Time Complexity and Big O

* Time Complexity is the study of efficiency of algorithms.

- If is time taken to execute an algorithm grows with the size of inputs.

Ex :- Sorting of elements with 2 algorithms depending on no. of element n.

n	Algo 1	Algo 2
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2s	800 ms

This is difficult to find which ~~element~~ algorithm is better.

- In my opinion Algo 2 is better, as the no. of elements are increasing the time is increasing.

Let us assume formula for algorithms in terms of n are :-

Algo 1 :- $k_1 n^2 + k_2 n + k_3 \Rightarrow O(n^2)$

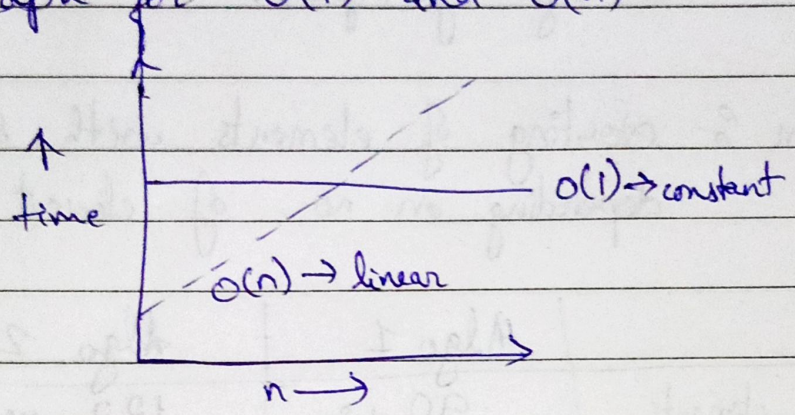
($k_1 n^2$: highest order term)

Algo 2 :- $k_1(n^0) + k_2 \Rightarrow O(n^0)$ or $O(1)$

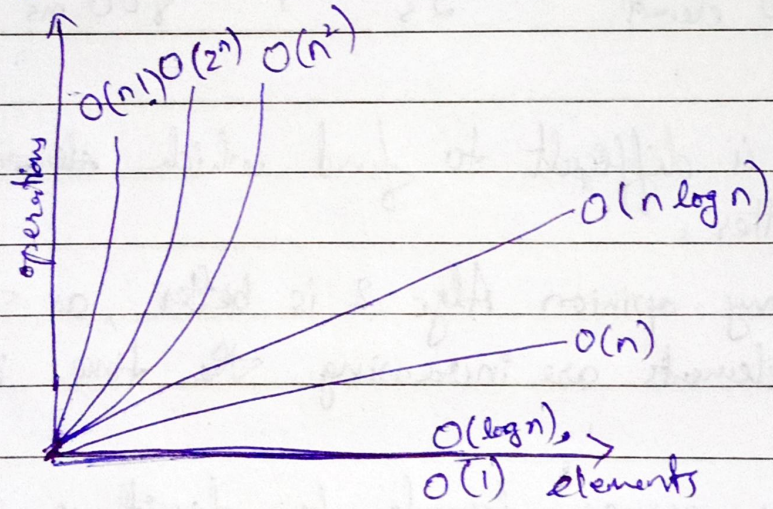
Constant.

where n is size of inputs.

Graph for O(1) and O(n)



Big - O - Complexity chart



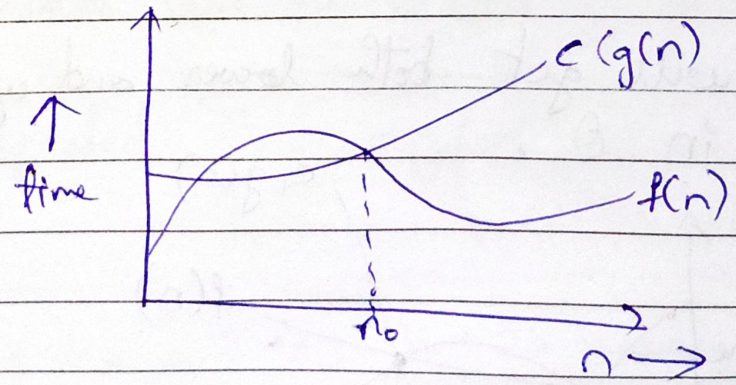
Asymptotic Notations

- It is used to compare a algorithm with other.
- It is
- Types :-
  1. Big Oh (O)
  2. Big omega (Ω)
  3. Big theta (Θ)

I. Big oh (O) :-

- If $f(n)$ describes the running time of an algorithm ; $f(n)$ is $O(g(n))$ if and only if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$
  Complexity for upper bond
- When we calculate Big O, if $f(n) = O(n^3)$ then it is $O(n^4)$ and $O(n^5)$, but we have to take minimum power.

Graphical Example for Big Oh (O)



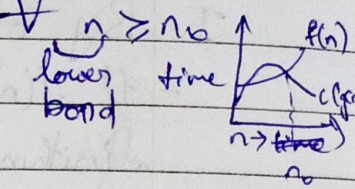
II. Big Omega (Ω)

- Reverse the $cg(n)$ & $f(n)$ of O.

- Let $f(n)$ define the running time of an algorithm $f(n)$ is said to be $\Omega(g(n))$ if and only if there exists positive constant c and n_0 such that

$$0 \leq c\,g(n) \leq f(n) \quad \forall\ n \geq n_0$$

lower bond



III. Big theta (Θ)

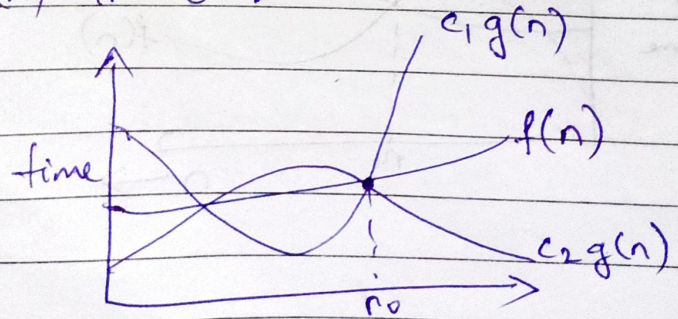
- A fun $f(n)$ is said to be $\Theta(g(n))$ if their exist c_1, c_2 & a constant n_0 such that

$$0 \leq f(n) \leq c_1 g(n) \quad \forall\ n \geq n_0$$
$$0 \leq c_2 g(n) \leq f(n) \quad \forall\ n \geq n_0$$

$$\Downarrow$$

$$0 \leq c_2 g(n) \leq f(n) \leq c_1 g_1(n)$$

- We will get both lower and upper bond for $f(n)$ in Θ.



If big Θ → big O
→ big Ω

- Try to give answer in big O, for time complexity and order of
- Better picture.

Increasing order of Common runtimes

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n^n$$

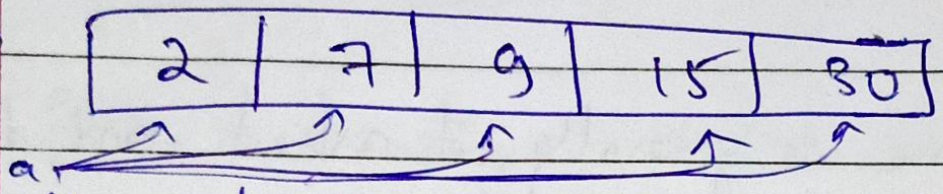
↑ Better — ↑ Common — ↑ worst

→

Best, Worst and Expected Case

Algo 1 (Search a in arr)

Sorted array (arr)



size (arr) = 5

If finding a = 2 is best case.
Similarly finding a = 30 is worst case.

Best Case → O(1) [$T_n = k$]
Worst Case → O(n) [$T_n = nk$]

Average / Expected Case = $O\left(\frac{\sum \text{all possible runtime}}{\text{total no. of possibility}}\right)$

(Ex) Avg case of $T = \frac{k + 2k + 3k + \cdots + nk + \underbrace{nk}_{\text{not present}}}{\underbrace{n+1}_{\text{element not present.}}}$

∵ k is time to search one element

$$\text{Avg case } T = \frac{k\left[(1+2+3+\cdots+n)+n\right]}{n+1}$$

$$= \frac{k\left[\frac{n(n+1)}{2}+n\right]}{n+1}$$

$$\frac{k\left[\frac{(n^2+n)+2n}{2}\right]}{n+1}$$ → Non dominating term (Graph)

$$= \frac{k\,n(n+1)}{2(n+1)}$$

$$= \frac{kn}{2} \Rightarrow k'n$$

$$\text{Avg case} = O(n)$$

⇒ Average case is generally not asked, and difficult.

Algo 2

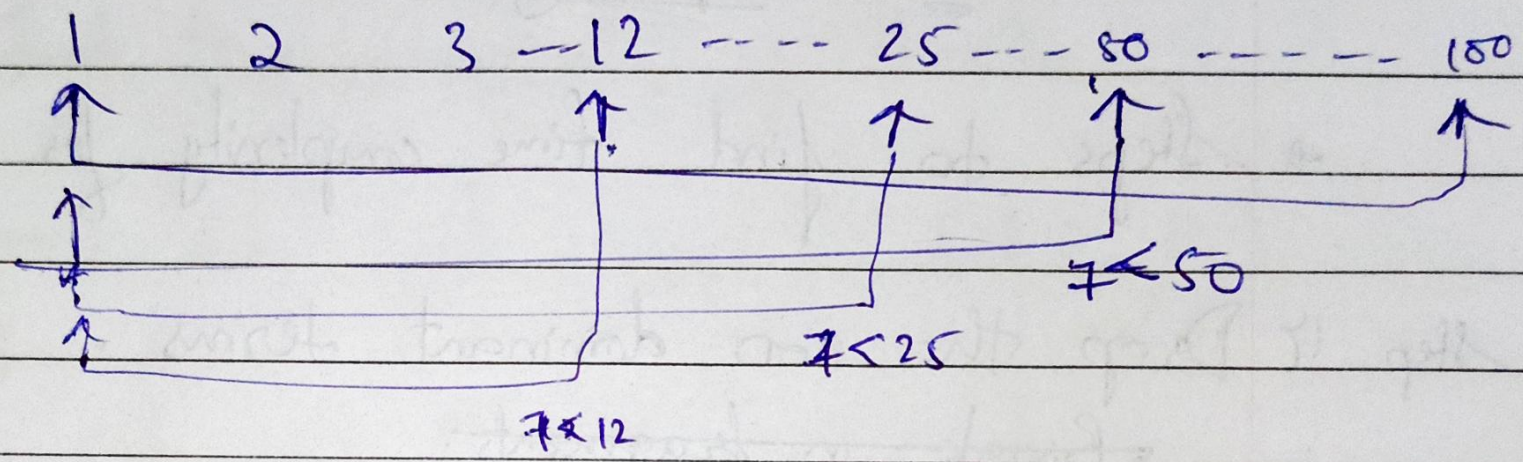
(Binary search)

2	7	9	15	30

Finding a in this first & last term, if not find goes the mid (In even any of 2 can be taken), then compare then do the same. with first array [lower ---- mid] if mid > a or with second array [mid+1 ---- upper]

mid < a

Eg.



first element is a

Best Case → O(1)

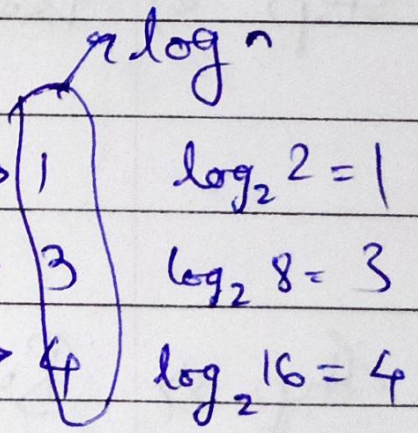
Worst Case → log(n)

Dividing elements

2 → 1

8 → 4 → 2 → 1

16 → 8 → 4 → 2 → 1



Space Complexity.

- Space is equally important with time for algorithm.

Ex. Creating an array of size n → O(n) space.

- If a fun calls itself recursively n times its space complexity is O(n)

- Memory allocation

- We can not calculate complexity in seconds as not everyones computer equally powerful..

- Asymptotic analysis is the measure of how time (run time) grows with input.

Time Complexity

* Steps to find time complexity for any algorith

Step 1) Drop the non dominant terms.

~~Break in fragments.~~

Like ignore declaring time (variables).

Step 2) Drop the constant term.

Step 2 ↑ Step 1 ↑

$T_n = k_2 n + k_1 \Rightarrow O(n)$

Step 3) Break the code into fragments :-

take time for every fragments.

for nested for, time complexity $O(n^2)$

```
Initializing 2 times.
for (i=0 ; i≤n ; i++)
{
    for (j=0 ; j≤n ; j++)
    {
        // code        } k2
    }
}
```

$T_n = k_1 + (k_2 n) n \Rightarrow O(n^2)$

Q. Find time complexity of func1

i)
```
void func1 (int array[], int length)
{
    int sum = 0;
    int product = 1;
    for (int i=0; i < length; i++)
    {
        sum += array[i];
    }
    for (int i=0; i < length; i++)
    {
        product *= array[i];
    }
}
```

$t_1 = k_1$

$t_2 = k_2 n$

$t_3 = k_3 n$

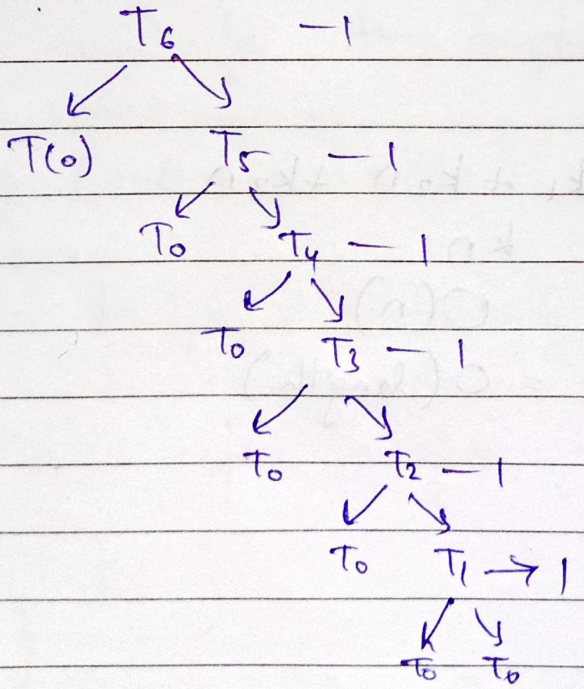
$T_n = k_1 + k_2 n + k_3 n$

$= kn$

$= O(n)$

$= O(length)$

ii)
```
int function (int n)
{
    int i;
    if (n<=0)
    {
        return 0;
    }
    else
    {
        i= random(n-1);  // It takes 1 unit of time. Any element in [0,n]
        cout << "this";
        return function(i)+function(n-1-i);
    }
}
```



$T_6 = 6$

$T_n = n(1) = n$

iii)
```
int isPrime (int n){
    if (n==1){
        return 0;
    }
    for(int i=2 ; i*i <n ; i++){
        if (n%i==0)
            retur 0;
    }
    return 1;
}
```

Negligible time neglected

Start $i=2$
$i=3$
$\vdots$
$i=[\sqrt{n}]$

$i^2 = n-1$

$\sqrt{n}$ - ~~X~~

$T_n = O(\sqrt{n})$