UNIT - I

Introduction: Algorithm, Performance Analysis - Space complexity, Time complexity, Asymptotic Notations- Big oh notation, Omega notation, Theta notation and Little oh notation.

Divide and conquer: General method, applications-Binary search, Quick sort, Merge sort, Strassen's matrix multiplication.

PART – I - INTRODUCTION

INTRODUCTION TO ALGORITHM

History of Algorithm

- The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who wrote a textbook on mathematics.
- He is credited with providing the step-by-step rules for adding, subtracting, multiplying, and dividing ordinary decimal numbers.
- When written in Latin, the name became Algorismus, from which algorithm is but a small step
- This word has taken on a special significance in computer science, where "algorithm" has come to refer to a method that can be used by a computer for the solution of a problem
- Between 400 and 300 B.C., the great Greek mathematician Euclid invented an algorithm
- Finding the greatest common divisor (gcd) of two positive integers.
- The gcd of X and Y is the largest integer that exactly divides both X and Y .
- Eg.,the gcd of 80 and 32 is 16.
- The Euclidian algorithm, as it is called, is considered to be the first non-trivial algorithm ever devised.

What is an Algorithm?

Algorithm is a set of steps to complete a task.

For example,

Task: to make a cup of tea.
Algorithm:
· add water and milk to the kettle,
· boil it, add tea leaves,
· Add sugar, and then serve it in cup.

''a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it''.

Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.

These algorithms run on computers or computational devices..For example, GPS in our smartphones, Google hangouts.

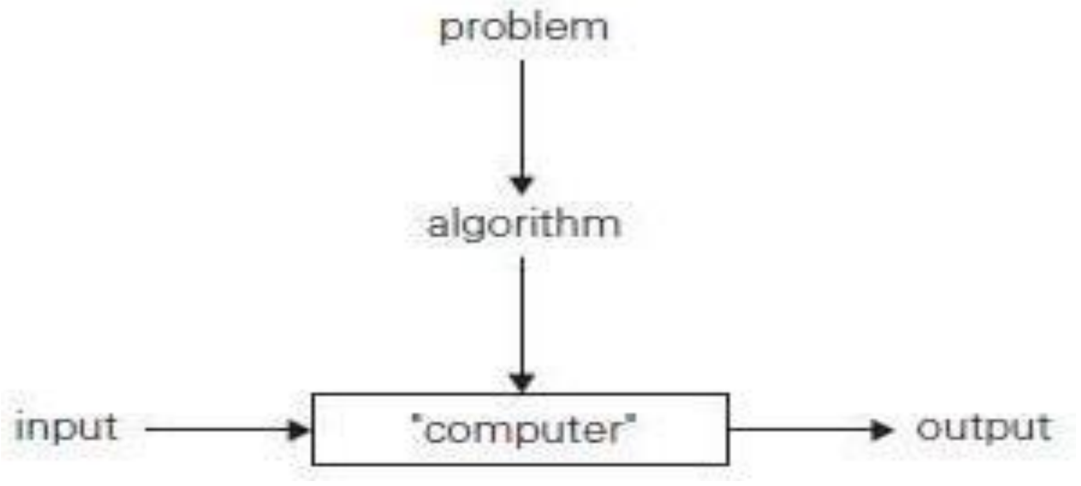
GPS uses shortest path algorithm.. **Online shopping** uses cryptography which uses RSA algorithm.

- Algorithm Definition1:
- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:
    - Input. Zero or more quantities are externally supplied.
    - Output. At least one quantity is produced.
    - Definiteness. Each instruction is clear and unambiguous.
    - Finiteness. The algorithm terminates after a finite number of steps.
    - Effectiveness. Every instruction must be very basic enough and must be feasible.

- Algorithm Definition2:

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

- Algorithms that are definite and effective are also called *computational procedures*.
- A program is the expression of an algorithm in a programming language



- **Algorithms for Problem Solving**

The main steps for Problem Solving are:

1. Problem definition
2. Algorithm design / Algorithm specification
3. Algorithm analysis
4. Implementation
5. Testing
6. [Maintenance]

- Step1. Problem Definition

What is the task to be accomplished?
Ex: Calculate the average of the grades for a given student

- Step2.Algorithm Design / Specifications:

Describe: in natural language / pseudo-code / diagrams / etc

- Step3. Algorithm analysis

Space complexity - How much space is required
Time complexity - How much time does it take to run the algorithm Computer Algorithm

An algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some tasks which, given an initial state terminate in a defined end-state

The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.

- Steps 4,5,6: Implementation, Testing, Maintainance
  - Implementation:

Decide on the programming language to use C, C++, Lisp, Java, Perl, Prolog, assembly, etc. , etc.

Write clean, well documented code

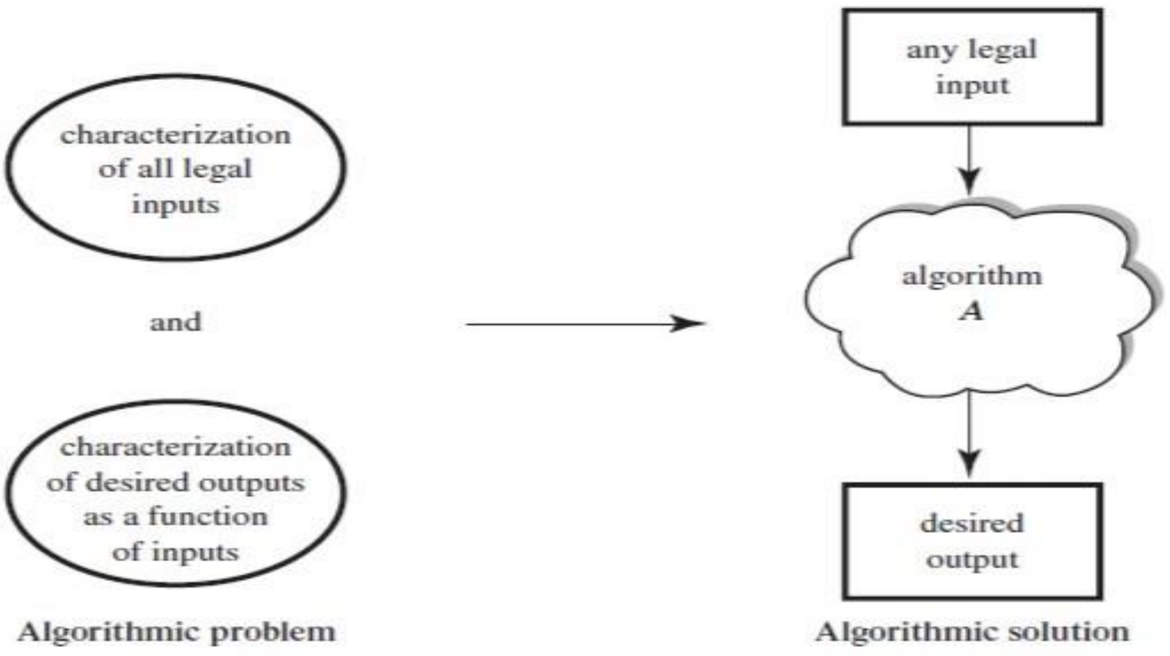
- Test, test, test

Integrate feedback from users, fix bugs, ensure compatibility across different versions

- Maintenance.

Release Updates,fix bugs

Keeping illegal inputs separate is the responsibility of the algorithmic problem, while treating special classes of unusual or undesirable inputs is the responsibility of the algorithm itself.


characterization of all legal inputs
and
characterization of desired outputs as a function of inputs
Algorithmic problem
any legal input
algorithm A
desired output
Algorithmic solution

- **4 Distinct areas of study of algorithms:**

- *How to devise algorithms*. Techniques – Divide & Conquer, Branch and Bound , Dynamic Programming
- *How to validate algorithms*.
- Check for Algorithm that it computes the correct answer for all possible legal inputs. algorithm validation. First Phase
- Second phase Algorithm to Program Program Proving or Program Verification Solution be stated in two forms:
- First Form: Program which is annotated by a set of assertions about the input and output variables of the program predicate calculus
- Second form: is called a specification
- 4 Distinct areas of study of algorithms (..Contd)
- *How to analyze algorithms*.

- Analysis of Algorithms or performance analysis refer to the task of determining how much computing time & storage an algorithm requires
- *How to test a program* □ 2 phases □
- Debugging - Debugging is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
- Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results

PSEUDOCODE:

- Algorithm can be represented in Text mode and Graphic mode
- Graphical representation is called Flowchart
- Text mode most often represented in close to any High level language such as C, Pascal□Pseudocode
- **Pseudocode: High-level description of an algorithm.**
- □More structured than plain English.
- □Less detailed than a program.
- □Preferred notation for describing algorithms.
- □Hides program design issues.
- **Example of Pseudocode**:

- To find the max element of an array

```
Algorithm arrayMax(A, n)
 Input array A of n integers
 Output maximum element of A
 currentMax □ A[0]
 for i □ 1 to n □ 1 do
 if A[i] □ currentMax then
     currentMax □ A[i]
```

return *currentMax*
- Control flow
- if … then … [else …]
- while … do …
- repeat … until …
- for … do …
- Indentation replaces braces
- Method declaration
- Algorithm *method* (*arg* [, *arg*…])
-     Input …
-     Output …
- Method call
- *var.method* (*arg* [, *arg*…])
- Return value
- return *expression*
- Expressions
- Assignment (equivalent to □)
- Equality testing (equivalent to □□)
- n^2 Superscripts and other mathematical formatting allowed

PERFORMANCE ANALYSIS:

- What are the Criteria for judging algorithms that have a more direct relationship to performance?

- computing time and storage requirements.

- **Performance evaluation** can be loosely divided into two major phases:
- a priori estimates and
- a posteriori testing.
- refer as *performance analysis* and *performance measurement* respectively

- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The time complexity of an algorithm is the amount of computer time it needs to run to completion.

SPACE COMPLEXITY:

- Space Complexity Example:
- Algorithm abc(a,b,c)

```
{
    return a+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 component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.
The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

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.

Example 2:

```
Algorithm sum(a,n)
{
s=0.0;
for I=1 to n do
s= s+a[I];
return s;
}
```

The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by 'n' is one word, since it is of type integer.
The space needed by 'a'a is the space needed by variables of tyepe array of floating point numbers.
This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
So,we obtain Ssum(n)>=(n+s)

- [n for a[],one each for n,I a& s]

TIME COMPLEXITY:

- The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

- The number of steps any problem statement is assigned depends on the kind of statement.

- For example, comments à 0 steps.

Assignment statements is 1 steps.
[Which does not involve any calls to other algorithms]
Interactive statement such as for, while & repeat-untilà Control part of the statement.

We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

Algorithm:

```
Algorithm sum(a,n)
{
s= 0.0;
count = count+1;
for I=1 to n do
{
count =count+1;
s=s+a[I];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}
```

If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	*Steps per execution*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

<u>How to analyse an Algorithm?</u>

Let us form an algorithm for Insertion sort (which sort a sequence of numbers).The pseudo code for the algorithm is give below.

<u>Pseudo code for insertion Algorithm</u>:

Identify each line of the pseudo code with symbols such as C1, C2 ..

PSeudocode for Insertion Algorithm	**Line Identification**
for j=2 to A length	C1
key=A[j]	C2
//Insert A[j] into sorted Array A[1.... j-1]	C3
i=j-1	C4
while i>0 & A[j]>key	C5
A[i+1]=A[i]	C6
i=i-1	C7
A[i+1]=key	C8

Let Ci be the cost of ith line. Since comment lines will not incur any cost C3=0.

Cost	**No. Of times Executed**
C1	N
C2	n-1
C3=0	n-1
C4	n-1
C5	
C6	
C7	
C8	n-1

Running time of the algorithm is:

T(n)=C1n+C2(n-1)+0(n-1)+C4(n-1)+C5( )+C6( )+C7( )+C8(n-1)

<u>Best case:</u>

It occurs when Array is sorted.

All t_j values are 1.

T(n)=C1n+C2(n-1)+0 (n-1)+C4(n-1)+C5( ) +C6( )+C7( )+ C8(n-1)

=C1n+C2 (n-1) +0 (n-1) +C4 (n-1) +C5 + C8 (n-1)

= (C1+C2+C4+C5+ C8) n-(C2+C4+C5+ C8) · Which is of the form an+b.

· Linear function of n.

So, linear growth.

Worst case:

It occurs when Array is reverse sorted, and tj =j

T(n)=C1n + C2(n-1)+0 (n-1)+C4(n-1)+C5( ) +C6( )+C7 ( ) + C8(n-1)

=C1n+C2(n-1)+C4(n-1)+C5(______) +C6(______)+C7(_)+ C8(n-1)

which is of the form an^2+bn+c

Quadratic function. So in worst case insertion set grows in n2.

Why we concentrate on worst-case running time?

· The worst-case running time gives a guaranteed upper bound on the running time for any input.

· For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

· Why not analyze the average case? Because it's often about as bad as the worst case.

Order of growth:

It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

Example: We found out that for insertion sort the worst-case running time is of the form an^2 + bn + c.

Drop lower-order terms. What remains is an2.Ignore constant coefficient. It results in n^2.But we cannot say that the worst-case running time T(n) equals n^2 .Rather It grows like n^2 . But it doesn't equal n^2.We say that the running time is Θ (n^2) to capture the notion that the order of growth is n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

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. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'.

Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. **Best Case** : The minimum possible value of f(n) is called the best case.

2. **Average Case** : The expected value of f(n).

3. **Worst Case** : The maximum value of f(n) for any key possible input.

ASYMPTOTIC NOTATION

- **Formal way notation to speak about functions and classify them**

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) and
4. Little–OH (o)

Asymptotic Analysis of Algorithms:

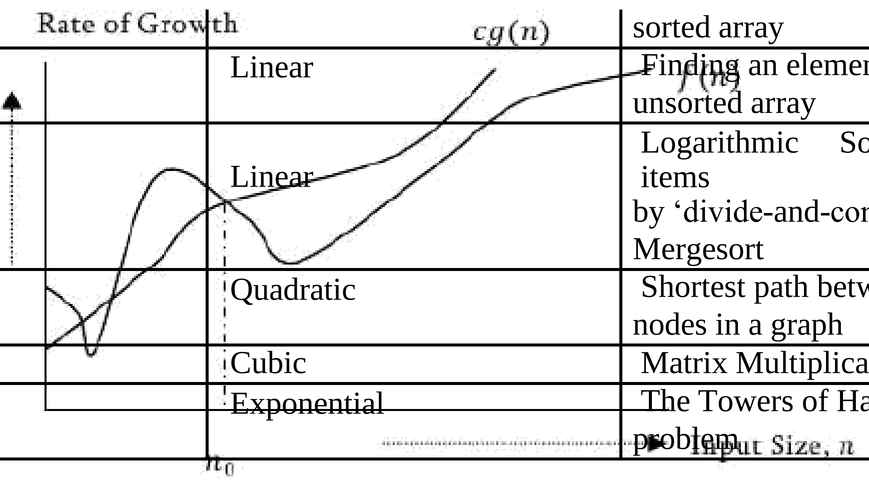
Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

· It is a way to describe the characteristics of a function in the limit.
· It describes the rate of growth of functions.
· Focus on what's important by abstracting away low-order terms and constant factors.
· It is a way to compare "sizes" of functions:

$O \approx \leq$
$\Omega \approx \geq$
$\Theta \approx =$
$o \approx <$
$\omega \approx >$

Time complexity	Name	Example
O(1)	Constant	Adding an element to the front of a linked list
O(logn)	Logarithmic	Finding an element in a

		sorted array
O (n)	Linear	Finding an element in an unsorted array
O(nlog n)	Linear	Logarithmic Sorting n items by 'divide-and-conquer'-Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem



Big 'oh': the function f(n)=O(g(n)) iff there exist positive constants c and no such that f(n)<=c*g(n) for all n, n>= no.
Omega: the function f(n)=(g(n)) iff there exist positive constants c and no such that f(n) >= c*g(n) for all n, n >= no.
Theta: the function f(n)=(g(n)) iff there exist positive constants c1,c2 and no such that c1 g(n) <= f(n) <= c2 g(n) for all n, n >= no

BIG-O NOTATION

This notation gives the tight upper bound of the given function. Generally we represent it as f(n) = O(g (11)). That means, at larger values of n, the upper bound off(n) is g(n). For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is g(n). That means g(n) gives the maximum rate of growth for f(n) at larger values of n.

O —notation defined as O(g(n)) = {f(n): there exist positive constants c and n_o such that 0 <= f(n) <= cg(n) for all n >= n_o}. g(n) is an asymptotic tight upper bound for f(n). Our objective is to give some rate of growth g(n) which is greater than given algorithms rate of growth f(n).

In general, we do not consider lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_o is the point from which we consider the rate of growths for a given algorithm. Below n_o the rate of growths may be different.

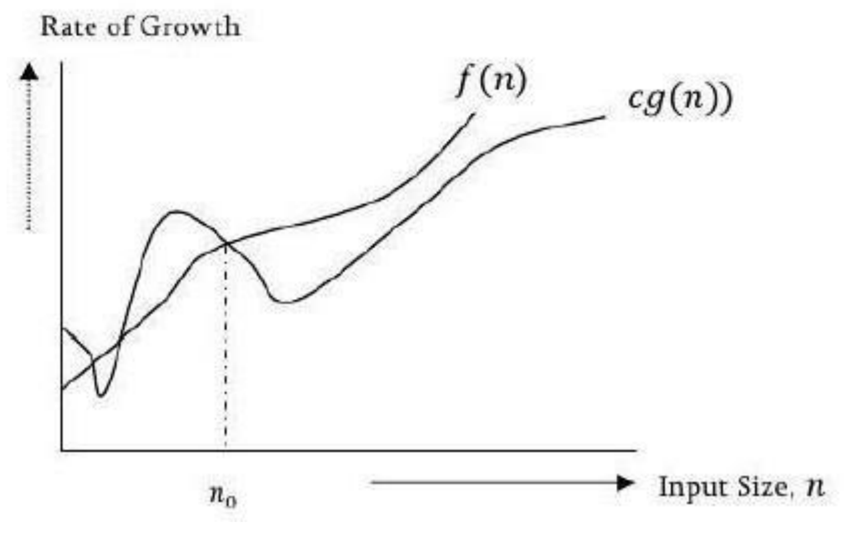
Note Analyze the algorithms at larger values of n only What this means is, below no we do not care for rates of growth.

OMEGA— Ω NOTATION

Similar to above discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as f(n) = Ω (g(n)). That means, at larger values of n, the tighter lower bound of f(n) is g

For example, if $f(n) = 100n^2 + 10n + 50$, g(n) is $\Omega(n^2)$.

The . Ω. notation as be defined as Ω (g (n)) = {f(n): there exist positive constants c and n_o such that 0 <= cg (n) <= f(n) for all n >= n_o}. g(n) is an asymptotic lower bound for f(n). Ω (g (n)) is the set of functions with smaller or same order of growth as f(n).

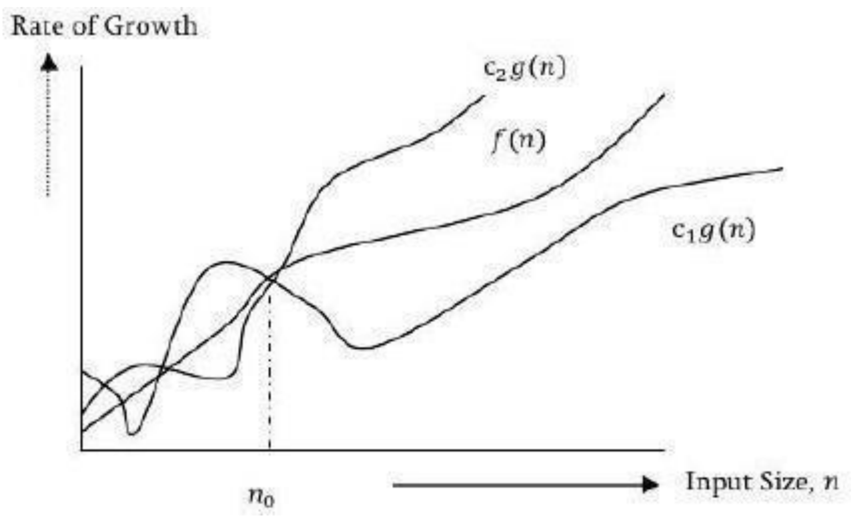


THETA- Θ NOTATION

This notation decides whether the upper and lower bounds of a given function are same or not. The average running time of algorithm is always between lower bound and upper bound.

If the upper bound (O) and lower bound (Ω) gives the same result then Θ notation will also have the same rate of growth. As an example, let us assume that f(n) = 10n + n is the expression. Then, its tight upper bound g(n) is O(n). The rate of growth in best case is g (n) = 0(n). In this case, rate of growths in best case and worst are same. As a result, the average case will also be same.

None: For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth Θ case may not be same.



Now consider the definition of Θ notation It is defined as Θ (g(n)) = {f(71): there exist positive constants C1, C2 and no such that O<=5 c_1g(n) <= f(n) <= c_2g(n) for all n >= n_o}. g(n) is an asymptotic tight bound for f(n). Θ (g(n)) is the set of functions with the same order of growth as g(n).

Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (Θ) may not be possible always.
For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (Θ).
In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance and we use 9 notation if upper bound (O) and lower bound (Ω) are same.

LITTLE OH NOTATION

The little Oh is denoted as o. It is defined as : Let, f(n} and g(n} be the non negative functions then such that f(n}= o(g{n)} i.e f of n is little Oh of g of n.

f(n) = o(g(n)) if and only if f'(n) = o(g(n)) and f(n) != Θ {g(n))

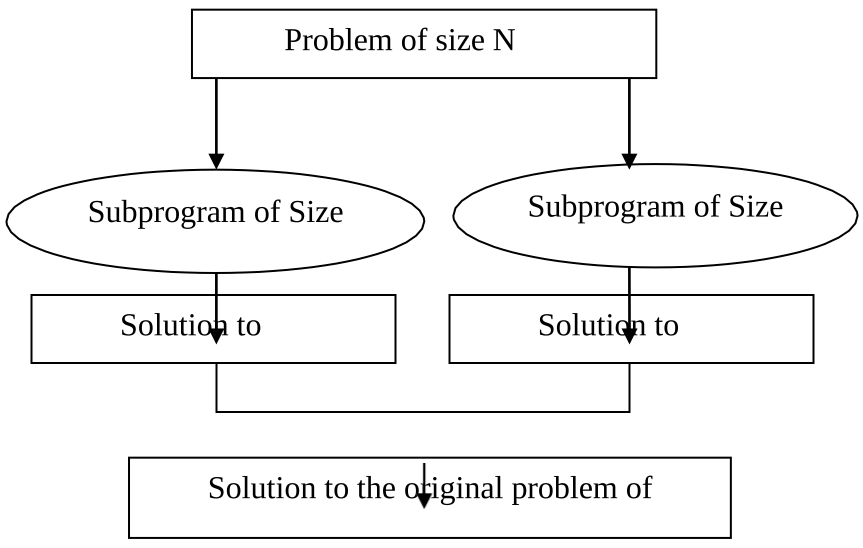
PART – II - DIVIDE AND CONQUER

GENERAL METHOD

In divide and conquer method, a given problem is,
i) Divided into smaller subproblems.
ii) These subproblems are solved independently.
iii) Combining all the solutions of subproblems into a solution of the whole.

If the subproblems are large enough then divide and conquer is reapplied.
The generated subproblems are usually of some type as the original problem.

Hence recurssive algorithms are used in divide and conquer strategy.



Pseudo code Representation of Divide and conquer rule for problem "P"

```
Algorithm DAndC(P)
{
if small(P) then return S(P)
else{
divide P into smaller instances P1,P2,P3…Pk;
apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2)…..
DAndC(Pk)
return combine(DAndC(P1), DAndC(P2)….. DAndC(Pk));
}
}
```

//P Problem
//Here small(P) Boolean value function. If it is true, then the function S is //invoked

Time Complexity of DAndC algorithm:

T(n) = T(1) if n=1
aT(n/b)+f(n) if n>1

a,b contants.
This is called the **general divide and-conquer recurrence.**

Example for GENERAL METHOD:
As an example, let us consider the problem of computing the sum of n numbers a_0, ... a_{n-1}.
If n > 1, we can divide the problem into two instances of the same problem. They are sum of the first | n/2|numbers
Compute the sum of the 1st [n/2] numbers, and then compute the sum of another n/2 numbers.
Combine the answers of two n/2 numbers sum.
i.e.,
$a_0 + . . . + a_{n-1} =(a_0 + ...+ a_{n/2}) + (a_{n/2} +........ + a_{n-1})$
Assuming that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time T(n).
T(n)=aT(n/b)+f(n)

This is called the general **divide and-conquer recurrence**.
f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example, a = b = 2 and f (n) = 1.

Advantages of DAndC:

The time spent on executing the problem using DAndC is smaller than other method.
This technique is ideally suited for parallel computation.
This approach provides an efficient algorithm in computer science.

<u>Master Theorem for Divide and Conquer</u>

In all efficient divide and conquer algorithms we will divide the problem into subproblems, each of which is some part of the original problem, and then perform some additional work to compute the final answer. As an example, if we consider merge sort [for details, refer Sorting chapter], it operates on two problems, each of which is half the size of the original, and then uses O(n) additional work for merging. This gives the running time equation:

T(n) = 2T(−)+ O(n)

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program or algorithm, first we try to find the recurrence relation for the problem. If the recurrence is of below form then we directly give the answer without fully solving it.

If the reccurrence is of the form T(n) = aT(‾) + Θ (nklog^{p}n), where a >= 1, b > 1, k >= O and p is a real number, then we can directly give the answer as:

1) If a > bk, then T(n) =)
Θ (

2) If a = bk

a. If p > -1, then T(n) = Θ ( )

b. If p = -1, then T(n) = Θ ( )

c. If $p < -1$, then $T(n) = \Theta(s)$
3) If $a < b^k$
a. If $p >= 0$, then $T(n) = \Theta(n^k \log p^n)$
b. If $p < 0$, then $T(n) = 0(n^k)$

Applications of Divide and conquer rule or algorithm:

- Binary search,
- Quick sort,
- Merge sort,
- Strassen’s matrix multiplication.

BINARY SEARCH OR HALF-INTERVAL SEARCH ALGORITHM:

1. This algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.
2. In each step, the algorithm compares the search key value with the key value of the middle element of the array.
3. If the keys match, then a matching element has been found and its index, or position, is returned.
4. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the **left** of the middle element or, if the search key is greater, then the algorithm repeats on sub array to the **right** of the middle element.
5. If the search element is less than the minimum position element or greater than the maximum position element then this algorithm returns not found.

Binary search algorithm by using recursive methodology:

Program for binary search (recursive)	Algorithm for binary search (recursive)
int binary_search(int A[], int key, int imin, int imax)	**Algorithm binary_search(A, key, imin, imax)**

**{
if (imax < imin)
return array is empty;
if(key<imin || K>imax)
return element not in array list
else
{
int imid = (imin +imax)/2;
if (A[imid] > key)
return binary_search(A, key, imin, imid-1);
else if (A[imid] < key)
return binary_search(A, key, imid+1, imax);
else
return imid;
}
}**
	**{
	if (imax < imin) then
	return “array is empty”;
	if(key<imin || K>imax) then
	return “element not in array list”
	else
	{
	imid = (imin +imax)/2;
	if (A[imid] > key) then
	return binary_search(A, key, imin, imid-1);
	else if (A[imid] < key) then
	return binary_search(A, key, imid+1, imax);
	else
	return imid;
	}
	}**

Time Complexity:
Data structure:- Array

For successful search	**Unsuccessful search**
Worst case□ O(log n) or θ(log n) Average case□O(log n) or θ(log n) Best case□ O(1) or θ(1)	θ(log n):- for all cases.

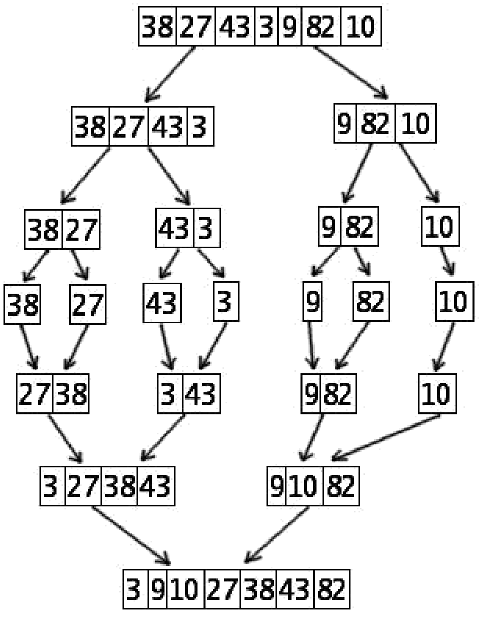
Binary search algorithm by using iterative methodology:

Binary search program by using iterative methodology:	**Binary search algorithm by using iterative methodology:**

int binary_search(int A[], int key, int imin, int imax)
{
while (imax >= imin)
{
i nt imid = midpoint(imin, imax);
if(A[imid] == key)
return imid;
else if (A[imid] < key)
imin = imid + 1;
else
imax = imid - 1;
}
}
	Algorithm binary_search(A, key, imin, imax)
	{
	While < (imax >= imin)> do
	{
	int imid = midpoint(imin, imax);
	if(A[imid] == key)
	return imid;
	else if (A[imid] < key)
	imin = imid + 1;
	else
	imax = imid - 1;
	}
	}

MERGE SORT:

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of *O* (*n* log *n*). Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort.



Advantages of Merge Sort:

1. Marginally faster than the heap sort for larger sets
2. Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.
3. Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

Program for Merge sort

```
#include<stdio.h>
#include<conio.h>
int n;
void main(){
int i,low,high,z,y;
int a[10];
void mergesort(int a[10],int low,int high);
void display(int a[10]);
clrscr();
printf("\n \t\t mergesort \n");
printf("\n enter the length of the list:");
scanf("%d",&n);
printf("\n enter the list elements");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
low=0;
high=n-1;
mergesort(a,low,high);
display(a);
getch();
}
void mergesort(int a[10],int low, int high)


{
int mid;
void combine(int a[10],int low, int mid, int high);
if(low<high)
{
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
combine(a,low,mid,high);
}
}
void combine(int a[10], int low, int mid, int high){ int i,j,k;
int temp[10];
k=low;
i=low;
j=mid+1;
while(i<=mid&&j<=high){
if(a[i]<=a[j])
{
```

```
temp[k]=a[i];
i++;
k++;
}
else
{
temp[k]=a[j];
j++;
k++;
}
}
while(i<=mid){
temp[k]=a[i];
i++;
k++;
}

while(j<=high){
temp[k]=a[j];
j++;
k++;
}
for(k=low;k<=high;k++)
a[k]=temp[k];
}
void display(int a[10]){
int i;
printf("\n \n the sorted array is \n");
for(i=0;i<n;i++)
printf("%d \t",a[i]);}
```

Algorithm for Merge sort:

```
Algorithm mergesort(low, high)
{
if(low<high) then
{
mid=(low+high)/2;
mergesort(low,mid);
mergesort(mid+1,high); //Solve the sub-problems
Merge(low,mid,high); // Combine the solution
}
}
void Merge(low, mid,high){
k=low;
i=low;
j=mid+1;
while(i<=mid&&j<=high) do{
if(a[i]<=a[j]) then
{
temp[k]=a[i];
i++;
k++;
```

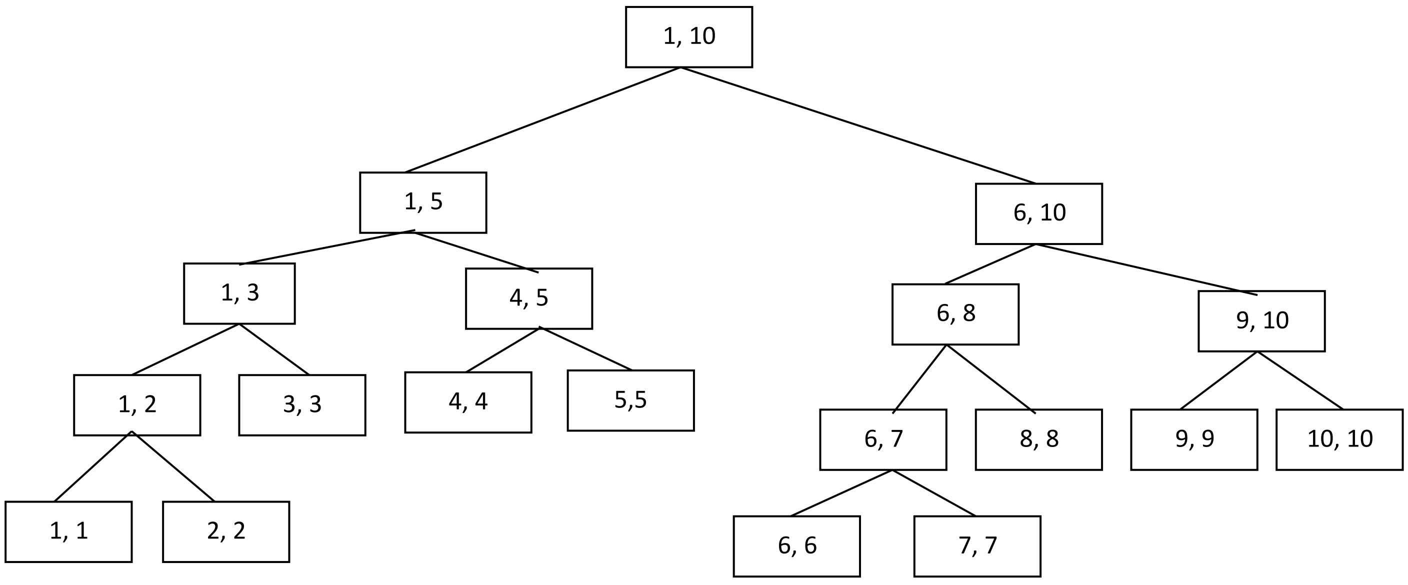
// Dividing Problem into Sub-problems and this “mid” is for finding where to split the set.

```
}
else
{
temp[k]=a[j];
j++;
k++;
}
}
while(i<=mid) do{
temp[k]=a[i];
i++;
k++;
}
while(j<=high) do{
temp[k]=a[j];
j++;
k++;
}
For k=low to high do
a[k]=temp[k];
}
For k:=low to high do a[k]=temp[k];
}
```

Tree call of Merge sort

Consider a example: (From text book)

A[1:10]={310,285,179,652,351,423,861,254,450,520}



“Once observe the explained notes in class room”

Computing Time for Merge sort:

The time for the merging operation in proportional to n, then computing time for merge sort is described by using recurrence relation.

T(n)= a	**if n=1;**
2T(n/2)+ cn	**if n>1**

Here c, a Constants.
If n is power of 2, $n=2^k$

Form recurrence relation

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

2[2T(n/4)+cn/2] + cn

4T(n/4)+2cn

2^2 T(n/4)+2cn

2^3 T(n/8)+3cn

2^4 T(n/16)+4cn

2^k T(1)+kcn

an+cn(log n)

By representing it by in the form of Asymptotic notation O is

T(n)=O(nlog n)

QUICK SORT

Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is O (n log n).

- Auxiliary space used in the average case for implementing recursive function calls is O (log n) and hence proves to be a bit space costly, especially when it comes to large data sets.
- Its worst case has a time complexity of O (n^2) which can prove very fatal for large data sets. Competitive sorting algorithms

Quick sort program

```
#include<stdio.h>
#include<conio.h>
int n,j,i;
void main(){
int i,low,high,z,y;
int a[10],kk;
void quick(int a[10],int low,int high);
int n;
clrscr();
printf("\n \t\t mergesort \n");
printf("\n enter the length of the list:");
scanf("%d",&n);
printf("\n enter the list elements");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
low=0;
high=n-1;
quick(a,low,high);
printf("\n sorted array is:");
for(i=0;i<n;i++)
printf(" %d",a[i]);
getch();
}
int partition(int a[10], int low, int high){
int i=low,j=high;
int temp;
int mid=(low+high)/2;
int pivot=a[mid];
while(i<=j)
{
while(a[i]<=pivot)
i++;
while(a[j]>pivot)
j--;
if(i<=j){
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
        i++;
        j--;
}}
return j;
}
void quick(int a[10],int low, int high)
{
int m=partition(a,low,high);
if(low<m)
quick(a,low,m);
if(m+1<high)
quick(a,m+1,high);
}
```

Algorithm for Quick sort

```
Algorithm quickSort (a, low, high) {
If(high>low) then{
m=partition(a,low,high);
if(low<m) then quick(a,low,m);
if(m+1<high) then quick(a,m+1,high);
}}

Algorithm partition(a, low, high){
i=low,j=high;
```

```
 mid=(low+high)/2;
 pivot=a[mid];
while(i<=j) do { while(a[i]<=pivot)
                    i++;
                  while(a[j]>pivot)
                   j--;
                     if(i<=j){  temp=a[i];
                     a[i]=a[j];
                     a[j]=temp;
                   i++;
         j--;
}}
return j;
}
```

Name	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	
Bubble	O(n)	-	O(n^2)	O(n)
Insertion	O(n)	O(n^2)	O(n^2)	O(n)
Selection	O(n^2)	O(n^2)	O(n^2)	O(n)
Quick	O(log n)	O(n log n)	O(n^2)	O(n + log n)
Merge	O(n log n)	O(n log n)	O(n log n)	O(2n)
Heap	O(n log n)	O(n log n)	O(n log n)	O(n)

Comparison between Merge and Quick Sort:

- Both follows Divide and Conquer rule.
- Statistically both merge sort and quick sort have the same average case time i.e., O(n log n).
- Merge Sort Requires additional memory. The pros of merge sort are: it is a stable sort, and there is no worst case (means average case and worst case time complexity is same).
- Quick sort is often implemented in place thus saving the performance and memory by not creating extra storage space.
- But in Quick sort, the performance falls on already sorted/almost sorted list if the pivot is not randomized. Thus why the worst case time is O(n^2).

Randomized Sorting Algorithm: (Random quick sort)

- While sorting the array a[p:q] instead of picking a[m], pick a random element (from among a[p], a[p+1], a[p+2]---a[q]) as the partition elements.
- The resultant randomized algorithm works on any input and runs in an expected O(n log n) times.

Algorithm for Random Quick sort

```
Algorithm RquickSort (a, p, q) {
If(high>low) then{
If((q-p)>5) then
Interchange(a, Random() mod (q-p+1)+p, p);
 m=partition(a,p, q+1);
quick(a, p, m-1);
quick(a,m+1,q);
}}
```

STRASSEN'S MATRIX MULTIPLICATION:

Let A and B be two n×n Matrices. The product matrix C=AB is also a n×n matrix whose i, j^th^ element is formed by taking elements in the i^th^ row of A and j^th^ column of B and multiplying them to get

C(i, j)=

Here $1 \leq i$ & $j \leq n$ means i and j are in between 1 and n.

To compute C(i, j) using this formula, we need n multiplications.

The divide and conquer strategy suggests another way to compute the product of two n×n matrices.

For Simplicity assume n is a power of 2 that is $n=2^k$ Here k any nonnegative integer.

If n is not power of two then enough rows and columns of zeros can be added to both A and B, so that resulting dimensions are a power of two.

Let A and B be two n×n Matrices. Imagine that A & B are each partitioned into four square sub matrices. Each sub matrix having dimensions n/2×n/2.

The product of AB can be computed by using previous formula.

If AB is product of 2×2 matrices then

=

C11=A11B11+A12B21
C12=A11B12+A12B22
C21=A21B11+A22B21
C22= A21B12+A22B22

Here 8 multiplications and 4 additions are performed.

Note that Matrix Multiplication are more Expensive than matrix addition and

$$T(n) = \begin{cases} b & \text{if } n \leq 2; \\ 8T(n/2) + cn^2 & \text{if } n > 2 \end{cases}$$

subtraction.

Volker strassen has discovered a way to compute the $C_{i,j}$ of above using 7 multiplications and 18 additions or subtractions.

For this first compute 7 n/2×n/2 matrices P, Q, R, S, T, U & V

P=(A11+A22)(B11+B22)
Q=(A21+A22)B11
R=A11(B12-B22)
S=A22(B21-B11)
T=(A11+A12)B22
U=(A21-A11)(B11+B12)
V=(A12-A22)(B21+B22)

C11 = P+S-T+V
C12 = R+T
C21 = P+R-Q+U