smallest element in the list that is larger than the given target.

Letters also wrap around. For example, if the target is `target = 'z'` and `letters = ['a', 'b']`, the answer is `'a'`.

```
// soln-1: binary search (be careful about corner cases)
char nextGreatestLetter(vector<char>& letters, char target) {

   int low = 0, hi = letters.size() - 1, len = letters.size();
   while (low + 1 < hi) {
       int m = low + (hi - low) / 2;
       letters[m] > target ? hi = m : low = m;
   }

   if (letters[low] > target) return letters[low];
   if (letters[hi] > target) return letters[hi];

   while (low + 1 < len && letters[low + 1] <= target) ++low;
   return letters[(low + 1) % len];
}
```

Ref:

748. Shortest completing word

```
// 748 - soln-1: easy hashmap
string shortestCompletingWord(string licensePlate, vector<string>& words) {
   string ans;
   for (auto& w : words) {
       vector<int> mp(26, 0);
       for (auto l : w) mp[tolower(l) - 'a']++;

       bool match = true;
       for (int i = 0; match && i < licensePlate.length(); ++i) {
           if (isalpha(licensePlate[i]) && --mp[tolower(licensePlate[i]) - 'a'] < 0)) match = false;
       }
       if (match) {
           if (ans.empty() || w.length() < ans.length()) ans = w;
       }
   }
   return ans;
}
```

Ref:

750. TODO

Solution:

Ref:

751. TODO

Solution:

Ref:

752. Open the lock

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: `'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'`. The wheels can rotate freely and wrap around: for example we can turn `'9'` to be `'0'`, or `'0'` to be `'9'`. Each move consists of turning one wheel one slot.

The lock initially starts at `'0000'`, a string representing the state of the 4 wheels.

You are given a list of `deadends` dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a `target` representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

Example 1:

```
Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
Output: 6
Explanation:
A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" ->
```

```
"0202".
Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,
because the wheels of the lock become stuck after the display becomes the dead end "0102".
```

Solution:
BFS: from src to dst or the opposite, or bi-directional search.

```
union _lock_number {
   int x;
   struct {
       unsigned int a:4, b:4, c:4, d:4;
   }_x;
};

int str2LockNum(string s) {
   _lock_number x; x.x = 0;
   x._x.a = s[0] - '0'; x._x.b = s[1] - '0';
   x._x.c = s[2] - '0'; x._x.d = s[3] - '0';
   return x.x;
}

vector<int> getNextMove(int x) {
   vector<int> ans;
   { _lock_number i; i.x = x; i._x.a = (i._x.a + 1) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.b = (i._x.b + 1) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.c = (i._x.c + 1) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.d = (i._x.d + 1) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.a = (i._x.a + 9) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.b = (i._x.b + 9) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.c = (i._x.c + 9) % 10; ans.push_back(i.x); }
   { _lock_number i; i.x = x; i._x.d = (i._x.d + 9) % 10; ans.push_back(i.x); }
   return ans;
}

int helper(unordered_set<int>& dead, int target) {
   unordered_set<int> visited;
   queue<int> q; q.push(target);
   int ans = 0;
   while (!q.empty()) {
       ++ans;
       for (int n = q.size(); n > 0; --n) {
           int x = q.front(); q.pop();

           for (int k : getNextMove(x)) {
               if (visited.find(k) != visited.end() || dead.find(k) != dead.end()) continue;
               if (0 == k) return ans;
               q.push(k);  visited.insert(k);
           }
       }
   }
   return -1;
}

int openLock(vector<string>& deadends, string target) {
   unordered_set<int> dead;
   for (string& s : deadends) dead.insert(str2LockNum(s));
   return helper(dead, str2LockNum(target));
}
```

Ref: #675 #743

753. TODO

Solution:

Ref:

754. Reach a number

You are standing at position `0` on an infinite number line. There is a goal at position `target`. On each move, you can either go left or right. During the *n*-th move (starting from 1), you take *n* steps.

Return the minimum number of steps required to reach the destination.

```
// 754 - reach a number
// soln-1: greedy
// the least steps would be all the steps if we add all of them.
// if sum(steps) > target, let x = target - sum(steps)
//     - if x is even, that means we can find a step = x/2 to jump left side
//     - if x is odd,
//         * if next step is odd, then 1 more step is ok
//         * else need 2 more steps (guarantee odd number) to make x even
// ex.  target | steps
//       2     | 1, 2, x = 3 - 2 = 1, next step is 3 (odd) will make x be even
//       12    | 1, 2, 3, 4, 5, x = 15 - 12 = 3, need 2 more steps 6, 7 to make x be even
//       13    | 1, 2, 3, 4, 5, x = 15 - 13 = 2, (2/2 = 1 can be picked up to jump left)
//       14    | 1, 2, 3, 4, 5, x = 1, need 6,7
int reachNumber(int target) {
   int total = 0, step = 0;
   while (total < target) step++, total += step;
   if (total == target) return step;

   if ((target - total) % 2 == 0) return step;
   return step % 2 == 0 ? step + 1 : step + 2;
}
```

Ref:

757. TODO

Solution:

Ref:

759. TODO

Solution:

Ref:

761. TODO

Solution:

Ref:

762. Prime number of set bits in binary representation

Given two integers `L` and `R`, find the count of numbers in the range `[L, R]` (inclusive) having a prime number of set bits in their binary representation.

Example 1:

```
Input: L = 6, R = 10, Output: 4
Explanation:
6 -> 110 (2 set bits, 2 is prime)
7 -> 111 (3 set bits, 3 is prime)
9 -> 1001 (2 set bits , 2 is prime)
10->1010 (2 set bits , 2 is prime)
```

```
int countPrimeSetBits(int L, int R) {
   // we have those primes under 32.
   unordered_set<int> primes {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
   int ans = 0;
   for (int i = L; i <= R; ++i) {
       // count bits
       int bits = 0;
       for (int n = i; n; n &= n-1) ++bits;

       if (primes.find(bits) != primes.end()) ++ans;
   }
   return ans;
}
```

Ref:

765. TODO

Solution:

Ref:

768/769. Max chunks to make sorted/II (review)

768 - Given an array `arr` that is a permutation of `[0, 1, ..., arr.length - 1]`, we split the array into some number of "chunks" (partitions), and individually sort each chunk. After concatenating them, the result equals the sorted array.

What is the most number of chunks we could have made?

769 - *the given array are not necessarily distinct, the input array could be up to length `2000`, and the elements could be up to `10**8`.*

```
// 769 - soln-1: array
// a[i] indicates current chunk must reach that indx, given arr[i] within 0~n-1.
// for example, [4, 3, 2, 1, 0], for i = 0, to make a[0] = 4 properly positioned,
// current chunk has to extend until index-4.
int maxChunksToSorted(vector<int>& arr) {
   int ans = 0, maxInChunk = -1;
   for (int i = 0; i < arr.size(); ++i) {
       if (i > maxInChunk) ans++;
       maxInChunk = max(maxInChunk, arr[i]);
   }

   return ans;
}
```

```
// 768 - soln-1: array
// make another sorted array, then compare:
// 2 1  3  4 3  5  <- input
// 1 2  3  3 4  5  <- sorted
// the trick is to compare the sum of sorted and original array
int maxChunksToSortedII(vector<int>& arr) {
   vector<int> sorted(arr);
   sort(sorted.begin(), sorted.end());

   int ans = 0;
   for (int i = 0, s1 = 0, s2 = 0; i < arr.size(); ++i) {
       s1 += arr[i], s2 += arr[i];
       if (s1 == s2) ++ans;
   }
   return ans;
}

// 768 - soln-2: monotonically stack
// 1, cur >= max, then push into stack. for example, [1, 1, 1]
// 2, if cur < max, we need to find a point that is safe to start for cur, ex. [0, 1, 1, 0, 1]
int maxChunksToSortedII(vector<int>& arr) {
   stack<int> stk;
   for (int i = 0, mx = INT_MIN; i < arr.size(); ++i) {
       if (stk.empty() || arr[i] >= mx) {
           stk.push(arr[i]);
       } else {
           while (!stk.empty() && stk.top() > arr[i]) stk.pop();
           stk.push(mx);
       }
       mx = max(mx, arr[i]);
   }
   return stk.size();
}
```

Ref:

770. TODO

Solution:

Ref:

771. Jewels and stones

trivial hashmap.

Ref:

772. TODO

Solution:

Ref:

773. TODO

Solution:

Ref:

775. Global and local inversions

We have some permutation A of [0, 1, ..., N - 1], where N is the length of A. The number of (global) inversions is the number of i < j with 0 <= i < j < N and A[i] > A[j]. The number of local inversions is the number of i with 0 <= i < N and A[i] > A[i+1].

Return true if and only if the number of global inversions is equal to the number of local inversions.

```
// 775 - soln-1: by observation
bool isIdealPermutation(vector<int>& A) {
    int global = 0, local = 0;
    for (int i = 0; i < A.size(); ++i) {
        if (A[i] > i) global += A[i] - i;
        if (A[i] != i && i < A.size() - 1 && A[i] > A[i + 1]) local++;
    }
    return global == local;
}
```

Ref:

778/847/864/909. Shortest path visiting all nodes/Snake and ladder (review)

```
// 847 - shortest path visiting all nodes in connected/undirected graph
// soln-1: bfs/dynamic programming (different approach)
// 1. starting from each node, do bfs, and mark visited: cur-node|visited-mask
// 2. finish when we see every node is visited
int shortestPathLength(vector<vector<int>>& graph) {
   int N = graph.size(), target = (1 << N) - 1, steps = 0;
   if (N <= 1) return 0;

   queue<pair<int, int>> Q;    // <node, visited-mask>
   unordered_set<int> seen;
   for (int i = 0; i < N; ++i) Q.push({i, 1 << i}), seen.insert((i << 16) | (1 << i));

   while (!Q.empty()) {
       ++steps;
       for (int i = Q.size(); i > 0; --i) {
```

```
            auto n = Q.front(); Q.pop();
            for (auto nxt : graph[n.first]) {
                auto mask = n.second | (1 << nxt), visit = (nxt << 16) | mask;
                if (mask == target) return steps;
                if (seen.find(visit) != seen.end()) continue;
                Q.push({nxt, mask}), seen.insert(visit);
            }
        }
    }
    return steps;
}
```

```
// 864 - shortest path to get all keys
// soln-1: bfs/dynamic programming (847 - shortest path visiting all nodes)
// 1. from starting point, do bfs, mark visited state as: pos | keys-mask
// 2. return current steps as soon as all keys are found
int shortestPathAllKeys(vector<string>& grid) {
    int row = grid.size(), col = grid[0].size(), keys = 0, ans = 0;
    queue<pair<int, int>> Q;
    unordered_set<int> seen;
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            if ('@' == grid[i][j]) Q.push({i * col + j, 0}), seen.insert((i * col + j) << 16);
            else if ('a' <= grid[i][j] && grid[i][j] <= 'f') keys |= 1 << (grid[i][j] - 'a');
        }
    }

    while (!Q.empty()) {
        for (int k = Q.size(); k > 0; --k) {
            auto p = Q.front(); Q.pop();
            int x = p.first / col, y = p.first % col, key = p.second;
            if (key == keys) return ans;
            for (auto& dir : vector<pair<int, int>>{{-1, 0}, {1, 0}, {0, 1}, {0, -1}}) {
                int nx = x + dir.first, ny = y + dir.second;
                if (nx < 0 || nx >= row || ny < 0 || ny >= col || '#' == grid[nx][ny]) continue;

                int nk = key, s = ((nx * col + ny) << 16) | nk;
                if ('a' <= grid[nx][ny] && grid[nx][ny] <= 'f') {       // pickup the key
                    nk = key | (1 << (grid[nx][ny] - 'a')), s = ((nx * col + ny) << 16) | nk;
                } else if ('A' <= grid[nx][ny] && grid[nx][ny] <= 'F'){ // hit lock, check key
                    int lock = grid[nx][ny] - 'A';
                    if (0 == (key & (1 << lock))) continue;
                }
                if (!seen.count(s)) seen.insert(s), Q.push({nx * col + ny, nk});
            }
        }
        ++ans;
    }
    return -1;
}
```

```
// 909 - snakes and ladders
// soln-1: bfs (dynamic programming would not be able to handle loops)
int getBoard(vector<vector<int>>& board, int pos) {
    int row = board.size(), col = board[0].size();
    int x = row - 1 - (pos - 1) / col, y = (pos - 1) % col;
    if (((pos - 1) / col) % 2) y = col - 1 - y;
    return board[x][y];
}
int snakesAndLadders(vector<vector<int>>& board) {
    int row = board.size(), col = board[0].size(), ans = 0;
    vector<bool> visited(row * col + 1);
    queue<int> Q;
    Q.push(1), visited[1] = true;
    while (!Q.empty()) {
        for (int k = Q.size(); k > 0; --k) {
```

```
            auto cur = Q.front();   Q.pop();
            if (cur == row * col) return ans;
            for (int i = 1; i <= 6 && cur + i <= row * col; ++i) {
                auto next = cur + i, jump = getBoard(board, next);
                if (jump != -1) next = jump;
                if (!visited[next]) Q.push(next), visited[next] = true;
            }
        }
        ++ans;
    }
    return -1;
}
```

```
// 778 - swim in rising water
// soln-1: bfs (lower water height overwrite existing )
int swimInWater(vector<vector<int>>& grid) {
    int row = grid.size(), col = grid[0].size();
    vector<vector<int>> dist(row, vector<int>(col, INT_MAX));
    queue<pair<int, int>> Q;

    dist[0][0] = grid[0][0], Q.push({0, dist[0][0]});
    while (!Q.empty()) {
        auto p = Q.front(); Q.pop();
        int x = p.first / col, y = p.first % col, d = p.second;
        if (dist[x][y] < d) continue;

        for (auto& dir : vector<pair<int, int>>{{1, 0}, {-1, 0}, {0, -1}, {0, 1}}) {
            int nx = x + dir.first, ny = y + dir.second;
            if (nx < 0 || nx >= row || ny < 0 || ny >= col) continue;
            int nd = max(d, grid[nx][ny]);  // check if we need to go with higher elevation
            if (nd < dist[nx][ny]) Q.push({nx * row + ny, nd}), dist[nx][ny] = nd;
        }
    }
    return dist.back().back();
}

struct Node {
    int h, x, y;
    Node(int height, int i, int j) : h(height), x(i), y(j) {}
    bool operator< (const Node& n) const { return h > n.h; }    // min-heap
};
// 778 - swim in rising water
// soln-2: priority-queue (greedy find a path, return max level in the path)
int swimInWater(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size(), ans = 0;
    priority_queue<Node> pq;
    vector<vector<bool>> visited(m, vector<bool>(n));
    pq.push(Node(grid[0][0], 0, 0)), visited[0][0] = true;
    while (!pq.empty()) {
        auto t = pq.top();    pq.pop();
        ans = max(ans, t.h);    // max height in the path
        if (t.x == (m - 1) && t.y == (n - 1)) break;

        for (auto& d : vector<pair<int, int>>{{1, 0}, {-1, 0}, {0, -1}, {0, 1}}) {
            int i = t.x + d.first, j = t.y + d.second;
            if (i < 0 || i >= m || j < 0 || j >= n || visited[i][j]) continue;
            pq.push(Node(grid[i][j], i, j)), visited[i][j] = true;
        }
    }
    return ans;
}

// 778 - swim in rising water
// soln-3: binary search (trial and error)
// use range-based [0, N*N-1] binary search to find the min height which can reach (dfs) to the right-bottom.
```

Ref:

780. TODO

Solution:

Ref:

781/991. Rabbits in forest/Broken calculator

```
// 781 - Rabbits in Forest
// soln-1: math (greedy)
int numRabbits(vector<int>& answers) {
    sort(answers.begin(), answers.end());
    int ans = 0;
    for (int i = 0; i < answers.size(); nullptr) {
        int n = answers[i], num = n;
        ans += n + 1;           // update rabbits number

        // jump at most n rabbits and answers must be same
        for (int j = ++i; n > 0 && j < answers.size(); ++j, --n) {
            if (answers[j] != num) break;
            ++i;
        }
    }
    return ans;
}
```

```
// 991 - broken calculator
// soln-1: math (binary search / greedy)
// 1. brute-force bfs would TLE
// 2. Greedy cut Y into half as long as X < Y
int brokenCalc(int X, int Y) {
    int ans = 0;
    while (X < Y) {
        if (0 == Y % 2) {
            Y >>= 1, ++ans;
        } else {
            Y = (Y >> 1) + 1, ans += 2;     // 2X and X-1 operations
        }
    }
    return ans + (X - Y);
}
```

Ref:

782. TODO

Solution:

Ref:

783/938. Min distance between BST nodes/Range sum of BST

Given a Binary Search Tree (BST) with the root node `root`, return the minimum difference between the values of any two different nodes in the tree.

Example :

```
Input: root = [4,2,6,1,3,null,null], Output: 1
Explanation: it occurs between node 1 and node 2, also between node 3 and node 2.
```

```
// 783 - min distance between BST
void h(TreeNode* &pre, TreeNode* cur, int& ans) {
   if (!cur) return;
   h(pre, cur->left, ans);
   if (pre) ans = min(ans, abs(pre->val - cur->val));
   pre = cur;
   h(pre, cur->right, ans);
}
int minDiffInBST(TreeNode* root) {
   TreeNode* pre = NULL;
   int ans = root->val;
   h(pre, root, ans);
   return ans;
}
```

```
// 938 - range sum of BST
int rangeSumBST(TreeNode* root, int L, int R) {
    if (nullptr == root) return 0;
    if (root->val < L) return rangeSumBST(root->right, L, R);
    if (root->val > R) return rangeSumBST(root->left, L, R);
    return rangeSumBST(root->left, L, R) + root->val + rangeSumBST(root->right, L, R);
}
```

Ref: #94

784. Letter case permutation

Given a string S, we can transform every letter individually to be lowercase or uppercase to create another string. Return a list of all possible strings we could create.

```
Examples:
Input: S = "a1b2", Output: ["a1b2", "a1B2", "A1b2", "A1B2"]
```

```
// soln-1: backtracking to enumerate
void helper(int start, string& str, vector<string>& ans) {
   ans.push_back(str);
   for (int i = start; i < str.length(); ++i) {
       if (isalpha(str[i])) {
           str[i] ^= (1 << 5);          // toggle case
           helper(i + 1, str, ans);
           str[i] ^= (1 << 5);
       }
   }
}
vector<string> letterCasePermutation(string S) {
   vector<string> ans;
   helper(0, S, ans);
   return ans;
}
```

Ref:

785/802/851/886. Bipartite/directed graph cycle detect/bi-partition-able (review)

```
// 785 - bipartite
// soln-1: color graph (dfs)
bool isBipartite(vector<vector<int>>& graph) {
   vector<int> color(graph.size());

   for (int i = 0; i < graph.size(); ++i) {
       if (graph[i].empty() || color[i]) continue;     // colored or not
       if (!helper(graph, i, color)) return false;
   }
   return true;
}

bool helper(vector<vector<int>>& g, int from, vector<int>& color) {
   if (color[from] == 0) color[from] = 1;          // color it if not yet
   for (auto& n : g[from]) {
       if (color[n] == color[from]) return false;  // color conflict
       if (color[n] != 0) continue;                // visited

       color[n] = (color[from] == 1 ? 2 : 1);
       if (!helper(g, n, color)) return false;
   }
   return true;
}
```

```
// 886 - possible bipartition
// soln-1: color graph with greedy (dfs)
bool possibleBipartition(int N, vector<vector<int>>& dislikes) {
    unordered_map<int, unordered_set<int>> g;
    int start = 0;
    for (auto& e : dislikes) {
        g[e[0]].insert(e[1]), g[e[1]].insert(e[0]), start = e[0];
    }

    vector<int> color(N + 1);
    stack<int> stk;
    stk.push(start), color[start] = 1;
    while (!stk.empty()) {
        start = stk.top();  stk.pop();
        for (auto neigh : g[start]) {       // try to color my disliked neighbors
            int toColor = (color[start] == 1 ? 2 : 1);
            if (0 == color[neigh]) stk.push(neigh), color[neigh] = toColor;
            else if (color[neigh] != toColor) return false;
        }
    }
    return true;
}
```

```
// 1042 - Flower Planting With No Adjacent (with 4 diff. colors)
// soln-1: greedy
void color(int n, vector<unordered_set<int>>& g, vector<int>& ans) {
   for (int c = 1; c <= 4; ++c) {
       bool used = false;
       for (auto nxt : g[n]) {
           if (ans[nxt] == c) used = true;
           if (used) break;
       }
       if (!used) {
           ans[n] = c;
           return;
       }
   }
}
vector<int> gardenNoAdj(int N, vector<vector<int>>& paths) {
```

```
    vector<int> ans(N);
    vector<unordered_set<int>> g(N);
    for (auto& e : paths) g[e[0] - 1].insert(e[1] - 1), g[e[1] - 1].insert(e[0] - 1);
    for (int i = 0; i < N; ++i) color(i, g, ans);
    return ans;
}
```

```
// 802 - find eventual safe states (direct graph cycle detect)
// soln-1: dfs with coloring
// 1. use state to color node (from CLRS book): unprocessed, processing, processed
// 2. if node is under processing, it means it's in dfs stack, which means a cycle.
// return true if cycle found from start node
enum {init = 0, processing = 1, cycle = 2, nocycle = 3};
bool helper(vector<vector<int>>& g, int start, vector<int>& dp) {
    if (nocycle == dp[start]) return false;
    if (processing == dp[start] || cycle == dp[start]) return true;

    dp[start] = processing;
    for (auto neigh : g[start]) {
        if (helper(g, neigh, dp)) {     // cycle found if go to this way
            dp[start] = cycle;
            break;
        }
    }
    processing == dp[start] ? dp[start]= nocycle : dp[start] = cycle;
    return cycle == dp[start];
}
vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
    vector<int> ans, dp(graph.size());
    for (int i = 0; i < graph.size(); ++i) {
        if (!helper(graph, i, dp)) ans.push_back(i);
    }
    return ans;
}
```

```
// 851 - loud and rich
// soln-1: topology sort (dfs)
// f(x) = min{f(children-of-x)}
// return the least quiet person
int helper(unordered_map<int, unordered_set<int>>& g, vector<int>& q, int start, vector<int>& dp) {
    if (-1 == dp[start]) {
        dp[start] = start;
        for (auto n : g[start]) {
            auto person = helper(g, q, n, dp);
            if (q[dp[start]] > q[person]) dp[start] = person;
        }
    }
    return dp[start];
}
vector<int> loudAndRich(vector<vector<int>>& richer, vector<int>& quiet) {
    unordered_map<int, unordered_set<int>> g;
    for (auto& r : richer) g[r[1]].insert(r[0]);

    vector<int> ans(quiet.size()), dp(quiet.size(), -1);
    for (int i = 0; i < quiet.size(); ++i) {
        ans[i] = helper(g, quiet, i, dp);
    }
    return ans;
}
```

Ref:

786. TODO

Solution:

Ref:

788. Rotated digits (review)

X is a good number if after rotating each digit individually by 180 degrees, we get a valid number that is different from X. Each digit must be rotated - we cannot choose to leave it alone.

A number is valid if each digit remains a digit after rotation. 0, 1, and 8 rotate to themselves; 2 and 5 rotate to each other; 6 and 9 rotate to each other, and the rest of the numbers do not rotate to any other number and become invalid.

Now given a positive number `N`, how many numbers X from `1` to `N` are good?

```
Example:
Input: 10, Output: 4
Explanation:
There are four good numbers in the range [1, 10] : 2, 5, 6, 9.
Note that 1 and 10 are not good numbers, since they remain unchanged after rotating.
```

```
// soln-1: dynamic programming
// [0..9] could be divided into 3 groups:
// - unconvertable: 3, 4, 7
// - invalid: 0, 1, 8 (since they are same after rotation)
// - valid: 2, 5, 6, 9
// let dp(i) be in one of above states. consider how the last digit will contribute:
// - if last digit is {2, 5, 6, 9} and previous is not unconvertable, then dp(i) = valid
// - if last digit is {0, 1, 8} and,
// -     if previous is valid, then dp(i) = valid
// -     if previous is invalid, then dp(i) = invalid
int rotatedDigits(int N) {
   int ans = 0;
   enum {ILLEGAL, INVALID, VALID};
   vector<int> dp(N + 1);
   for (int i = 0; i <= N; ++i) {
       if (i < 10) {
           if (2 == i || 5 == i || 6 == i || 9 == i)   dp[i] = VALID, ++ans;
           else if (0 == i || 1 == i || 8 == i)        dp[i] = INVALID;
       } else {
           int last = i % 10;
           if ((2 == last || 5 == last || 6 == last || 9 == last) && dp[i / 10] != ILLEGAL) dp[i] = VALID,
++ans;
           else if (0 == last || 1 == last || 8 == last) {
               if (dp[i / 10] == VALID)            dp[i] = VALID, ++ans;
               else if (dp[i / 10] == INVALID)     dp[i] = INVALID;
           }
       }
   }
   return ans;
}
```

Ref:

792. Number of matching subsequence

Given string `S` and a dictionary of words `words`, find the number of `words[i]` that is a subsequence of `S`.

Example:

Input: `S = "abcde", words = ["a", "bb", "acd", "ace"]`, **Output:** `3`

Explanation: There are three words in words that are a subsequence of S: "a", "acd", "ace".

```
// soln-1: hashmap + binary seach
int numMatchingSubseq(string S, vector<string>& words) {
   vector<vector<int>> mp(26);
   for (int i = 0; i < S.length(); ++i) mp[S[i] - 'a'].push_back(i);

   int ans = 0;
   for (string& str : words) {
       bool seq = true;
       for (int i = 0, cur_pos = 0; seq && i < str.length(); ++i) {
           vector<int>& pos = mp[str[i] - 'a'];
           auto p = lower_bound(pos.begin(), pos.end(), cur_pos);
           if (p == pos.end()) seq = false;
           else cur_pos = (*p) + 1;
       }
       if (seq) ans++;
   }
   return ans;
}
```

Ref: #392

793. TODO

Solution:

Ref:

796. Rotate string

We are given two strings, `A` and `B`.

A *shift on* `A` consists of taking string `A` and moving the leftmost character to the rightmost position. For example, if `A = 'abcde'`, then it will be `'bcdea'` after one shift on `A`. Return `True` if and only if `A` can become `B` after some number of shifts on `A`.

```
// soln-1: one-liner
bool rotateString(string A, string B) {
   return A.length() == B.length() && (A + A).find(B) != string::npos;
}

// soln-2: normal ways
bool rotateString(string A, string B) {
   if (A.length() != B.length()) return false;
   if (A.empty()) return true;

   int i = B.find(A[0], 0), n = A.length();
   while (i != string::npos) {
       // check from A[0] against B[i]
       bool mismatch = false;
       for (int k = 0; !mismatch && k < n; ++k) {
           if (A[k] != B[(k + i) % n]) mismatch = true;;
       }
       if (!mismatch) return true;
```

```
        i = B.find(A[0], i + 1);
    }

    return false;
}
```

Ref:

798. TODO

Solution:

Ref:

799. TODO

Solution:

Ref:

800. Similar RGB color

The red-green-blue color "#AABBCC" can be written as "#ABC" in shorthand. For example, "#15c" is shorthand for the color "#1155cc".

Now, say the similarity between two colors "#ABCDEF" and "#UVWXYZ" is -(AB - UV)^2 - (CD - WX)^2 - (EF - YZ)^2.

Given the color "#ABCDEF", return a 7 character color that is most similar to #ABCDEF, and has a shorthand (that is, it can be represented as some "#XYZ"

```
// 800 - soln-1: brute-force
// 1. only the 2nd digit (from right to left) matters most, it could be -1/+0/+1, for example,
//    "e1" == 3 options => ["dd", "ee", "ff"]
// 2. smart way to pick up from 3 candidates directly:
//    n / 0x11 + n % 0x11 > 8 ? 1 : 0
// 0x11 is the difference in between [0x11, 0x22, 0x33, 0x44, ...],
// 1. for 0xab, only when b >= a, we will get 'a', otherwise ‘a-1’
// 2.           when b > half-of-difference, we need to promote to higher
string similarRGB(string color) {
    string ans, dict("0123456789abcdef");
    for (int i = 1; i < color.length(); i += 2) {
        int n = stoi(color.substr(i, 2), nullptr, 16);
        int idx = n / 0x11 + (n % 0x11 > 8 ? 1 : 0);
        ans += string(2, dict[idx]);
    }
    return ans;
}
```

Ref:

801. TODO

Solution:

Ref:

803. TODO

Solution:

Ref:

804. Unique Morse code words

trivial question

Ref:

805. TODO

Solution:

Ref:

806. Number of lines to write string

We are to write the letters of a given string `S`, from left to right into lines. Each line has maximum width 100 units, and if writing a letter would cause the width of the line to exceed 100 units, it is written on the next line. We are given an array `widths`, an array where widths[0] is the width of 'a', widths[1] is the width of 'b', ..., and widths[25] is the width of 'z'.

Now answer two questions: how many lines have at least one character from `S`, and what is the width used by the last such line? Return your answer as an integer list of length 2.

```
vector<int> numberOfLines(vector<int>& widths, string S) {
   int lines = 0, ll = 0, width = 100;
   for (auto& ch : S) {
       if ((ll + widths[ch - 'a']) > width) {
           lines++, ll = widths[ch - 'a'];
       } else {
           ll += widths[ch - 'a'];
       }
   }
   lines++;
   return {lines, ll};
}
```

Ref:

808. TODO

Solution:

Ref:

809/966. Expressive words/Vowel spellchecker

```cpp
// 809 - expressive words
// soln-1: brute-force
// if next is same char, then match 1 from S.
// else match 1 or >= 3 from S.
int expressiveWords(string S, vector<string>& words) {
    int ans = 0;
    for (auto& word : words) {
        // stretch match
        int k = 0, matched = true;;
        for (int i = 0; matched && i < S.length(); ++i) {
            if (k < word.length() && S[i] == word[k]) ++k;
            else if (i > 1 && S[i-2] == S[i-1] && S[i-1] == S[i]) continue;
            else if (0 < i && i < S.length() - 1 && S[i-1] == S[i] && S[i] == S[i+1]) continue;
            else matched = false;
        }
        if (k == word.length() && matched) ++ans;
    }
    return ans;
}
```

```cpp
// 966 - vowel spellchecker
// soln-1: hashmap
// 1. use set for exactly match
// 2. for capitalization and replace-able vowels:
// - transform to lowercase
// - because vowels are replaceable, so replace them as same char, to treat them as same
string normString(string word) {
   transform(word.begin(), word.end(), word.begin(), ::tolower);
   for (auto& ch : word) {
       if ('a' == ch || 'e' == ch || 'i' == ch || 'o' == ch || 'u' == ch) ch = '#';
   }
   return word;
}

vector<string> spellchecker(vector<string>& wordlist, vector<string>& queries) {
   unordered_set<string> words{wordlist.begin(), wordlist.end()};

   unordered_map<string, string> lo;           // <lower, origin>
   unordered_map<string, string> norm;         // <normalized, origin>
   for (auto word : wordlist) {
       norm.insert({normString(word), word});  // insert method = put-if-absent in Java.
       auto l = word;
       transform(l.begin(), l.end(), l.begin(), ::tolower);
       lo.insert({l, word});
   }

   vector<string> ans;
   for (auto q : queries) {
       if (words.find(q) != words.end()) {
           ans.push_back(q);
       } else {
           auto l = q;
           transform(l.begin(), l.end(), l.begin(), ::tolower);
           if (lo.find(l) != lo.end()) {
               ans.push_back(lo[l]);
           } else {
               auto it = norm.find(normString(q));
               it != norm.end() ? ans.push_back(it->second) : ans.push_back("");
           }
       }
   }
```

```
    return ans;
}
```

Ref:

810. TODO

Solution:

Ref:

811. subdomain visit count

We are given a list `cpdomains` of count-paired domains. We would like a list of count-paired domains, (in the same format as the input, and in any order), that explicitly counts the number of visits to each subdomain.

Example 2:

Input: ["900 google.mail.com", "50 yahoo.com", "1 intel.mail.com", "5 wiki.org"]

Output:

["901 mail.com","50 yahoo.com","900 google.mail.com","5 wiki.org","5 org","1 intel.mail.com","951 com"]

Explanation: We will visit "google.mail.com" 900 times, "yahoo.com" 50 times, "intel.mail.com" once and

```
vector<string> subdomainVisits(vector<string>& cpdomains) {
    unordered_map<string, int> mp;
    for (auto& p : cpdomains) {
        auto c = stoi(p.substr(0, p.find(' ')));
        for (int i = p.find(' ') + 1; i < p.length(); ++i) {
            mp[p.substr(i)] += c;
            for (int j = i; j < p.length() && p[j] != '.'; ++j, i = j);
        }
    }

    vector<string> ans;
    for (auto& p : mp)ans.push_back(to_string(p.second) + " " + p.first);

    return ans;
}
```

Ref:

813. TODO

Solution:

Ref:

814/865. Binary tree pruning/smallest substree with all deepest nodes

```
// 814 - binary tree pruning if does not contain 1
// soln-1: dfs
```

```
TreeNode* pruneTree(TreeNode* root, TreeNode* parent = nullptr) {
    if (nullptr == root) return nullptr;
    pruneTree(root->left, root);
    pruneTree(root->right, root);
    if (0 == root->val && nullptr == root->left && nullptr == root->right) {
        if (nullptr == parent) return nullptr;
        parent->left == root ? parent->left = nullptr : parent->right = nullptr;
    }
    return root;
}
```

```
// 865 - smallest substree with all deepest nodes
// soln-1: bottom up dfs
pair<TreeNode*, int> h(TreeNode* root) {
    if (nullptr == root) return {nullptr, 0};
    auto l = h(root->left), r = h(root->right);

    if (l.second == r.second) return {root, l.second + 1};
    if (l.second < r.second) return {r.first, r.second + 1};
    return {l.first, l.second + 1};
}

TreeNode* subtreeWithAllDeepest(TreeNode* root) {
    return h(root).first;
}
```

Ref:

815. TODO

Solution:

Ref:

819/884. Most common word / uncommon word

```
// 819 - most common word
string mostCommonWord(string paragraph, vector<string>& banned) {
   unordered_set<string> b(banned.begin(), banned.end());

   unordered_map<string, int> mp;
   pair<string, int> ans;
   for (int i = 0; i < paragraph.length(); ++i) {
       int j = i;
       while (j < paragraph.length() && isalpha(paragraph[j])) ++j;
       string w = paragraph.substr(i, j - i);
       i = j;

       std::transform(w.begin(), w.end(), w.begin(), ::tolower);
       if (w.empty() || w == " " || b.find(w) != b.end()) continue;

       if(++mp[w] > ans.second) ans.second = mp[w], ans.first = w;
   }

   return ans.first;
}
```

```
// 884 - uncommon words from two sentences
vector<string> uncommonFromSentences(string A, string B) {
   unordered_map<string, int> mp;
```

```
    istringstream iss(A + " " + B);    // combine string, uncommon words would only appear once
    while (iss >> A) mp[A]++;

    vector<string> ans;
    for (auto& p : mp) {
        if (p.second == 1) ans.push_back(p.first);
    }
    return ans;
}
```

Ref:

820. TODO

Solution:

Ref:

821. Shortest distance to a character

Given a string `S` and a character `C`, return an array of integers representing the shortest distance from the character `C` in the string.

Example 1:

Input: S = "loveleetcode", C = 'e', **Output:** [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]

#238 - product of array except self

```
// soln-1: binary search
vector<int> shortestToChar(string S, char C) {
    set<int> pos;
    for (int i = 0; i < S.length(); ++i) {
        if (S[i] == C) pos.insert(i);
    }

    vector<int> ans;
    for (int i = 0; i < S.length(); ++i) {
        auto it = pos.lower_bound(i);
        if (it == pos.begin())      ans.push_back(*it - i);
        } else if (it == pos.end()) ans.push_back(i - *pos.rbegin());
        } else {
            int d = *it - i;        ans.push_back(min(d, i - *(--it)));
        }
    }
    return ans;
}

// soln-2: two passes scan (better than soln-1)
vector<int> shortestToChar(string S, char C) {
    int n = S.length();
    vector<int> ans(n, n);
    for (int i = 0, latest = -n; i < S.length(); ++i) {
        if (S[i] == C) latest = i;
        ans[i] = min(ans[i], i - latest);
    }
    for (int i = n - 1, latest = 0; i >= 0; --i) {
        if (S[i] == C) latest = i;
        ans[i] = min(ans[i], latest - i);
```

```
    }
    return ans;
}
```

Ref:

822. TODO

Solution:

Ref:

824. Goat latin

trivial

```
string helper(const string& w, int idx) {
    string ans;
    if (w[0] == 'a' || w[0] == 'A' || w[0] == 'e' || w[0] == 'E' || w[0] == 'i' ||
        w[0] == 'I' || w[0] == 'o' || w[0] == 'O' || w[0] == 'u' || w[0] == 'U') {
        ans = w + "ma";
    } else {
        ans = w.substr(1) + w[0] + "ma";
    }
    for (int i = 0; i < idx; ++i) ans += "a";
    return ans;
}

string toGoatLatin(string S) {
    string ans;
    for (int i = 0, idx = 1; i < S.length(); ++i, ++idx) {
        // extract a word
        int j = i;
        while (j < S.length() && S[j] != ' ') ++j;

        if (!ans.empty()) ans += " ";
        ans += helper(S.substr(i, j - i), idx);

        i = j;
    }
    return ans;
}
```

Ref:

825. Friends of appropriate ages

Some people will make friend requests. The list of their ages is given and ages[i] is the age of the ith person. Person A will NOT friend request person B (B != A) if any of the following conditions are true:

- age[B] <= 0.5 * age[A] + 7
- age[B] > age[A]
- age[B] > 100 && age[A] < 100

Otherwise, A will friend request B. Note that if A requests B, B does not necessarily request A. Also, people will not friend request themselves. How many total friend requests are made?

```
// 825 - soln-1: hashmap
// group people by its age, if there is a request, then
// - People with diff. age, request time = i * j, i/j are the number-of-person
// - People with same age, times = k * (k-1), k is number-of-person
bool request(int a, int b) {
   return !(b <= (a/2 + 7) || (b > a) || (b > 100 && a < 100));
}
int numFriendRequests(vector<int>& ages) {
   unordered_map<int, int> grp;    // <age, persons>
   for (auto& age : ages) grp[age]++;

   int ans = 0;
   for (auto& p1 : grp) {
       for (auto& p2 : grp) {
           if (request(p1.first, p2.first)) {
               ans += p1.second * (p2.second - (p1.first == p2.first));
           }
       }
   }
   return ans;
}
```

Ref:

828/891/907. Unique letter string/Sum of subsequence width/subarray mins. (review)

828 - Given a string S with only uppercases, calculate the sum of UNIQ(substring) over all non-empty substrings of S. If there are two or more equal substrings at different positions in S, we consider them different.

891 - Given an array of integers A, consider all non-empty subsequences of A. For any sequence S, let the *width* of S be the difference between the maximum and minimum element of S. Return the sum of the widths of all subsequences of A.

907 - Given an array of integers A, find the sum of min(B), where B ranges over every (contiguous) subarray of A.

Example 1:

```
Input: [3,1,2,4], Output: 17
Explanation: Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4].
Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.  Sum is 17.
```

Monotone stack:
496/503 - Next great element/Online stock plan

```
// 828 - soln-1: math
// Instead of consider each substring, consider each letter, how many substring would
// contain me as unique letter? For example, XA(XAXX)A, consider 2nd A, how many substring
// would contain me as unique substring?
// It's about the ways of inserting '(' between 1st and 2nd A, and inserting ')' between 2nd and 3rd.
// 1. for '(', 2 ways: XA(XAXXA / XAX(AXXA
// 2. for ')', 3 ways: XAXA)XXA/XAXAX)XA/XAXAXX)A
// So, there are 2 * 3 = 6 substrings which would count the 2nd A as unique letter.
//
// Algorithm:
// 1. keep the last 2 occurrence for each letter
```

```
// 2. when hit 2nd time, calculate the number of substrings containning the letter at PREV position
// 3. at the end of string, calculate all (it's like hitting every letter again)
int uniqueLetterString(string S) {
    vector<vector<int>> idx(26, vector<int>(2, -1));
    int ans = 0, mod = 1e9+7;
    for (int i = 0; i < S.length(); ++i) {
        auto ch = S[i] - 'A';
        int l = idx[ch][1] - idx[ch][0], r = i - idx[ch][1];
        ans += (l * r) % mod;
        idx[ch][0] = idx[ch][1], idx[ch][1] = i;
    }
    // step-3
    for (int i = 0; i < 26; ++i) {
        int l = idx[i][1] - idx[i][0], r = S.length() - idx[i][1];
        ans = (ans + (l * r) % mod) % mod;
    }
    return ans;
}
```

```
// 891 - soln-1: math
// 1. consider each number, how many subsequence would it belong to as min/max respectively?
// 2. after step-1, the final answer would be sum-for-max(i * number-of-subseq(i)) - sum-for-min
//     |2   1   3
// ----+----------
// min |2   4   1           [2], [2, 3] subsequence would consider 2 as min.
// max |2   1   4           => 3 * (4-1) + 1 * (1-4) + 2 * (2-2) = 6
//
// Consider [3], how many subsequence would contain it as min. of subsequence:
//   [4, 2, 5, 1, 3, 6]
// On left side, we have 2 numbers bigger than it, so it's 2^2 = 4 subsequence (1 if asking for subarray)
// On right side, we have 1 numbers, it's 2^1 = 2 subsequence, total 4 * 2 = 8 subsequence.
//
// I was thinking to use BST to count numbers bigger than current, it turns out there position doesn't matter.
// So we can simply sort, then go through one-by-one.
// For A[i], there are 2^i smaller numbers, A[i] would contribute as max number.
//                      2^(n-1-i) bigger number, A[i] as min number.  A[i] * (2^i - 2^(n-1-i)) % mod
// Another trick is about dealing with overflow, 2^(n-1-i) is too big if we calculate it first.
int sumSubseqWidths(vector<int>& A) {
   sort(A.begin(), A.end());
   long ans = 0, mod = 1e9+7, c = 1, n = A.size();
   for (long i = 0; i < n; ++i, c = (c << 1) % mod) {
       ans = (ans + A[i] * c - A[n - 1 - i] * c) % mod; // A[n-1-i] also has c subseq. contributing as min.
   }
   return (ans + mod) % mod;
}
```

```
// 907 - sum of subarray mins
// soln-1: monotone stack .:|
// Instead of consider each substring, consider each number,
// how many substrings we have such that current value shall be the min. of substring?
// For ex. given [2, 9, 7, 8, 3, 4, 6, 1], how to find any every subarray with min. 3.
// 1. find the first element bigger than 3: [9, 7, 8]
// 2. find the last element bigger than 3: [4, 6]
// 3. there are 12 = 4 * 3 subarray with min. 3
// 4. hence the sum of all min(subarray) = 12 * 3.
// 5. monotone stack can easily get previous smaller element in O(n)
// 6. given array: [71, 55, 82, 55], trick for dups - can choose only from one side:
//                 [1, 2, 1, 4] <= #s on my left side before hitting 1st number less than me.
//                 [1, 2, 1, 1] <= #s on my right side before hitting 1st number great than me.
int sumSubarrayMins(vector<int>& A) {
    vector<int> left(A.size()), right(A.size()), stk;
    for (int i = 0; i < A.size(); ++i) {
        while (!stk.empty() && A[stk.back()] >= A[i]) stk.pop_back();   // choose from left side
        left[i] = !stk.empty() ? i - stk.back() : i + 1;
        stk.push_back(i);
    }
```

```
    stk.clear();
    for (int i = A.size() - 1; i >= 0; --i) {
        while (!stk.empty() && A[stk.back()] > A[i]) stk.pop_back();    // avoid dup result, > not >=
        right[i] = !stk.empty() ? stk.back() - i : A.size() - i;
        stk.push_back(i);
    }

    unsigned long M = 1e9+7, ans = 0;
    for (int i = 0; i < A.size(); ++i) ans += (unsigned long)(left[i] * right[i] * A[i]) % M;

    return ans % M;
}
```

Ref:

829. TODO

Solution:

Ref:

830. Positions of large groups

In a string `S` of lowercase letters, these letters form consecutive groups of the same character.

For example, a string like `S = "abbxxxxzyy"` has the groups `"a"`, `"bb"`, `"xxxx"`, `"z"` and `"yy"`.

Call a group *large* if it has 3 or more characters. We would like the starting and ending positions of every large group.

The final answer should be in lexicographic order.

Example 1:

Input: "abbxxxxzzy", **Output:** [[3,6]]

Explanation: "xxxx" is the single large group with starting 3 and ending positions 6.

```
// 830 - positions of large groups
vector<vector<int>> largeGroupPositions(string S) {
   vector<vector<int>> ans;

   int l = 0, c = 0, ch = 0;
   for (int i = 0; i < S.length(); ++i) {
       if (ch == 0) {
           ch = S[i], l = i, c = 1;
       } else if (ch != S[i]) {
           if (c >= 3) ans.push_back({l, i - 1});
           ch = S[i], l = i, c = 1;
       } else {
           c++;
       }
   }
   if (c >= 3) ans.push_back({l, S.length() - 1});

   return ans;
}
```

Ref:

831. TODO

Solution:

Ref:

832. Flipping an image

Given a binary matrix `A`, we want to flip the image horizontally, then invert it, and return the resulting image.

To flip an image horizontally means that each row of the image is reversed. For example, flipping `[1, 1, 0]` horizontally results in `[0, 1, 1]`.

To invert an image means that each `0` is replaced by `1`, and each `1` is replaced by `0`. For example, inverting `[0, 1, 1]` results in `[1, 0, 0]`.

```
// 832 - flipping an image (trivial)
vector<vector<int>> flipAndInvertImage(vector<vector<int>>& A) {
   for (auto& r : A) reverse(r.begin(), r.end());
   for (auto& r : A) {
       for (auto& i : r) i ^= 1;
   }
   return A;
}
```

Ref:

833. Find and replace in string

To some string `S`, we will perform some replacement operations that replace groups of letters with new ones (not necessarily the same size).

Each replacement operation has `3` parameters: a starting index `i`, a source word `x` and a target word `y`. The rule is that if `x` starts at position `i` in the **original string `S`**, then we will replace that occurrence of `x` with `y`. If not, we do nothing.

For example, if we have `S = "abcd"` and we have some replacement operation `i = 2, x = "cd", y = "ffff"`, then because `"cd"` starts at position `2` in the original string `S`, we will replace it with `"ffff"`.

Using another example on `S = "abcd"`, if we have both the replacement operation `i = 0, x = "ab", y = "eee"`, as well as another replacement operation `i = 2, x = "ec", y = "ffff"`, this second operation does nothing because in the original string `S[2] = 'c'`, which doesn't match `x[0] = 'e'`.

All these operations occur simultaneously. It's guaranteed that there won't be any overlap in replacement: for example, `S = "abc", indexes = [0, 1], sources = ["ab","bc"]` is not a valid test case.

Example 1:

```
Input: S = "abcd", indexes = [0,2], sources = ["a","cd"], targets = ["eee","ffff"], Output: "eeebffff"
Explanation: "a" starts at index 0 in S, so it's replaced by "eee". "cd" starts at index 2 in S, so it's replaced by "ffff".
```

```
// soln-1: sort given indexes decendingly, then replace from right to left
string findReplaceString(string S, vector<int>& indexes, vector<string>& sources, vector<string>& targets) {
   vector<pair<int, int>> sorted;          // sort by index-of-S
   for (int i = 0; i < indexes.size(); ++i) sorted.push_back({indexes[i], i});
   sort(sorted.rbegin(), sorted.rend());   // sort by index descendingly
   for (auto& p : sorted) {
       int idx = p.first;
       string& src = sources[p.second], &tgt = targets[p.second];
```

```
        if (S.substr(idx, src.length()) == src) S.replace(idx, src.length(), tgt);
    }
    return S;
}
```

Ref:

834. Sum of distance in tree (review)

An undirected, connected tree with N nodes labelled 0...N-1 and N-1 edges are given. The ith edge connects nodes edges[i][0]and edges[i][1] together. Return a list ans, where ans[i] is the sum of the distances between node i and all other nodes.

```
// 834 - sum of distances in tree
// soln-1: dynamic programming + dfs + bfs
// 1. pick any node, let us say root, as starting node, sum the distance in O(n).
// 2. how to use the know result is the key to this question.
//    let f(root) be the sum starting from root. consider immediate child of root,
//    all the nodes under this child would be 1-step closer to root, the rest nodes
//    would be 1-step further to root. hence
//    f(child) = f(parent) - number-of-child(child-node) + N - number-of-child(child-node)
//                           ~~~~~~~~~~~~~~~~~~~~~~~~~~    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//                              1-step closer                    1-step further
// 3. dfs to get 1) sum of distance to all other nodes. 2) number of child for current node.
// 4. bfs/dfs to calculate sum of distance for immediate children.
void dfs(t_graph& g, int start, int dist, vector<int>& visited, vector<int>& child, int& sum) {
  sum += dist, child[start] += 1, visited[start] = 1;
  for (auto& c : g[start]) {
      if (visited[c]) continue;
      dfs(g, c, dist + 1, visited, child, sum);
      child[start] += child[c];
  }
}

// update the reset sum of distance using bfs
void bfs(t_graph& g, int start, vector<int>& child, vector<int>& ans) {
  int N = ans.size();
  queue<int> q;
  vector<bool> visited(N);
  q.push(start), visited[start] = true;
  while (!q.empty()) {
      for (int i = q.size(); i > 0; --i) {
          int n = q.front();  q.pop();
          for (auto& c : g[n]) {
              if (visited[c]) continue;
              ans[c] = ans[n] - child[c] + N - child[c], q.push(c), visited[c] = true;
          }
      }
  }
}

// update the reset sum of distance using dfs too
void update(t_graph& g, int start, vector<int>& visited, vector<int>& child, vector<int>& ans) {
  visited[start] = 1;
```

```
  for (auto& c : g[start]) {
      if (visited[c]) continue;
      ans[c] = ans[start] - child[c] + ans.size() - child[c];
      update(g, c, visited, child, ans);
  }
}

vector<int> sumOfDistancesInTree(int N, vector<vector<int>>& edges) {
  unordered_map<int, unordered_set<int>> g;
  for (auto& e : edges) g[e[0]].insert(e[1]), g[e[1]].insert(e[0]);

  vector<int> ans(N), children(N), visited(N);
  dfs(g, 0, 0, visited, children, ans[0]);

  visited.assign(N, 0);
  update(g, 0, visited, children, ans);       // bfs(g, 0, children, ans);
  return ans;
}
```

Ref:

835. Image overlap (review)

Two images A and B are given, represented as binary, square matrices of the same size. (A binary matrix has only 0s and 1s as values.)

We translate one image however we choose (sliding it left, right, up, or down any number of units), and place it on top of the other image. After, the *overlap* of this translation is the number of positions that have a 1 in both images.

What is the largest possible overlap?

```
// 835 - soln-1: hashmap
// let offset of A[i][j] to B[m][n] be (i-m) + (j-n), then
//    offset(A[i][j], B[m][n]) = offset(A[i+1][j+1], B[m+1][n+1])
int largestOverlap(vector<vector<int>>& A, vector<vector<int>>& B) {
    vector<pair<int, int>> As, Bs;
    for (int i = 0; i < A.size(); ++i) {
        for (int j = 0; j < A.size(); ++j) {
            if (A[i][j]) As.push_back({i, j});
            if (B[i][j]) Bs.push_back({i, j});
        }
    }
    unordered_map<int, int> mp;
    for (auto& p1 : As) {
        for (auto& p2 : Bs) mp[myHash(p1.first - p2.first, p1.second - p2.second)]++;
    }

    int ans = 0;
    for (const pair<int, int>& kv : mp) ans = max(ans, kv.second);
    return ans;
}

// proof no collision under given condition:
// assume there is collision, which means,
// i * p1 + j * p2 = i' * p1 + j' * p2, given i != i' && j != j'
// ==> (i - i') * p1 = (j' - j) * p2
// ==> i - i' = p2
```

```
// ==> i = p2 + i', given i != i', i && i' within [1, 30]
// as long as p1/p2 is > 30 and p1 != p2, we can guarantee no collision.
int myHash(int x, int y) {
    return x * 37 + y * 31;
}
```

Ref:

837. TODO

Solution:

Ref:

839. TODO

Solution:

Ref:

840. Magic squares in grid

In a row of `seats`, `1` represents a person sitting in that seat, and `0` represents that the seat is empty.

There is at least one empty seat, and at least one person sitting.

Alex wants to sit in the seat such that the distance between him and the closest person to him is maximized.

Return that maximum distance to closest person.

Example 1:

```
Input: [1,0,0,0,1,0,1], Output: 2
Explanation:
If Alex sits in the second open seat (seats[2]), then the closest person has distance 2.
If Alex sits in any other open seat, the closest person has distance 1.
Thus, the maximum distance to the closest person is 2.
```

trivial brute force ...

Ref:

842. Split array into Fibonacci sequence

Given a string `S` of digits, such as `S = "123456579"`, we can split it into a *Fibonacci-like sequence* `[123, 456, 579].`

Formally, a Fibonacci-like sequence is a list `F` of non-negative integers such that:

- `0 <= F[i] <= 2^31 - 1`, (that is, each integer fits a 32-bit signed integer type);
- `F.length >= 3`;
- and `F[i] + F[i+1] = F[i+2]` for all `0 <= i < F.length - 2`.

Also, note that when splitting the string into pieces, each piece must not have extra leading zeroes, except if the piece is the number 0 itself.

Return any Fibonacci-like sequence split from `S`, or return `[]` if it cannot be done.

Example 1:

`Input: "123456579", Output: [123,456,579]`

```
// test if string is splittable given 2 initial numbers
bool doable(const string& S, int pos, int f0, int f1, vector<int>& seq) {
   if (pos >= S.length()) return false;

   bool ans = true;
   seq = {f0 , f1};
   do {
       long f2 = f0 + f1;
       if (pos == S.length() || f2 > INT_MAX) break;
       seq.push_back(f2);

       string s2 = to_string(f2);
       if (s2.length() + pos > S.length()) ans = false;
       for (int i = 0; ans && i < s2.length(); ++i, ++pos) {
           if (s2[i] != S[pos]) ans = false;
       }
       f0 = f1, f1 = f2;
   } while (ans);

   if (!ans) seq.clear();
   return ans;
}

// soln-1: brute force split and test
vector<int> splitIntoFibonacci(string S) {
   vector<int> ans;
   const string imax = to_string(INT_MAX);
   for (int i = 1; i <= min(S.length(), imax.length()); ++i) {
       string f0 = S.substr(0, i);
       if (i > 1 && '0' == f0[0]) continue;
       if (f0.length() == imax.length() && f0 > imax) continue;

       for (int j = i; j < min(S.length(), imax.length()); ++j) {
           string f1 = S.substr(i, j - i + 1);
           if (j - i + 1 > 1 && '0' == f1[0]) continue;
           if (f1.length() == imax.length() && f1 > imax) continue;

           if (doable(S, j + 1, stoi(f0), stoi(f1), ans)) return ans;
       }
   }
   return ans;
}
```

Ref:

843. TODO

Solution:

Ref:

844. Backspace string cmp

Given two strings `S` and `T`, return if they are equal when both are typed into empty text editors. `#` means a backspace character.

Example 1:

```
Input: S = "ab#c", T = "ad#c", Output: true
Explanation: Both S and T become "ac".
```

```
// soln-1: compare from back directly O(n) time O(1) space
bool backspaceCompare(string S, string T) {
   int i = S.length() - 1, j = T.length() -1;
   while(1) {
       for (int c = 0; i >= 0 && (S[i] == '#' || c > 0); ) c += (S[i--] == '#') ? 1 : -1;
       for (int c = 0; j >= 0 && (T[j] == '#' || c > 0); ) c += (T[j--] == '#') ? 1 : -1;
       if (i < 0 || j < 0 || S[i] != T[j]) return i == -1 && j == -1;

       --i, --j;
   }
   return true;    // unreachable
}
```

Ref:

848. TODO

Solution:

Ref:

849. Maximize distance to closest person

In a row of `seats`, `1` represents a person sitting in that seat, and `0` represents that the seat is empty.

There is at least one empty seat, and at least one person sitting.

Alex wants to sit in the seat such that the distance between him and the closest person to him is maximized.

Return that maximum distance to closest person.

Example 1:

```
Input: [1,0,0,0,1,0,1], Output: 2
Explanation:
If Alex sits in the second open seat (seats[2]), then the closest person has distance 2.
If Alex sits in any other open seat, the closest person has distance 1.
Thus, the maximum distance to the closest person is 2.
```

```
// 849 - max distance to closest person (0 - empty, 1 - occupied)
// soln-1: brute-force (keep last 1's position)
int maxDistToClosest(vector<int>& seats) {
   int ans = 0, p1 = -1;    // last 1's position
   for (int i = 0; i < seats.size(); ++i) {
       if (seats[i] == 0) continue;
       if (-1 == p1) {
           ans = i;
       } else {
```

```
            ans = max(ans, (i - p1) / 2);
        }
        p1 = i;
    }
    ans = max(ans, (int)seats.size() - 1 - p1);

    return ans;
}
```

Ref:

854. TODO

Solution:

Ref:

857. TODO

Solution:

Ref:

859. Buddy strings

Given two strings `A` and `B` of lowercase letters, return `true` if and only if we can swap two letters in `A` so that the result equals `B`.

Example 1: **Input:** A = "ab", B = "ba", **Output:** true	**Example 2:** **Input:** A = "ab", B = "ab", **Output:** false

```
// buddy string if:
// 1, position-mismatch-exactly-twice && (m1 ^ m2) == 0, OR
// 2, no mismatch at all && at least 1 char appeared twice
bool buddyStrings(string A, string B) {
    if (A.empty() || (A.length() != B.length())) return false;

    bool dup_char = false;
    vector<bool> m(26, false);

    int m1 = 0, m2 = 0;
    for (int i = 0; i < A.length(); ++i) {
        if (A[i] ^ B[i]) {
            if (m1 == 0) m1 = A[i] ^ B[i];
            else if (m2 == 0) m2 = A[i] ^ B[i];
            else return false;          // mismatch more than twice
        }

        if (m[A[i] - 'a']) dup_char = true;
        m[A[i] - 'a'] = true;           // mark it as appeared
    }
```

```
    if (m1 == 0) return dup_char;  // mismatch not happen at all
    return (m1 ^ m2) == 0;
}
```

Ref:

860. Lemonade change

At a lemonade stand, each lemonade costs `$5`.

Customers are standing in a queue to buy from you, and order one at a time (in the order specified by `bills`).

Each customer will only buy one lemonade and pay with either a `$5`, `$10`, or `$20` bill. You must provide the correct change to each customer, so that the net transaction is that the customer pays $5.

Note that you don't have any change in hand at first.

Return `true` if and only if you can provide every customer with correct change.

Example 1:

```
Input: [5,5,5,10,20], Output: true
Explanation:
From the first 3 customers, we collect three $5 bills in order.
From the fourth customer, we collect a $10 bill and give back a $5.
From the fifth customer, we give a $10 bill and a $5 bill.
Since all customers got correct change, we output true.
```

```
bool lemonadeChange(vector<int>& bills) {
    int c5 = 0, c10 = 0;
    for (int b : bills) {
        if (b == 20) {
            if (c10 > 0) b -= 10, c10 -= 10;     // exchange $10 first
            if ((b - 5) > c5) return false;      // enough $5 ?
            c5 -= (b - 5);
        } else if (b == 10) {
            if (c5 == 0) return false;           // enough $5 ?
            c5 -= 5, c10 += 10;
        } else {
            c5 += 5;
        }
    }
    return true;
}
```

Ref:

864. TODO

Solution:

Ref:

868. Binary gap

Given a positive integer N, find and return the longest distance between two consecutive 1's in the binary representation of N.

If there aren't two consecutive 1's, return 0.

```
// 868 - soln-1: bit manipulation
// as it counts only between 2 consecutive 1's
int binaryGap(int N) {
   int ans = 0;
   for (int pre = -1, cur = 0; N; N >>= 1, cur++) {
       if (N & 1) {
           if (-1 != pre) ans = max(ans, cur - pre);
           pre = cur;
       }
   }
   return ans;
}
```

Ref:

869. TODO

Solution:

Ref:

870. Advantage shuffle

Given two arrays A and B of equal size, the *advantage of A with respect to B* is the number of indices i for which A[i] > B[i].

Return any permutation of A that maximizes its advantage with respect to B.

```
// 870 - soln-1: greedy with BST
vector<int> advantageCount(vector<int>& A, vector<int>& B) {
    multiset<int> s(A.begin(), A.end());
    for (int i = 0; i < B.size(); ++i) {
        auto it = *s.rbegin() <= B[i] ? s.begin() : s.upper_bound(B[i]);
        A[i] = *it, s.erase(it);
    }
    return A;
}
```

Ref:

871. TODO

Solution:

Ref:

872. Leaf -similar trees

Two binary trees are considered *leaf-similar* if their leaf value sequence is the same.

```
// 872 - soln-1: dfs
// brute-force would be comparing two leaf list node
// depends on tree structure, we can getNextLeafNode helper then compare
bool leafSimilar(TreeNode* root1, TreeNode* root2) {
   stack<TreeNode*> stk1, stk2;
   stk1.push(root1), stk2.push(root2);

   while (!stk1.empty() && !stk2.empty()) {
       if (dfs(stk1) != dfs(stk2)) return false;
   }
   return stk1.empty() && stk2.empty();
}

int dfs(stack<TreeNode*>& stk) {
   while (!stk.empty()) {
       auto n = stk.top(); stk.pop();
       if (n->right) stk.push(n->right);
       if (n->left) stk.push(n->left);
       if (!n->left && !n->right) return n->val;
   }
   return INT_MIN; // never reach here
}
```

Ref:

873. Max length of Fib subsequence

Given a strictly increasing array `A` of positive integers forming a sequence, find the length of the longest fibonacci-like subsequence of `A`. If one does not exist, return 0.

```
// 873 - soln-1: dynamic programing (TLE related to inefficient memory usage I guess)
// let dp[a][b] be the max length of fib subsequence, when forming [..., a, b] fib sequence,
// dp[a][b] = dp[b][b-a] + 1, for example,
// dp[8][5] = dp[5][3] + 1          , if [3] is in arr, otherwise break
//          = dp[3][2] + 1 + 1      , if [2] is in arr
//          = dp[2][1] + 1 + 1      , if [1] is in arr
//          = dp[1][-1] + 1 + 1 + 1 , dp[a][b] = 1 when b <= a.
int lenLongestFibSubseq(vector<int>& A) {
    int ans = 0;
    unordered_map<int, unordered_map<int, int>> dp;
    for (int i = 1; i < A.size(); ++i) {
        for (int j = i - 1; j >= 0; --j) {
            int a = A[i], b = A[j];
            dp[a][b] = (dp[b].find(a - b) == dp[b].end()) ? 2 : dp[b][a - b] + 1;
            ans = max(ans, dp[a][b]);
        }
    }
    return ans;
}

// 873 - soln-2: brute-force in O(n^2*lg(n))
int lenLongestFibSubseq(vector<int>& A) {
    int ans = 0;
    unordered_set<int> s(A.begin(), A.end());
    for (int i = 0; i < A.size(); ++i) {
        for (int j = i + 1; j < A.size(); ++j) {
            // greedy check the length of fib starting with (A[i], A[j])
```

```
            // this would be finished in O(lg(n)) as fib sequence increase exponentially
            int len = 2, a = A[i], b = A[j];
            while (s.find(a + b) != s.end()) len++, a += b, swap(a, b);
            ans = max(ans, len);
        }
    }
    return 2 == ans ? 0 : ans;
}
```

Ref:

874. Walking robot simulation

```
struct pair_hash_t {
   size_t operator() (const pair<int, int>& p) const {
       return std::hash<int>{}(p.first ^ p.second);
   }
};
// 874 - walking robot simulation
int robotSim(vector<int>& commands, vector<vector<int>>& obstacles) {
   unordered_set<pair<int, int>, pair_hash_t> s;
   for (auto& p : obstacles) s.insert({p[0], p[1]});

   // moving forward under 4-dirs(east -> south -> west -> north)
   vector<pair<int, int>> dirs { {1,  0}, {0, -1}, {-1, 0}, {0,  1} };

   int ans = 0;
   pair<int, int> pos;
   for (int i = 0, d = 3; i < commands.size(); ++i) {  // by default, head to north
       switch (commands[i]) {
           case -1:    // turn right
               d = (d + 1) % 4;        break;
           case -2:    // turn left
               d = (d - 1 + 4) % 4;    break;
           default:    // move forward
               for (int step = 0; step < commands[i]; ++step) {
                   int nx = pos.first + dirs[d].first, ny = pos.second + dirs[d].second;
                   if (s.find({nx, ny}) != s.end()) break;
                   pos = {nx, ny};
                   ans = max(ans, pos.first * pos.first + pos.second * pos.second);
               }
               break;
       }
   }
   return ans;
}
```

Ref:

TODO

Solution:

Ref:

TODO
Solution:
Ref:

TODO
Solution:
Ref:

TODO
Solution:
Ref:

881. TODO
Solution:
Ref:

882. TODO
Solution:
Ref:

883. TODO
Solution:
Ref:

886. TODO
Solution:

Ref:

887. TODO

Solution:

Ref:

888. Fair candy swap

```
// 888 - soln-1: easy hashmap
vector<int> fairCandySwap(vector<int>& A, vector<int>& B) {
   // x must get this amount of candy
   int x = (accumulate(B.begin(), B.end(), 0) - accumulate(A.begin(), A.end(), 0)) / 2;
   unordered_set<int> s(B.begin(), B.end());
   for (auto n : A) {
       if (s.find(n + x) != s.end()) return vector<int>{n, n + x};
   }
   return vector<int>{-1, -1};
}
```

Ref:

883/892. Projection/Surface area of 3D shapes

Each value `v = grid[i][j]` represents a tower of `v` cubes placed on top of grid cell `(i, j)`.

Return the total surface area of the resulting shapes.

```
// 883 - soln-1: view the grid from col/row/top
// but we could do it in 1-pass instead of 2/3
int projectionArea(vector<vector<int>>& grid) {
    int x = 0, y = 0, z = 0;
    for (int i = 0; i < grid.size(); ++i) {
        int mx = 0, my = 0;
        for (int j = 0; j < grid[0].size(); ++j) {
            mx = max(mx, grid[i][j]), my = max(my, grid[j][i]);
            if (grid[i][j]) ++z;
        }
        x += mx, y += my;
    }
    return x + y + z;
}
```

```
// 892 - soln-1: math
// surface = 4 * v + 2 (if no adjacent)
// if v1 and v2 are adjacent, surface -= min(v1, v2) * 2
// at each position, check its up and left side neighbors
int surfaceArea(vector<vector<int>>& grid) {
   int ans = 0;
   for (int i = 0; i < grid.size(); ++i) {
       for (int j = 0; j < grid[0].size(); ++j) {
```

```
            if (grid[i][j]) ans += 4 * grid[i][j] + 2;
            if (i) ans -= min(grid[i][j], grid[i - 1][j]) * 2;
            if (j) ans -= min(grid[i][j], grid[i][j - 1]) * 2;
        }
    }
    return ans;
}
```

Ref:

893. TODO

Solution:

Ref:

895/981. Max frequency stack/Time based key-value store (review)

895 - FreqStack has two functions:

- push(int x), which pushes an integer x onto the stack.
- pop(), which removes and returns the most frequent element in the stack.
  - If there is a tie for most frequent element, the element closest to the top of the stack is removed and returned.

981 - Create a time based key-value store class TimeMap, that supports two operations.

1. set(string key, string value, int timestamp)

- Stores the key and value, along with the given timestamp.

2. get(string key, int timestamp)

- Returns a value such that set(key, value, timestamp_prev) was called previously, with timestamp_prev <= timestamp.
- If there are multiple such values, it returns the one with the largest timestamp_prev.
- If there are no values, it returns the empty string ("").

```
// 895 - soln-1: two hash maps
class FreqStack {
    unordered_map<int, int> _freq;          // x -> frequency
    unordered_map<int, stack<int>> _val;    // freq -> val
    int _maxFreq;
public:
    FreqStack() : _maxFreq(0) {
    }

    void push(int x) {
        _maxFreq = max(_maxFreq, ++_freq[x]);
        _val[_freq[x]].push(x);
    }

    int pop() {
        int x = _val[_maxFreq].top(); _val[_maxFreq].pop();
```

```
        if (_val[_freq[x]--].empty()) --_maxFreq;

        return x;
    }
};
```

```
// 981 - soln-1: hash map
class TimeMap {
    unordered_map<string, vector<pair<int, string>>> _mp;
public:
    void set(string key, string value, int timestamp) {
        _mp[key].push_back({timestamp, value});
    }

    string get(string key, int timestamp) {
        if (_mp.find(key) == _mp.end()) return "";

        auto& ts = _mp[key];
        int lo = 0, hi = ts.size() - 1;
        while (lo + 1 < hi) {
            int m = lo + (hi - lo) / 2;
            ts[m].first < timestamp ? lo = m : hi = m;
        }
        if (ts[hi].first <= timestamp) return ts[hi].second;
        if (ts[lo].first <= timestamp) return ts[lo].second;
        return "";
    }
};
```

Ref:

896. Monotonic array

```
// 896 - monotonic array
bool isMonotonic(vector<int>& A) {
   bool incr = false, decr = false;
   for (int i = 1; i < A.size(); ++i) {
       if (A[i] > A[i - 1]) incr = true;
       if (A[i] < A[i - 1]) decr = true;
       if (incr && decr) return false;
   }
   return !(incr && decr);     // increasing and decreasing are exclusive
}
```

Ref:

897. Flatten binary tree as in-order sequence

```
// 897 - flatten binary tree as in-order sequence
// soln-1: in-order recursion
TreeNode* increasingBST(TreeNode* root) {
   if (!root) return nullptr;
   auto left = increasingBST(root->left), right = increasingBST(root->right);
   if (left == nullptr) {
       root->left = nullptr, root->right = right;
       return root;
   } else {
       auto tail = left;
       while (tail->right) tail = tail->right;
```

```
        tail->right = root, root->left = nullptr, root->right = right;
        return left;
    }
}
```

Ref:

898. TODO

Solution:

Ref:

899. TODO

Solution:

Ref:

900. RLE iterator

```
class RLEIterator {
    vector<int> _A;
    int _idx, _avail;
public:
    RLEIterator(vector<int> A) {
        _A = A, _idx = 0, _avail = _A[0];
    }

    int next(int n) {
        for (; _avail < n; n -= _avail, _avail = _A[_idx]) {
            _idx += 2;
            if (_idx >= _A.size()) {
                _avail = 0;     // consumed anyway
                return -1;
            }
        }
        _avail -= n;            // _avail >= n
        return _A[_idx + 1];
    }
};
```

Ref:

902. TODO

Solution:

Ref:

903. TODO

Solution:

Ref:

905/922. Sort array by parity / II

```
// 905 - soln-1: two pointers
vector<int> sortArrayByParity(vector<int>& A) {
    int left = 0, right = A.size() - 1;
    while (left < right) {
        while (left < right && A[left] % 2 == 0) ++left;
        while (left < right && A[right] % 2 == 1) --right;
        if (left < right) swap(A[left++], A[right--]);
    }
    return A;
}
```

```
// 922 - soln-1: two pointers
vector<int> sortArrayByParityII(vector<int>& A) {
   int even = 0, odd = 1;
   while (even < A.size() && odd < A.size()) {
       while (even < A.size() && A[even] % 2 == 0) even += 2;
       while (odd < A.size() && A[odd] % 2 == 1) odd += 2;
       if (even < A.size() && odd < A.size()) swap(A[even], A[odd]), even += 2, odd += 2;
   }
   return A;
}
```

Ref:

TODO

Solution:

Ref:

914. X of a kind in a deck of cards

```
// 914 - X of a kind in a deck of cards
// soln-1: Great Common Divisor
bool hasGroupsSizeX(vector<int>& deck) {
    unordered_map<int, int> mp;
    for (auto& x : deck) mp[x]++;

    int ans = 0;
    for (auto& p : mp) ans = gcd(ans, p.second);
    return ans > 1;
}
```

Ref:

915. Partition array into 2 subarray

Given an array A, partition it into two (contiguous) subarrays left and right so that:

- Every element in left is less than or equal to every element in right.
- left and right are non-empty.
- left has the smallest possible size.

Return the length of left after such a partitioning. It is guaranteed that such a partitioning exists.

```
// 915 - soln-1: greedy
int partitionDisjoint(vector<int>& A) {
    vector<int> mx(A);
    for (int i = 1; i < A.size(); ++i) mx[i] = max(mx[i - 1], A[i]);

    int ans = 0;
    for (int i = A.size() - 1, mi = A.back(); i > 0; --i, mi = min(mi, A[i])) {
        if (mx[i - 1] > mi) continue;  // not a partition yet
        ans = i;                       // keep potential ans, keep searching to left
    }
    return ans;
}
```

Ref:

TODO

Solution:

Ref:

TODO

Solution:

Ref:

926. Flip string to monotone increasing

A string of '0's and '1's is *monotone increasing* if it consists of some number of '0's (possibly 0), followed by some number of '1's (also possibly 0.)

We are given a string S of '0's and '1's, and we may flip any '0' to a '1' or a '1' to a '0'. Return the minimum number of flips to make S monotone increasing.

```
// 926 - soln-1: dynamic programming
// let f0(i)/f1(i) be the min. flips when ending with 0/1 respectively. now we could see either 0 or 1:
// - 0, f0(i) = f0(i-1), no flip; f1(i) = min{f0(i-1), f1(i-1)} + 1, flip
// - 1, f0(i) = f0(i-1) + 1, flip; f1(i) = min{f0(i-1), f1(i-1)}, no flip
```

```cpp
// space can be in O(1) as we only depend on previous result.
int minFlipsMonoIncr(string S) {
   vector<int> dp0(S.length()), dp1(S.length());

   dp0[0] = '0' == S.front() ? 0 : 1, dp1[0] = '0' == S.front() ? 1 : 0;
   for (int i = 1; i < S.length(); ++i) {
       if ('0' == S[i]) dp0[i] = dp0[i - 1], dp1[i] = min(dp0[i - 1], dp1[i - 1]) + 1;
       else dp0[i] = dp0[i - 1] + 1, dp1[i] = min(dp0[i - 1], dp1[i - 1]);
   }
   return min(dp0.back(), dp1.back());
}
```

Ref:

TODO

Solution:

Ref:

929. Unique email addresses

```cpp
// 929 - soln-1: easy hashmap
int numUniqueEmails(vector<string>& emails) {
   unordered_set<string> s;
   for (auto& email : emails) {
       bool skip = false, domain = false;;
       string normalized;
       for (auto ch : email) {
           switch(ch) {
               case '+':
                   skip = true;                                    break;
               case '.':
                   if (domain) normalized.push_back(ch);           break;
               case '@':
                   skip = false, domain = true, normalized.push_back(ch);    break;
               default:
                   if (!skip) normalized.push_back(ch);            break;
           }
       }
       s.insert(normalized);
   }
   return s.size();
}
```

Ref:

TODO

Solution:

Ref:

TODO

Solution:

Ref:

933. Number of recent calls

Any ping with time in `[t - 3000, t]` will count, including the current ping.

It is guaranteed that every call to `ping` uses a strictly larger value of `t` than before.

```
Input:  ["RecentCounter","ping","ping","ping","ping"], inputs =[[],[1],[100],[3001],[3002]]
Output: [null,1,2,3,3]
```

```
// 933 - soln-1: ring buffer
// having a ring buffer with size of k+1, when pinged, record time and move head if value < cur - k
class RecentCounter {
   vector<long> _a;
   int _len, _head;
public:
   RecentCounter(int sz = 3000) {
       _a.assign(sz + 1, 0);       // 1 more space, because [..], not (..]
       _len = 0, _head = 0;
   }

   int ping(int t) {
       int n = _a.size();
       while (_len > 0 && _a[_head] < t - (n - 1)) _len--, _head = (_head + 1) % n;
       _a[(_head + _len) % n] = t, _len++;
       return _len;
   }
};
```

Ref:

TODO

Solution:

Ref:

TODO

Solution:

Ref:

TODO

Solution:

Ref:

945/954/969. Min increase to make array unique/Array of doubled pairs/Pancake sorting

945 - Given an array of integers A, a *move* consists of choosing any A[i], and incrementing it by 1.
Return the least number of moves to make every value in A unique.

954 - Given an array of integers A with even length, return true if and only if it is possible to reorder it such that A[2 * i + 1] = 2 * A[2 * i] for every 0 <= i < len(A) / 2.

969 - Given an array A, we can perform a *pancake flip*: We choose some positive integer k <= A.length, then reverse the order of the first k elements of A. We want to perform zero or more pancake flips (doing them one after another in succession) to sort the array A.
Return the k-values corresponding to a sequence of pancake flips that sort A. Any valid answer that sorts the array within 10 * A.length flips will be judged as correct.

```
// 945 - soln-1: greedy
// sort the input, each current must be greater than previous one.
int minIncrementForUnique(vector<int>& A) {
    int ans = 0;
    sort(A.begin(), A.end());
    for (int i = 1; i < A.size(); ++i) {
        if (A[i] <= A[i - 1]) {
            ans += A[i - 1] - A[i] + 1;
            A[i] = A[i - 1] + 1;
        }
    }
    return ans;
}
```

```
// 954 - soln-1: greedy
// use multimap (bst) to keep each number, greedy consume and find its peer.
// for negative value, find x/2, because we consume from least number
// for non-negative value, find x*2.
bool canReorderDoubled(vector<int>& A) {
    map<int, int> mp;
    for (int x : A) mp[x]++;

    while (!mp.empty()) {
        auto x = mp.begin()->first, y = x < 0 ? x / 2 : x * 2;
        if (mp.find(y) == mp.end()) return false;

        mp[y] -= mp[x], mp.erase(x);
        if (mp[y] < 0) return false;
        if (mp[y] == 0) mp.erase(y);
    }
    return mp.empty();
}
```

```
// 969 - soln-1: greedy/recursion
// 1. find the max, then use 2 flips to make it in position.
// 2. the problem size reduces 1. now repeat.
vector<int> pancakeSort(vector<int>& A) {
```

```
    vector<int> ans;
    for (int i = 0; i < A.size() - 1; ++i) {
        auto mx = max_element(A.begin(), A.end() - i);
        ans.push_back(mx - A.begin() + 1), ans.push_back(A.size() - i);

        reverse(A.begin(), mx + 1), reverse(A.begin(), A.end() - i);
    }
    return ans;
}
```

Ref:

TODO

Solution:

Ref:

948. Bag of tokens

```
// 948 - bag of tokens
// soln-1: greedy + two pointers
int bagOfTokensScore(vector<int>& tokens, int P) {
    int ans = 0, cur = 0;

    sort(tokens.begin(), tokens.end());
    for (int l = 0, r = tokens.size() - 1; l <= r; nullptr) {
        if (P >= tokens[l]) {
            P -= tokens[l], ++cur, ++l;
        } else if (cur) {
            P += tokens[r], --cur, --r;
        } else {
            break;      // no power to get tokens and no token to get power
        }
        ans = max(ans, cur);
    }
    return ans;
}
```

Ref:

949. TODO

Solution:

Ref:

950. Reveal cards in increasing order

do the following steps repeatedly, until all cards are revealed:

1. Take the top card of the deck, reveal it, and take it out of the deck.
2. If there are still cards in the deck, put the next top card of the deck at the bottom of the deck.

3. If there are still unrevealed cards, go back to step 1. Otherwise, stop.

Return an ordering of the deck that would reveal the cards in increasing order. The first entry in the answer is considered to be the top of the deck.

```
// 950 - soln-1: queue
vector<int> deckRevealedIncreasing(vector<int>& deck) {
    queue<int> q;
    for (int i = 0; i < deck.size(); ++i) q.push(i);

    sort(deck.begin(), deck.end());
    vector<int> ans(deck.size());
    for (int i = 1, j = 0; !q.empty(); ++i) {
        if (i % 2) {                        // consume a card from deck or not
            ans[q.front()] = deck[j++];
        } else {
            q.push(q.front());
        }
        q.pop();
    }
    return ans;
}
```

Ref:

952. TODO

Solution:

Ref:

953. TODO

Solution:

Ref:

954. TODO

Solution:

Ref:

955. TODO

Solution:

Ref:

956. TODO

Solution:

Ref:

960. TODO

Solution:

Ref:

961. N-repeated in 2N array (review)

In a array A of size 2N, there are N+1 unique elements, and exactly one of these elements is repeated N times. Return the element repeated N times.

```
// 961 - N-Repeated Element in Size 2N Array
// soln-1: if A[i] != A[i-1] and A[i] != A[i-2], then the answer must be A[0].
// [a, a, b, c]
// [a, b, c, a]
//
// soln-2: random check
// lets assume the pattern is [a a a a b c d], repeated numbers on same side.
// 1. if we pick up 2 value randomly, the probabilty of failure is 1/2 * 1/2,
// which means pick up 1 from left side and the other from right side.
// 2. so probability of success is 1 - (1/4)^x, x is runs.
// while (i == j or A[i] != A[j]) i = rand() % n, j = rand() % n;
int repeatedNTimes(vector<int>& A) {
    for (int i = 2; i < A.size(); ++i) {
        if (A[i] == A[i-1] || A[i] == A[i-2]) return A[i];
    }
    return A[0];
}
```

Ref:

962. TODO

Solution:

Ref:

964. TODO

Solution:

Ref:

965. TODO

Solution:

Ref:

967. Numbers with same consecutive diff.

```
// 967 - numbers with same consecutive differences
// soln-1: backtracking
void helper(int N, int K, int d, string cur, vector<int>& ans) {
   if (d >= N) {
       if (d == N) ans.push_back(stoi(cur));
       return;
   }

   int last = cur.back() - '0';
   if (last + K < 10)      helper(N, K, d + 1, cur + to_string(last + K), ans);
   if (K && last - K >= 0) helper(N, K, d + 1, cur + to_string(last - K), ans);
}
vector<int> numsSameConsecDiff(int N, int K) {
   if (1 == N) return vector<int> {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

   vector<int> ans;
   for (int i = 1; i < 10; ++i) {
       if (i + K <= 9 || i - K >= 0)  helper(N - 1, K, 0, to_string(i), ans);
   }
   return ans;
}
```

Ref:

969. TODO

Solution:

Ref:

970. Powerful integers

```
// 970. Powerful Integers. soln-1: brute-force
vector<int> powerfulIntegers(int x, int y, int bound) {
    unordered_set<int> s;
    for (int xx = 1; xx < bound; xx *= x) {
        for (int yy = 1; xx + yy <= bound; yy *= y) {
            s.insert(xx + yy);
            if (1 == y) break;
        }
        if (1 == x) break;
    }
    return vector<int>(s.begin(), s.end());
```

```
}
```

Ref:

972. TODO

Solution:

Ref:

973. TODO

Solution:

Ref:

974. TODO

Solution:

Ref:

975. TODO

Solution:

Ref:

978. TODO

Solution:

Ref:

979. Distribute coins in binary tree

Given the `root` of a binary tree with `N` nodes, each `node` in the tree has `node.val` coins, and there are `N` coins total.

In one move, we may choose two adjacent nodes and move one coin from one node to another. (The move may be from parent to child, or from child to parent.)

Return the number of moves required to make every node have exactly one coin.

```
// 979 - soln-1: dfs (bottom-up)
int distributeCoins(TreeNode* node, TreeNode* parent = nullptr) {
    if (nullptr == node) return 0;
    int l = distributeCoins(node->left, node), r = distributeCoins(node->right, node);

    int toParent = node->val - 1;    // how many coins go to my parent
    if (parent) parent->val += toParent;
    return l + r + abs(toParent);
}
```

Ref:

981. TODO

Solution:

Ref:

982. TODO

Solution:

Ref:

984. String without AAA or BBB

Given two integers A and B, return any string S such that:

- S has length A + B and contains exactly A 'a' letters, and exactly B 'b' letters;
- The substring 'aaa' does not occur in S;
- The substring 'bbb' does not occur in S.

```
// 984 - soln-1: greedy
string strWithout3a3b(int A, int B) {
    string ans;
    while (A + B) {
        int len = ans.length();
        if (len && ans[len - 1] == ans[len - 2]) {
            if (ans[len - 1] == 'a') {
                ans.push_back('b'), --B;
            } else {
                ans.push_back('a'), --A;
            }
        } else {
            if (A > B) {
                ans.push_back('a'), --A;
            } else {
                ans.push_back('b'), --B;
            }
        }
    }
    return ans;
}
```

Ref:

985. Sum of even numbers after queries

```
// 985 - sum of even numbers after queries (trivial)
vector<int> sumEvenAfterQueries(vector<int>& A, vector<vector<int>>& queries) {
    vector<int> ans;
    int sum = 0;
    for (auto x : A) if (0 == x % 2) sum += x;
    for (auto& q : queries) {
        int old = A[q[1]];
        A[q[1]] += q[0];
        if (0 == A[q[1]] % 2) {
            0 == old % 2 ? sum += A[q[1]] - old : sum += A[q[1]];
        } else if (0 == old % 2) {
            sum -= old;
        }
        ans.push_back(sum);
    }
    return ans;
}
```

Ref:

TODO

Solution:

Ref:

1000. TODO

Solution:

Ref:

Common Classes/Methods

```
// good slides about union-find: https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf
// summary: (WQUPC - Weighted Quick-Union Path Compression)
// - WQUPC is not quite linear in theory, but it is linear in practice.
// - WQUPC reduces time from 3000 years to 1 minute.(supercomputer won't help much)
// - path compression worst-case time: N + MlogN (M union-find ops on a set of N objects)
class DisjointSet {
   vector<int> _v;
   int _sz;
public:
   DisjointSet(int sz) : _sz(sz) {
       _v.resize(sz);
       iota(_v.begin(), _v.end(), 0);
   }
   int find(int x) {
       if (x != _v[x]) _v[x] = find(_v[x]);
       return _v[x];
```

```cpp
    }
    void join(int x, int y) {
        x = find(x), y = find(y);
        if (x != y) {
            _v[y] = x, _sz--;
        }
    }
    int size() { return _sz; }
};
```

```cpp
// Trie tree node
struct Trie {
    struct TrieNode {
        string word;
        TrieNode* next[26];
        TrieNode() { memset(next, 0, sizeof(next)); }
    };

    TrieNode* _root;
    Trie() : _root(nullptr) {}

    void build(const vector<string>& dict) {
        if (!_root) _root = new TrieNode;
        for (auto& w : dict) insert(0, w, _root);
    }

    vector<string> startsWith(const string& prefix) const {
        vector<string> ans;
        search(0, prefix, _root, ans);
        return ans;
    }

private:
    void insert(int start, const string& word, TrieNode* cur) {
        if (nullptr == cur || start >= word.length()) return;

        if (!cur->next[word[start] - 'a']) cur->next[word[start] - 'a'] = new TrieNode;
        cur = cur->next[word[start] - 'a'];
        if (start == word.length() - 1) cur->word = word;
        insert(start + 1, word, cur);
    }
    void search(int start, const string& prefix, TrieNode* cur, vector<string>& ans) const {
        if (nullptr == cur) return;
        if (start < prefix.length()) {
            search(start + 1, prefix, cur->next[prefix[start] - 'a'], ans);
            return;
        }

        if (!cur->word.empty()) ans.push_back(cur->word);
        for (int i = 0; i < sizeof(cur->next) / sizeof(TrieNode*); ++i) {   // in case the word length are 
different
            search(start, prefix, cur->next[i], ans);
        }
    }
};
```

```cpp
// Fenwick tree for range sum query
#define LOWBIT(x) (x & (-x))
class BITree {
    vector<int> _sum;
public:
    BITree(vector<int>& nums) {
        _sum.assign(nums.size() + 1, 0);
```

```
        for (int i = 0; i < nums.size(); ++i) edit(i + 1, nums[i]);
    }

    // update all sum related to i.
    void edit(int i, int delta) {
        for (NULL; i < _sum.size(); i += LOWBIT(i)) _sum[i] += delta;
    }

    // get sum from 1 to i
    int sum(int i) {
        int ans = 0;
        for (ans = 0; i > 0; i -= LOWBIT(i)) ans += _sum[i];    // sum all its parent
        return ans;
    }
};
```

```
// return the first target if having multiple
int bsearch(vector<int>& nums, int low, int hi, int target) {
    while (low + 1 < hi) {
        int m = low + (hi - low) / 2;
        target <= nums[m] ? hi = m ? low = m;
    }
    if (target == nums[low]) return low;
    if (target == nums[hi]) return hi;

    return -1;
}
```

```
// lcm(a, b) = (a * b) / gcd(a, b)
int gcd(int a, int b) {
   return 0 == b ? a : gcd(b, a % b);
}
```

```
// 1071 - Greatest Common Divisor of Strings
// soln-1: brute-force
string gcdOfStrings(string str1, string str2) {
    if (str1.length() < str2.length()) return gcdOfStrings(str2, str1);
    if (str2.empty()) return str1;

    if (str1.substr(0, str2.length()) != str2) return "";
    return gcdOfStrings(str1.substr(str2.length()), str2);
}
```

Trivial/Not-interesting Questions

```
// 6 - ZigZag Conversion
string convert(string s, int numRows) {
    if (1 == numRows) return s;
    vector<string> v(numRows);
    for (int i = 0, incr = 1, r = 0; i < s.length(); ++i, incr ? ++r : --r) {
        v[r].push_back(s[i]);
        if (incr && r + 1 >= numRows) incr = 0;
        if (!incr && r - 1 < 0) incr = 1;
    }
    string ans;
    for (auto& x : v) ans += x;
    return ans;
}
```

```
// 14 - longest common prefix
string longestCommonPrefix(vector<string>& strs) {
```

```
    if (strs.empty()) return "";

    for (int j = 0; j < strs[0].size(); ++j) {
        for (int i = 1; i < strs.size(); ++i) {
            if (j >= strs[i].size() || strs[i][j] != strs[0][j]) return strs[0].substr(0, j);
        }
    }
    return strs[0];;
}
```

```
// 58 - length of last word
int lengthOfLastWord(string s) {
    int ans = 0, tail = s.length() - 1;
    while (tail >= 0 && s[tail] == ' ') --tail;
    while (tail >= 0 && s[tail] != ' ') ++ans, --tail;
    return ans;
}
```

```
// 165 - Compare Version Numbers
// soln-1: brute-force
string trim(string str) {
    int i = 0, n = str.length();
    while (i < n && str[i] == '0') ++i;
    return i == n ? "0" : str.substr(i);
}
int compareVersion(string v1, string v2) {
    if (v1.empty() && v2.empty()) return 0;
    auto i1 = v1.find_first_of('.'), i2 = v2.find_first_of('.');
    if (i1 == string::npos) i1 = v1.length();
    if (i2 == string::npos) i2 = v2.length();

    string pre1 = trim(v1.substr(0, i1)), pre2 = trim(v2.substr(0, i2));
    int ans = 0;
    if (pre1.length() != pre2.length()) ans = pre1.length() > pre2.length() ? 1 : -1;
    else                                ans = pre1 > pre2 ? 1 : (pre1 == pre2 ? 0 : -1);
    if (0 != ans) return ans;

    v1 = (i1 == v1.length() ? "" : v1.substr(i1 + 1)), v2 = (i2 == v2.length() ? "" : v2.substr(i2 + 1));
    return compareVersion(v1, v2);
}
```

```
// 278 - first bad version
int firstBadVersion(int n) {
    int low = 1, hi = n;
    while (low + 1 < hi) {
        int m = low + (hi - low) / 2;
        isBadVersion(m) ? hi = m : low = m;
    }
    return isBadVersion(low) ? low : hi;
}
```

```
// 365 - water and jug problem
// soln-1: math
// the question is essentially a * x + b * y = z
// as long as 0 == z % gcd(a, b), we have the integer x and y.
bool canMeasureWater(int x, int y, int z) {
    return 0 == z || (x + y >= z && 0 == z % gcd(x, y));
}
```

```
// 439 - Ternary Expression Parser (contains only 0-9, :, ?, T/F)
// "F?1:T?4:5"  ==> T?4:5  ==> 4
// "T?T?F:5:3"  ==> T?F:5  ==> F
string parseTernary(string expression) {
    auto ans = expression;
    while (ans.length() > 1) {
```

```
        if (ans.front() == 'T') {
            ans = ans.substr(2, ans.find_last_of(':') - 2);
        } else {
            ans = ans.substr(ans.find_first_of(':') + 1);
        }
    }
    return ans;
}
```

```
// 476/1012 - number complement
int findComplement(int num) {
    int hb = 0;      // get highest bit 1
    for (int i = num; i >= 1; i >>= 1) ++hb;
    if (0 == hb) return 1;
    return ~num & ((1 << hb) - 1);
}
```

```
// 537 - Complex Number Multiplication
// (a1+b1i)*(a2+b2i) = a1a2 + (a1b2 + a2b1)i - b1b2
string complexNumberMultiply(string a, string b) {
     auto pa = a.find_first_of("+"), pb = b.find_first_of("+");
     auto a1 = stoi(a), a2 = stoi(b), b1 = stoi(a.substr(pa + 1)), b2 = stoi(b.substr(pb + 1));
     return to_string(a1 * a2 - b1 * b2) + "+" + to_string(a1 * b2 + a2 * b1) + "i";
}
```

```
// 592 - fraction add/sub
// soln-1: brute-force (istringstream usage is nice)
string fractionAddition(string expression) {
    istringstream iss(expression);
    int A = 0, B = 1, a, b;
    char _;
    while (iss >> a >> _ >> b) {
        A = A * b + B * a, B = b * B;
        auto g = abs(__gcd(A, B));
        A /= g, B /= g;
    }
    return to_string(A) + "/" + to_string(B);
}
```

```
// 609 - Find Duplicate File in System (group file by its content)
// soln1-: hashmap
vector<vector<string>> findDuplicate(vector<string>& paths) {
    unordered_map<string, vector<string>> mp;
    for (auto& str : paths) {
        istringstream iss(str);
        string path, filename;
        iss >> path;                 // extract path
        while (iss >> filename) {    // extract filename and content
            auto idx = filename.find('(');
            auto content = filename.substr(idx);
            mp[content].push_back(path + "/" + filename.substr(0, idx));
        }
    }
    vector<vector<string>> ans;
    for (auto& kv : mp) if (kv.second.size() > 1) ans.push_back(kv.second);
    return ans;
}
```

```
// 616/758 - Add Bold Tag in String
// soln-1: brute-force
string boldWords(vector<string>& words, string S) {
    unordered_set<string> dict;
    int mi = INT_MAX, mx = INT_MIN;
    for (auto& w : words) {
        dict.insert(w), mi = min(mi, (int)w.length()), mx = max(mx, (int)w.length());
    }
```

```
    vector<bool> bold(S.length());          // mark it if its bold in current position
    for (int i = 0; i <= S.length() - mi; ++i) {
        for (int len = mx; len >= mi; --len) {
            auto ss = S.substr(i, len);
            if (dict.find(ss) != dict.end()) {      // mark ss as bold
                for (int k = i; k < i + len; ++k) bold[k] = true;
                break;
            }
        }
    }
    string ans;
    for (int i = 0; i < S.length(); ++i) {
        if (bold[i]) {
            if (i > 0 && bold[i - 1]) {}    // do nothing if previous is bold too
            else ans += "<b>";
        } else {
            if (i > 0 && bold[i - 1]) ans += "</b>";
        }
        ans.push_back(S[i]);
    }
    if (bold.back()) ans += "</b>";
    return ans;
}
```

```
// 640 - solve the equation
// soln-1: brute-force
pair<int, int> helper(string equation) {
    if (equation.empty()) return {0, 0};
    int k = 0, b = 0, start = 0;
    for (int i = 0; i < equation.length(); ++i) {
        if ('x' == equation[i]) {
            auto co = equation.substr(start, i - start);
            if (co.empty()) co = "1";
            (start > 0 && '-' == equation[start - 1]) ? k -= stoi(co) : k += stoi(co);
            start = i + 1;
        } else if ('-' == equation[i] || '+' == equation[i]) {
            auto co = equation.substr(start, i - start);
            if (!co.empty()) {
                (start > 0 && '-' == equation[start - 1]) ? b -= stoi(co) : b += stoi(co);
            }
            start = i + 1;
        }
    }
    if ('x' != equation.back()) {
        auto co = equation.substr(start);
        (start > 0 && '-' == equation[start - 1]) ? b -= stoi(co) : b += stoi(co);
    }
    return {k, b};
}
string solveEquation(string equation) {
    auto idx = equation.find('=');
    auto p1 = helper(equation.substr(0, idx)), p2 = helper(equation.substr(idx + 1));
    auto k = p1.first - p2.first, b = p1.second - p2.second;
    if (0 == k) {
        return b ? "No solution" : "Infinite solutions";
    }
    return "x=" + to_string(-b / k);
}
```

```
// 708 - insert into a cyclic sorted list
// case-by-case:
// 1. insert in-between      : 10--(15)--20, pre=10, cur=20, pre < v < cur
// 2. insert a smallest value: (9)--10--20,  pre=20, cur=10, v > cur
// 3. insert a largest value : 10--20--(30), pre=20, cur=10, v > pre
Node* insert(Node* head, int V) {
     if (nullptr == head) {
```

```
        auto h = new Node(V, nullptr);
        h->next = h;
        return h;
    }

    auto pre = head, cur = head->next;
    while (pre != cur) {
        if (pre->val <= V && V <= cur->val) break;
        if (pre->val >= cur->val && V >= cur->val) break;
        if (pre->val >= cur->val && V >= pre->val) break;
        pre = cur, cur = cur->next;
    }
    pre->next = new Node(V, cur);
    return head;
}
```

```
// 709 - to lowercase
// 'a' => 0x61, 'A' => 0x41
string toLowerCase(string str) {
   for (auto& ch : str) {
       if (ch <= 'Z' && ch >= 'A') ch += 0x20;
   }
   return str;
}
```

```
// 721 - accounts merge based on email
// soln-1: graph dfs
void helper(vector<vector<string>>& accounts, unordered_map<int, unordered_set<int>>& g,
           int start, vector<bool>& visited, set<string>& ans) {
   visited[start] = true, ans.insert(accounts[start].begin() + 1, accounts[start].end());
   for (auto n : g[start]) {
       if (!visited[n]) helper(accounts, g, n, visited, ans);
   }
}

vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
   unordered_map<int, unordered_set<int>> g;
   unordered_map<string, int> emails;
   for (int i = 0; i < accounts.size(); ++i) {
       g[i].insert(i);                  // in case itself is a graph component.
       for (int j = 1; j < accounts[i].size(); ++j) {
           auto it = emails.find(accounts[i][j]);
           if (it != emails.end()) {
               g[it->second].insert(i), g[i].insert(it->second);
           } else {
               emails[accounts[i][j]] = i;
           }
       }
   }
   vector<bool> visited(accounts.size());
   vector<vector<string>> ans;
   for (auto i : g) {
       if (!visited[i.first]) {
           set<string> email;
           helper(accounts, g, i.first, visited, email);
           ans.push_back({email.begin(), email.end()});
           ans.back().insert(ans.back().begin(), accounts[i.first][0]);
       }
   }
   return ans;
}
```

```
// 733 - flood fill ()
void helper(vector<vector<int>>& image, int sr, int sc, int nc, int oc) {
   if (image[sr][sc] == oc && image[sr][sc] != nc) {
```

```
        image[sr][sc] = nc;
        if (sr + 1 < image.size())    helper(image, sr + 1, sc, nc, oc);
        if (sc + 1 < image[0].size()) helper(image, sr, sc + 1, nc, oc);
        if (sr - 1 >= 0)              helper(image, sr - 1,  sc, nc, oc);
        if (sc - 1 >= 0)              helper(image, sr, sc - 1, nc, oc);
    }
}

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {
    helper(image, sr, sc, newColor, image[sr][sc]);
    return image;
}
```

```
// 734/737 - sentence similarity
// soln-1: graph dfs
bool helper(unordered_map<string, unordered_set<string>>& g, string start, string& target,
unordered_set<string>& visited) {
    if (visited.find(start) == visited.end()) {
        visited.insert(start);
        for (auto& n : g[start]) {
            if (visited.find(n) == visited.end()) {
                if (helper(g, n, target, visited)) return true;
            }
        }
    }
    return start == target;
}
bool areSentencesSimilarTwo(vector<string>& words1, vector<string>& words2, vector<pair<string, string>> pairs)
{
    if (words1.size() != words2.size()) return false;
    unordered_map<string, unordered_set<string>> g;
    for (auto& pair : pairs) {
        g[pair.first].insert(pair.second), g[pair.second].insert(pair.first);
    }
    for (int i = 0; i < words1.size(); ++i) {
        if (words1[i] == words2[i]) continue;
        unordered_set<string> visited;
        if (!helper(g, words1[i], words2[i], visited)) return false;
    }
    return true;
}
```

```
// 749 - contain virus
// soln-1: dfs/bfs (brute-force, not interesting)
// 1. get islands and extend 1 step to know the infection size.
// 2. quarantine the biggest possible infection first, then repeat.
```

```
// 756 - pyramid transition matrix
// soln-1: brute-force dfs
bool helper(string bottom, int idx, string upper, unordered_map<string, unordered_set<string>>& bricks) {
    if (bottom.length() < 2) return bottom.length() == 1;
    if (idx > bottom.length() - 2) return helper(upper, 0, "", bricks);

    for (auto& brick : bricks[bottom.substr(idx, 2)]) {
        if (helper(bottom, idx + 1, upper + brick.substr(2, 1), bricks)) return true;
    }
    return false;
}
bool pyramidTransition(string bottom, vector<string>& allowed) {
    unordered_map<string, unordered_set<string>> bricks;
    for (auto& brick : allowed) bricks[brick.substr(0, 2)].insert(brick);
    return helper(bottom, 0, "", bricks);
}
```

```
// 807 - Max Increase to Keep City Skyline
```

```cpp
int maxIncreaseKeepingSkyline(vector<vector<int>>& grid) {
    int row = grid.size(), col = grid[0].size(), ans = 0;
    vector<int> rmax(row), cmax(col);
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            rmax[i] = max(rmax[i], grid[i][j]), cmax[j] = max(cmax[j], grid[i][j]);
        }
    }
    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            ans += min(rmax[i], cmax[j]) - grid[i][j];
        }
    }
    return ans;
}
```

```cpp
// 817. Linked List Components
int numComponents(ListNode* head, vector<int>& G) {
    bool miss = true;
    unordered_set<int> s(G.begin(), G.end());
    int ans = 0;
    while (head) {
        if (0 == s.count(head->val)) {
            miss = true;
        } else {
            if (miss) ++ ans;
            miss = false;
        }
        head = head->next;
    }
    return ans;
}
```

```cpp
// 853 - Car Fleet
// soln-1: sort by position
// if cur car is faster, then it will catch up to prev cars. otherwise itself form new fleet
int carFleet(int target, vector<int>& position, vector<int>& speed) {
    map<int, double> m;
    for (int i = 0; i < position.size(); ++i) {
        m[-position[i]] = double(target - position[i]) / speed[i];
    }
    int ans = 0;    double time = .0f;
    for (auto& it : m) {
        if (it.second > time) ans++, time = it.second;    // slower car will form new fleet
    }
    return ans;
}
```

```cpp
// 867 - transpose matrix
vector<vector<int>> transpose(vector<vector<int>>& A) {
    if (A.empty()) return A;
    vector<vector<int>> ans(A[0].size(), vector<int>(A.size()));
    for (int i = 0; i < A.size(); ++i){
        for (int j = 0; j < A[0].size(); ++j) {
            ans[j][i] = A[i][j];
        }
    }
    return ans;
}
```

```cpp
// 911 - online election
// soln-1: binary search (keep time stamp and leader, use upperbound to search)
class TopVotedCandidate {
    vector<pair<int, int>> _v;
public:
    TopVotedCandidate(vector<int>& persons, vector<int>& times) {
```

```cpp
        int mx = 0, p = 0;
        unordered_map<int, int> mp;     // person <-> count
        for (int i = 0; i < persons.size(); ++i) {
            auto count = ++mp[persons[i]];
            if (count >= mx) p = persons[i], mx = count, _v.push_back({times[i], p});
            else _v.push_back({times[i], p});
        }
    }

    int q(int t) {
        int l = 0, r = _v.size() - 1;
        while (l + 1 < r) {
            auto m = l + (r - l) / 2;
            if (_v[m].first == t) return _v[m].second;
            _v[m].first > t ? r = m : l = m;
        }
        if (_v[r].first <= t) return _v[r].second;
        return _v[l].second;
    }
};
```

```cpp
// 993 - cousins in binary tree
// soln-1: dfs (brute-force find height and parent)
// soln-2: bfs (able to terminate earlier)
void helper(TreeNode* node, int height, vector<int>& x, vector<int>& y) {
    if (nullptr == node) return;
    if (node->left && node->left->val == x[0]) x[1] = height + 1, x[2] = node->val;
    if (node->right && node->right->val == x[0]) x[1] = height + 1, x[2] = node->val;
    if (node->left && node->left->val == y[0]) y[1] = height + 1, y[2] = node->val;
    if (node->right && node->right->val == y[0]) y[1] = height + 1, y[2] = node->val;
    helper(node->left, height + 1, x, y), helper(node->right, height + 1, x, y);
}
bool isCousins(TreeNode* root, int x, int y) {
    vector<int> xx{x, 0, 0}, yy{y, 0, 0};
    helper(root, 0, xx, yy);
    return (xx[1] == yy[1]) && (xx[2] != yy[2]);
}
```

```cpp
// 994 - rotting oranges
// soln-1: bfs
int orangesRotting(vector<vector<int>>& grid) {
    int ans = 0, fresh = 0, m = grid.size(), n = grid[0].size();
    queue<pair<int, int>> q;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (2 == grid[i][j]) q.push({i, j});
            if (1 == grid[i][j]) ++fresh;
        }
    }

    while (fresh && !q.empty()) {
        ++ans;
        for (int i = q.size(); i > 0; --i) {
            auto p = q.front();     q.pop();
            for (auto& d : vector<pair<int, int>>{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}) {
                int nx = p.first + d.first, ny = p.second + d.second;
                if (nx < 0 || nx >= m || ny < 0 || ny >= n) continue;
                if (1 == grid[nx][ny]) {
                    --fresh, q.push({nx, ny}), grid[nx][ny] = 2;
                }
            }
        }
    }
    return fresh ? -1 : ans;
}
```

```cpp
// 999 - available captures for rook
// soln-1: brute-force
int numRookCaptures(vector<vector<char>>& board) {
    int x = 0, y = 0;
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            if ('R' == board[i][j]) x = i, y = j;
        }
    }
    int ans = 0;
    for (auto& d : vector<pair<int, int>>{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}) {
        int nx = x + d.first, ny = y + d.second;
        while (nx > 0 && nx < 8 && ny > 0 && ny < 8) {
            if (board[nx][ny] == 'p') {
                ++ans;
                break;
            } else if ('B' == board[nx][ny]) {
                break;
            }
            nx += d.first, ny += d.second;
        }
    }
    return ans;
}
```

```cpp
// 1002 - find common chars
// soln-1: brute-force
vector<string> commonChars(vector<string>& A) {
    vector<int> hits(26, INT_MAX);
    for (int i = 0; i < A.size(); ++i) {
        vector<int> hit(26);
        for (auto ch : A[i]) hit[ch - 'a']++;
        for (int k = 0; k < hits.size(); ++k) hits[k] = min(hits[k], hit[k]);
    }
    vector<string> ans;
    for (int i = 0; i < hits.size(); ++i) {
        for (int c = 0; c < hits[i]; ++c) ans.push_back(string(1, i + 'a'));
    }
    return ans;
}
```

```cpp
// 1005 - max sum of array after k negations
// soln-1: priority queue in O(klgn) time
// soln-2: math
// 1. sort input, negate min(K, all negative numbers)
// 2. ans = sum(A) - K % 2 ? 2 * min : 0 (keep negating the minimum)
int largestSumAfterKNegations(vector<int>& A, int K) {
    for (int i = 0; i < K; ++i) {
        make_heap(A.begin(), A.end(), [](int a, int b){
            return a > b;
        });
        A[0] = -A[0];
    }
    return accumulate(A.begin(), A.end(), 0);
}
```

```cpp
// 1010 - Pairs of Songs With Total Durations Divisible by 60
// soln-1: hashmap
int numPairsDivisibleBy60(vector<int>& time) {
    int ans = 0;
    vector<int> mp(60);
    for (auto t : time) {
        t = t % 60;
        if (0 == t) ans += mp[0], mp[0]++;
        else ans += mp[60 - t], mp[t]++;
    }
```

```
    return ans;
}
```

```
// 1013 - Partition Array Into Three Parts With Equal Sum
// soln-1: two pointers
bool canThreePartsEqualSum(vector<int>& A) {
   int sum = accumulate(A.begin(), A.end(), 0);
   if (sum % 3) return false;
   sum /= 3;
   int left = 0, right = A.size() - 1, L = 0, R = 0;
   while (left < right) {
       if (sum != L) L += A[left], left++;
       if (sum != R) R += A[right], right--;
       if (sum == L && sum == R) break;
   }
   if (left > right || L != sum || R != sum) return false;
   return true;
}
```

```
// 1017 - Convert to Base -2
// soln-1: math
// https://www.geeksforgeeks.org/convert-number-negative-base-representation/
string baseNeg2(int N) {
    if (0 == N) return "0";
    string ans;
    while (N) {
        int rem = N % -2;
        N /= -2;
        if (rem < 0) rem += 2, N += 1;
        ans = to_string(rem) + ans;
    }
    return ans;
}
```

```
// 1090 - Largest Values From Labels
// soln-1: hashmap + greedy (sort by value then greedy pick it, almost brute-force)
int largestValsFromLabels(vector<int>& values, vector<int>& labels, int num_wanted, int use_limit) {
    vector<pair<int, int>> items;
    for (int i = 0; i < values.size(); ++i) items.push_back({values[i], labels[i]});
    sort(items.rbegin(), items.rend());
    unordered_map<int, int> cnt;    // <label, used-count>
    int ans = 0;
    for (int i = 0; i < items.size() && num_wanted; ++i) {
        auto v = items[i].first, lbl = items[i].second;
        if (cnt[lbl] + 1 > use_limit) continue;
        ++cnt[lbl], ans += v, --num_wanted;
    }
    return ans;
}
```