Geeks Man Algorithms Lesson 1

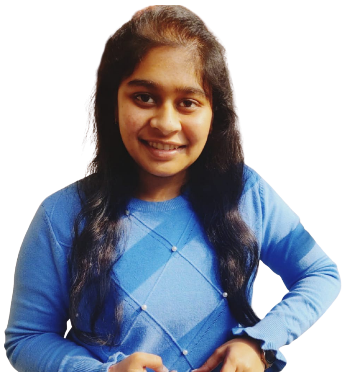
GeeksMan

We code therefore we are

Trilok Kaushik

Founder of Geeksman

Aditya Aggarwal



Aditya Gupta

Suhani Mittal

Team Coordinators

Design and Analysis of Algorithms:-

Problem -----> ALGORITHM ------> Program.

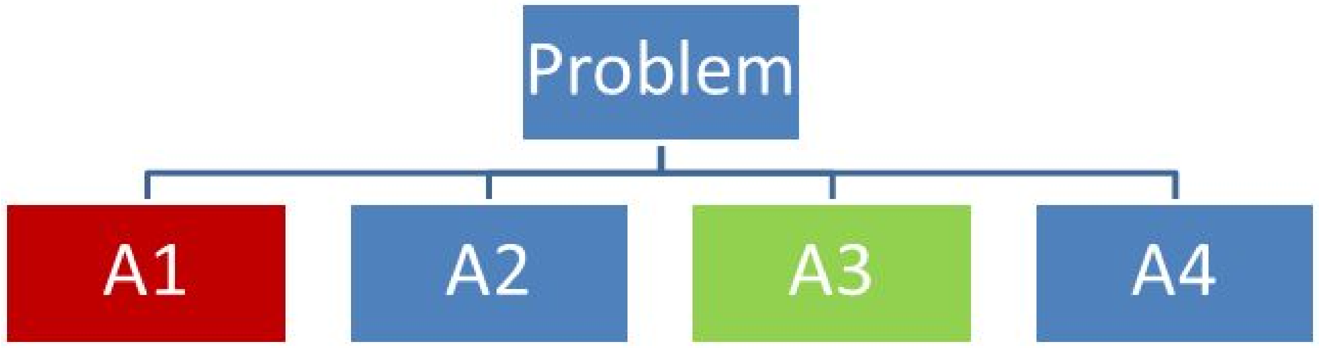
Data Structure :- It is a kind of container ie. how to store the elements in an efficient way such that manipulation with data gets faster . eg= Array, Linkedlist, Trees etc.

Algorithm:- A set of instructions in a human readable format in order to obtain the expected output from the given input.

Design:- Making Various types of algorithms for a given set of problems.

Analysis:- Calculating the efficient algorithm to perform a given task.

Implementation:- After Designing and Analysis, finally writing a program via computer language in order to execute the task from the computer.



In this above Diagram, suppose we have a Problem (P) and there are more than one algorithm present to solve the Problem (P) i.e. A1,A2,A3,A4 . Out of these algorithms, A1 takes more computation and resources to solve the problem than other algorithms and A3 computes the problem in less resources as compared to A1,A2 and A4. Therefore we can say that the A3 algorithm is the best algorithm out of other Algorithms in order to find the solution of the given set problem P.

1. <u>Factors Affecting the algorithms :-</u>

1. ***Machine Configuration :-*** Algorithm depends on the Hardware Configuration like processors(i3 / i5 / i7), RAM (4GB , 8GB), Architecture(32-bit / 64-bit) etc.
2. ***Size of input :-*** How much data we are entering in the algorithm to compute the problem.

❖ ***Time Complexity*** :- It tells how much approximate time is taken by the algorithm to complete the task.
❖ ***Space Complexity*** :- It tells how much extra space is taken by the algorithm to perform a given task for 'n' size of input.

2. <u>Asymptotic Analysis :-</u>

Asymptotic Analysis is an idea to handle the issues like time and space in analyzing algorithms. In Asymptotic Analysis, we have to evaluate the performance of an algorithm in terms of input size. We calculate how the time (or space) taken by an algorithm increases with the input size.

We can have 3 cases to analyze the algorithm :-

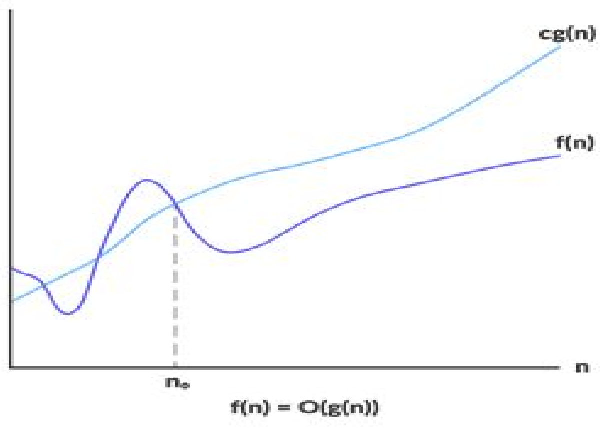
1. **Best Case** - In the best case analysis, we calculate the lower bound of running time of an algorithm.
2. **Average Case** - In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.
3. **Worst Case** - In the worst case analysis, we calculate the upper bound of running time of an algorithm.

3. <u>Asymptotic Notations :-</u>

1. **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above i.e. It tells the worst case time an algorithm takes and it cannot exceed from that time .

The Big O notation is useful if we have to compute upper bound on time complexity of an algorithm.

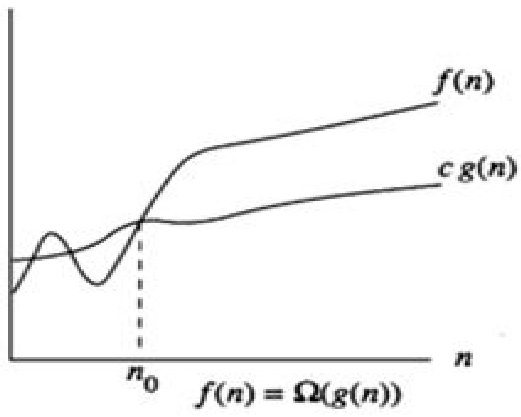
- ***f (n) <= C g(n)*** , n>= n0 , c>0 -----------> ***f(n) = O(g(n))***



- F(n) = 3n+2 , g(n) = n ---------> 3n+2 <= 5n -------> ***F(n) = O(n) / O(n^2) / O(2^n).***

2. **Big-Ω Notation:** Just as Big O notation, Ω notation provides an asymptotic lower bound.
Ω Notation tells the lower bound on time complexity of an algorithm. The Omega notation is the least used notation among all three as this notation tells the best case of an algorithm which can be achieved

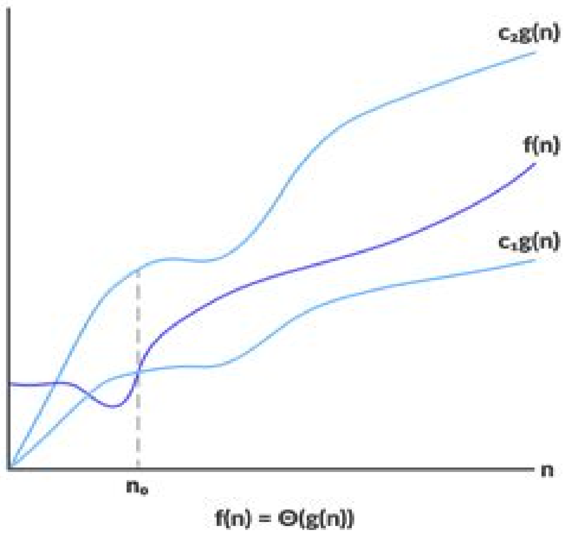
- ***F(n) >= C g(n) ,*** n >=n0 , C>0 ----------> ***F(n) = omega (g(n))***



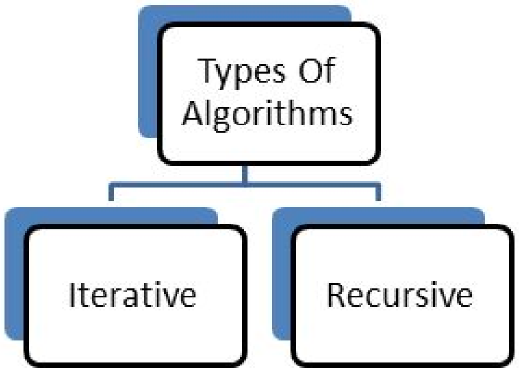
- F(n)=3n+2 , g(n) = n --------> 3n+2 >= n ------->***F(n) = omega(n) / omega(1) / omega(logn)***

3. Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. It is useful in giving the exact order of time taken by the Algorithm.

- **C1 g(n) <= F(n) <= C2 g(n) ----------> F(n) = Theta(g(n))**



- F(n)=3n+2 , g(n) = n ------>1n <= 3n+2 <= 5n ------> ***F(n) = theta(n)***



If Algorithm Doesn’t depend upon the above 2, Therefore algorithm doesn’t depend upon the input size.

1. Iterative Algorithm Analysis :-

In order to analyse an iterative program we have to count the number of times a loop gets executed. Let us understand with the help of examples.

1.

```
Assume N = User Input;
int count = 0;
for (int i = 0; i < N; i++)
    count++;
```

Here the **for loop** runs N times ,
The time complexity of this code is considered to be **O(N)**

2.

```
for (int i = 1; i <= N ; i++)
    for (int j = 1; j <= i ; j++)
        cout<<”Ram”<<endl;
```

Let us see how many times Ram gets printed.

When i=1 it will run 1 time.
When i=2 it will run 2 times.
When i=3 it will run 3 times.
When i=N it will run N times.
Total number of times Ram will be printed is

$$1+2+3+4+....+N = N*(N+1)/2$$

So the time complexity will be **O(N^2)**

3.

```
Assume N = User input
int count = 0;
for (int i = 1; i <= N; i=i*2)
    count++;
```

Let us see how many times this loop gets executed
The series formed by variable i is 1, 2, 4, 8, 16 .... this series will continue till we reach N.
Simplified form can be 2^0 , 2^1 , 2^2 , …., 2^k (suppose 2^k = N)
We can observe that the loop runs k times and k can be found by taking logarithm both the sides

k=log$_2$n , the time complexity of this code is **O(log$_2$n)**

4.

```
for (int i = 1; i <= N ; i++)
    for (int j = 1; j <= N ; j+=i)
        cout<<"Ram"<<endl;
```

Let us see how many times Ram gets printed.
When i=1 it will run N times.
When i=2 it will run N/2 times.
When i=3 it will run N/3 times.
When i=N it will run 1 (N/N) time.
Total number of times Ram will be printed is

$$N+N/2+N/3+.....+1 = N(1+ ½+⅓+....+1/n)$$
$$= N(\log N)$$

Time complexity of this code is **O(N logN)**

5.

```
User Input :  N= 2^m where m = 2^k
for (int i = 1; i <= N ; i++)
    j=2;
    while( j<=n)
        j=j*2;
```

```
6.         cout<<"Ram"<<endl;
```

Outer for loop is running n times ,now lets see how many times inner loop runs,
For k=1, n would be **4** ,
j can be 2 and 4 ,the inner loop will run 2 times , totally it will run n*2 times
For k=2, n would be **16** ,
j can be 2,4,16 , the inner loop will run 3 times, totally it will run n*3 times
For k=3, n would be **2^8** ,
j can be 2,4,16, 2^8, the inner loop will run 4 times,totally it will run n*4 times
In general,the the total number of times this code gets executed in the **k^{th}** iteration is **n*(k+1)** times.
where **$\log_2 n = 2^k$** , again taking logarithm both sides we get
$k = \log \log_2 n$
The time complexity of this code is **O(nlog(logn))**

2. Recursive Algorithm Analysis :-

Recursion : Function that calls itself and has a termination condition.

Recursive Algorithm technique is based on dividing the large problem into subproblem tasks in order to achieve the output.

Recursive Algorithm is a top to bottom approach while **Iterative Algorithm** is a bottom up approach.

The way we analyse the time complexity of recursive functions is very much different from the way we used in iterative functions as there is nothing to count.We write recursive functions and then try to analyse them.

While practising Recursion, we can divide the problem into smaller parts by calling:

1. Decreasing Functions : n---> n-1 ---> n-2 ---> n-3 …… 1
2. Dividing Functions : n ---> n/2 ------> n/4 ------> n/8 ……… 1
3. Square Root Functions :

   and much more.

There are 3 ways to solve the recurrences problem :-

1. Substitution Method
2. Recurrence Tree Method
3. Masters theorem

1. Substitution method :

```
A(n)
 if ( n>1)  // takes O(1) time
    return A(n/2) + A(n/2);
```

Here suppose time taken by **A(n)** is **T(n)**
Then time taken by **A(n/2)** will be **T(n/2)**
Hence we can write the the recursive function for this code as,

$$T(n) = 2T(n/2) + 1$$

Let's understand with the help of some examples,

1.

```
A(n)
    if(n>1)
      return A(n-1);
```

If condition takes O(1) time hence the recursive function can be written as

$$T(n) = 1+T(n-1) \qquad \text{.... (1)}$$

One way to solve this recursive function is ***Back substitution***
In order to find T(n-1) substituting n-1 in place of n in equation 1 ,

$$T(n-1) = 1+ T(n-2) \qquad \text{.... (2)}$$

Similarly , $T(n-2) = 1+ T(n-3)$ .... (3)
Substituting equation 2 in 1, we get,

$$T(n) = 1+1+T(n-2) = 2+T(n-2) \qquad \text{.... (4)}$$

Again substituting 3 in 4,

$$T(n) = 2+1+T(n-3) = 3+T(n-3)$$

In general we can write, $T(n)= k +T(n-k)$ .... (5)
According to the algorithm,

$$\begin{aligned} \mathbf{T(n)} &= \mathbf{1+ T(n-1)}\ , \quad \textbf{is valid only when } \mathbf{n>1} \\ &= \mathbf{1} \qquad , \quad \mathbf{n=1} \end{aligned}$$

n is a very large number which is gradually decreasing and hence the program will stop when it reaches 1 ,this is known as the base condition.
To compute T(n) we require T(1) in equation 5 ,
hence to eliminate k , n-k =1 , k=n-1
Substituting n-1 in place of k equation 5,

$$\begin{aligned} T(n) &= n-1 + T(n-(n-1)) \\ &= n-1 + T(1) \\ &= n-1 +1 = n \end{aligned}$$

T(n)=n, time complexity is **O(n)**

2. Recursive function is given as

$$\begin{aligned} T(n) &= n + T(n-1) \quad , \ n>1 \\ &= 1 \quad\quad\quad\quad\quad , \ n=1 \end{aligned}$$

$T(n) = n + T(n-1)$ ....(1)

Now $T(n-1) = (n-1) + T(n-2)$ ....(2)

Also $T(n-2) = (n-2) + T(n-3)$ ....(3)

Substituting 2 and 3 in 1 , we get,

$T(n) = n + (n-1) + (n-2) + T(n-3)$

In general we can write , $T(n) =n + (n-1) + (n-2) + \dots + (n-k) +T(n-(k+1))$ ....(4)

To eliminate the $T(n-(k+1))$ we require the base condition ,

$n-(k+1)=1$, $k=n-2$ ....(5)

Substituting equation 5 in 4 ,

$$\begin{aligned} T(n) &= n + (n-1) + (n-2) +\dots+ (n-(n-2))+ T(n-(n-2+1)) \\ &= n + (n-1) + (n-2) +\dots+ 2 +1 = \\ &= (n*(n+1))/2 \end{aligned}$$

(sum of first n natural numbers)

$T(n) = O(n^2)$

2. Recursion tree method :

Another way to solve recurrence relations is, recursion tree method. In this method, we draw a recursion tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels.

1.

$$\begin{aligned} T(n) &= 2T(n/2) + c \quad ,n>1 \\ &= c \quad\quad\quad\quad\quad ,n=1 \end{aligned}$$

Consider the recurrence relation

```
                      c
                  /       \
             T(n/2)    T(n/2)
```

Further breaking down T(n/2) and T(n/2) we get the recursion tree as

```
                    c
                /       \
             c        T(n/2)
```

```
          /      \
     T(n/4)    T(n/4)
```

Breaking down further gives us the following

```
                   c                    // total work done - c
                /         \
             c              c          //  total work done - 2c
          /    \          /    \
        c        c        c      c     //  total work done - 4c
     /    \    /  \     /  \   /  \
 T(n/8)  T(n/8)                        //  total work done - 8c
```

The tree will continue to the level it reaches T(1) i.e T(n/n).

T(n) = c+ 2c + 4c + 8c +...... nc

= c ($2^0 + 2^1 + 2^2 + ... + 2^k$)   assuming $n= 2^k$

this forms a gp consisting of (k+1) terms with 1 as first term and 2 as common difference

= c (2^{k+1} -1) = c (2n -1)

T(n) = O(n)

Consider one more example.

2.

T(n) = 2T(n/2) + n   ,n>1
= 1   ,n=1

T(n) = 2T(n/2) +n

T(n/2) = 2T(n/4) + n/2

The recursion tree can be created as:

```
                   n                       // total work done- n
                /        \
            n/2         n/2            // total work done- n
          /      \       /    \
        n/4   n/4   n/4     n/4        // total work done- n
```

```
         /     \
        .       .
        .       .
      T(1)     T(1)                              // total work done- n
```

Work done on each level = n

Total work done = number of levels * work done on each level

Lets see the way in which the tree is shrinking, initially it is n then n/2 , n/4 and so on.

$n/2^0$, $n/2^1$, $n/2^2$….. , $n/2^k$ (assuming that **$n/2^k$ =1** where we are stopping)

Therefore the levels are 0,1,2,...k i.e total number of levels are (k+1) (also **$k = \log_2 n$**)

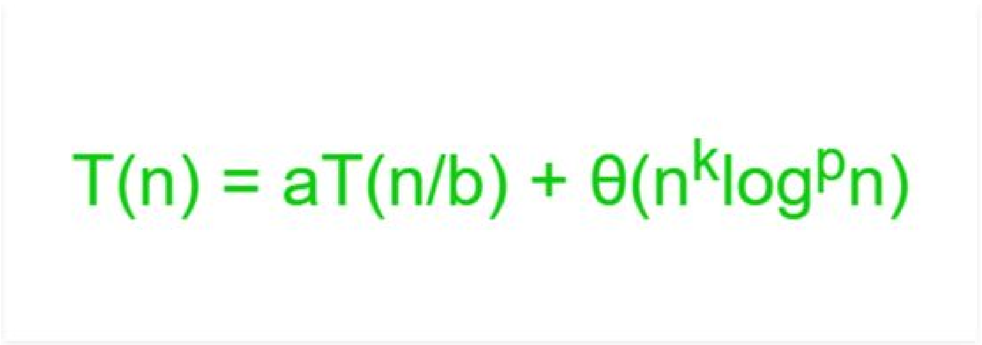
$T(n) = (\log_2 n+1)*n$

$T(n) = O(n \log_2 n)$

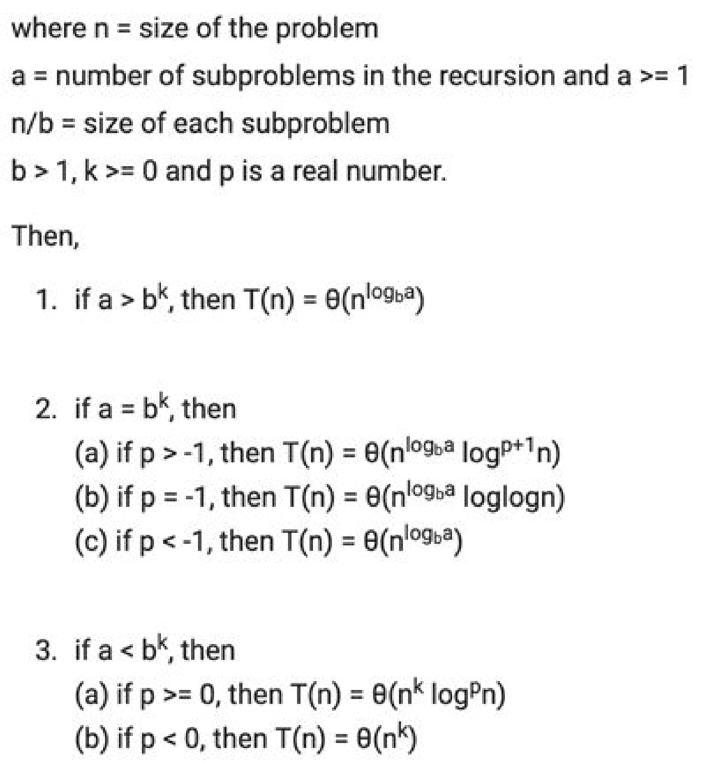
3. Masters Theorem:-

- It is used to calculate the time complexity of Recursive Algorithms.
- Masters Theorem can be applied on 3 types of Recursive Algorithms:-
  - Dividing Recursive Algorithms
  - Decreasing Recursive Algorithms
  - Square root Recursive Algorithms

Masters Theorem of Dividing Recursive Algorithms :-



$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$



where n = size of the problem

a = number of subproblems in the recursion and a >= 1

n/b = size of each subproblem

b > 1, k >= 0 and p is a real number.

Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$

2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$

3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Examples on master's method

1. **$T(n) = 3T(n/2) + n^2$**

   Here, a=3 , b=2, k=2, p=0
   All conditions for masters method are satisfied
   $b^k = 2^2 = 4$
   $a < b^k$ this falls in condition **3 a** of master's method
   T(n) $= \Theta(n^2 \log^0 n) = \Theta$ **(n^2)**

2. **$T(n) = 2^n T(n/2) + n^n$**

Here $a=2^n$ which is not a constant
According to Master's theorem , a must be a constant, also a>=1
This fails to satisfy the condition of master's theorem hence master's theorem cannot be applied.

3. **$T(n) = 2T(n/2) + n\log n$**

Here a=2, b=2, k=1, p=1
All conditions for masters method are satisfied
$b^k = 2^1 = 2$
$a = b^k$ this is the **condition 2 a**
$T(n) = \Theta(n^l \log^2 n) = \Theta(n \log^2 n)$

4. **$T(n) = 0.5T(n/2) + 1/n$**

Here a=0.5 which is not a valid condition of master's method since it fails to satisfy (a>=1)
Hence , Master's theorem is **not valid.**

5. **$T(n) = 64\, T(n/8) - n^2 \log n$**

Generally , we do some work then the problem is divided into sub problem but here if we don't do any work then the problem is getting divided.
Hence whenever there is minus sign(-) master's method is not valid
Master's theorem **cannot be applied.**

6. **$T(n) = 4\, T(n/2) + cn$**

Comparing with the standard equation we get ,
a=4, b=2, k=1, p=0
$b^k = 2^1$ comparing with a
$a > b^k$ (4 > 2) this satisfies the **condition 1** of master's method
$T(n) = \Theta(n^2)$