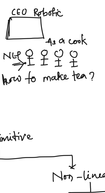
Data Structures and algorithms :-

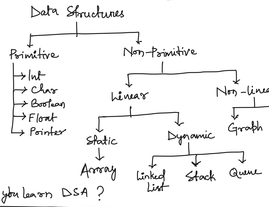
how to store, organize and process data efficiently.

Data Structures :- How to store data

Algorithms :- A sequence of steps followed to solve a problem.



Types of Data Structures :-



Why should you learn DSA ?

1. Improves your logic
2. Make your code efficient
3. Needed for Interviews
4. Foundations for AI, ML, DS, Gen AI, LLM models.

Real Life Examples of DSA :-

1. Any e-commerce website – linear DS, searching, sorting
2. ticket booking website – matrix
3. Google map – non-linear DS
4. Any search engine – linear DS - Stack

Complexity Measures :-

1. time complexity
2. Space Complexity

Algorithm Analysis :-

- Running time (speed)
- Memory usage
- Effort (number of operations)

Examples: check 19 is prime or not ?

```
a=19
(1) for(i=2; i<a; i++){
      if (a%i==0){
(16)    return False;
      }
    }
    return True;
```

```
(2) for (i=2; i<a/2; i++){
      -do-
(7) }
```

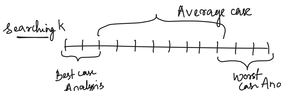
```
(3) for(i=2;
      i < math.sqrt(a);
(2)   i++)
      -do
    }
```

Complexity Categories :-

Best Case :- the min. time taken to solve a problem.

Average Case :- Expected Running time

Worst Case :- the max time take taken to solve a problem.




```
Fun( )
{
  ======
}
```

Input	
10	23 ms
100	37 ms
1000	45 ms
10000	52 ms
1000000	58 ms

Worst case Analysis

Asymptotic Notations :-

- a function's behaviour when input size n becomes very large.
- It helps to compare algorithms by ignoring machine-dependent constraints.

For Ex :- Algo-A — $T(n) = 3n+2$

Algo-B — $T(n) = n^2+3n$

$n = 10$

a) $T(n) = 3*10+2 = 32$ unit

b) $T(n) = (10)^2+3*10 = 130$ unit

Algo-A runs faster as compared to Algo-B.

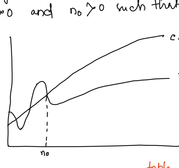
Major Acymptotic Notations :-

There are 5 major acymptotic notations.

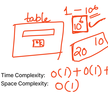
1) Big-O – (O) – Upper Bound
2) Big-Omega (Ω) – Lower Bound
3) Big-Theta (Θ) – Tight Bound
4) Little-o (o) – Non-tight Upper
5) Little Omega (ω) – Non-tight Lower Bound

Big O

- gives upper bound
- worst case time Complexity.
- Formally, $f(n) = O(g(n))$ if there exist constant $c > 0$ and $n_0 > 0$ such that $f(n) \leq c * g(n)$ for all $n \geq n_0$



Practice :-




```
1.  int add(int a, int b){
      int sum;          ——— O(1)
      sum = a+b;        ——— O(1)
      return sum;       ——— O(1)
    }
```

Time Complexity: $O(1)+O(1)+O(1) = O(1)$
Space Complexity: $O(1)$

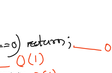
```
    void add( ){
      int a=10;         ——— O(1)
      int b=20, c;      ——— O(1)
      c = a+b;          ——— O(1)
      System.out.print(c);  ——— O(1)
    }
```

Time Complexity: $O(1)$
Space Complexity: $O(1)$

```
2.  void printItem(int[] arr){
      if (arr == null || arr.length ==0) return;  ——— O(1)
      System.out.print(arr[0]);  ——— O(1)
    }
```

Time Complexity: $O(1)$
Space Complexity: $O(1)$

```
3.  void sumArr(int n){
      int[] arr = new int[n];  ——— O(n)
      int sum;                 ——— O(1)
      for(int i=0; i<arr.length; i++)  ——— n(1)
        sum += arr[i];                 ——— O(n)
      System.out.print(sum);           ———
    }
```



Time Complexity: $O(n)$
Space Complexity: $O(n)$

```
4.  void Fun( ){
      for(int i=0; i<10; i++)
        System.out.println(i);
    }
```

10 times

Time Complexity: $O(1)$
Space Complexity: $O(1)$

```
5.  void Fun1(n){
      for(int i=1; i<n; i++)
        System.out.print(i);
    }
```

(n-1) times

Time Complexity: $O(n)$
Space Complexity: $O(1)$

```
6.  void Fun2(n){
      for(int i=1; i<n; i+=2)
        System.out.print(i);
    }
```

(n-1)/2 times

Time Complexity: $O(n/2) \approx O(n)$
Space Complexity: $O(1)$

```
7.  void Fun3(int m, int n){
      for(int i=0; i<m; i++) System.out.print(i);  ——— O(m)
      for(int j=0; j<n; j++) System.out.print(j);  ——— O(n)
    }
```

(m-1) times
+ (n-1) times
(m+n-2) times

Time Complexity: $O(m+n)$
Space Complexity: $O(1)$