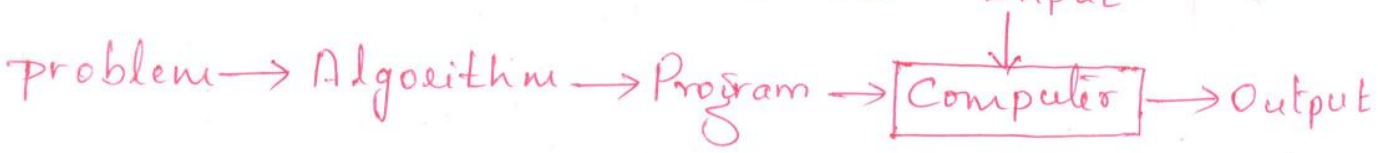
Design and Analysis of Algorithms

Module - 1

What is an Algorithm?

It is a finite set of instructions followed to accomplishes a particular task.

The notion of an algorithm is pictorially represented as shown below



All algorithms must satisfy the following criteria:

1. Input - Zero or more inputs are externally supplied.
2. Output - It should produce correct result. Atleast one output has to be produced.
3. Definiteness - Each instruction should be clear and unambiguous.
4. Finiteness - The algorithm must terminate after a finite sequence or instructions.
5. Effectiveness - The instruction should be simple and should transform the given input to the desired output.

Computing GCD

Different ways of computing GCD
- Euclid's algorithm
- Repetitive subtraction
- Consecutive integer checking algorithm
- Middle school procedure using prime factors

Euclid's Algorithm: GCD(M, N)

Algorithm to compute GCD of two numbers using Euclid's Algorithm is

step 1 - Start

step 2 - Read M and N value

Repeat as long as N is not zero

step 3 - Comput $R \leftarrow M .\!/. N$

step 4 - Assign N to M ie., $M \leftarrow N$

step 5 - Assign R to N ie., $N \leftarrow R$

end of Repeat

return M as the GCD

eg GCD(6, 10)

M	N	$R \leftarrow M .\!/. N$
6	10	$6 \leftarrow 6 .\!/. 10$
10	6	$4 \leftarrow 10 .\!/. 6$
6	4	$2 \leftarrow 6 .\!/. 4$
4	2	$0 \leftarrow 4 .\!/. 2$
2	0	stop when n is zero

Recursive Algorithm:

$GCD(M, N) = GCD(N, M \%. N)$

$$GCD(M, N) = \begin{cases} M & \text{if } N = 0 \\ GCD(N, M \%. N) & \text{otherwise} \end{cases}$$

Recursive algorithm to compute GCD of two numbers using Euclids algorithm

// Computes GCD(m, n)
// Input : M and N should be positive integers
// Output : GCD of M and N
step1 - if (N = 0) return M
step2 - return GCD(N, M %. N)

eg- GCD(10, 6) = GCD(6, 10 %. 6)

M	N	R ← M %. N
6	10	6 ← 6 %. 10
10	6	4 ← 10 %. 6
6	4	2 ← 6 %. 4
4	2	0 ← 4 %. 2
2	0	stop when n is zero

Consecutive integer checking algorithm:

Algorithm GCD(m, n)
// Purpose : To find the GCD of two numbers
// Description : This algorithm computes GCD of two non-negative integers as parameters
// Input : Two non-negative and non-zero integers values m and n
// Output : GCD of m and n

step 1 : small ← (m, n)

step 2 : Divide m and n by small

if remainder is 0 in both cases

return small as the answer

else

goto step 3

end if

step 3 : Decrement small by 1 and goto step 2

small	m %. small	n %. small	small is GCD?
6	10 %. 6 = 4	6 %. 6 = 0	6 is not GCD
5	10 %. 5 = 0	6 %. 5 = 1	5 is not GCD
4	10 %. 4 = 2	6 %. 4 = 2	4 is not GCD
3	10 %. 3 = 1	6 %. 3 = 0	3 is not GCD
2	10 %. 2 = 0	6 %. 2 = 0	2 is GCD (both remainder is 0)

m = 10, n = 6

GCD(10, 6) = 2

Repititive Subtraction :

Algorithm GCD(m, n)

// Description : Computes GCD(m, n)

// Input : m and n should be positive integ

// Output : GCD of m and n

step 1 - Repeat as long as M and N are different

step1 - if M > N, subtract N from M, stor result in M

step 2 : if M < N, subtract N from N, store result in N

step 2 - return M

M	N	Description
10	6	M = 10 - 6 = 4
4	6	N = 6 - 4 = 2
4	2	M = 4 - 2 = 2
2	2	Since M and N are same, GCD will be M or N

GCD(10,6) = 2

Using Middle School Method

Algorithm GCD(m,n)

// Purpose : To find the GCD of two numbers

// Description : This algorithm computes the GCD of two non-negative and non-zero values as parameters.

// Input : Two non-negative and non-zero values as m and n

// Output : GCD of m and n

step 1 - Find the prime factors of m

step 2 - Find the prime factors of n

step 3 - Identify the common prime factors obtained in step 1 and step 2

step 4 - Find the product of all common factors and return the result as gcd of given numbers.

GCD(60, 120)

2 | 60
2 | 30
3 | 15
5

prime factors of 60 = 2.2.3.5

$$
\begin{array}{r|l}
2 & 120 \\
\hline
2 & 60 \\
\hline
2 & 30 \\
\hline
3 & 15 \\
\hline
 & 5
\end{array}
$$

prime factors of $120 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5$

Common prime factors $= 2 \cdot 2 \cdot 3 \cdot 5 = 60$

GCD $(60, 120) = 60$

Algorithm Specification

* Pseudocode Conventions:

1. Comments begin with // and continue until the end of line
2. Blocks are indicated with matching braces { and }.
3. An identifier begins with a letter.
4. Assignment of values to variables is done using the assignment statement.

   <variable> := <expression>;
5. There are two boolean values true and false. To produce these values, the logical and relational operators are provided.
6. Elements of multidimensional arrays are accessed using [and].
7. Looping statements are employed like

```
while <condition> do
{
    <statement 1>
    :
    <statement n>
}
```

get executed. When <condition> becomes false, the loop is exited.

General form of a for loop

```
for variable := value1 to value2 step step do
{
    <statement 1>
        :
    <statement n>
}
```

A repeat until is constructed as follows

```
repeat
    <statement 1>
        :
    <statement n>
until <condition>
```

The statements are executed as long as <condition is false. The value of <condition> is computed after executing the statements.

break: results in the exit of the loop
return: results in the exit of the function itself.

8. A conditional statement has the following forms: if <condition> then <statement>
   if <condition> the <statement1> else <statement2>

9. Input and Output are done using the instructions read and write.

10. There is only one type of procedure:
    Algorithm Name (<parameter list>)

Eg 1- Algorithm finds and returns the maximum of n given numbers.

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4     Result := A[1];
5     for i := 2 to n do
6        if A[i] > Result then Result := A[i];
7     return Result;
8  }
```

Eg 2- Algorithm SelectionSort(a, n) correctly sorts a set of n>=1 elements ; the result remains in a[1:n] such that a[1]<=a[2]<=...<=a[n].

```
1  Algorithm SelectionSort(a, n)
2  //sort the array a[1:n] into nondecreasing
                                    order
3  {
4     for i := 1 to n do
5     {
6        j := i;
7        for k := i+1 to n do
8           if (a[k] < a[j]) then j := k;
9        t := a[i]; a[i] := a[j]; a[j] := t;
10    }
11 }
```

Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly an algorithm is said to be recursive, if the same algorithm is invoked in the body.

Two examples show how to develop a recursive algorithm.

1. Towers of Hanoi Problem

Assume that the number of disks is n. To get the largest disk to the bottom of tower B, we move the remaining n-1 disks to tower C and then move the largest to tower B. Now Left with moving the disks from tower C to B. To do this we have towers A and B availab the fact that tower B has a disk on it can be ignored as the disk is larger than the disk being moved from tower C and so any disk can be placed on top of it.

So n-disk problem is formulated in terms of solutions to two (n-1)-disk problems

```
1  Algorithm Towers Of Hanoi (n, x, y, z)
2  // Move the top n disks from tower x to y
3  {
4      if (n >= 1) then
```

```
5     {
6       TowersOfHanoi (n-1, x, z, y);
7       write ("move top disk from tower", x,
          "to top of tower", y);
8       TowersOfHanoi (n-1, z, y, x);
9     }
10  }
```

2. Permutation Generator: [list of characters]

Given a set of $n >= 1$ elements, the problem is to print all possible permutations of this set.

Eg. if the set is {a, b, c}, then the set of permutations is {(a,b,c), (a,c,b), (b,a,c), (b,c,a) (c,a,b), (c,b,a)}.

When n elements, there are n! different permutations.

In case of four elements (a, b, c, d). The answer can be constructed by writing

1. a followed by all the permutations (b,c,d)
2. b followed by all the permutations (a,c,d)
3. c followed by all the permutations (a,b,d)
4. d followed by all the permutations (a,b,c)

The expression "followed by all the permutation" is the clue to recursion.

```
1  Algorithm Perm (a, k, n)
2  {
3      if (k = n) then write (a[1:n]); // output
4      else // a[k:n] has more than one
                                    permutation
```

```
5        // Generate these recursively
6        for i:=k to n do
7        {
8           t:=a[k]; a[k]:=a[i]; a[i]:=t;
9           Perm(a, k+1, n);
10          // All permutations of a[k+1:n]
11          t:=a[k]; a[k]:=a[i]; a[i]:=t;
12       }
13    }
```

Analysis Framework.

There are two kinds for analyzing the efficiency of algorithm

- Time efficiency
- Space efficiency

Time Efficiency - indicates how fast an algorithm in question runs. Since the time efficiency of an algorithm depends on various factors like

- Speed of the computer
- Choice of the programming Language
- Compiler used
- Choice of the algorithm
- Number (size) of inputs / Outputs

Space Efficiency - deals with the extra space the algorithm requires. The space complexity of an algorithm depends on various factors like

- Data Space
- Stack Space

Program Space - The space required for storing the machine program generated by the compiler or assembler.

Data Space - The space required to store the constants, variables etc.

Stack Space - The space required to store the return address along with parameters that are passed to the function, local variables etc.

Measuring an Input's size-

- All algorithms run longer on larger inputs
- For eg, it takes longer to sort larger arrays, multiply larger matrices & so on.
- Therefore an algorithm's efficiency depends on some parameter n indicating the algorithm's input size.
- So choice of a parameter indicating an input size matters in problems like sorting, searching, finding the list's smallest element, evaluating polynomial equation etc.
- For eg, to measure an input's size for a spell-checking algorithm, the algorithm examines individual characters of its input, then we should measure the size by the number of characters, if it works by processing words, we should count their number in the input.

Since it is present in innermost loop.
- multiplication operation in matrix multiplica since it is present in innermost loop.
- The running time of an algorithm or time efficiency is analyzed by determining the number of times the basic operation is executed on inputs of size (n).
- Let C_{op} be the execution time of an algorithm's basic operation on a particular computer
- Let $C(n)$ be the number of times this basic operation needs to be executed for this algorithm.

To estimate the running time $T(n)$ the formula is

$$T(n) \approx C_{op} C(n)$$

How much faster would this algorithm run on a machine that is ten times faster than the one we have? Answer is ten times.

$$C(n) = \frac{1}{2} n(n-1)$$

How much does longer will the algorithm run if we double its input size? Answer is four times longer.

$$C(n) = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 - \frac{1}{2} n \approx \frac{1}{2} n^2$$

and therefore $$\frac{T(2n)}{T(n)} \approx \frac{C_{op} C(2n)}{C_{op} C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2} n^2} = 4$$

Order of Growth

- We expects the algorithms to work faster for all values of n. Some algorithms executes faster for smaller values of n. But as the value of n increases, they tend to be very slow.

So the behaviour of some algorithm changes

with increase in value of n.

- This change in behaviour of algorithm and efficiency can be analyzed by considering the highest order of n.

The order of Growth is normally determined for largest values of n for the following reasons:

* The behaviour of algorithm changes as the value of n increases
* In real time applications, we normally encounter large values of n.

The order of Growth to two running times is given

* Suppose $T(n) \approx c_{op} C(n)$ - T(n) varies linearly Order of Growth is linear
* Suppose $T(n) \approx c_{op} C(n^2)$ - The order of growth is quadratic.

As n increases Order of Growth increases

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^5		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

values of some of the functions

The basic efficiency of asymptotic classes are shown below

* 1 or any constant - Indicates that running time of a program is constant.

* $\log n$ - indicates running time of a program is logarithmic. (eg- binary search)

* n - indicates running time is linear. When n is 1000, the running time is 1000 units. (eg- Linear search)

* $n \log n$ - indicates that running time is linear or quadratic. The algorithms to sort elements in ascending order such as quick sort, merge sort and heap sort will have this running time.

* n^2 - indicates running time is quadratic. Algorithm normally have two loops. Eg - sorting algorithms such as bubble sort, selection sort, addition and subtraction of two matrices have this running time.

* n^3 - indicates the running time of program is cubic. The algorithm with running time will have three loops. Eg - matrix multiplication, algorithm to solve simultaneous equation using gauss-elimination method will have this running time.

* 2^n - indicates running time is exponential. The tower of hanoi problem and algorithms that generate subsets of a given set will have this running time.

* $n!$ - indicates that running time of an algorithm is factorial. The algorithm that generate all permutations of set will have this running time

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

Worst case, Best-case and Average case efficiencies

Eg. while searching for a specific item in an array of n elements using linear search, we have the following situations.

* An item we are searching for may be present in the very first location itself. In this case only one item is compared and this is the best case.
* The item may be present somewhere in the middle which definitely takes some time. Running time is more, we have to consider the average number of cases and hence this situation is an average case.
* The item we are searching for may not be present in the array and running time is more, this is the worst case.

knowing n alone itself is not enough to estimate the run time of an algorithm or a function, we have to find the worst-case, best-case and average-case efficiency.

Best-case efficiency

The efficiency of an algorithm for the input of size n for which the algorithm takes least time during execution among all possible inputs of that size is called best case efficiency.

In the best case, the algorithm runs fastest for the input specified.

For eg. In linear search, if the item to be

present at the beginning, then
is an best-case efficiency

Worst-case efficiency

The efficiency of an algorithm for the input of size n for which the algorithm ta longest time to execute among all possible inputs is called worst case efficiency.

For eg. If we use linear search, item be searched is not present in the array, is an worst case efficiency.

Average case efficiency - the average number of ba operations executed will be considered. This is required only for the randomized input.

Performance Analysis

Space Complexity - Space complexity of an algor is the amount of memory it needs to run completion.

Algorithm abc computes a+b+b*c+(a+b-c)/(a+b 4.0

```
Algorithm abc(a, b, c)
{
 return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

The space needed by each of these algorithms is seen to be the sum of the following components

1. A fixed part that is independent of the characteristics (eg., number, size) of the inputs and outputs. This part includes the instruction spac (space for the code), space for simple variable f

2) $100n^2$ and $0.01n^3$

$100n^2$ $\quad$ $0.01n^3$

order 2 $\quad$ order 3

$100n^2$ is lower order of growth than $0.01n^3$

3) $\log_2 n$ and $\ln n$

$$\log_2 n = \frac{\log_e n}{\log_e 2}$$

since, $\log_e n = \ln n$

$$\log_2 n = \frac{\ln n}{\log_e 2}$$

$$\ln n = \log_e 2 * \log_2 n$$

$$\ln n \approx \log_2 n \qquad (\log_e 2 \text{ is constant multiple})$$

$\log_2 n$ has same order of growth as $\ln n$.

4) $\log_2^2 n$ and $\log_2 n^2$

can be written as, $\log_2 n * \log_2 n$ $\quad$ $2\log_2 n$

$$\log_2 n * \log_2 n \gg 2\log_2 n$$

So, $\log_2^2 n$ has higher order of growth $\log_2 n^2$

5) 2^{n-1} and 2^n

can be written as, $\frac{2^n}{2}$ and 2^n

2^n is same as 2^n

2^{n-1} has order of growth as 2^n

6) $(n-1)!$ and $n!$

can be written as, $(n-1)!$ and $n*(n-1)!$

$(n-1)!$ is less than $n*(n-1)!$

$(n-1)!$ has lower order of growth than $n!$.

Problem - For each of the following functions, indicate how much the function's value will change if its argument is increased fourfold

1) $\log_2 n$

Let $f(n) = \log_2 n$ . . . . (1)

$g(n) = \log_2 4n$ . . . . (2)

$g(n) = \log_2 4n$

$= \log_2 4 + \log_2 n$

$= 2 + \log_2 n$

$= 2 + f(n)$ . . . substituting (1)

So, the function's value is increased by 2

2) $\sqrt{n}$

Let $f(n) = \sqrt{n}$ . . . (1)

$g(n) = \sqrt{4n}$ . . . . (2)

$g(n) = \sqrt{4n}$

$= \sqrt{4} \cdot \sqrt{n}$

$= 2\sqrt{n}$

$= 2 \cdot f(n)$

So, the function's value is increased by 2 times

3) n

Let $f(n) = n$ . . . . (1)

$g(n) = 4n$ . . . . (2)

$g(n) = 4n$

$= 4 \cdot f(n)$

So, the function's value is increased by 4 times

4) n^2

Let $f(n) = n^2$ . . . . (1)

$g(n) = (4n)^2$ . . . . (2)

$g(n) = (4n)^2$

$= 4^2 \cdot n^2$

$= 4^2 \cdot f(n)$

So, the function's value is increased by 4^2 times

5) 2^n

Let $f(n) = 2^n$ ..... (1)

$g(n) = 2^{4n}$ ..... (2)

$g(n) = 2^{4n}$

$= 2^{3n} \cdot 2^n$

$= (2^n)^3 \cdot 2^n$

$= (2^n)^3 \cdot f(n)$

So, the function's value is increased by $(2^n)^3$-tim

Asymptotic Notations.

The efficiency of the algorithm is expressed using asymptotic notations.

The order of growth ~~using asymptotic notations~~ can be expressed using two methods.

1. Order of Growth using asymptotic notations
2. Order of Growth using limits.

What do you mean by asymptotic behaviour of a function?

The value of the function may increase or decrease as the value of n increases. Based on the order of growth of n, the behaviour of the function varies.

Asymptotic notations are the notations using which two algorithms can be compared with respect to efficiency based on the order of growth of an algorithm's basic operation.

Different types of asymptotic notations

- O (Big Oh)
- Ω (Big Omega)
- θ (Big Theta)

Informal Definitions of Asymptotic Notations

Informal Definitions of Big-Oh (O)

Assuming n indicates the size of input and g(n) is a function, informally O(g(n)) is defined as a set of functions with a small or same order of growth as g(n) as n goes to infinity.

eg1- Let $g(n) = n^3$, Since n and n^2 have smaller order of growth and n^3 has same order of growth when compared to n^3

$$n \in O(n^3)$$
$$n^2 \in O(n^3)$$
$$n^3 \in O(n^3)$$

eg2- Let $g(n) = n^3$. Since n^4, $0.001n^4$, n^5+n+3 do not have smaller order or same order as that of n^3, they do not belong to O(g(n)) and can be represented as shown

$$n^4 \notin O(n^3)$$
$$0.001n^4 \notin O(n^3)$$
$$n^5+n+3 \notin O(n^3)$$

Informal Definitions of Big-Omega (Ω)

Assuming n indicates the size of input & g(n) is a function, informally $\Omega(g(n))$ is defined as set of functions with a Larger or same order of growth as g(n) as n goes to infinity.

Eg-1 Let $g(n) = n^3$. Note that n^3, n^4, $0.001n^4$, n^5+n+3 have Larger order or same order as growth as g(n) as n goes to infinity, that of n^3. Hence, they belong to $\Omega(g(n))$ and can be

$n^3 \in \Omega(n^3)$

$n^4 \in \Omega(n^3)$

$0.001n^4 \in \Omega(n^3)$

$n^5+n+3 \in \Omega(n^3)$

eg 2. Let $g(n) = n^3$. Since n and n^2 do not have a larger order or same order of growth when compared to n^3, we say

$n \notin \Omega(n^3)$

$n^2 \notin \Omega(n^3)$

Let $g(n) = n^2$. what is the relation between n^3, $\frac{1}{2}(n)(n-1)$ and $100n+5$ using Ω-notation?

Let $g(n) = n^2$, a) n^3 and n^2

b) $\frac{1}{2}(n)(n-1)$ and n^2

c) $100n+5$ and n^2

a) n^3 and n^2

$n^3 > n^2$

$\boxed{n^3 \in \Omega(n^2)}$

b) $\frac{1}{2}(n)(n-1)$ and n^2

$\Downarrow$

$\frac{n^2}{2} - \frac{n}{2}$ $\qquad$ n^2

same order of growth $\boxed{\frac{1}{2}(n)(n-1) \in \Omega(n^2)}$

c) $100n+5$ $\qquad$ n^2

$n < n^2$

So, $\boxed{100n+5 \notin \Omega(n^2)}$

Informal Definitions of Big-Theta (Θ)

Assuming n indicates the size of input and $g(n)$ is a function, informally $\Theta(g(n))$ is defined as set of functions that have same order of growth as $g(n)$ as n goes to infinity.

Eg. Let $g(n) = n^2$. Note that an^2+bn+c for $a>0$, $n^2+\sin n$, $n^2+\log n$ have same order as that of n^2. Hence, they belong to $\Theta(g(n))$ can be represented as shown

$$an^2+bn+c \in \Theta(n^2)$$
$$n^2+\sin n \in \Theta(n^2)$$
$$n^2+\log n \in \Theta(n^2)$$

Problem - Use the informal definitions of O, Ω and Θ to determine the following assertions are true or false.

a) $n(n+1)/2 \in O(n^3)$

$$\frac{n^2}{2}+\frac{n}{2} \in O(n^3)$$

Since 2 has smaller order of growth than 3 so, the relation $n(n+1)/2 \in O(n^3)$ is true.

b) $n(n+1)/2 \in O(n^2)$

$$\frac{n^2}{2}+\frac{n}{2} \in O(n^2)$$

Since n^2 has same order of growth to n^2 So, the relation $n(n+1)/2 \in O(n^2)$ is true.

c) $n(n+1)/2 \in \Theta(n^3)$

$$\Downarrow$$
$$\frac{n^2}{2}+\frac{n}{2}$$

Since n^2 has smaller order of growth than n^3 so, the relation $n(n+1)/2 \in \Theta(n^3)$ is false.

d) $n(n+1)/2 \in \Omega(n)$

$\Downarrow$

$\frac{n^2}{2} + \frac{n}{2}$

Since 2 has larger order of growth than 1 so, the relation $n(n+1)/2 \in \Omega(n)$ is true.

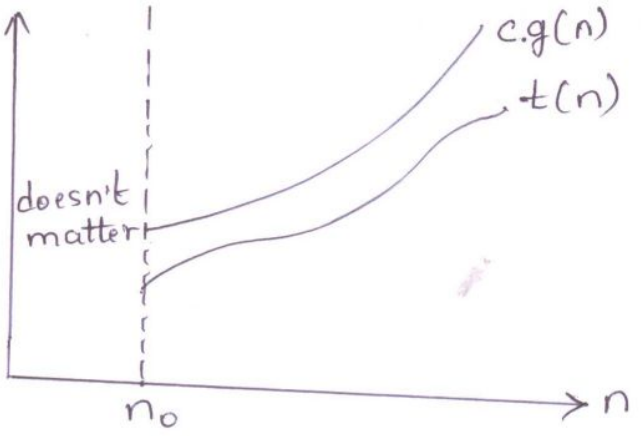
Formal Definitions of Asymptotic Notations

Big-Oh (O) –

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n ie., there exist some positive constant c and some non negative integer n_0 such that

$t(n) \leq c * g(n)$ for all $n \geq n_0$

So, if we draw the graph $t(n)$ and $c * g(n)$ versus n, the graph of $t(n)$ lies below the graph of $c * g(n)$ for sufficiently large value of n as below



Here, $c.g(n)$ is the upper bound.

$t(n)$ be the time efficiency of an algorithm.

Eg-1 Let $t(n) = 100n + 5$. Express $t(n)$ using big-oh

Given $t(n) = 100n + 5$. Replacing 5 with n

We get $100n + n$ and call it $c.g(n)$

ie., $c.g(n) = 100n + n$ for $n = 5$

$= 101n$ for $n = 5$

$t(n) \leq c * g(n)$ for $n \geq n_0$

ie., $100n+5 \leq 101\ n$ for $n \geq 5$

Here, $c = 101$, $g(n) = n$ and $n_0 = 5$

So, $t(n) \in O(g(n))$

ie., $\boxed{t(n) \in O(n)}$

Eg-2 Let $t(n) = 10n^3+8$. Express $t(n)$ using Big-Oh

Given $t(n) = 10n^3+8$. Replacing 8 with n^3 we get $10n^3+n^3$ and call it $c.g(n)$

ie., $c.g(n) = 10n^3+n^3 = 11n^3$ for $n=8$

Now, the following constraint is satisfied

$t(n) \leq c * g(n)$ for $n \geq n_0$

ie., $10n^3+8 \leq 11n^3$ for $n \geq 8$

here, $c=11$, $g(n)=n^3$ and $n_0=8$

So, $t(n) \in O(g(n))$

ie., $\boxed{t(n) \in O(n^3)}$

Eg-3. Prove that every polynomial $p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 n^1 + a_0$ with $a_k > 0$ belongs to $O(n^k)$

Given, $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 n^1 + a_0$

Each term in the summation is of the form $a_i n^i$.

Since, n is non-negative, a particular term will be negative only if $a_i < 0$.

So, $|f(n)| = |a_k n^k + a_{k-1} n^{k-1} + \ldots a_1 n^1 + a_0|$

$\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \ldots |a_1| n^1 + |a_0|$

$= n^k \left[|a_k| + \frac{|a_{k-1}|}{n} + \ldots \frac{|a_1|}{n^{k-1}} + \frac{|a_0|}{n^k} \right]$

$\leq n^k \left(|a_k| + |a_{k-1}| + \ldots |a_1| + |a_0| \right)$

$\leq c * n^k$ where $c = (|a_k| + |a_{k-1}| + \ldots |a_1| + |a_0|)$

ie., $f(n) \leq c * n^k$ where $c = (|a_k| + |a_{k-1}| + \ldots |a_1| + |a_0|)$ for $n \geq 1$

$f(n) \leq c * g(n)$ for $n > n_0$ where $g(n) = n^k$ and $n_0 = 1$

So by definition we can write

$$\boxed{f(n) \in O(n^k)}$$

Eg-4 · $f(n) = 6 * 2^n + n^2$. Express $f(n)$ using Big-Oh

Given, $f(n) = 6 * 2^n + n^2$. Replacing n^2 with 2^n we get $6 * 2^n + 2^n$ and call it $c.g(n)$.

ie., $c.g(n) = 6 * 2^n + 2^n = 7 * 2^n$ for $n >= 0$

Now, the following constraint is satisfied

$f(n) \leq c * g(n)$ for $n \geq n_0$

ie., $6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 0$

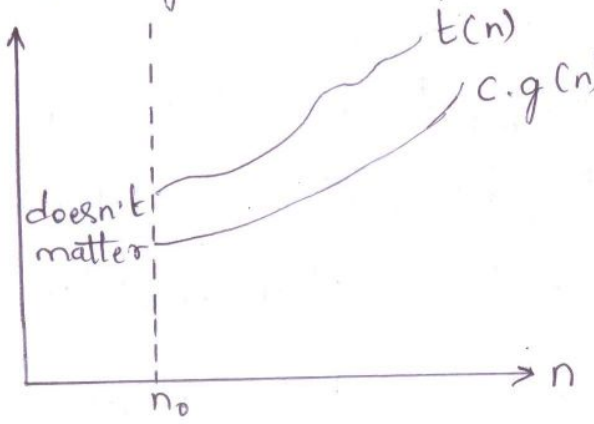
here, $c = 7$, $g(n) = 2^n$ and $n_0 = 0$

So, $\boxed{f(n) \in O(2^n)}$

Big-Omega (Ω)

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$ if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, ie, if there exist some positive constant c and some non negative integer n_0 such that

$t(n) \geq c.g(n)$ for all $n \geq n_0$.



$\leq c * n^k$ where $c = (|a_k| + |a_{k-1}| + \ldots |a_1| + |a_0|)$

ie., $f(n) \leq c * n^k$ where $c = (|a_k| + |a_{k-1}| + \ldots |a_1| + |a_0|)$ for $f(n) \leq c * g(n)$ for $n > n_0$ where $g(n) = n^k$ and $n_0 =$

So by definition we can write

$$\boxed{f(n) \in O(n^k)}$$

Eg-4 · $f(n) = 6 * 2^n + n^2$. Express $f(n)$ using Big-Oh

Given, $f(n) = 6 * 2^n + n^2$. Replacing n^2 with 2^n we get $6 * 2^n + 2^n$ and call it $c.g(n)$.

ie., $c.g(n) = 6 * 2^n + 2^n = 7 * 2^n$ for $n >= 0$

Now, the following constraint is satisfied

$f(n) \leq c * g(n)$ for $n \geq n_0$

ie., $6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 0$

here, $c = 7$, $g(n) = 2^n$ and $n_0 = 0$

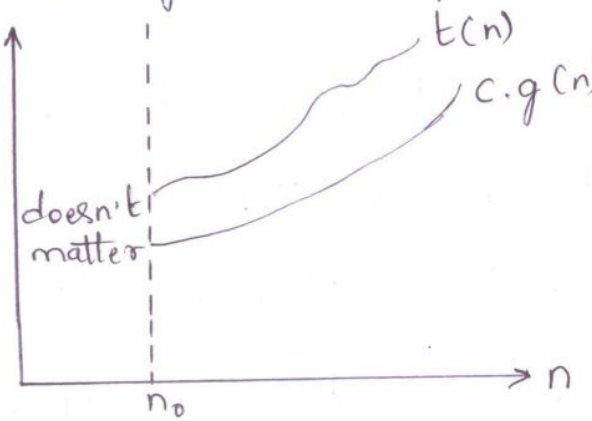
So, $\boxed{f(n) \in O(2^n)}$

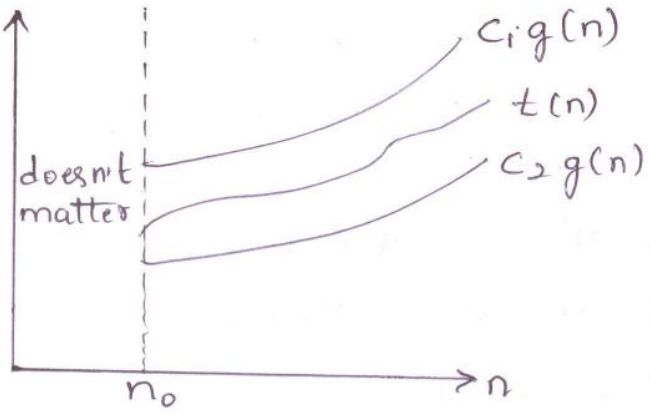
Big-Omega (Ω)

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$ if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large ie, if there exist some positive constant c and some non negative integer n_0 such that

$$t(n) \geq c.g(n) \text{ for all } n \geq n_0.$$

Eg:-





Eg 1- Let $t(n) = 100n + 5$. Express $t(n)$ using big-the

The constraint to be satisfied is

$c_1 * g(n) \leq t(n) \leq c_2 * g(n)$ for $n \geq n_0$

$100 * n \leq 100n + 5 \leq 105 * n$ for $n \geq 5$

here, $c_1 = 100$, $c_2 = 105$, $n_0 = 1$, $g(n) = n$.

So, $\boxed{t(n) \in \Theta(g(n))}$

Eg 2- Let $t(n) = 10n^3 + 5$.

The constraint to be satisfied is

$c_1 * g(n) \leq t(n) \leq c_2 * g(n)$ for $n \geq n_0$

$10 * n^3 \leq 10n^3 + 5 \leq 11 * n^3$ for $n \geq 5$

here, $c_1 = 10$, $c_2 = 11$, $n_0 = 2$, $g(n) = n^3$.

So, $\boxed{t(n) \in \Theta(n^3)}$

Eg 3. Let $t(n) = 6 * 2^n + n^2$. Express $t(n)$ using Big-Oh(theta)

The constraint to be satisfied is

$c_1 * g(n) \leq t(n) \leq c_2 * g(n)$ for $n \geq n_0$

$6 * 2^n \leq 6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 0$

here, $c_1 = 6$, $c_2 = 7$, $n_0 = 0$, $g(n) = 2^n$

So, $\boxed{t(n) \in \Theta(2^n)}$

Property of Asymptotic Notations

The following property is useful in analyzing algorithms that comprise two consecutively executed parts.

Theorem: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

(The analogous assertions are true for the Ω and Θ notations as well.)

Proof: The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2 and b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exists some positive constant c_1 and some non negative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities.

Adding the two equalities above yields the follg.

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3\, 2\max\{g_1(n), g_2(n)\} \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$ respectively.

This property imply for an algorithm that comprises two consecutively executed parts.

$$\left.\begin{array}{l} t_1(n) \in O(g_1(n)) \\ t_2(n) \in O(g_2(n)) \end{array}\right\} \; t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

For eg, we can check whether an array has identical elements by means of the following two part algorithm:

first, sort the array by applying some known sorting algorithm

second, scan the sorted array to check its consecutive elements for equality.

If, for eg, a sorting algorithm used in the first part makes no more than $\frac{1}{2}n(n-1)$ comparisons while second part makes no more than $n-1$ comparisons, the efficiency of entire algorithm will be in $O(\max\{n^2, n\}) = O(n^2)$.

Using Limits for Comparing Orders of Growth

Instead of using asymptotic notations, the order of growth can be obtained by computing the limit of the ratio of the two functions as shown

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c>0 & \text{implies that } t(n) \text{ has same order of growth than } g(n) \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, & the second case means that $t(n) \in \Theta(g(n))$.

The limit-based approach is often more convenient than the one based on the definition because it can take advantage of the powerful calculus techniques developed for computing limits,

such as,

- L'Hospital's rule $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

* Stirling's formula $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n

Note - $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{ie., } t(n) \in O(g(n)) \\ c & \text{ie., } t(n) \in \Theta(g(n)) \\ \infty & \text{ie., } t(n) \in \Omega(g(n)) \end{cases}$

Eg 1- Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) = \frac{1}{2}$$

Since, the limit is equal to a positive constant, the functions have the same order of growth.

So, $\boxed{\frac{1}{2}n(n-1) \in \Theta(n^2).}$

Eg 2- Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} \quad \text{(According to L'Hopital's rule)}$$

$$= \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}$$

$$= 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n}$$

$$= 0$$

Since, the limit is equal to zero, $\log_2 n$ has smaller order of growth than $\sqrt{n}$.

(since $\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we use little-oh notation)

So, $\boxed{\log_2 n \in o(\sqrt{n})}$

Eg 3- Compare the orders of growth of $n!$ and 2^n

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}$$ (according to stirling's formula)

$$= \lim_{n\to\infty} \frac{\sqrt{2\pi n}\, n^n}{2^n e^n}$$

$$= \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n$$

$$= \infty$$

Thus, though 2^n grows very fast, $n!$ grows still faster, so $\boxed{n! \in \Omega(2^n)}$

Problems on Asymptotic notations

1) Prove that $3n^3 + 2n^2 = O(n^3)$

Given, $f(n) = 3n^3 + 2n^2$. Replacing 2 with n

we get $3n^3 + n * n^2$ for $n = 2$

$c.g(n) = 4n^3$ for $n = 2$

Now, the following constraint is satisfied

$f(n) \leq c * g(n)$ for $n \geq n_0$

ie., $3n^3 + 2n^2 \leq 4 * n^3$ for $n \geq 2$

here, $c = 4$, $g(n) = n^3$, $n_0 = 2$

So, $f(n) \in O(g(n))$

$\boxed{3n^3 + 2n^2 \in O(n^3)}$

2) Prove that $3^n \neq O(2^n)$

Let $f(n) = 3^n$ and $g(n) = 2^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$$

By definition, ∞ indicates that $f(n)$ has a larger order of growth when compared to $g(n)$

Hence, $\boxed{3^n \neq O(2^n)}$ is true.

Eg. For each of the following functions, indicate the class $\Theta(g(n))$ the function belongs to. Prove your assertions.

a) $(n^2+1)^{10}$

Informally, $(n^2+1)^{10} \approx (n^2)^{10} = n^{20} \in \Theta(n^{20})$

So, $(n^2+1)^{10} \in \Theta(n^{20})$

Assertion, Let $f(n) = (n^2+1)^{10}$ and $g(n) = n^{20}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(n^2+1)^{10}}{n^{20}}$$

$$= \lim_{n\to\infty} \frac{(n^2+1)^{10}}{(n^2)^{10}}$$

$$= \lim_{n\to\infty} \left(\frac{n^2+1}{n^2}\right)^{10}$$

$$= \lim_{n\to\infty} \left(1+\frac{1}{n^2}\right)^{10}$$

$$= 1$$

ie., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ (constant) So, $(n^2+1)^{10} \in \Theta(n^{20})$

b) $\sqrt{10n^2+7n+3}$

Informally, $\sqrt{10n^2+7n+3} \approx \sqrt{10n^2} = \sqrt{10}\, n \in \Theta(n)$.

So, $\sqrt{10n^2+7n+3} \in \Theta(n)$.

Assertion, Let $f(n) = \sqrt{10n^2+7n+3}$, $g(n) = n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{10n^2+7n+3}}{n}$$

$$= \lim_{n\to\infty} \sqrt{\frac{10n^2+7n+3}{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{10 + \frac{7}{n} + \frac{3}{n^2}}$$

$$= \sqrt{10}$$

$$= \text{c}$$

So, $\sqrt{10n^2+7n+3} \in \Theta(n)$

Mathematical Analysis of Nonrecursive Algorithms

The general plan of analyzing non-recursive algorithm is

* Based on the size of input, determine the number of parameters to be considered.
* Identify the basic operation in the algorithm
* Check whether the number of times the basic operation is executed depends only on size of the input. If the basic operation to be executed depends on some other conditions, then it is necessary to obtain the worst case, best case and average case seperately.
* Obtain the total number of times a basic operation is executed.
* Simplify using standard formulas and obtain the order of growth.

Example 1- Consider the problem of finding the value of the largest element in a List of n numbers.

Algorithm MaxElement(A[0...n-1])
// Determines the value of the largest element in a given array
// Input: An array A[0...n-1] of real numbers
// Output: The value of the largest element in A

```
maxval ← A[0]
for i ← 1 to n-1 do
```

```
    if A[i] > maxval
        maxval ← A[i]
    return maxval.
```

Input's size is the number of elements in the array ie., n.

Time efficiency can be calculated as shown

Let us denote, n represents the size of the input and C(n) denotes the number of times (A[i] > maxval) comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n-1.

$$\therefore C(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1) - 1 + 1 \qquad \text{(upperbound - lowerbound +1)}$$

$$= n-1$$

$$= n$$

$$\boxed{C(n) \in \Theta(n)}$$

Example 2- Consider the element uniqueness problem: check whether all the elements in a given array are distinct.

Algorithm. UniqueElements (A[0....n-1])
// Determines whether all the elements in a given array are distinct
// Input : An array A[0...n-1]
// Output : Returns "true" if all elements in A are distinct and "false" otherwise

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
```

$$= \lim_{n\to\infty} \sqrt{10 + \frac{7}{n} + \frac{3}{n^2}}$$

$$= \sqrt{10}$$

So, $\sqrt{10n^2+7n+3} \in \Theta(n)$

Mathematical Analysis of Nonrecursive Algorithms

The general plan of analyzing non-recursive algorithm is

* Based on the size of input, determine the number of parameters to be considered.
* Identify the basic operation in the algorithm
* Check whether the number of times the basic operation is executed depends only on size of the input. If the basic operation to be executed depends on some other conditions, then it is necessary to obtain the worst case, best case and average case seperately.
* Obtain the total number of times a basic operation is executed.
* Simplify using standard formulas and obtain the order of growth.

Example 1- Consider the problem of finding the value of the largest element in a List of n numbers.

Algorithm MaxElement(A[0...n-1])
// Determines the value of the largest element in a given array
// Input : An array A[0...n-1] of real numbers
// Output : The value of the largest element in A

```
maxval ← A[0]
for i ← 1 to n-1 do
```

if A[i] > maxval
    maxval ← A[i]
return maxval.

Input's size is the number of elements in the array ie., n.

Time efficiency can be calculated as shown

Let us denote, n represents the size of the input and C(n) denotes the number of times (A[i] > maxval) comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n-1.

$$\therefore C(n) = \sum_{i=1}^{n-1} 1$$

$$= (n-1) - 1 + 1 \qquad \text{(upperbound - lowerbound + 1)}$$

$$= n-1$$

$$= n$$

$$\boxed{C(n) \in \Theta(n)}$$

Example 2- Consider the element uniqueness problem: check whether all the elements in a given array are distinct.

Algorithm. UniqueElements (A[0....n-1])
// Determines whether all the elements in a given array are distinct
// Input : An array A[0...n-1]
// Output : Returns "true" if all elements in A are distinct and "false" otherwise

for i ← 0 to n-2 do
    for j ← i+1 to n-1 do

```
            if A[i] = A[j] return false
        end for
    end for
    return true
```

Time efficiency can be calculated as shown below.

By definition, the worst case input is an array for which the number of elements comparison $C_{worst}(n)$ is the largest among all array of size n. An inspection of innermost loop reveals that there are two kinds of worst case inputs: arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$= \frac{(n-1)n}{2}$$

$$= \frac{1}{2} n^2 \in \Theta(n^2)$$

Example 3 - Matrix Multiplication

Given two n-by-n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product $C = AB$.

By definition, C is an n-by-n matrix whose elements are computed as the scalar products of the rows

matrix A and B.

```
Algorithm Multiplication (a[ ], b[ ], c[ ], n)
// Purpose: Multiply two matrices a and b of size
            n x n
// Inputs  n: size of the array
           a: First matrix of size n x n
           b: Second matrix of size n x n
// Output  c: Resultant matrix where the product
   of two matrices should be stored.
    for i ← 0 to n-1
        for j ← 0 to n-1
            sum ← 0
            for k ← 0 to n-1
                sum ← sum + a[i][k] * b[k][j]
            end for
            c[i][j] ← sum
        end for
    end for
```

Time complexity in the best case, worst case remains same.

1. n is input size
2. The basic operation is the multiplication statement in the innermost for loop ie.,
   sum ← sum + a[i][k] * b[k][j] is the basic operation
3. The number of multiplications depends on the value of n only and not on any other factors. So, the total number of times the multiplication statement is executed can be obtained as

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n-1-0+1$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n \sum_{j=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} n(n-1-0+1)$$

$$= \sum_{i=0}^{n-1} n(n) = \sum_{i=0}^{n-1} n^2$$

$$= n^2 \sum_{i=0}^{n-1} 1$$

$$= n^2(n-1-0+1)$$

$$= n^3$$

Time complexity is $\boxed{f(n) \in \theta(n^3)}$

General Plan for Analyzing Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter indicating an input's size
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, if necessary best-case efficiencies have to be investigated seperately
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count, establish its order of growth.

Mathematical Analysis of Recursive Algorithms

Recursive function is a function that calls itself during execution. This enables the function to repeat itself several times to solve a given problem.

Direct Recursion- A recursive function that invokes itself is said to have direct recursion.

eg-
```
int fact(int n)
{
    if (n==0) return 1;
    return n*fact(n-1);
}
```

Indirect recursion. A function which contains a call to another function which intuon calls another function and so on and eventually calls the first function is called indirect recursion

```
void f1()
{
.....
.....
f2();
}
```

```
void f2()
{
.....
.....
f3();
}
```

```
void f3()
{
f1();
.....
}
```

Example 1. Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n.

$$F(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * F(n-1) & \text{otherwise} \end{cases}$$

Algorithm F(n)
// computes n! recursively
// A nonnegative integer n
// Output: The value of n!
if n = 0 return 1
else return n * F(n-1)

Since the function F(n) is computed according to the formula

$$F(n) = n * F(n-1) \quad \text{for } n > 0$$

the number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0$$

So, M(n-1) multiplications are spent to compute F(n-1) and one more multiplication is needed to multiply the result by n.

Example 2. Towers of Hanoi Puzzle. In this puzzle, we have n disks of different sizes and three pegs. Initially all the disks are on the first peg in order of size, the largest on the bottom and smallest on top. The goal is to move all the disks to third peg, using second one as

$F(n) = n * F(n-1)$ for $n > 0$

the number of multiplications $M(n)$ needed to compute it must satisfy the equality

Recurrence relation for factorial of n

$$M(n) = 1 + M(n-1)$$

$$M(n-1) = 1 + M(n-2)$$

$$= 1 + 1 + M(n-2)$$

$$= 2 + M(n-2)$$

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

$$= 3 + M(n-3)$$

$$= 4 + M(n-4)$$

$$\vdots$$

$$= i + M(n-i) \qquad \text{let } i = n$$

$$= n + M(n-n)$$

$$= n + M(0)$$

$$= n + 0$$

$$= n$$

Time complexity of factorial of N is $t(n) \in O(n)$

Time efficiency of Algorithm is better than ...

an auxiliary if necessary.

Algorithm TowerOfHanoi(n, source, temp, destination)
// Inputs : n total number of disks to be moved
// Output : all n disks should be available on the destination needle

step1- if (n=0) return

step2 - [Recursively move n-1 disk from source to temp]
TowerOfHanoi(n-1, source, destination, temp)

step3 - [Move nth disk from source to destination]
write ("Move disk", n, "from", source, "to", destination)

step4 - [Recursively move n-1 disks from temp to destination]
TowerOfHanoi(n-1, temp, source, destination)

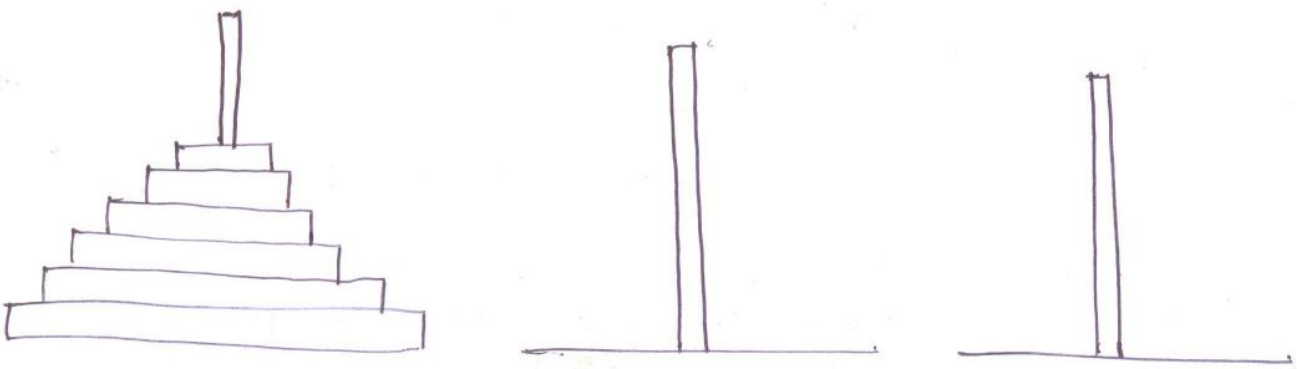
The number of moves M(n) depends on n only we get the following recurrence equation for it

$M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$

$= 2M(n-1) + 1$ for $n > 1$

So, $M(1) = 1$ if $n = 1$

$M(n) = 2M(n-1) + 1$ for $n > 1$



General Plan for analyzing Time Efficiency of Recursive Algorithm

1) Decide on a parameter indicating an input's size
2) Identify the algorithm's basic operation
3) Check whether the number of times the basic operation is executed can vary on different inputs of the same size, if it - the worst case, average case, best-case efficiencies must be investigated seperately.
4) Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5) Solve the recurrence or at least ascertain the order of growth of its solution.

Recurrence relation for Towers of Hanoi problem

$M(n) = M(n-1) + 1 + M(n-1)$ if $M(n) = 2M(n-1)+1$

$= 2M(n-1) + 1$ $M(n-1) = 2M(n-2)+1$

$= 2[2M(n-2)+1]+1$

$= 2^2 M(n-2) + 2 + 1$ $M(n-2) = 2M(n-3)+1$

$= 2^2[2M(n-3)+1]+2+1$

$= 2^3 M(n-3) + 2^2 + 2 + 1$

$= 2^4 M(n-4) + 2^3 + 2^2 + 1$

∴ total number of recursive calls $= 2^n - 1$

∴ $t(n) \in \Theta(2^n)$

5) $\sum_{i=1}^{n}\sum_{j=1}^{n} i \cdot j = \sum_{i=1}^{n}\left[i\left(\sum_{j=1}^{n} j\right)\right]$

$= \sum_{i=1}^{n}\left[i\,\frac{n(n+1)}{2}\right]$

$= \frac{n(n+1)}{2}\sum_{i=1}^{n} i$

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

$= \frac{n^2(n+1)^2}{4}$

6) Simplify $\sum_{j=1}^{n} 3^{j+1}$

Given $\sum_{j=1}^{n} 3^{j+1} = \sum_{j=1}^{n} 3^j \cdot 3^1$

$= 3^1 \sum_{j=1}^{n} 3^j = 3\left[3^1 + 3^2 + \dots 3^n\right]$

$= 3 \cdot \frac{a(r^n-1)}{r-1}$

$= 3 \cdot \frac{3(3^n-1)}{3-1}$

$= \frac{9}{2}(3^n-1)$

Times

Important mathematical formulas and simplifications

1) $\sum_{i=lb}^{ub} 1 = ub - lb + 1$

2) $\sum_{i=lb}^{ub} k = k \sum_{i=lb}^{ub} 1 = k(ub - lb + 1)$

3) $\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots n = \frac{n(n+1)}{2}$

4) $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \ldots n^2 = \frac{n(n+1)(2n+1)}{6}$

5) $\sum_{i=1}^{n} i^k = 1^k + 2^k + 3^k + \ldots n^k \approx \frac{n^{k+1}}{k+1}$

6) $\sum_{i=0}^{n} a^i = 1 + a + a^2 + a^3 + \ldots a^n = \frac{a(r^n - 1)}{r-1} = \frac{1(a^{n+1} - 1)}{a-1} = \frac{a^{n+1} - 1}{a-1}$

7) $\sum_{i=1}^{n} i \cdot 2^i = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \ldots n \cdot 2^n = (n-1)2^{n+1} + 2$

8) $\sum_{i=1}^{n} \log i \approx n \log n$

Find the sum of the series $1 + 3 + 5 + 7 + \ldots 999$

Given: $1 + 3 + 5 + 7 + \ldots 999$

$= 1 + 2 + 3 + \ldots 1000 - [2 + 4 + 6 + 8 + \ldots 1000]$

$= \sum_{i=1}^{1000} i - 2 \sum_{i=1}^{500} i$

$= \frac{1000 * 1001}{2} - 2 * \left(\frac{500 * 501}{2} \right)$

$= 2,50,000$

2) Find the sum of the series 2+4+8+16+....1024

Given, $2+4+8+16+\ldots.1024$

$= 2^1+2^2+2^3+2^4+\ldots.2^{10}$

$= \frac{a(r^n-1)}{r-1}$

$= \frac{2(2^{10}-1)}{2-1} = \underline{2046}$

3) Simplify $\sum_{i=3}^{n+1} 1$

Given, $\sum_{i=3}^{n+1} 1 = ub - lb + 1$

$= n+1-3+1$

$= n-1$

4) Simplify $\sum_{i=3}^{n+1} i$

Given, $\sum_{i=3}^{n+1} i = \left(\sum_{i=1}^{n+1} i\right) - 3$

$= \frac{(n+1)(n+2)}{2} - 3$

$= \frac{1}{2}(n^2+3n+2-6)$

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

$= \frac{(n-1)(n+4)}{2}$

Important Problem Types

Sorting - used to rearrange the items of a given list in ascending order. We need to sort lists of numbers, characters from an alphabet, character strings, records maintained by schools about their students, libraries about their holdings, companies about their employees.

For eg, we can sort student records in alphabetical order of names or by number or by grade point average. Such a specially chosen piece of information is called key.

Sorting makes many questions about the list easier to answer. eg searching in dictionaries, telephone books, class lists and so on.

Two properties of sorting algorithm deserve special mention.

* A sorting algorithm is called stable, if it preserves the relative order of any two equal elements in its input. For eg, we have a list of students sorted alphabetically and we want to sort it according to student GPA, a stable algorithm will yield a list in which students with the same GPA will still be sorted alphabetically.
* An algorithm is said to be in place if it does not require extra memory.

Searching - deals with finding a given value called a search key in a given set. There are plenty of searching algorithm like sequential search, binary search.

For searching, there is no single algorithm that fits all situations best.

require more memory, some are very fast but applicable only to sorted arrays.

In applications where data may change frequently relative to the number of searches, searching has to be considered with two operations addition to and deletion from the data set of an item.

In such situations, data structures and algorithms should be chosen to balance among the requirement of each operation.

String Processing -

String is a sequence of characters from an alphabet. Strings of particular interest are text strings, which comprise letters, numbers & special characters. Bit strings which comprise zeros & ones. Gene sequences which can be modeled by strings of characters from the four-character alphabet {A, C, G, T}.

One particular problem - searching for a given word in a text has attracted special attention from researchers called string-matching.

Graph Problem

Graph is a collection of points called vertices, some of which are connected by line segments called edges.

Graphs can be used for modeling a wide variety of real life applications, including transportation and communication network, project scheduling & games. One recent application is an estimation of the web's diameter, which is the maximum number of links one needs to follow to reach web page from another

Basic graph algorithms include graph traversal algorithms, shortest path algorithms, topological sorting for graphs with directed edge

Some graph problems are very hard - eg: Traveling Salesman Problem (TSP) and Graph coloring problem

1. TSP is the problem of finding the shortest tour through n cities that visits every city exactly on In addition to applications involving route planning used in modern applications as circuit board and VLSI chip fabrication, X-ray chrystallography and genetic engineering.
2. The graph coloring problem asks to assign the smallest number of color to vertices of the graph so that no two adjacent vertices are the same color This problem arises in several application such as event scheduling.

Combinatorial Problems

The traveling salesman problem and the graph coloring problem are examples of combinatorial problems. These are problem asked to find a combinatorial object - such as permutation, a combination or a subset that satisfy certain constraints f has some desired property.

Difficulties - First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances.

Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

Some combinatorial problems can be solved by efficient algorithm with exception.

1 Fundamental Data Structures

Stacks - Stack is a list in which insertion & deletions can be done only at the one end. This end is called the top because a stack is usually visualized not horizontally but vertically.
When elements are added to (pushed onto) a stack and deleted from (popped off) it, the structures operates in the "Last in first Out" (LIFO) fashion. stacks have a multitude of applications, they are indispensable for implementing recursive algorithms.

Queue - is a list from which elements are deleted from one end of the structure called front (this operation is called dequeue) and new elements are added to the other end called rear (this operation is called enqueue). A queue operates in "first in first out" fashion.
Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such application is called a priority queue. A better implementation of a priority queue is based on data structure called heap.

Graphs - is a collection of points in the plane called "vertices" or "nodes", some of them connected by line segments called "edges" or "arcs".
Formally, a graph $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite set V of items called vertices and set E of pairs of these items called edges.
If a pair of vertices (u, v) is same as the pair (v, u), we say vertices u and v are adjacent to each other. A graph G is called undirected if every edge

in it is undirected

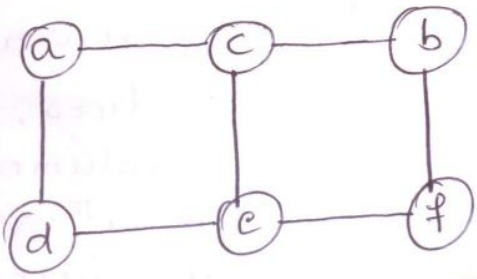


fig (a)

If a pair of vertices (u, v) is not the same as the pair (v, u), we say that the edge (u, v) is directed from vertex u (head) to v (tail). A graph G whose every edge is directed is called directed graph or Digraph.

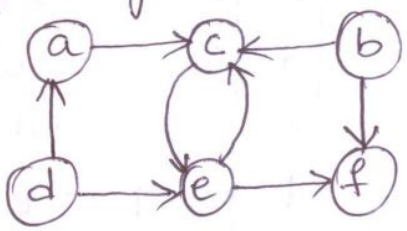


fig (b)

fig (a) has six vertices and seven edges

$V = \{a, b, c, d, e, f\}$

$E = \{(a, c), (a-d), (b, c), (b, f), (c, e), (d, e), (e, f)\}$

fig (b) has six vertices and eight directed edges

$V = \{a, b, c, d, e, f\}$

$E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$.

A graph with every pair of its vertices connected by an edge is called complete.

A graph with few edges relative to the number of its vertices is called sparse.

A graph with relatively few possible edges missing is called dense.

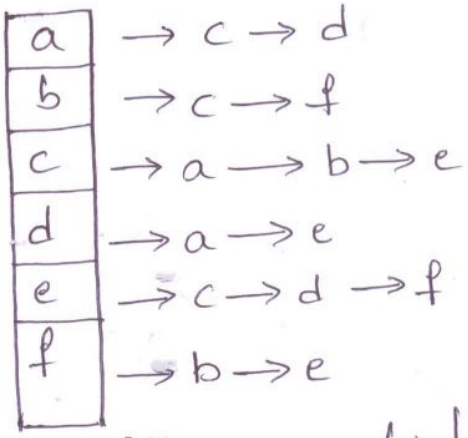
Graph representations -

two principal ways - Adjacency Matrix
Adjacency lists

Adjacency matrix - of a graph with n vertices is an n-by-n boolean matrix with one row and one column for each of the graph's vertices, in which the element in the i^{th} row and j^{th} column is equal to 1 if there is an edge from the i^{th} vertex to the j^{th} vertex, and equal to 0 if there is no such edge.

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

Adjacency matrix

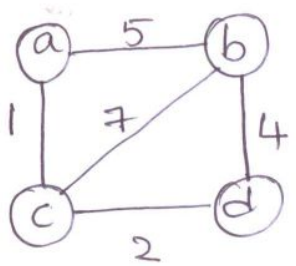


Adjacency List

Adjacency Lists - of a graph or digraph is a collection of linked list, one for each vertex, that contain all the vertices adjacent to the list's vertex.

Weighted graphs - is a graph with numbers assigned to its edges. These numbers are called weights or costs.

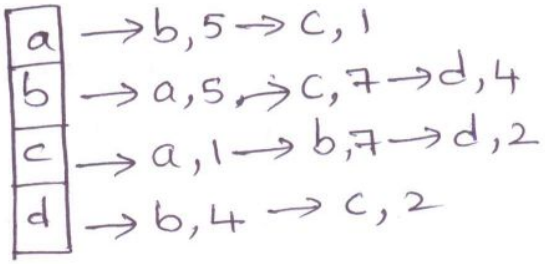
Weight matrix or Cost matrix - if a weighted graph is represented by its adjacency matrix. if there is no edge, a special symbol ∞ is used



Weighted graph

	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

Weight matrix



Adjacency List

Paths and Cycles - Connectivity and acyclicity are interesting properties of graphs.

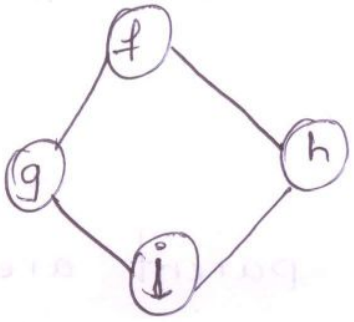
A path from vertex u to vertex v of a graph G can be defined as a sequence of adjacent vertices that starts with u and ends with v.

If all vertices of a path are distinct, the path is said to be simple.

A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.

A Cycle is a path of a positive length that starts and ends at same vertex and does not traverse the same edge more than once

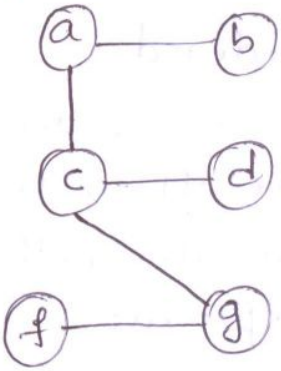
eg.



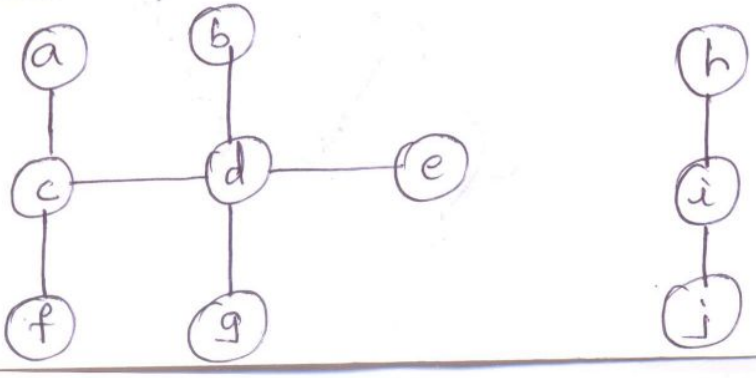
eg, f, g, i, h, f is a cycle

A graph with no cycles is said to be acyclic

Trees. A tree is a connected acyclic graph..

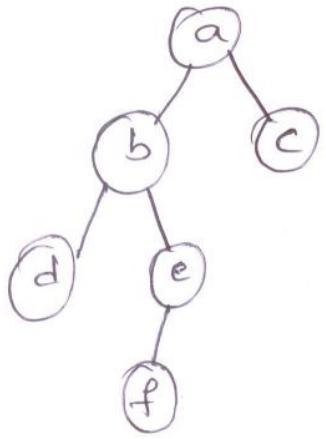


A graph that has no cycles but is not necessarily connected is called forest



Rooted tree - it as the root

State-space trees that underline two important algorithm design techniques: backtracking and branch-and-bound.



Root of the vertex are called ancestors

b - parent

d, e - child

f - leaf

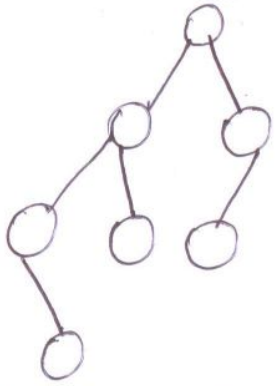
Vertices that have same parent are said to be siblings.

A vertex with no children is called a leaf

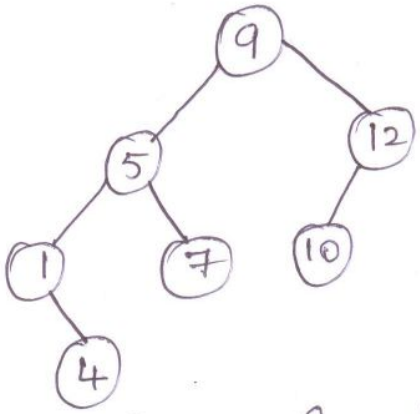
A vertex with atleast one child is called parental

Ordered trees - is a rooted tree in which all the children of each vertex are ordered.

Binary tree - can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either left child or right child of its parent.



Binary Tree



Binary Search tree

Sets and Dictionaries

A set can be described as an unordered collection of distinct items called elements of the set. A specific set is defined either by an explicit listing of its elements (eg $S = \{2, 3, 5, 7\}$) or by specifying a property that all the set's elements & only they must satisfy (eg $S = \{n : n$ is a prime number and $n < 10\}$).

The most important set are checking membership of a given item in a given set, finding the union of two sets, and finding the intersection of two sets.

Sets can be implemented in computer applications in two ways.

The first, considers only sets that are subsets of some large set U, called the universal set. If set U has n elements, then any subset S of U can be represented by a bitstring of size n called a bit vector.

eg- if $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$S = \{2, 3, 5, 7\}$

bitstring is 011010100.

Second, unordered collection of items that are not necessarily distinct multiset or bag. The operations like searching for a given item, adding a new item and deleting an item can be performed for a set or multiset. A data structure that implements these three operation called the dictionary.

Abstract data type (ADT): a set of abstract objects representing data items with a collection of operations that can be performed on them.