Data Structure and algorithm

Topics

① → Time and Space Complexity
② → Basic Data structures
→ String and Array
→ Linked List
→ Stack & Queue
③ Advanced Data structure
→ Trees
→ Heap
→ Graphs
④ Basic Algorithms
→ Sorting
• Bubble, Selection, Insertion
• Merge, Quick
→ Searching
• Linear & Binary Search
• DFS & BFS in Graph
⑤ Advanced Algorithms
→ Dynamic
• overlapping Sub Problems
• Memoization
• Tabulation
→ Greedy
• Concepts Examples

→ Other Topics -
• Back tracking
• Shortest Path
• Minimum Spanning tree

Date:
P. No:

DSA

→ DSA Prep → Problem Solving, logic solving, Patterns, Interview Prep.

→ Big O → → Space & time complexity, scalability, rules and graph, few exercises to practice

→ Strings → · Basic Nature, Immutable nature, Indexing, Iterating, Concatenation & Substring Extraction
· Case Conversions, Trimming, Split & join (with Array), replace characters, substring, Reversing, Palindromes, Anagram, Longest Substring
· Leetcode easy

→ Array → · Understand Array
· Dynamic vs Static Arrays
· Memory Allocation (Access & Storage)
· Operations
→ Accessing & Updating Elements
→ Insertion, Deletion (End, Start, middle)
→ Forward, Backward traversal
→ Multi-dimensional
· Access, Matrics
→ Manipulate
· Reverse
· Rotate
· Shuffle
· Rad

Date:
P. No:

→ Patterns →

- Two pointers, Sliding Window, Sub Array
- Searching & Sorting
  - Bubble, Selection, Insertion, Merge, Quick
  - Linear Search, Binary Search

→ Leet code (easy, medium)

Linked List → → Nodes & Pointers. Comparision with array

→ Singly, Doubly, Circular

→ Implementation

- Node with Data & Pointers
- Constructors
- Head/Tail Reference
- Utility functions (isEmpty, size)
- Insertion, Deletion, Searching

→ Advance techniques

- Two pointers
- Reverse
- Merge & Split

→ Patterns

- Detecting Cycle
- Finding Mid
- Nth node from End

→ Leet Code

→ Stack & Queue →

- → Stack (LIFO)
- → Queue (FIFO)
- → Basic Operation
  - Push, Pop, peek
- → Basic operation
  - Circular
  - Add, Remove, Front
  - Underflow & Overflow
- → BFS, Round Robin
- → Balancing Symbol, Reverse a Queue
- → Thread-safe Implementation
- → Leet Code

Data Structures & Algorithms

Date:
P. No:

- **Data Structure :-**
  Data structures are specialized formats for organizing and storing data to perform operations efficiently.

- **Algorithms :-**
  Algorithms are step-by-step procedures or formulas for solving problems and performing tasks.

Abstract Data Type

- An abstract data type (ADT) is an abstraction of a data structure which provides only the interface to which a data structure must adhere to.

- The interface does not give any specific details about how something should be implemented or in what programming language.

Ex Example of Abstract Data Type in DS are list, stack, queue etc.

Date:
P. No:

Example

ListADT interface :-

```
public interface ListADT {
    void add(int element);
    boolean contains(int element);
}
```

Arraylist Implementation :-

```
import java.util.ArrayList;

public class ArrayListWrapper implements
ListADT {
    private ArrayList<Integer> elements;

    public ArrayListWrapper() {
        elements = new ArrayList<>();
    }

    public void add(int element)
    {
        elements.add(element);
    }
```

Date:
P. No:

```
public boolean contains(int element){
    return elements.contains(element);
}

psvm(){
    ListADT list = new ArrayListWrapper();
    list.add(1);
    list.add(2);

    SOP(list.contains(1)); // true
    Sop(list.contains(3)); // false
```

Explanation:-

- ListADT serves as the abstract data type interface specifying operations like 'add' and 'contains'.

- You can implement another class like 'LinkedWrapper' like 'ArrayListWrapper' using different internal structures (ArrayList and LinkedList) but adhere to the same abstract interface

Conclusion:- Users of 'ListADT' interface can use the same set of operations ('add' and 'contains') regardless of whether the

specification.

Date:
P. No:

→ stack ~~or~~ a underlying implementation is an Arraylist or LinkedList.

Complexities in DSA

Apna College :-

1. Time and Space Complexity :-

Time Complexity :-

- An ~~web~~ app or a web site is as efficient as the space ~~or~~ and time it takes
- Relation between Input size and Running Time (Operation)

Suppose :-

```
main(){
    print("hello");
}
```

```
main(){
    print("hello"); print("Hello");
}
```

- In first there is one print statement and in second the number of operations increases
- ~~But~~ According to that the program will take the time to run.

- But these times of very small operations are nearly negligible. But it is when these operations become huge than it is very important to consider them.

- These operations depend on the input size.

Now Suppose:-

scanner → input a variable "n".
for(int i=0 to n)
{
    print("hello"); → (1)
}
Now here print is one operation but we are iterating it n times i.e.
the time
1 x n

- So the our code is going to take largely depends on input size.

- do in the above case → time ∝ input n
complexity

i.e.
as we increase the input n the time complexity will increase accordingly.

There are three ways to find time complexity:-

→ Best Case
→ Average Case
→ Worst Case

Suppose:- Numbers: 1, 2, 3, 4, 5
Search for '1'

① → On traversing through the element we will get 1 at position = 1 only.

→ So it took us 1 operation
i.e. 1 unit of time

So in this case this is our best case scenario.

② Now there can be many possible combinations:-

21345 }
23415 } $\frac{1+2+3+4+5}{5}$

OR

$\frac{1+2+3 \ldots n}{n} \Rightarrow \frac{n(n+1)}{2 \times n} \Rightarrow \frac{(n+1)}{2}$

$\frac{n+1}{2}$ will be the time in the average case.

③ Worst Case

5,4,3,2,1

So here time $\propto n$

$\Rightarrow$ To find n this will be maximum no. of iterations.

→ Worst case → that our code will not exceed this time in any way.

To represent these there are some fixed ways.

i.e.

Best case → $\Omega(1)$ (omega of 1)

Average case → $\Theta(\frac{n+1}{2})$ (theta of $(\frac{n+1}{2})$)

Worst case → $O(n)$ (Big-O of n)

We will always take worst case.

Examples

①
```
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        sop("Hello");
    }
}
```

Worst Case :- $O(n \times n)$

↓

ds for every n the loop iterates n times.

→ Suppose n = 5

i = 0 → j = 0, 1, 2, 3, 4, 5

i = 1 → j = 0, 1, 2, 3, 4, 5

:

:

so on

(2)

```
for(int i=0; i<n; i++)
{
  for(int j=0; i<m; i++)
  { sop("Hello");
  }
}
```

time complexity (worse case) → $O(n \times m)$

(Big O of n x m)

(3)

```
for(int i=0; i<n; i++){}
for(int j=0; j<m; j++){}
```

time complexity → $O(n+m)$

Date:
P. No:

1

Time complexity :-

Definition :-

Measures of the amount of time an algorithm takes to compute as a function of the size of its input.

Cases

Types of Time complexity :-

→ Best case :-

The minimum time required by an algorithm to complete.

→ Average case :-

The average time an algorithm takes over all possible inputs

→ Worst case :-

The maximum time an algorithm takes over all possible inputs.

Notation (Big-O) :-

→ Time complexity is often expressed using Big-O notation.

→ It describes upper bound of the growth rate.

UpperBound :- The maximum amount of time or space an algorithm might need for the largest possible in

CRM tools i.e.
specification

Date:
P. No:

→ stack

Types of Time Complexity:-

1. Constant Time Complexity (O(1)):-
   → Algorithms with constant time complexity have execution times that do not depend on the size of input

Example:-

```
int constantTimeOperation(int[] arr)
{ return arr[0]; // O(1)
}
```

2. Linear Time Complexity (O(n)):
   → The execution time of the algorithm grows linearly with the size of the input

Example:-

```
void linearTimeOperation(int[] arr)
{
    for(int num : arr) {
        // some operation
    }
}
```

3. Logarithmic Time Complexity (O(log n)):
→ Algorithms with logarithmic time complexity often divide the problem in each step, reducing the size of the input significantly.

Example:-

```
int binarySearch(int[] arr, int target)
{
    // Binary search implementation
}
```

4. Quadratic Time Complexity (O(n^2)):-
→ Algorithm with quadratic time complexity have execution times proportional to the square of the size of the input.

Example:-

```
for(int i=0; i<arr.length; i++){
for(int j=0; j<arr.length; j++){}}
```

5. Polynomial Time Complexity (O(n^k)):-
→ Generalization of quadratic time complexity to higher degrees.

Example:
```
void polynomialTimeOperation(int[] arr)
{
// Nested loops or recursive calls leading to higher degrees.
}
```

6. Exponential Time Complexity (O(2^n)):
→ Algorithms with exponential time complexity often involve recursive solutions where the problem size grows exponentially.

Example:-
```
int exponentialTimeOperation(int n){
    if (n<=1) return n;
    return exponentialTimeOperation(n-1) + exponentialTimeOperation(n-2);
}
```

Asymptotic notations are classified into three types :-
1. Big-Oh (O) notation → For worst case
2. Big-Omega (Ω) notation → For Best case
3. Big-Theta (Θ) notation → For Best case

Date
P. No.

⑥ Space Complexity

Definition:-
Measure of the amount of memory space an algorithm uses with respect to its input size.

→ • Auxiliary Space Complexity →
The extra space use by the algorithm, excluding the input space.

Auxiliary Space Complexity = Total space comp - input space comp

→ • Total Space Complexity :-
TSC = Auxiliary Space Complexity + Input Space Complexity.

• Types of Space Complexity :-

1. Constant Space Complexity (O(1)):-
→ Algorithms with constant Space complexity use a fixed amount of memory regardless of the input size.

Date:
P. No:

Example:-

```
void constantSpaceOperation(int[] arr)
{ int sum = 0; // O(1) space
  for(int num : arr) {
      sum += sum;
  }
}
```

2. Linear space complexity (O(n));
→ Algorithms with linear space complexity use memory that grows linearly with the input size.

Example:-

```
int[] linearSpaceOperation(int[] arr)
{
  int[] result = new int[arr.length];
                 // O(n) space
  // some operation
  return result;
}
```

Logarithmic space complexity (O(log n));

Date:
P. No:

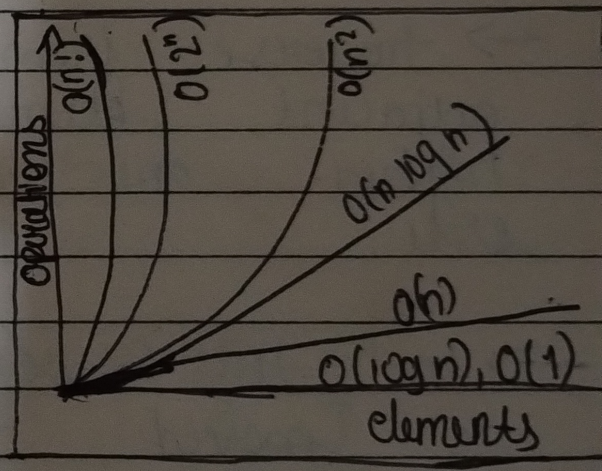
⇒ Algorithms with logarithmic space complexity use memory that grows logarithmically with the input size.

example :-

```
int logarithmicSpaceOperation (int n) {
    if (n<=1) return n;
    return logarithmicSpaceOperation(n-1) +
           logarithmicSpaceOperation(n/2);
}
```

Big O Notation

- O(1) : Constant Time
- O(log n): Logarithmic Time
- O(n): Linear Time
- O(n log n): Linearithmic Time
- O(n²): Quadratic Time
- O(2ⁿ): Exponential Time
- O(n!): Factorial Time



Date:
P. No:

Sorting Algorithms

1. Bubble Sort:-

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- Not suitable for large data sets as its average and worst-case time complexity is quite high (n^2).

In Bubble Sort Algorithm:-

→ traverse from left and compare adjacent elements and the higher one is placed at right side.

→ In this way, the largest element is moved to the right most end at first.

→ This process is then continued to find the second largest and place it and so on until the data is sorted.

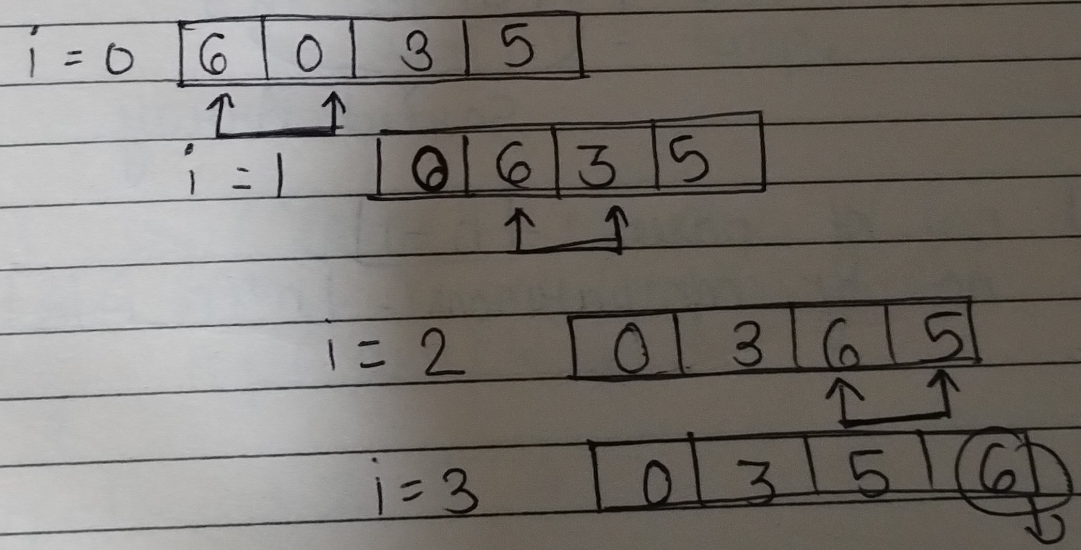
• How bubble Sort Work?

input : arr[] = {6, 3, 0, 5}

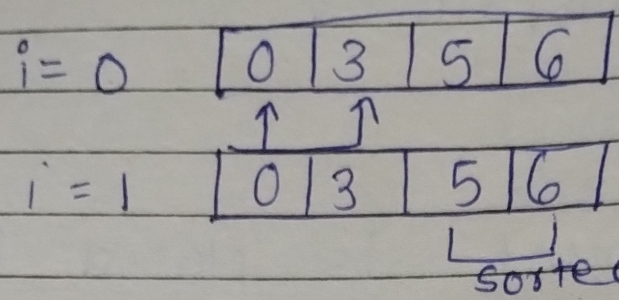
First Pass :-

The largest element is placed in its correct position i.e, the end of the array.

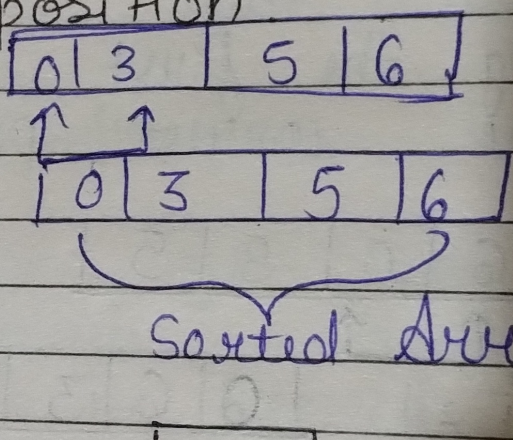
Step 1 Placing the first largest elment at the correct position



Step 2 Placing 2nd largest element at correct position



Step 3 Placing 3rd largest element at correct position



Total no. of passes :- n - 1

Total no. of comparisons :- n*(n-1)/2

Date:
P. No:

code :-

```
Class Main{

    psvm(){
        int i, j, temp, count = 0;
        int arr[] = {2,3,1,4};
        for(i=0; i< arr.length-1; i++)
        {
            for(j=0; j< arr.length-i-1; j++)
            {
                sop(count++);
                if(arr[j] > arr[j+1])
                {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
        sop("count" + count); // 6
        for(int k=0; k< arr.length; k++){
            sop(arr[k]);
        }
    }
```

Output :- 1, 2, 3, 4.

Complexity Analysis :-

→ Time Complexity :- $O(N^2)$

→ Auxiliary Space :- $O(1)$

Date:
P. No:

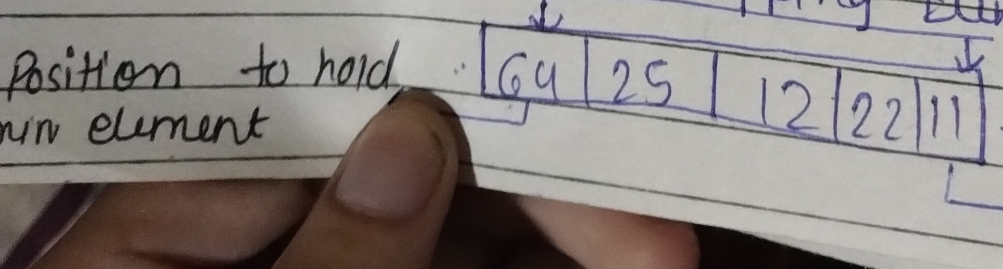
2. Selection Sort :-
→ Simple but inefficient for large datasets.
→ Time Complexity : $O(n^2)$ in the worst case.

In Selection Sort :-

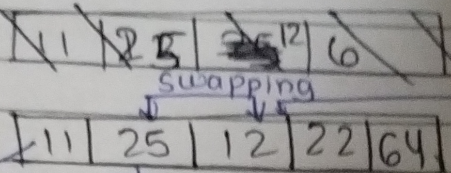
→ The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part.
→ This process is repeated for the remaining unsorted portion until the entire list is sorted.

How doesn't it work?

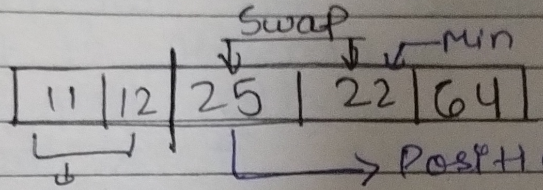
Step 1 Swaping first element with the minimum in array.



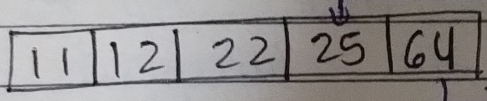
Step 2



Step 3



Step 4



Step 5 Sorted Array :- 11 | 12 | 22 | 25 | 64

Code :-

```
for (int i = 0; i<n-1; i++)
{
    int min = i;
    for (int j = i+1; j<n; j++)
        if (arr[j] < arr[min])
            min = j;

    int temp = arr[min];
    arr[min] = arr[i];
    arr[i] = temp; }}
```

③ Insertion Sort

In Insertion Sort:-
→ To sort an array of size N in ascending order iterate over the array and compare the current (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Code:-

```
int arr[] = [5, 4, 3, 2, 1];
for(int i=1; i< arr.length; i++)
{
    int key = arr[i];
    int j = i-1;
    while( j>=0 && arr[j] > key)
    {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = key;
}
```