19-6-25:- DATA STRUCTURES AND ALGORITHMS:- 10:00 AM

## Analysis of Algorithm:-

→ Algorithm :- A step-by-step procedure to solve a problem

→ Analysis of Algorithm :- Determining the time/space requirements to solve a problem

## → Why Data structures & Algorithms :-

* Efficient data structure + optimal algorithm = high performance application.
* Helps to solve Complex Complex Problems faster & manage data efficiently.

## → Types of DS:-

* Linear : Array, Stack, queue, Linked list
* Non linear : Tree, graph
* Static : Array (Fixed size)
* Dynamic : Linked list (Flexible size).

## → Notation :-

→ T(n) → Time taken by an algorithm to solve an problem

→ S(n) → Space taken.

## → Framework for describing & analyzing algorithm:-

→ Input size identification

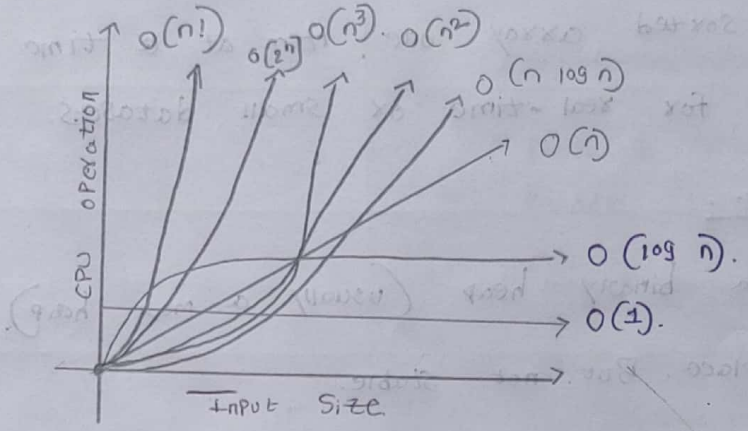
→ Basic operation selection.

→ Worst case, Average case.

→ Use recurrence relations for recursive algorithms.

## → Asymptotic Notations:-

* Used to express time complexity without constant factors.
* Big O (O) :- upper bound - worst case
* Omega (Ω) :- Lower bound - Best case
* Theta (Θ) :- Tight bound - Average case.




Ex:

```
for (i=0; i<n; i++)
```
↓
O(n).

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
```
↓
O(n²)

```
int fact(int n)
{
  if (n==0)
    return 1;
  return n * fact(n-1);
}
```
↓
O(2^n)

## SORTINGS:-

→ A sorting algorithm is used to rearrange a give array or list of elements in a order (Either in ascending/descending)

| Algorithms | Best case | Average case | Worst case | Space comple |
|---|---|---|---|---|
| Bubble sort | O(n) | O(n²) | O(n²) | O(1) |
| Insertion sort | O(n) | O(n²) | O(n²) | O(1) |
| Heap sort | O(n log n) | O(n log n) | O(n log n) | O(1) |
| Quick sort | O(n log n) | O(n log n) | O(n²) | O(log n) |
| Merge sort. | O(n log n) | O(n log n) | O(n log n) | O(n). |

→ Bubble sort :-

* Repeatedly swaps adjacent elements if they are in wrong order
* Optimized version can stop early if no swaps are made in a pass.

→ Insertion sort :-

* Builds sorted array one item at a time
* Good for real-time or small datasets.

→ Heap sort :-

* Use a binary heap (usually a max heap).
* In-place. But not stable.

→ Quick sort :-

* use divide & conquer by choosing a pivot.
* Worst case can be avoided by using random or median pivot.

→ Merge sort :-

* Recursively splits array, then merges sorted subarrays
* Requires extra space for merging.

## Array Traversal & representation :-

* Array representation in memory

→ Arrays are stored in contiguous memory blocks
→ For an array arr[i] :

Address of arr[i] = Base Address + i * size of element

Ex: For int (size = 4 bytes).

Base Address = 100 then arr[3] = 100 + 3 * 4 = 1012

## * Measuring Time Complexity of Traversal.

```
for (int i=0; i<n; i++)
    System.out.println(arr[i]);
```

TC: O(n)
SC: O(1).

## SEARCHING:

### Linear Search

* checks each element one by 1

* Time Complexity:-

→ BC :- O(1)
→ AC :- O(n/2) ≅ O(n)
→ WC :- O(n)

* Space Complexity :- O(1).

```
int search (int arr[], int n, int key)
{
  for (int i=0; i<n; i++)
      if (arr[i] == key)
          return i;
  return -1;
}
```

### Binary search:

* Divide & conquer by comparing middle element

* Time Complexity

* BC: O(1)
* AC: O(log n)
* WC: O(log n)

* Space :- O(1).

```
int BS (int arr[], int low, int high, int k
{
  while (low <= high)
  {
    int mid = (low+high)/2.
    if (arr[mid] == k)
        return mid.
    else if (arr[mid] < key)
        low = mid+1;
    else
        high = mid-1.
  }
  return -1.
}
```

# LINKED LIST :-

## Types of Linked List :-

| Type | Description |
|---|---|
| Singly Linked list (SLL) | Each node Points to next node only |
| Circular singly Linked List (CSLL) | Last node Points Back to head |
| Doubly Linked list (DLL) | Each node has both next & pre Pt |
| Circular Doubly Linked list (CDLL) | Last node connects to head & vice versa. |

## Common Operations & Time Complexities.

| Operation | SLL | CSLL | DLL | CDLL |
|---|---|---|---|---|
| Traverse | O(n) | O(n) | O(n) | O(n) |
| Search | O(n) | O(n) | O(n) | O(n) |
| Insert at beginning | O(1) | O(1) | O(1) | O(1) |
| Insert at end | O(n) | O(n) | O(n) | O(n) |
| Delete from beginning | O(1) | O(1) | O(1) | O(1) |
| Delete from End | O(n) | O(n) | O(n) | O(n) |