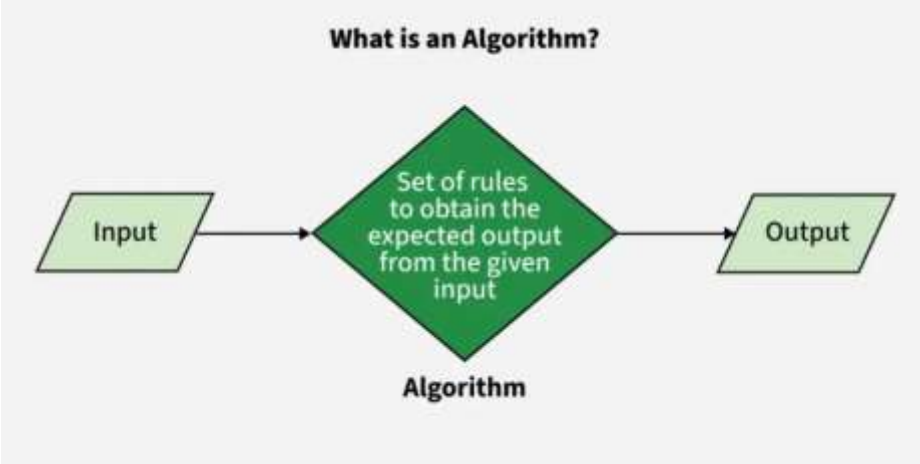
Outline

- Algorithm
- Complexity Notations
- Representation of Arrays & storage structure
- Sparse matrix and its representation
- Application of Arrays
- String
- Pointer
- Structure

Algorithm & Asymptotic Notations

Algorithm

The word **Algorithm** means "*A set of finite rules or instructions to be followed in calculations or other problem-solving operations*" Or "*A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations*".
Therefore Algorithm refers to a sequence of finite steps to solve a particular problem.



Use of Algorithm

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

Computer Science: Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.

Mathematics: Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.

Operations Research: Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.

Artificial Intelligence: Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.

Data Science: Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

Characteristics of Algorithms

1. Clear and Unambiguous: Every step must be precisely defined.

2. Input: It should have clearly defined input(s).

3. Output: It should produce at least one expected output**.**

4. Finiteness: The algorithm must end after a finite number of steps.

5. Effectiveness: Each step must be simple enough to be performed exactly and in a finite amount of time.

6. Feasibility: Performing the steps using available resources should be possible.

7. Language Independent: An algorithm is a logical process and should not depend on a specific programming language.

Complexity Notations

Asymptotic notation is a set of mathematical tools used in computer science to describe the efficiency and scalability of algorithms. It focuses on how an algorithm's running time or space requirements change as the input size grows, particularly for large inputs. This allows for a high-level understanding of an algorithm's behavior without being dependent on specific hardware or implementation details.

The three primary types of asymptotic notations are:

Big O Notation (O):

This describes the upper bound or worst-case time complexity of an algorithm. It provides a guarantee that the algorithm's performance will not exceed a certain growth rate as the input size increases. For example, if an algorithm has a time complexity of O(n^2), it means its running time grows no faster than a quadratic function of the input size 'n'.

Big Omega Notation (Ω):

This describes the lower bound or best-case time complexity of an algorithm. It indicates that the algorithm's performance will be at least as efficient as a certain growth rate. For example, Ω(n) means the algorithm will take at least linear time.

Big Theta Notation (Θ):

This describes the tight bound or average-case time complexity of an algorithm. It signifies that the algorithm's performance is bounded both from above and below by the same growth rate. If an algorithm is Θ(n log n), its running time grows proportionally to n log n for sufficiently large inputs.

Representation of Arrays & storage structure

Representation of Arrays

❑ What is Array?

- An **array** is a collection of elements of the **same data types**.
- Number of memory locations is **sequentially** allocated to the array.
- A array size is fixed and therefore requires a fixed number of memory locations.

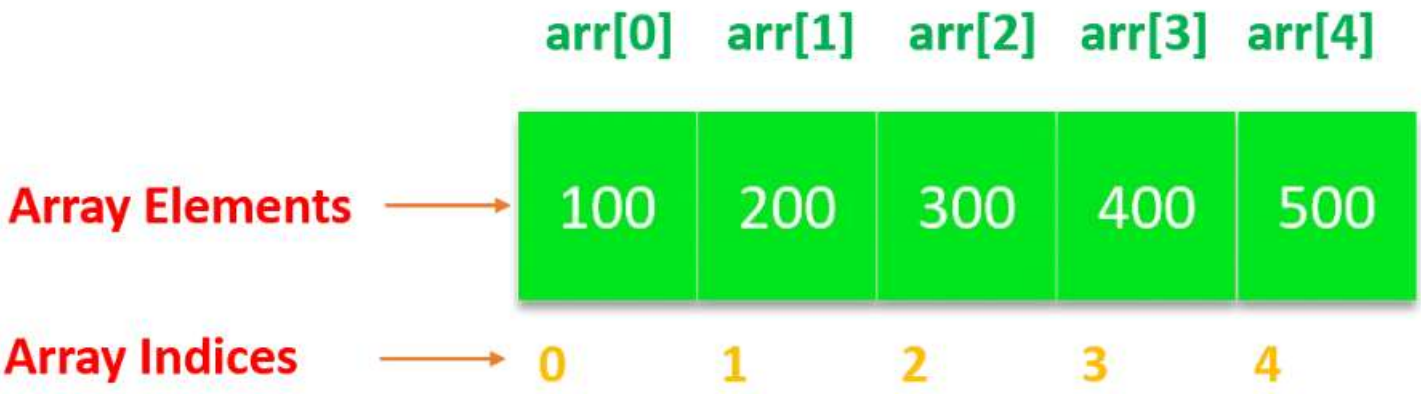
❑ Two types of Array

- 1-Dimension Array
- Multi Dimension Array

1-Dimensional Arrays

❑ 1-D Array

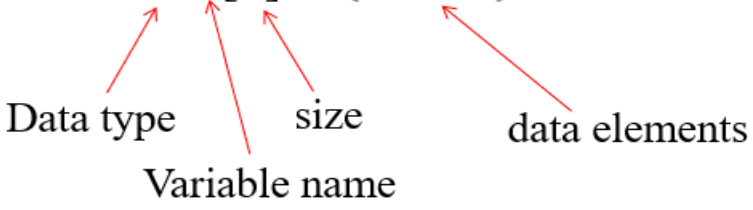
- Single dimensional array or 1-D array is the simplest form of array.
- This type of array consists of elements of similar types and these elements can be accessed through their indices.



1-Dimensional Arrays

❑ Declaration and Storing Array Elements

1. `int a[3] = {1, 2, 5}` [Compile Time]



2. [Run time]

```
int i, a[3];
for (i=0 ;  i < 3; i++)
    a[i] = i;
```

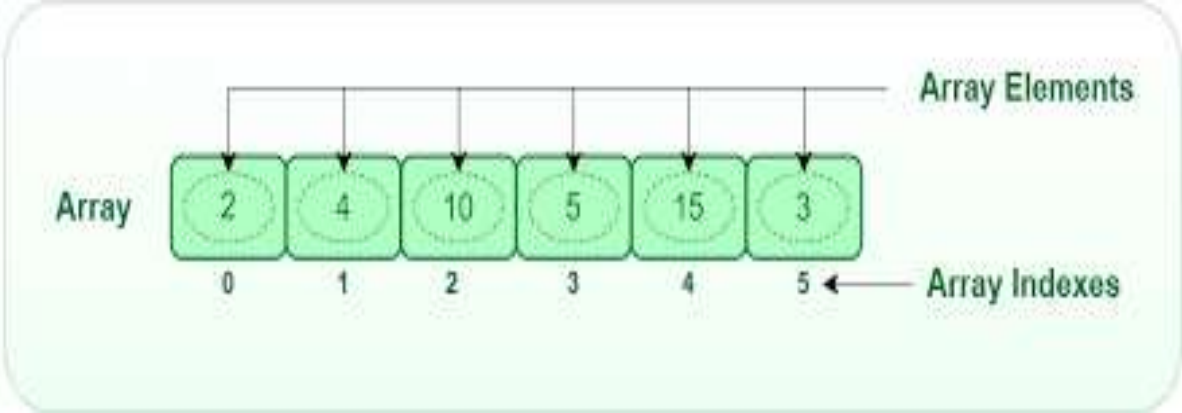
3. [Run time]

```
int i, a[3];
for (i=0 ; i < 3 ; i++)
    scan("%d", &a[i]);
```

1-Dimensional Arrays

❑ **Store array elements**

```
for (i = 0; i < n; i++)
{
        scanf("%d", &a[i]);
}
```



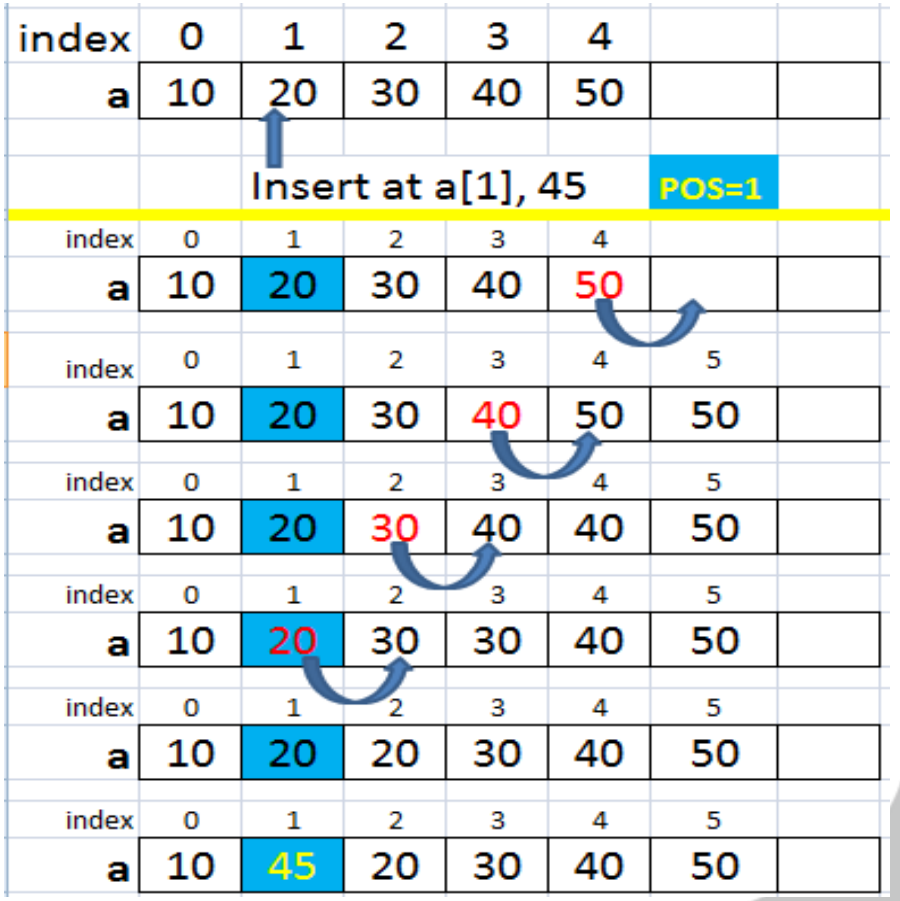
❑ **Display**

```
for (i = 0; i < n; i++)
{
        printf("%d ", a[i]);
}
```

1-Dimensional Arrays

❑ Insert elements at any position

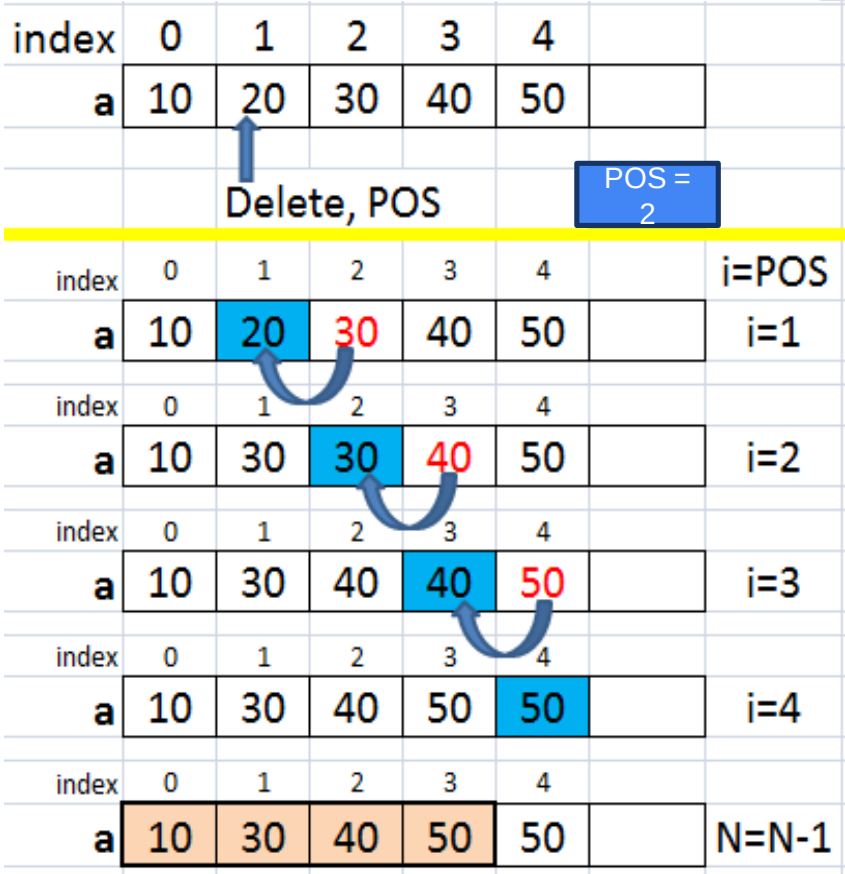
```
n=n+1;                  //n=size of array
for(i = n; i > pos; i--)
{
        a[i]=a[i-1];
}
a[pos-1]=x; //x=element
```



1-Dimensional Arrays

❑ Delete an element

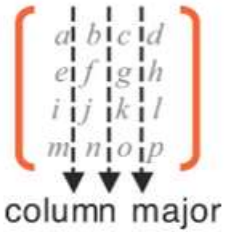
```
for (i = pos - 1; i < num -1; i++)
    {
       arr[i] = arr[i+1]; // assign arr[i+1] to arr[i]
    }
```

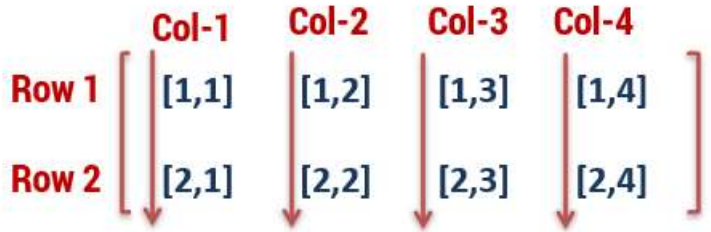


2-Dimensional Arrays

❑ 2-D Array

- Two dimensional arrays are also called ***table*** or ***matrix***
- Two dimensional arrays have two subscripts.
- **Column major order matrix:** Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array consisting of **two rows** and **four columns** is stored sequentially by columns : A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3], A[1,4], A[2,4]

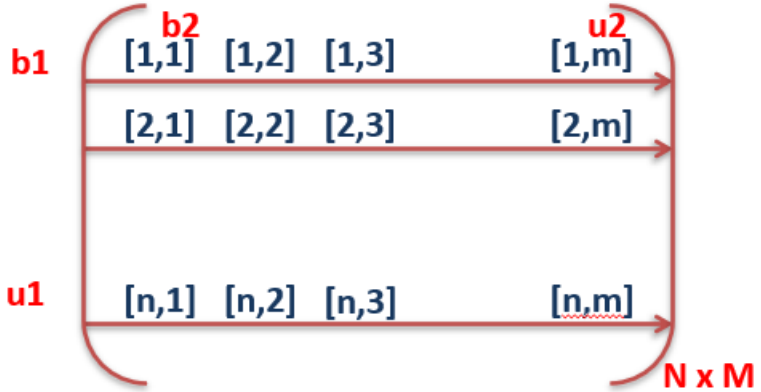




2-Dimensional Arrays

❑ 2-D Array

- **Row major order matrix:** Two dimensional array in which elements are stored row by row is called as row major matrix.



n = no of rows, **m** = no of columns

b1 = lower bound subscript of row
u1 = upper bound subscript of row
n = u1 – b1 + 1

b2 = lower bound subscript of column
u2 = upper bound subscript of column
m = u2 – b2 + 1

Memory Address Calculation in an Array

Address Calculation in single (one) Dimension Array:

Actual Address of the **1st** element of the array is known as **Base Address (B)**
Here it is 1100

Memory space acquired by every element in the Array is called **Width (W)**
Here it is 4 bytes

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	**15**	**7**	**11**	**44**	**93**	**20**
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Lower Limit/Bound of Subscript **(LB)**

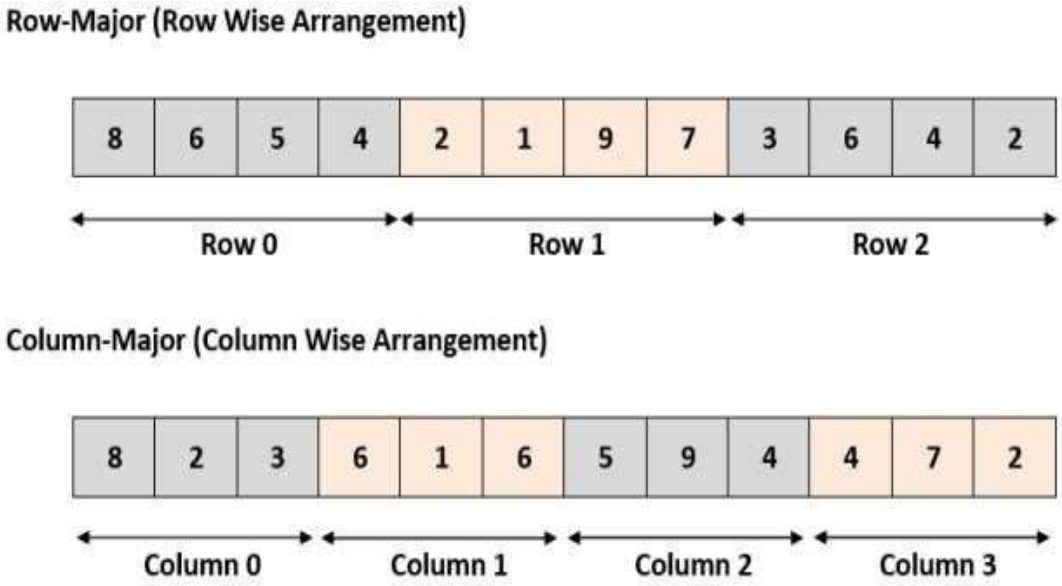
Memory Address Calculation in an Array

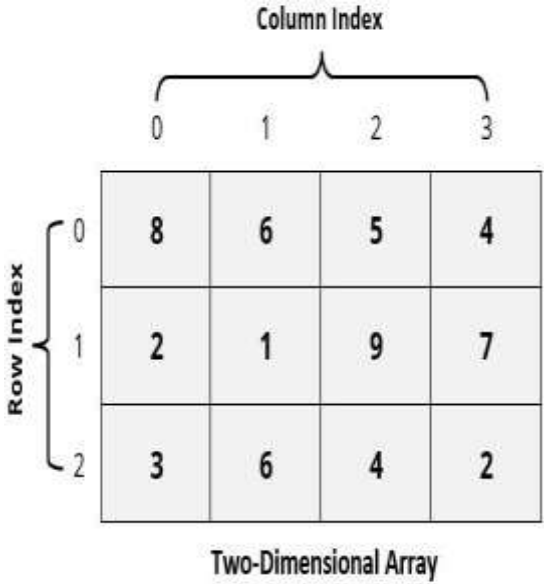
Array of an element of an array say “A[I]” is calculated using the following formula:

Address of A [I] = B + W * (I – LB)
Where,
B = Base address
W = Storage Size of one element stored in the array (in byte)
I = Subscript of element whose address is to be found
LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)
Example:
Given the base address of an array **B[1300…..1900]** as 1020 and size of each element is 2 bytes in the memory. Find the address of **B[1700]**.
Solution:
The given values are: B = 1020, LB = 1300, W = 2, I = 1700
Address of A [I] = B + W * (I – LB)
= 1020 + 2 * (1700 – 1300)
= 1020 + 2 * 400
= 1020 + 800
= 1820 **[Ans]**

Address Calculation in Double (Two) Dimensional Array:

While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations. Therefore, a 2-D array must be linearized so as to enable their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.





Address Calculation in Double (Two) Dimensional Array:

Address of an element of any array say "**A[I][J]**" is calculated in two forms as given:
(1) Row Major System (2) Column Major System

Row Major System:

The address of a location in Row Major System is calculated using the following formula:

Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

Column Major System:

The address of a location in Column Major System is calculated using the following formula:

Address of A [I][J] Column Major Wise = B + W * [(I – Lr) + M * (J – Lc)]

Where,

B = Base address
I = Row subscript of element whose address is to be found
J = Column subscript of element whose address is to be found
W = Storage Size of one element stored in the array (in byte)
Lr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
Lc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)
M = Number of row of the given matrix
N = Number of column of the given matrix

Address Calculation in Double (Two) Dimensional Array:

Important:
Usually number of rows and columns of a matrix are given (like A[20][30] or A[40][60]) but if it is given as **A[Lr- – – – – Ur, Lc- – – – – Uc]**. In this case number of rows and columns are calculated using the following methods:
Number of rows (**M**) will be calculated as = **(Ur – Lr) + 1**
Number of columns (**N**) will be calculated as = **(Uc – Lc) + 1**
And rest of the process will remain same as per requirement (Row Major Wise or Column Major Wise).

Examples:

Q 1. An array X [-15…….10, 15… 40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:
As you see here the number of rows and columns are not given in the question. So they are calculated as:

Address Calculation –Column Major System

Q 1. An array X [-15…….10, 15… 40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:
As you see here the number of rows and columns are not given in the question. So they are calculated as:

Number or rows say
M = (Ur – Lr) + 1 = [10 – (- 15)] +1 = 26
Number or columns say

N = (Uc – Lc) + 1 = [40 – 15)] +1 = 26

Column Major Wise Calculation of above equation
The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, M = 26
Address of A [I][J] = B + W * [(I – Lr) + M * (J – Lc)]
= 1500 + 1 * [(15 – (-15)) + 26 * (20 – 15)]
= 1500 + 1 * [30 + 26 * 5]
= 1500 + 1 * [160]
= 1660 **[Ans]**

Address Calculation –Row Major System

Q 1. An array X [-15…….10, 15… 40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:
Row Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, N = 26
Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]
= 1500 + 1* [26 * (15 – (-15))) + (20 – 15)]
= 1500 + 1 * [26 * 30 + 5] = 1500 + 1 * [780 + 5]
= 1500 +785
= 2285 **[Ans]**

Operations of Arrays

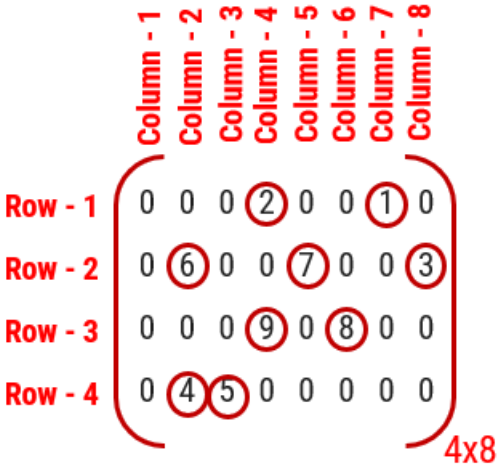
❑ Operation on Arrays

- Insertion of elements
- Deletion of elements
- Traversal
- Searching
- Merging

Sparse matrix and its representation

Spars Matrix and its representation

- An m x n matrix is said to be ***sparse*** if “many” of its elements are zero.
- A matrix that is not sparse is called a ***dense matrix***.
- **Example:**



Terms	0	1	2	3	4	5	6	7	8
Row	1	1	2	2	2	3	3	4	4
Column	4	7	2	5	8	4	6	2	3
Value	2	1	6	7	3	9	8	4	5

Linear Representation of given matrix

Polynomial matrix and its representation

- Matrix representation of polynomial equation
  - We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc…
  - Array can be used to represent Polynomial equation.

Polynomial matrix and its representation

- **Example:**

	Y	Y^2	Y^3	Y^4
X	XY	XY^2	XY^3	XY^4
X^2	X^2Y	X^2Y^2	X^2Y^3	X^2Y^4
X^3	X^3Y	X^3Y^2	X^3Y^3	X^3Y^4
X^4	X^4Y	X^4Y^2	X^4Y^3	X^4Y^4

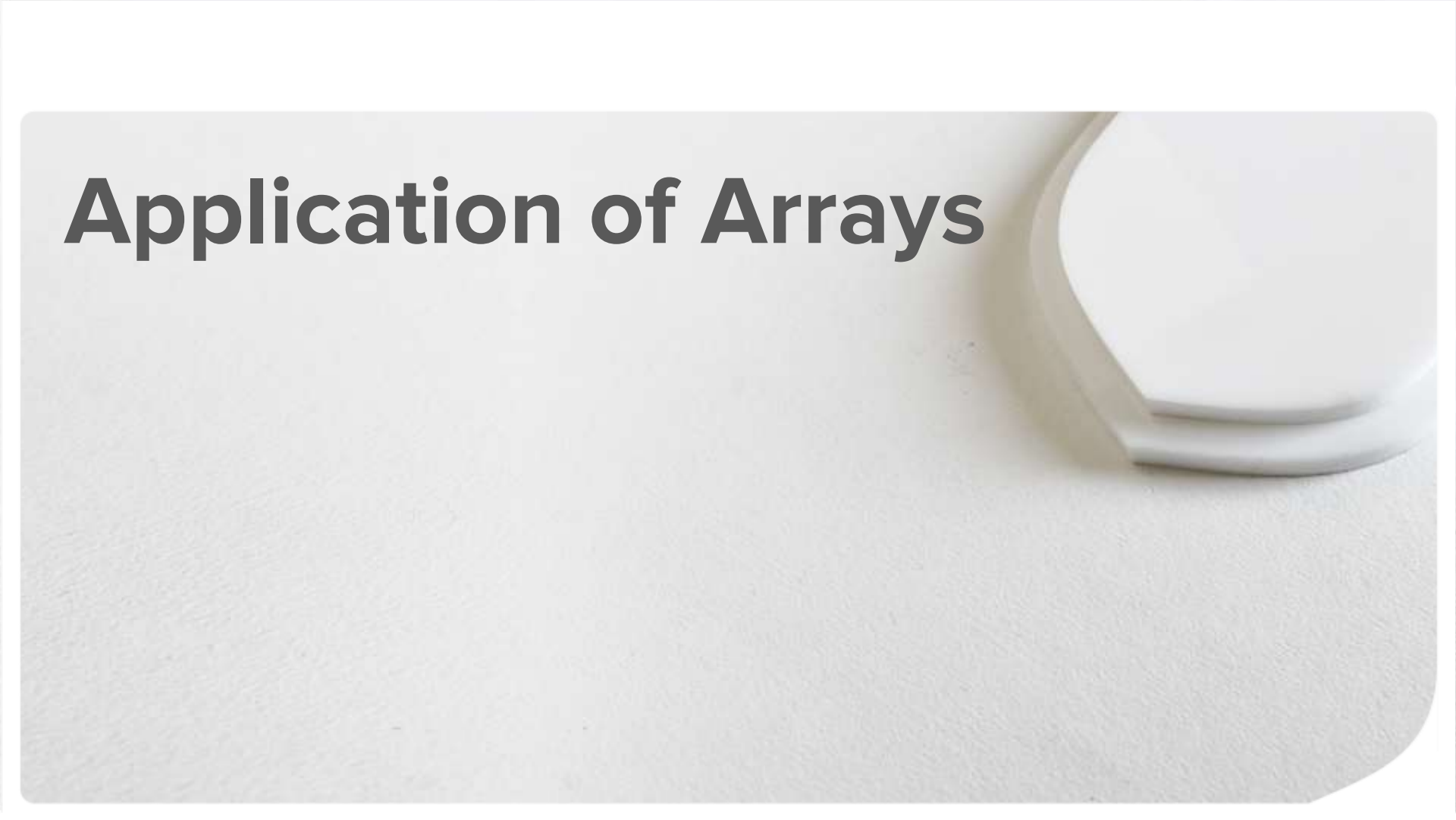
$2X^2 + 5XY + Y^2$

		Y	Y^2	Y^3	Y^4
	0	0	1	0	0
X	0	5	0	0	0
X^2	2	0	0	0	0
X^3	0	0	0	0	0
X^4	0	0	0	0	0

$X^2 + 3XY + Y^2 + Y - X$

		Y	Y^2	Y^3	Y^4
	0	1	1	0	0
X	-1	3	0	0	0
X^2	1	0	0	0	0
X^3	0	0	0	0	0
X^4	0	0	0	0	0

Application of Arrays



Application of Arrays

- To perform **arithmetic operation** on polynomial equation.
- Widely used to implement **mathematical vectors, matrices** and other kinds of **rectangular tables**.
- Used to implement **stack, queue, heap, hash table, string**, etc..
- Can be used for **dynamic memory allocation**.

Drawbacks of Linear Arrays

Fixed Size:
A primary limitation is their fixed size. Once an array is declared with a specific size, it cannot be dynamically increased or decreased during runtime. If more elements are needed, a new, larger array must be created and all existing elements copied over, which can be inefficient.

Inefficient Insertions and Deletions:
Because elements are stored in contiguous memory locations, inserting or deleting an element in the middle of an array requires shifting all subsequent elements. This can be a time-consuming operation, especially for large arrays.

Memory Wastage:
If an array is declared with a size larger than the actual number of elements stored, the unused allocated memory space is wasted. Conversely, if the declared size is too small, it can lead to "array index out of bounds" errors or require costly resizing operations.

String

Strings are used for storing text/characters.

For example, "Hello World" is a string of characters.

Unlike many other programming languages, C does not have a String type to easily create string variables. Instead, you must use the char type and create an array of characters to make a string in C

```
char greetings[] = "Hello World!";
```

To output the string, you can use the printf() function together with the format specifier %s to tell C that we are now working with strings:

Example

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

Memory Address

When a variable is created in C, a memory address is assigned to the variable.

The memory address is the location of where the variable is stored on the computer.

When we assign a value to the variable, it is stored in this memory address.

To access it, use the reference operator (&), and the result represents where the variable is stored:

Example

```
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

Note: The memory address is in hexadecimal form (0x..). You will probably not get the same result in your program, as this depends on where the variable is stored on your computer.

You should also note that &myAge is often called a "pointer". A pointer basically stores the memory address of a variable as its value. To print pointer values, we use the %p format specifier

Pointer

Creating Pointers

Example

```
int myAge = 43; // an int variable

printf("%d", myAge);  // Outputs the value of myAge (43)
printf("%p", &myAge); // Outputs the memory address of myAge (0x7ffe5367e044)
```

A pointer is a variable that stores the memory address of another variable as its value.

A pointer variable points to a data type (like int) of the same type, and is created with the * operator.

The address of the variable you are working with is assigned to the pointer:

Pointer Example

Example

```
int myAge = 43;     // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that stores the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Example explained

Create a pointer variable with the name ptr, that points to an int variable (myAge). Note that the type of the pointer has to match the type of the variable you're working with (int in our example).

Use the & operator to store the memory address of the myAge variable, and assign it to the pointer.

Now, ptr holds the value of myAge's memory address.

Pointers and Arrays

You can also use pointers to access arrays.

Consider the following array of integers:

Example
```
int myNumbers[4] = {25, 50, 75, 100};
```
You learned from the arrays chapter that you can loop through the array elements with a for loop:

Example
```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", myNumbers[i]);
}
```
Result:

```
25
50
75
100
```

Pointers and Arrays

Instead of printing the value of each array element, let's print the memory address of each array element:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
  printf("%p\n", &myNumbers[i]);
}
```

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
```

Note that the last number of each of the elements' memory address is different, with an addition of 4.

It is because the size of an int type is typically 4 bytes, remember:

Pointers and Arrays

Example

```
// Create an int variable
int myInt;

// Get the memory size of an int
printf("%zu", sizeof(myInt));
```

Result:

4

So from the "memory address example" above, you can see that the compiler reserves 4 bytes of memory for each array element, which means that the entire array takes up 16 bytes (4 * 4) of memory storage:

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the size of the myNumbers array
printf("%zu", sizeof(myNumbers));
```

Result:

16

Pointers Related to Arrays

The memory address of the first element is the same as the name of the array:
Example
```
int myNumbers[4] = {25, 50, 75, 100};

// Get the memory address of the myNumbers array
printf("%p\n", myNumbers);

// Get the memory address of the first array element
printf("%p\n", &myNumbers[0]);
```

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f0
```

This basically means that we can work with arrays through pointers!
How? Since myNumbers is a pointer to the first element in myNumbers, you can use the * operator to access it:

Example
```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the first element in myNumbers
printf("%d", *myNumbers);
```
Result:
25

Pointers Related to Arrays

To access the rest of the elements in myNumbers, you can increment the pointer/array (+1, +2, etc):

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the second element in myNumbers
printf("%d\n", *(myNumbers + 1));

// Get the value of the third element in myNumbers
printf("%d", *(myNumbers + 2));

// and so on..
```

Result:

```
50
75
```

Pointers Related to Arrays: print array elements using loop:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", *(ptr + i));
}
```

Output:

```
25
50
75
100
```

Structure

Structure

- Structures (also called structs) are a way to group several related variables into one place.
- Each variable in the structure is known as a member of the structure.
- Unlike an array, a structure can contain many different data types (int, float, char, etc.).

Create a Structure

- You can create a structure by using the struct keyword and declaring each of its members inside curly braces:

```
struct MyStructure {   // Structure declaration
 int myNum;           // Member (int variable)
  char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

Structure

To access the structure, you must create a variable of it.

Use the struct keyword inside the main() method, followed by the name of the structure and then the name of the structure variable:

Create a struct variable with the name "s1":

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

Access Structure Members

To access members of a structure, use the dot syntax (.):

Example

```
// Create a structure called myStructure
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  // Create a structure variable of myStructure called s1
  struct myStructure s1;

  // Assign values to members of s1
  s1.myNum = 13;
  s1.myLetter = 'B';

  // Print values
  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);

  return 0;
}
```

Create a Multiple Structure variable

Example

```
// Create different struct variables
struct myStructure s1;
struct myStructure s2;

// Assign values to different struct variables
s1.myNum = 13;
s1.myLetter = 'B';

s2.myNum = 20;
s2.myLetter = 'C';
```

Strings in Structures

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];  // String
};

int main() {
  struct myStructure s1;

  // Trying to assign a value to the string
  s1.myString = "Some text";

  // Trying to print the value
  printf("My string: %s", s1.myString);

  return 0;
}
```

Output:
An error will occur:
prog.c:12:15: error: assignment to expression with array type

Solution for String in Structure

There is a solution for this! You can use the strcpy() function and assign the value to s1.myString, like this:

Example

```
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30]; // String
};
int main() {
  struct myStructure s1;
 // Assign a value to the string using the strcpy function
  strcpy(s1.myString, "Some text");
  // Print the value
  printf("My string: %s", s1.myString);
return 0;
}
```

Result:

My string: Some text

String in Structure without strcpy function

You can also assign values to members of a structure variable at declaration time, in a single line. Just insert the values in a comma-separated list inside curly braces {}. Note that you don't have to use the strcpy() function for string values with this technique:

Example

```
// Create a structure
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};
int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};
  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);
  return 0;
}
```

Structure and Pointers

You can use pointers with structs to make your code more efficient, especially when passing structs to functions or changing their values.

To use a pointer to a struct, just add the * symbol, like you would with other data types.

To access its members, you must use the -> operator instead of the dot . syntax:

Structure and Pointers

Example

```
// Define a struct
struct Car {
  char brand[50];
  int year;
};
int main() {
  struct Car car = {"Toyota", 2020};
  // Declare a pointer to the struct
  struct Car *ptr = &car;
  // Access members using the -> operator
  printf("Brand: %s\n", ptr->brand);
  printf("Year: %d\n", ptr->year);

  return 0;
}
```

Passing Struct Pointers to Functions

Here's how you can pass a struct pointer to a function and change its values:

```
struct Car {
  char brand[20];
  int year;
};// Function that takes a pointer to a Car struct and updates the year
void updateYear(struct Car *c) {
  c->year = 2025;  // Change the year
}
int main() {
  struct Car myCar = {"Toyota", 2020};
  updateYear(&myCar);  // Pass a pointer so the function can change the year
  printf("Brand: %s\n", myCar.brand);
  printf("Year: %d\n", myCar.year);

  return 0;
}
```

Why Use Struct Pointers?

Using pointers with structs is helpful when:

1. You want to avoid copying large amounts of data.

Instead of copying a whole struct, you can just pass a pointer. This makes your program faster and uses less memory.

2. You want to change values inside a function.

If you pass a pointer to a struct into a function, the function can change the original values.

3. You want to create structs dynamically using memory allocation.

With pointers, you can use malloc() to create structs while the program is running.


Thank
You