BY LEARNSTACK

2026 Edition

DSA Cheat Sheet

Complete Data Structures & Algorithms Reference for Placements & Interviews

Arrays | Trees | Graphs | DP | Sorting | Recursion | Hashing

Learn. Build. Grow.

linktr.ee/LearnStack

Placement Series

About This Handbook

LearnStack DSA Complete Handbook 2026 is a premium, placement-focused reference for mastering Data Structures and Algorithms from fundamentals to advanced interview patterns. It is designed for CSE/IT students, developers refreshing DSA, self-learners preparing for product-company interviews, and competitive coding aspirants.

INFO: What makes it different
- Every topic has working Python code.
- Interview patterns are highlighted.
- Every algorithm includes time and space complexity.
- 100+ solved problems are included for revision and practice.

TIP: How to use this handbook
- Beginners should read chapters 1-6 linearly.
- Use chapters 11-16 as focused revision blocks.
- Use chapter 17 before interviews as a solved-problem bank.
- Use chapter 19 as a final-day quick reference.

Table of Contents

About This Handbook 2
Table of Contents 3
CHAPTER 1 - FOUNDATIONS OF DSA 4
1.1 What is DSA? . . . 4
1.2 DSA Learning Roadmap . . . 4
1.3 Choosing a Language . . . 4
1.4 Setup . . . 4
CHAPTER 2 - TIME & SPACE COMPLEXITY 6
2.5 Space Complexity . . . 8
2.6 Best, Average, Worst Case . . . 9
CHAPTER 3 - ARRAYS 10
3.6 Two Pointer, Sliding Window, Prefix Sum . . . 16
CHAPTER 4 - STRINGS 19
CHAPTER 5 - LINKED LISTS 27
CHAPTER 6 - STACK 33
CHAPTER 7 - QUEUE 39
CHAPTER 8 - TREES 44
CHAPTER 9 - HEAPS 52
CHAPTER 10 - GRAPHS 56
CHAPTER 11 - SORTING ALGORITHMS 64
CHAPTER 12 - SEARCHING ALGORITHMS 69
CHAPTER 13 - RECURSION & BACKTRACKING 74
CHAPTER 14 - DYNAMIC PROGRAMMING 81
CHAPTER 15 - HASHING 89
CHAPTER 16 - ADVANCED DATA STRUCTURES 94
CHAPTER 17 - TOP 100 INTERVIEW PROBLEMS 98
CHAPTER 18 - INTERVIEW PREPARATION GUIDE 132
CHAPTER 19 - QUICK REFERENCE CHEAT SHEET 135
Complexity Quick Reference . . . 135
Python Built-ins for DSA . . . 135
Must-Know Patterns . . . 135
Sorting Reference . . . 135
Tree Traversal Snippets . . . 136
Graph Traversal Snippets . . . 136
DP Pattern Identification Guide . . . 136
Most Used Templates . . . 136
Resources & Links 138

CHAPTER 1 - FOUNDATIONS OF DSA

CHAPTER 1
FOUNDATIONS OF DSA
Roadmap, language choice, setup, and interview mindset

1.1 What is DSA?

INFO: Definition
Data Structures are ways of organizing data. Algorithms are step-by-step procedures to solve problems. DSA matters because it controls how efficiently software stores, finds, transforms, and connects information.

- Used in databases, operating systems, compilers, search engines, APIs, and apps.
- Core of technical interviews and competitive programming.
- Improves problem-solving speed and code quality.

1.2 DSA Learning Roadmap

Step	Topic	Goal
1	Time & Space Complexity	Predict performance before coding
2	Arrays & Strings	Master indexing, two pointers, sliding window
3	Linked Lists	Pointers, reversal, fast/slow patterns
4	Stack & Queue	LIFO/FIFO and expression/traversal problems
5	Trees & BST	Recursive structures and search properties
6	Graphs	BFS, DFS, shortest paths, components
7	Sorting & Searching	Core algorithms and binary search patterns
8	Recursion & Backtracking	Explore solution spaces safely
9	Dynamic Programming	Optimize repeated subproblems
10	Advanced DS	Trie, DSU, segment tree, monotonic stack

1.3 Choosing a Language

Language	Pros	Cons	Best For
Python	Clean syntax, fast learning	Slower than C++	Beginners and interviews
Java	Strong OOP, common in campuses	Verbose	Campus placements
C++	Fast and STL-rich	Complex memory rules	Competitive coding
JavaScript	Flexible and popular	Less DS-focused	Web developers

TIP: Language tip
Use Python to understand DSA concepts quickly. Switch to C++ when contest speed and STL control matter.

1.4 Setup

First DSA Program

INFO: What it does
Install Python, use VS Code or Jupyter, and start by writing small functions that can be tested immediately.

```python
# Terminal setup
# python --version
# pip install notebook

print('Hello DSA!')
```

Output: Hello DSA!

COMPLEXITY
Time: O(1) | Space: O(1)

TIP
Keep each algorithm in a function so you can test it with multiple inputs.

CAUTION
Do not copy code without tracing it once.

CHAPTER 2 - TIME & SPACE COMPLEXITY

CHAPTER 2
TIME & SPACE COMPLEXITY
Big O, space usage, and analysis habits

INFO: Why complexity matters
Complexity tells how an algorithm grows when input size grows. In interviews, a correct but slow solution often needs optimization before it is accepted.

Notation	Name	Example	Speed
O(1)	Constant	Array access	Best
O(log n)	Logarithmic	Binary search	Fast
O(n)	Linear	Single loop	OK
O(n log n)	Linearithmic	Merge sort	Good for sorting
O(n^2)	Quadratic	Nested loops	Slow
O(2^n)	Exponential	Brute recursion	Avoid
O(n!)	Factorial	Permutations	Avoid

O(1) - Constant

INFO: What it does
Array access does not depend on input size.

Python

```
def first(arr):
    return arr[0] if arr else None
```

Output: Sample function returns the expected value.

COMPLEXITY
Time: O(1) | Space: O(1)

TIP
Drop constants and lower-order terms in Big O.

CAUTION
Big O usually means worst case unless the interviewer asks otherwise.

O(log n) - Logarithmic

INFO: What it does
Each step cuts the search space in half.

Python

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

print(binary_search([1, 3, 5, 7], 5))
```

Output: Sample function returns the expected value.

COMPLEXITY
Time: O(log n) | Space: O(1)

TIP
Drop constants and lower-order terms in Big O.

CAUTION
Big O usually means worst case unless the interviewer asks otherwise.

O(n) - Linear

INFO: What it does
A single loop visits each element once.

Python

```
def linear_search(arr, target):
    for i, x in enumerate(arr):
        if x == target:
            return i
    return -1

print(linear_search([10, 20, 30], 20))  # 1
```

Output: Sample function returns the expected value.

COMPLEXITY
Time: O(n) | Space: O(1)

TIP
Drop constants and lower-order terms in Big O.

CAUTION
Big O usually means worst case unless the interviewer asks otherwise.

O(n^2) - Quadratic

INFO: What it does
Nested loops compare pairs.

Python

```
def all_pairs(arr):
    ans = []
    for i in arr:
        for j in arr:
            ans.append((i, j))
    return ans
```

Output: Sample function returns the expected value.

COMPLEXITY
Time: O(n^2) | Space: O(n^2)

TIP
Drop constants and lower-order terms in Big O.

CAUTION
Big O usually means worst case unless the interviewer asks otherwise.

O(2^n) - Exponential

INFO: What it does
Each recursive call branches into two calls.

Python

```
def fib_slow(n):
    if n <= 1: return n
    return fib_slow(n-1) + fib_slow(n-2)
```

Output: Sample function returns the expected value.

COMPLEXITY
Time: O(2^n) | Space: O(n)

TIP
Drop constants and lower-order terms in Big O.

CAUTION
Big O usually means worst case unless the interviewer asks otherwise.

2.5 Space Complexity

INFO: Space complexity
Space complexity measures extra memory used by the algorithm excluding the input unless stated otherwise.
O(1) means constant extra variables, O(n) means memory proportional to input size.

Python

```
def copy_array(arr):
    copy = arr[:]     # O(n) extra space
    return copy

def in_place_increment(arr):
    for i in range(len(arr)):
        arr[i] += 1  # O(1) extra space
    return arr
```

INTERVIEW: Interview rule
Mention both time and space complexity after every solution. Explain why, not only the final notation.

2.6 Best, Average, Worst Case

Algorithm	Best	Average	Worst
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	O(n)	O(n^2)	O(n^2)
Quick Sort	O(n log n)	O(n log n)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)

CHAPTER 3 - ARRAYS

CHAPTER 3
ARRAYS
Contiguous storage, patterns, and interview problems

INFO: Array definition
An array stores elements in indexed order. Python lists are dynamic arrays; they support O(1) average append and O(1) index access.

Index: [0] [1] [2] [3] [4]
Value: [10][20][30][40][50]

Operation	Description	Time	Space
Access	arr[i]	O(1)	O(1)
Search	Find element	O(n)	O(1)
Insert end	append	O(1) avg	O(1)
Insert middle	Shift elements	O(n)	O(1)
Delete	Shift elements	O(n)	O(1)
Traverse	Visit all	O(n)	O(1)
Update	arr[i]=val	O(1)	O(1)

Declaration and Traversal

INFO: What it does
Create, access, update, and loop through a Python list.

Python

```
arr = [10, 20, 30]
arr[1] = 99
for i, value in enumerate(arr):
    print(i, value)
```

Output: Returned value depends on the input; sample calls are included where helpful.

COMPLEXITY
Time: O(n) | Space: O(1)

TIP
Trace the algorithm on a small example before coding.

CAUTION
Handle empty input and boundary values first.

Linear Search

INFO: What it does
Scan until target is found.

Python

```
def linear_search(arr, target):
    for i, x in enumerate(arr):
        if x == target:
            return i
    return -1

print(linear_search([10, 20, 30], 20))  # 1
```

Output: Returned value depends on the input; sample calls are included where helpful.

COMPLEXITY
Time: O(n) | Space: O(1)

TIP
Trace the algorithm on a small example before coding.

CAUTION
Handle empty input and boundary values first.

Find Maximum and Minimum

INFO: What it does
Track two values while traversing once.

Python

```
def max_min(arr):
    if not arr:
        return None, None
    mn = mx = arr[0]
    for x in arr[1:]:
        mn = min(mn, x)
        mx = max(mx, x)
    return mn, mx

print(max_min([4, 1, 9, 2]))
```

Output: Returned value depends on the input; sample calls are included where helpful.

COMPLEXITY
Time: O(n) | Space: O(1)

TIP
Trace the algorithm on a small example before coding.

CAUTION
Handle empty input and boundary values first.

Reverse an Array

INFO: What it does
Swap two ends and move inward.

Python

```
def reverse_array(arr):
    left, right = 0, len(arr) - 1
    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1
    return arr

print(reverse_array([1, 2, 3, 4]))
```

Output: Returned value depends on the input; sample calls are included where helpful.

COMPLEXITY
Time: O(n) | Space: O(1)

TIP
Trace the algorithm on a small example before coding.

CAUTION
Handle empty input and boundary values first.

Rotate Array

INFO: What it does
Normalize k and rebuild using slicing.

Python

```
def rotate_right(arr, k):
    n = len(arr)
    if n == 0: return arr
    k %= n
    return arr[-k:] + arr[:-k]

print(rotate_right([1, 2, 3, 4, 5], 2))
```

Output: Returned value depends on the input; sample calls are included where helpful.

COMPLEXITY
Time: O(n) | Space: O(n)

TIP
Trace the algorithm on a small example before coding.

CAUTION
Handle empty input and boundary values first.

LearnStack Free Preview

This was a free preview. Get the full book on LearnStack.

Visit: https://www.learnstack.co.in

Digital PDF delivery is handled through Gumroad email after purchase.