Analysis of Algorithms

By
Smita Sankhe
Email id: smitasankhe@somaiya.edu

Ms.Smita Sankhe(SRS)

Designation: Assistant Professor

Experience: 17+ Years

Graduation: B.Tech (Computer Science and Technology)

In KJSCE/KJSSE - December 2007

Post Graduation: M. E. Computer Engineering

Ph.D. : Computer Engineering(Pursuing)

Funded Research Project Completed : 02 (ML and AI based in Agriculture Domain)

Area of Interest: Artificial intelligence, Data Science, Machine Learning, Deep Learning .

Publication : 20 + Publication in reputed scopus index conference and journals

CopyRight : 02 (IOT Based Project in Agriculture Domain)

Life time Member of- ISTE, CSI

Faculty Advisor - CodeCell

Faculty coordinator- Student Development Committee, SIH

.

Algorithms and Programs

Algorithm: a method or a process followed to solve a problem.

- A recipe.

An algorithm takes the input to a problem (function) and transforms it to the output.

- A mapping of input to output.

A problem can have many algorithms.

- Preparing tea..

What is an Algorithm?

An algorithm is a finite set of instructions that, if followed, accomplishes particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input**. Zero or more quantities are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. Each instruction is clear and unambiguous.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**. Every instruction must be very basic so that it can be carried out, it also must be feasible.

Why Algorithms

- Data Science
  - DNA Analysis
  - Tweet Analysis
- E Commerce
  - Flipkart
  - Amazon
  - Myntra
- Ticket Booking
- GPS
- Web checking

Analysis of Algorithms

- Syllabus
- Theory CA Rubrics
- Lab CA Rubrics
- All experiments are to be done individually

Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Analysis of Algorithm

- Apriori analysis
  - Time
  - Space
  - Cost (Software Engineering)
- Posterior analysis

A Priori analysis	A Posteriori Testing
It is done before execution of an **algorithm**.	It is done after execution of an algorithm. Or after writing the **program**
Priori analysis is an absolute analysis.	Posteriori analysis is a relative analysis.
It is independent of language of compiler and types of hardware/OS.	It is dependent on language of compiler and type of hardware/OS
It will give approximate answer.	It will give exact answer.
It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.	It doesn't use asymptotic notations to represent the time complexity of an algorithm.

Apriori analysis in general

- Buying a cellphone
  - Budget
  - User age group
  - Technical specification
  - User reviews

Apriori analysis in general

- Buying a home
  - Budget
  - Area (price to area ratio)
  - Location & Locality
  - Amenities in the apartment
  - Amenities around the place
  - other factors

Apriori analysis in general

- Preparing for examination
  - # of chapters
  - #of days/hours in hand
  - Weightage given to every topic
  - Difficulty level
  - Importance of examination score

Apriori analysis in general

- Admission for higher studies
  - ?
  - ?
  - ?
  - ?

How to analyze algorithms

- Time
- Space
- Performance Analysis
  - ❖ Best Case
  - ❖ Average Case
  - ❖ Worst Case

Specifications of good Algorithm

- Work Correctly for all case
- Steps are clear
- Effective Time utilization
- Effective Space utilization
- Give best solution

Algorithm Classification

- Recursion
- Divide and Conquer Technique
- Greedy Technique
- Dynamic Programming Technique
- Backtracking Technique
- String Matching Algorithms
- Non-deterministic Polynomial Algorithms

Performance Analysis

- The performance of a program is the amount of computer memory and time needed to run a program.
- Time Complexity
- Space Complexity

- How to compare Algorithms?
- Execution time
- Number of statements executed
- Running time Analysis

Time Complexity

- The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm.
- The time complexity of a program is the amount of computer time it needs to run to completion.
- Time Complexity is mainly of 3 Types:
- Best Case
- Worst Case
- Average Case

Space Complexity

- The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:
- **Instruction space:** Instruction space is the space needed to store the compiled version of the program instructions.
- **Data space:** Data space is the space needed to store all constant and variable values.
- **Environment stack space:** used to save information needed to resume execution of partially completed functions.
- The space requirement S(P) of any algorithm P may therefore be written as,
- **S(P) = c+ Sp(Instance characteristics)**
- where “c” is a constant.

Complexity of Algorithms

- The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size “n” of the input data.

- Approaches to calculate Time/Space Complexity:
- Frequency count/Step count Method
- Asymptotic Notations – (Order of)

Frequency count/Step count Method

Rules:

1. For comments, declaration

   count = 0

2. return and assignment statement

   count = 1

3. Ignore lower order exponents when higher order exponents are present

   Ex. Complexity of following algo is as follows:

   f(n) = 6n3 + 10n2 + 15n + 3 ⟹ 6n3

4. Ignore constant multipliers

   6n3 ⟹ n3

   f(n) = O(n3)

Example 1: sum of n values of an array

Algorithm sum (int a[], int n
s = 0;
for(i=0; i<n; i++)
{
s=s + a[i];
}
return s;

Time Complexity
□ 1
□ n+1
declaration and n times execution
□ n
□ 1
□ 2n+3
f(n) = O(n)

Space Complexity
a[] = n words
n= 1 word
s= 1 word
i= 1 word
n+3
Space complexity = O(n)

Example 2 : Addition of two square Matrices of dimension $n \times n$

`Algorithm addMat (int a[][], int b[][])`
`{ int c[][];`
`for(i=0; i<n; i++) {`
`for(j=0; j<n; j++) {`
`c[i][j] = a[i][j] + b[i][j]`
`}`
`}`
`}`

Time Complexity
□ $n+1$
□ $n \times (n+1)$
□ $n \times n$
□ $n+1+n^2+n+n^2$ □ $2n^2+2n+1$ $f(n) = O(n^2)$

Space Complexity
$a[][] = n^2$ words
$b[][] = n^2$ words
$c[][] = n^2$ words
$i = 1$ word
$j = 1$ word
$n = 1$ word
□ $3n^2 + 3$ Space complexity = $O(n^2)$

Example 3 : Multiplication of two Matrices of dimension $n \times n$

Algorithm	Time Complexity	Space Complexity
`Algorithm matMul (int a[][], int b[][])`		
`{ int c[][];`		$a[][] = n^2$ words
`for(i=0; i<n; i++) {`	□ $n+1$	$b[][] = n^2$ words
`for(j=0; j<n; j++) {`	□ $n \times (n+1)$	$c[][] = n^2$ words
`c[i][j] = 0;`	□ $n \times n$	$i= 1$ word
`for(k=0; k<n; k++){`	□ $n \times n \times (n+1)$	$j= 1$ word
`c[i][j] = a[i][j] * b[i][j]`	$n \times n \times n$	$k=1$ word
`}`		$n= 1$ word
`}`	□ $n+1+n^2+n+n^2+n^3+1+n^3+n^2$	□ $3n^2+4$ Space complexity = $O(n^2)$
`}`	□ $2n^3+3n^2+2n+1$	
`}`	$f(n) = O(n^3)$	

Example: loops

1.

for(i=0; i<n; i++) {
statements;
}

Time Complexity
? n+1
? n
f(n) = 2n+1 f(n) = O(n)

2.

for(i=n; i>n; i--) {
statements;
}

Time Complexity
? n+1
? n
f(n) = 2n+1 f(n) = O(n)

Example: loops

3.

for(i=1; i<n; i=i+2) {
statements;
}

Time Complexity
? n+1
? n/2
$f(n) = 3n/2+1$ $f(n) = O(n)$

4.

for(i=0; i<n; i++) {
for(j=0; j<n; j++) {
statements;
}
}

Time Complexity
? n+1
$n(n+1)$
? $n \times n$
$f(n) = 2n^2+2n+1$ $f(n) = O(n^2)$

Example: loops (By tracing)

5.

```
for(i=0; i<n; i++) {
    for(j=0; j<i; j++) {
      statements;
  }
}
```

Time Complexity		
i	**j**	**statements**
0	0	0
1	0	1
	1	
2	0	2
	1	
	2	
3	0	3
	1	
	2	
	3	
...	...	...
N	0 to n-1	n

$$1 + 2 + 3 + 4 + \cdots + n = n(n+1)/2$$

$$T(n) = 1 + 2 + 3 + 4 + \cdots + n - 1 = \frac{(n-1)(n)}{2} = O(n^2)$$

Example 5: loops (By tracing)

6.

```
p=0 ;
  for(i=1; p<=n; i++) {
        p=p+i;
  }
}
```

Time Complexity		
i	**p**	**statements**
1	0+1	1
2	1+2	1
3	1+2+3	1
4	1+2+3+4	1
5	1+2+3+4+5	1
6	1+2+3+4+5+6	1
k	1+2+3+4+…+k	???

$=1+2+3+4+\ldots+k>n$

$= \ k(k+1)/2 > n$

$= \ k^2 + k/2 \ > n$

$\cong \ k^2 > n$

$$\boldsymbol{k = \sqrt{n} \ = \ O(n)}$$

Example: loops (By tracing)

6.
```
for(i=1; i<n; i=i*2) {
    statements;
  }
}
```

$$i >= n$$
$$i = 2^k$$
$$2^k >= n$$
$$2^k = n$$

$$\boldsymbol{k = log_2 n \; = \; O(log_2 n)}$$

Time Complexity	
i	**statements**
$1*2^0$	1
$1*2$	1
$1*2*2$	1
$1*2*2*2$	1
…	…
…	…
2^k	1

Example: loops (By tracing)

7.

```
for(i=n; i>=1; i=i/2) {
    statements;
  }
}
```

$$i<1$$

$$n/2^k = 1$$

$$n=2^k$$

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

Time Complexity
i
n
$n/2$
$n/2^2$
$n/2^3$
$n/2^4$
…
$n/2^k$

Example: loops (By tracing)

8.
```
for(i=0; i*i<n; i++) {
    statements;
  }
}
```

$$k * k >= n$$

$$k^2 = n$$

$$k=\sqrt{n}$$

$$\boldsymbol{k = \sqrt{n} = O(\sqrt{n})}$$

Time Complexity	
i	**statements**
1	1
2	2^2
3	3^2
4	4^2
5	5^2
...	...
k	k^2

Important

Example: loops (By tracing)

9.

```
for(i=1; i<n; i=i*2) {
        p++;
 }
for(j=1;j<p;j=j*2){
    statements;
}
```

$p = \log_2 n$

$T(n) = \log_2 p$

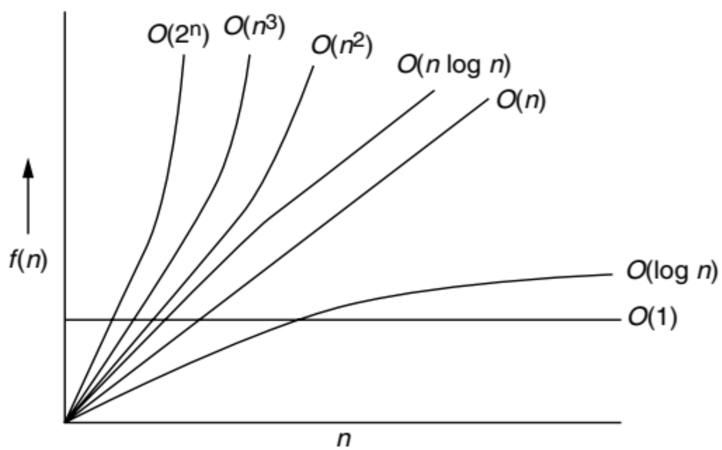
$T(n) = \log_2 \log_2 n$

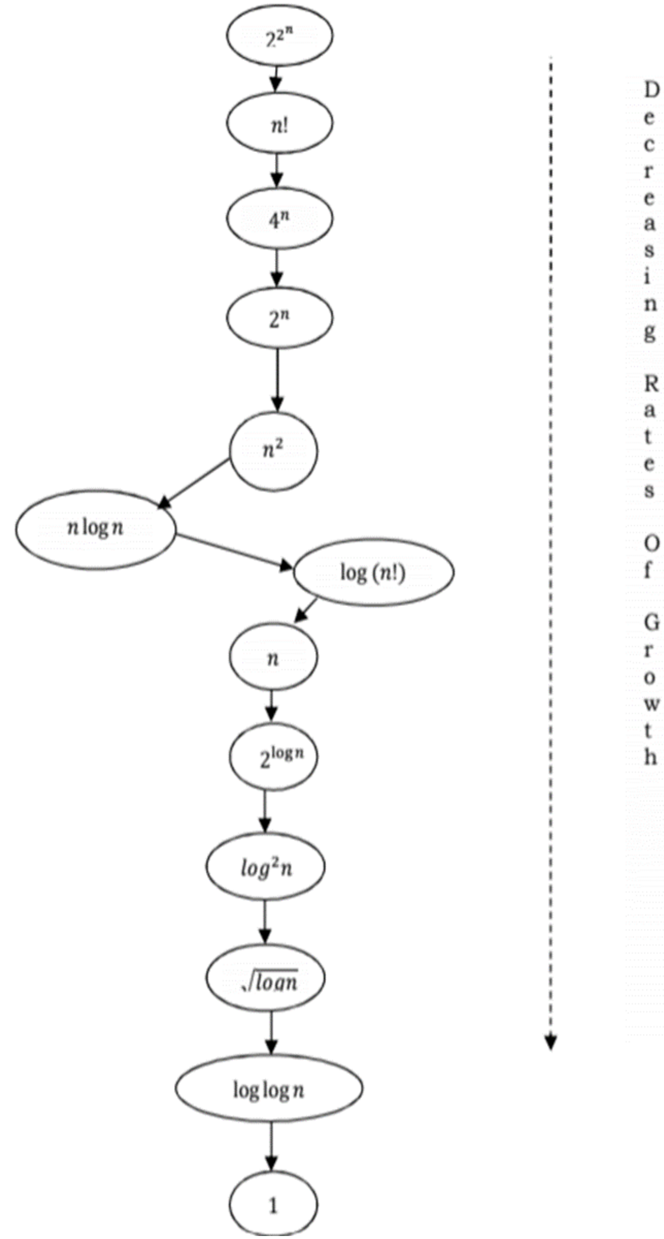
Rate of Growth

- Rate at which the running time
- increases as a function of input is called Rate of Growth.
- Example:

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

n^4 , $2n^2$,$100n$ *and* 500 *are individual cost of some functions and approximate to* n^4 *since* n^4 *is highest rate of growth.*





Numerical Comparison of Different Algorithms

n	log2 n	n*log2n	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Asymptotic Notations:

- Asymptotic notations have been developed for analysis of algorithms.
- By the word asymptotic means “for large values of n”
- The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

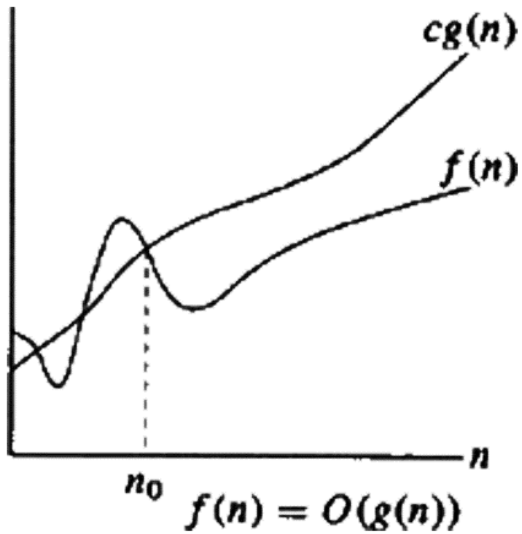
1. Big–OH(O)
2. Big–OMEGA(Ω),
3. Big–THETA (Θ)

Big O notation:

- This notation gives the tight upper bound of the given function
- Represented as:

$$f(n) = O(g(n))$$

that means, at larger values of n, upper bound of $f(n)$ is $g(n)$.



Definition:

Big O notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and no such that

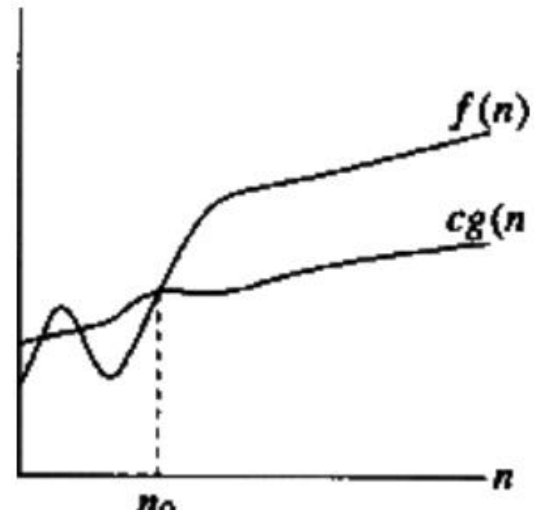
$$0 <= f(n) <= c.g(n) for\ all\ n > n_0\}$$

Big Omega (Ω) notation:

- This notation gives the tight lower bound of the given function
- Represented as:

$$f(n) = \Omega(g(n))$$

that means, at larger values of n, lower bound of $f(n)$ is $g(n)$.



$f(n) = \Omega(g(n))$

Definition:

Big Ω notation defined as $\Omega(g(n))$ = $\{f(n)$: there exist positive constants c and no such that

$$0 <= c.g(n) \leq= f(n) \quad for\ all\ n > n_0\}$$

Big Theta (θ) Notation:

- Average running time of an algorithm is always between lower bound and upper Bound
- Represented as:

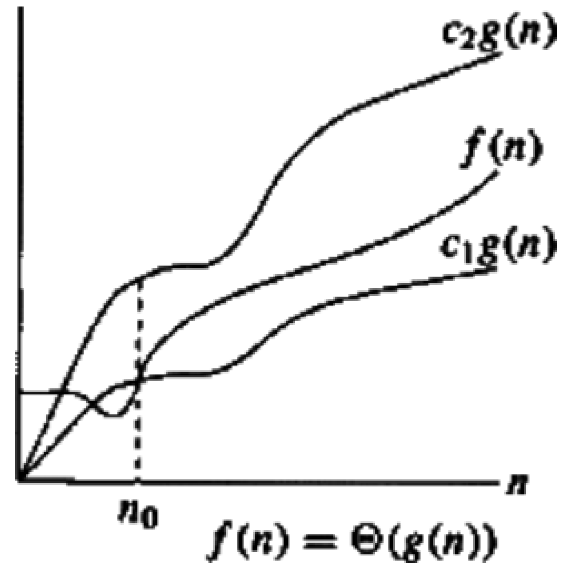
$$f(n) \ = \theta(g(n))$$

  that means, at larger values of n, lower bound of $f(n)$ is $g(n)$.

Definition:

Bigθ notation defined as $\theta(g(n))$ = $\{f(n)$: there exist positive constants c_1 and c_2 and n_o such that

$$0 \le c_1.\,g(n) \le f(n) \le c_1.\,g(n)$$
$$for\ all\ n > n_0\}$$



Recursion:

- Recursion is an ability of an algorithm to repeatedly call itself until a certain condition is met.
- Such condition is called the base condition.

- The algorithm which calls itself is called a recursive algorithm.

- **The recursive algorithms must satisfy the following two conditions:**

1. It must have the base case: The value of which algorithm does not call itself and can be evaluated without recursion.
2. Each recursive call must be to a case that eventually leads toward a base case.

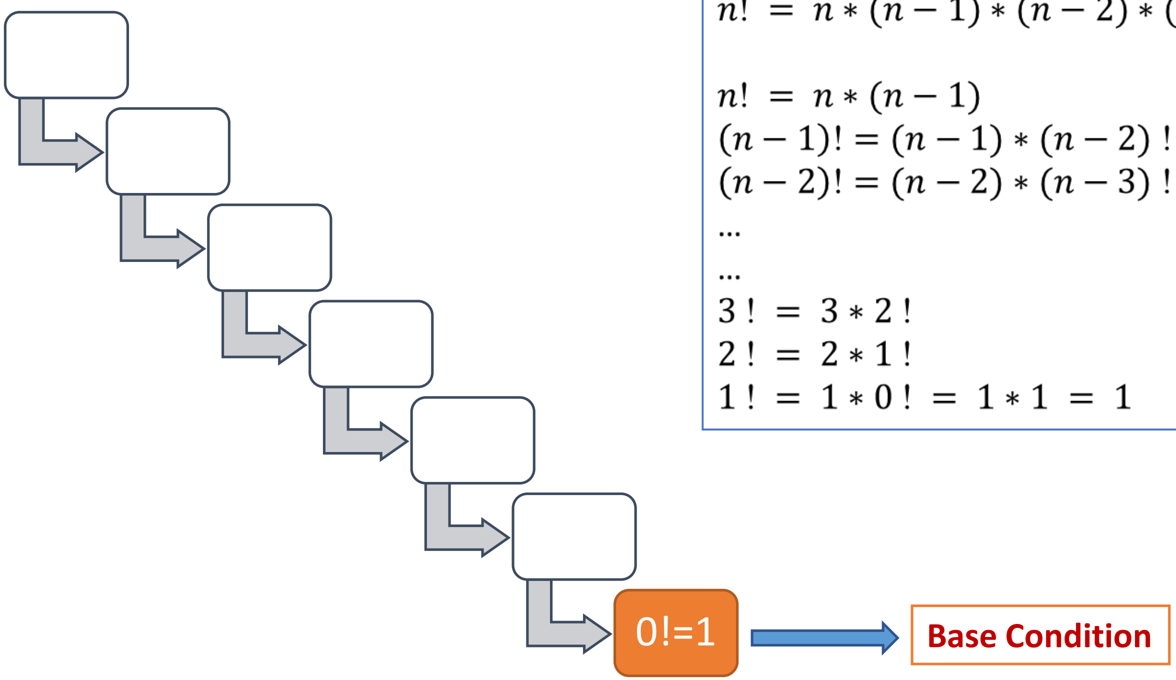
Recursion:

Recurrence Relation:

- An algorithm is said to be recursive if it can be defined in terms of itself.
- The running time of recursive algorithm is expressed by means of recurrence relations.
- A recurrence relation is an equation of inequality that describes a function in terms of its value on smaller inputs.
- It is generally denoted by $T(n)$ where n is the size of the input data of the problem.
- The recurrence relation satisfies both the conditions of recursion, that is, it has both the base case as well as the recursive case.
  - The portion of the recurrence relation that does not contain T is called the base case of the recurrence relation and
  - The portion of the recurrence relation that contains T is called the recursive case of the recurrence relation.

$$T(n) = \begin{cases} d & ; n = 1 \\ T(n-1) + c & ; n > 1 \end{cases}$$

Example: Factorial



$n! = n * (n-1) * (n-2) * (n-3) * \cdots 3 * 2 * 1$

$n! = n * (n-1)$
$(n-1)! = (n-1) * (n-2)\,!$
$(n-2)! = (n-2) * (n-3)\,!$
…
…
$3\,! = 3 * 2\,!$
$2\,! = 2 * 1\,!$
$1\,! = 1 * 0\,! = 1 * 1 = 1$

Factorial Algorithm:

Recursive Method:

```
Algorithm fact(n)
If (n<0) then return("error");
        else
If (n<2) then return(1);
        else
Return (n*fact(n-1);
End if
End if
End fact
```

$$T(n) = \begin{cases} d & ; n = 1 \\ T(n-1) + c & ; n > 1 \end{cases}$$

Iterative Method:

```
Algorithm fact(n)
If (n<0) then return("error");
            else
If (n<2) then return(1);
            else
prod=1;
End if
End if
For(i=n down to 0)
            do prod=prod*i
            end for
Return prod
End fact
```

Recursion:

Recurrence Relation:

There are various methods to solve recurrence:

1. Iterative Method / Substitution Method
2. Recurrence Tree
3. Master Method/ Master's Theorem

Recursion:

Recurrence Tree Method:

- Recurrence is converted into a tree
- Sum of cost of various nodes at each level is calculated, called pre-level cost then sum of pre-level cost is obtained to determine the total cost of all levels of recursion tree.
- If recurrence relation is:

No. of subproblems

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Cost incurred for dividing and combining

Size of subproblem

Then, $f(n)$ is the root of tree, and each node should have 'a' children and Size of each child node is $\frac{1}{b}$ of parent node.