Introduction (C++)

Data - Structures :-

DS is a particular way of storing and organizing data in a Computer so that it can be used efficiently. A DS is a special format for organizing and storing data.

1. Linear DS → Linked List, Array, Stack, Queues.
2. Non-Linear DS → Trees & graphs.

Algorithm :-

An algorithm is the step-by-step unambiguous instructions to solve a given problem.

→ Correctness
→ efficiency

Asymptopic Analysis :->

* The idea is to measure order of growth.
* Does not depend upon machine, programming language etc.
* No need to implement, we can analyze algorithms.

Example :-

add upto nth number.

```
int fun1 (int n) {
    return n * (n+1)/2;
}
```

Time Taken → C

```
int fun2 (int n) {
    int sum=0;
    for (int i=1; i<=n; i++) {
        sum += i;
    }
    return sum;
}
```

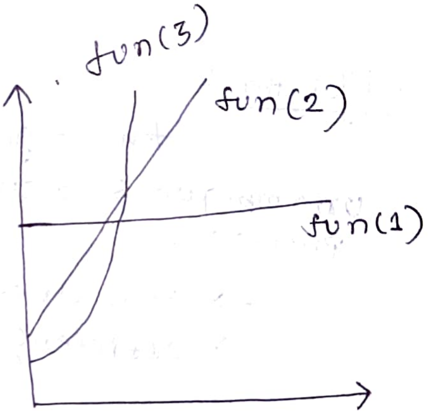
Time taken : $C_2 n + C_1$

```
int fun3 (int n) {
    int sum=0;
    for (int i=1; i<=n; i++) {
        for (int j=0; j<i; j++) {
            sum ++;
        }
    }
    return sum;
}
```

Time taken →

$C_4 n^2 + C_3 n + C_2$

fun (1) is most efficient here.



Order of Growth

A function f(n) is said to be growing faster that g(n) if

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

or,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Always Consider $n \to \infty$

(Input → ∞)

Direct way to find to Find and Compare growth! —

① Ignore lower order terms
② Ignore leading term Constant.

$f(n) = 2n^2 + n + 6$ order of growth → n^2 (Quadratic)
$f(n) = 5n + 8$ " " " $n \to n$ (Linear)

Some :→

$c < \log\log n < \log n < n^{1/3} < n^{1/2} < n < n^2 < n^3$

$< n^4 < 2^n < n^n$.

Sn'

$f(n) = c_1 \log n + c_2$
$g(n) = c_1 n + c_2 \log\log n + c_3$

$f(n) = \log n$ → good algo
$g(n) = n$ → bad algo

Best, Average and Worst Case

```
int getSum (int arr[], int n) {
    if (n%2==0) {
        return 0;
    }
    int sum = 0;
    for (int i = 0; i<n; i++) {
        Sum += arr[i]
    }
    return sum;
}
```

Best Case :- even no. (Constant)

Avg Case :- Linear

Worst Case :- Linear

Asymptotic Notations

Big O : Exact or upper

Theta θ : Exact

Omega : Exact or lower.

$\Theta(1)$
$O(1)$ > Constant.

$\Omega(1)$ → either Constant or bigger than 1 (Constant)

$O(n)$ → takes liner time or less than liner time.

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

$O(n)$
$\Omega(1)$

Big - O - Notation

$3n^2+5n+6 \rightarrow O(n^2)$

$3n+10n\log n+3 \rightarrow O(n\log n)$

$10n^3+40n+10 \rightarrow O(n^3)$

mathematically, $O(n)=O(n^2)=O(n^3)\ldots$

$\{100, \log 2000, 10^4, \ldots\} \in O(1)$

$\{\frac{n}{4}, 2n+3, \frac{n}{100}+\log n, n+10000+\ldots\} \in O(n)$

$\{n^2+n, 2n^2, n^2+1000n, n^2+n\log n, \ldots\} \in O(n^2)$

Multiple variable

$100n^2+1000m+n$: $O(n^2+m)$

$1000m^2+1200mn+30m+20n$: $O(m^2+mn)$

Amit

Some Common loops

(1)
```
for (int i=0; i<n; i=i+c) {
    some const work
}
```

loops run $\left(\frac{n}{c}\right)$ time

Time Complexity $\Theta(n)$

(2)
```
for (int i=n; i>0; i=i-c) {
    Some Const work
}
```

loops run $\left(\frac{n}{c}\right)$ times.

Time Complexity $\Theta(n)$

(3)
```
for (int i=1; i<n; i=i*c) {
    Some Const.
}
```

loops run $(\log_c n)$ times.

$c^0, c^1, \ldots \ldots c^{k-1}$

$c^{k-1} < n$

$k < \log_c n + 1$

time Comple → $\Theta(\log_c n)$

(4)
```
for (int i=1; i<n; i=i/c) {
    Const work.
}
```

Same as ago.

(5)
```
for (int i=2; i<n; i=pow(i,c)) {
    Const work
}
```

$\Theta(\log\log n)$

Analysis of Multiple Loops

(1)

```
void fun (int n)
{
    for (int i=0; i<n; i++) {          ] θ(n)

    }
    for (int i=0; i<n; i=i*2){         ] θ(logn)

    }
    for (int i=0; i<100; i++){         ] θ(1)

    }
}
```

Time Complexity => $\theta(n) + \theta(\log n) + \theta(1)$

$= \theta(n)$

(2)

```
void fun (int n) {
    for (int i=0; i<n; i++) {            θ(n)
        for (int j=1; j<n; j=j*2){       θ(logn)
            Const work.
        }
    }
}
```

T.C. :- $\theta(n \log n)$

(nesting loops → multiply)

(Consequent loops → add)

Analysis of Recursion

(1)

```
void fun (int n) {
    if (n <= 0)
        return;
    print ("GFG");
    fun (n/2);
    fun (n/2);
}
```

→ $n > 0$

$T(n) = T(n/2) + T(n/2) + \Theta(1) = 2T(n/2) + O(1)$

→ $n <= 0$

$T(n) = \Theta(1)$

$T(0) = \Theta(1)$

(2)

```
void fun (int n) {
    if (n <= 0)
        return
    for (int i=0; i<n; i++) {
        printf ("GFG");
    }
    fun (n/2);
    fun (n/3);
}
```

→ $n > 0$

$T(n) = T(n/2) + T(n/3) + \Theta(n)$

→ $n <= 0$

$T(n) = \Theta(1)$

3

```
void fun (int n) {
    if (n<=1)
        return;
    print("GFG");
    fun (n-1);
}
```

$T(n) = T(n-1) + \Theta(1)$

$T(1) = \Theta(1)$

Space Complexity

order of growth of memory space in terms of input type.

Auxiliary space:- order of growth of extra space or temporary space in terms of input size.