UNIT I

INTRODUCTION: Algorithm, pseudo code for expressing algorithms, performance analysis- space complexity, time complexity, asymptotic notation- big (O) notation, omega notation, theta notation and little (o) notation, recurrences, probabilistic analysis, disjoint set operations, union and find algorithms.

1.1 ALGORITHMS

An algorithm named for the ninth century Persian mathematician Abu Ja'far Mohammed ibn Musa al Khowarizmiis simply a set of rules used to perform some calculations, either by hand or more usually on a machine. An algorithm is referred as a method that can be used by the computer for solution of a problem.

Definition:

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. An Algorithm is a method of representing the step-by-step procedure for solving a problem. It is a method of finding the right answer to a problem or to a different problem by breaking the problem into simple cases.

All algorithms must satisfy following criteria:

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

An algorithm can be written in English like sentences or in any standard representations. The algorithm written in English language is called Pseudo code.

Example: To find the average of 3 numbers, the algorithm is as shown below.

```
Step1: Read the numbers a, b, c, and d.
Step2: Compute the sum of a, b, and c.
Step3: Divide the sum by 3.
Step4: Store the result in variable of d.
Step5: End the program.
```

There are various **issues** in the study of algorithms and those are-

1. **How to devise algorithms?**

The creation of an algorithm is a logical activity and one can not automate it. But there are certain algorithmic design strategies, and by using these strategies one can create many useful algorithms. Hence a study of such algorithmic strategy is made to solve certain set of problems.

2. **How to validate algorithms?**

The state after creation of algorithm is to validate the algorithms. The process of checking whether an algorithm computes the correct answer for all possible legal inputs is called algorithm validation. The purpose of validation of algorithm is to find whether algorithm works properly without being dependent upon programming languages. Once validation of algorithm is done then the program can be written using corresponding algorithm.

3. **How to analyze algorithms?**

Analysis of algorithm is a task of determining how much computing time and storage is required by an algorithm. Analysis of algorithm is also called performance analysis. This analysis is based on mathematics. Ana a judgment is often needed about better algorithm when two algorithms get compared. The behavior of algorithm in best case, worst case and average case needs to be obtained.

4. **How to test a program?**

After finding efficient algorithm is is necessary to test that the program written using efficient algorithm behaves properly or not. Testing of a program is an activity that can be carried out in two phases-

i) Debugging
ii) Performance measuring (profiling)

While debugging a program, it is checked whether program produces faulty results for valid set of input and if it is found then the program has to be corrected. Thus by debugging thoroughly the program is corrected.

Profiling is a process of measuring time and space required by a correct program for valid set of inputs.

Development Of An Algorithm

The steps involved in the development of an algorithm are as follows:

- Specifying the problem statement.
- Designing an algorithm.
- Coding.
- Debugging
- Testing and Validating
- Documentation and Maintenance.

Specifying the problem statement: The problem which has to be implemented in to a program must be thoroughly understood before the program is written. Problem must be analyzed to determine the input and output requirements of the program.

Designing an Algorithm: Once the problem is cleared then a solution method for solving the problem has to be analyzed. There may be several methods available for obtaining the required solution. The best suitable method is designing an Algorithm. To improve the **clarity** and **understandability** of the program flowcharts are drawn using algorithms.

Coding: The actual program is written in the required programming language with the help of information depicted in flowcharts and algorithms.

Debugging: There is a possibility of occurrence of errors in program. These errors must be removed for proper working of programs. The process of checking the errors in the program is known as 'Debugging'.
There are three types of errors in the program.

- Syntactic Errors: They occur due to wrong usage of syntax for the statements.
  Ex: x=a*%b
  Here two operators are used in between two operands.
- Runtime Errors : They are determined at the execution time of the program
  Ex: Divide by zero
  Range out of bounds.
- Logical Errors : They occur due to incorrect usage of instructions in the program. They are neither displayed during compilation or execution nor cause any obstruction to the program execution. They only cause incorrect outputs.

Testing and Validating: Once the program is written , it must be tested and then validated. i.e., to check whether the program is producing correct results or not for different values of input.

Documentation and Maintenance: Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references.
Maintenance is the process of upgrading the program, according to the changing requirements.

1.2 PSEUDO CODE FOR EXPRESSING ALGORITHMS

Algorithm is basically sequence of instructions written in simple English language. Based on algorithm there are two more representations used by programmer and those are flow chart and pseudo-code. **Flowchart** is a **graphical representation** of an algorithm. Similarly **pseudo code** is a **representation** of algorithm in which instruction sequence can be given with the help of **programming constructs.**

The conventions used for writing an algorithm using pseudo-code.

1. Algorithm is a procedure consisting of heading and body. The heading consists of name of the procedure and parameter list. The syntax is
   Algorithm name_of_procedure(parameter1, parameter2,………….,parameter N)

2. The beginning and end of block should be indicated by { and } respectively. The compound statements must be enclosed within { and } brackets.

3. The delimiters ; are used at the end of each statement.

4. Single line comments are written using // as beginning of comment.

5. The identifier should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.

6. Using assignment operator := an assignment statement can be given.
   Eg: variable:=expression

7. There are other types of operators such as Boolean operators such as True or False. Logical operators such as and or not. And relational operators such as <, <=, >, >=,≠.

8. The array indices are stored within [and] brackets. The index of array usually starts at zero. The multi dimensional array can also be used in algorithm.

9. The input and output can be done using read and write.

10. The conditional statements such as if-then or if-then-else are written in following form:
    if(condition) then statement
    if(condition) then statement else statement
    If the if-then statement is of compound type then { and } should be used for enclosing block.

11. While statement can be written as:
    while(condition) do

```
{
statement 1
statement 2
    :
    :
statement n
}
```

While the condition is true the block enclosed with { } gets executed otherwise statement after } will be executed.

12. The general form for writing for loop is:

```
for variable := value1 to valueN step do
{
statement 1
statement 2
    :
    :
statement n
}
```

Here value value 1 is initialization condition and value n is a terminating condition. The step indicates the increments or decrements in value 1 for executing the for loop.

13. The repeat-until statement can be written as:

```
Repeat
{
statement 1
statement 2
    :
    :
statement n
}
until(condition)
```

14. The break statement is used to exit from inner loop. The return statement is used to return control from one point to another. Generally used while exiting from function.

Example 1:
Write an algorithm to count the sum of n numbers.

```
Algorithm sum(1, n)
{
result := 0;
for i:= 1 to n do i := i+1
result := result + 1;
}
```

Example 2:
Write an algorithm to check whether given number is even or odd.

```
Algorithm evenodd(n)
{
if(n%2 == 0) then;
write("Given number is even");
else
write("Given number is odd");
}
```

Example 3:
Write an algorithm for sorting the elements.

```
Algorithm sort(a, n)
{
for i:= 1 to n do
for j:= i+1 to n-1 do
{
if(a[i] > a[j]) then
{
temp := a[i];
a[i] := a[j];
a[j] := temp;
}
}
write("List is sorted");
}
```

Example 4:
Write an algorithm to find factorial of n number.

```
Algorithm fact(n)
{
iIf n := 1 then
   return1;
else
   return(n*fact(n-1));
}
```

Example 5:
Write an algorithm to perform multiplication of two matrices.

```
Algorithm Mul(A, B, n)
{
for i:= 1 to n do
for j:= 1 to n do
        c[i, j] := 0;
for k:= 1 to n do
        c[i, j] = c[i, j] + A[i, k] * B[k, j];
}
```

1.3 PERFORMANCE ANALYSIS

When several algorithms can be designed for the solution of a problem, there arises the need to determine which among them is the best. The efficiency of a program or an algorithm is measured by computing its time and/or space complexities.

- The **time complexity** of an algorithm is a function of the running time of the algorithm.
- The **space complexity** is a function of the space required by it to run to completion.
- The time complexity is therefore given in terms of **frequency count**.
- Frequency count is basically a count denoting number of times of execution of statement.

How to compute time complexity

1	Algorithm Message(n)	0
2	{	0
3	for i=1 to n do	n+1
4	{	0
5	write(“Hello”);	n
6	}	0
7	}	0
	total frequency count	2n+1

While computing the time complexity we will neglect all the constants, hence ignoring 2 and 1 we will get n. Hence the time complexity becomes O(n).

1	Algorithm add(A,B,m,n)	0
2	{	0
3	for i=1 to m do	m+1
4	for j=1 to n do	m(n+1)
5	C[i,j] = A[i,j]+B[i,j]	mn
6	}	0
	total frequency count	2mn+2m+1

By neglecting the constants, we get the time complexity as $O(n^2)$. The maximum degree of the polynomial has to be considered.

Best Case, Worst Case and Average Case Analysis

- If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called best case complexity.
- If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called worst case time complexity.
- The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called average case time complexity.

Space Complexity

- The space complexity can be defined as amount of memory required by an algorithm to run.
- To compute the space complexity we use two factors: constant and instance characteristics. The space requirement S(p) can be given as
  S(p) = C + Sp

where C is a constant i.e.. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers.

- Sp is a space dependent upon instance characteristics. This is a variable part whose space requirement depend on particular problem instance.

Eg:1

```
Algorithm add(a,b,c)
{
return a+b+c;
}
```

If we assume a, b, c occupy one word size then total size comes to be 3
S(p) = C

Eg:2
```
Algorithm add(x,n)
{
sum=0.0;
foe i= 1 to n do
sum:=sum+x[i];
return sum;
}
```

$S(p) \geq (n+3)$
The n space required for x[], one space for n, one for i, and one for sum

1.4 ASYMPTOTIC NOTATIONS:

- To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.
- Using asymptotic notations we can give time complexity as “fastest possible”, “slowest possible” or “average time”.
- Various notations such as Ω, θ, O used are called asymptotic notions.

Big Oh Notation

Big Oh notation denoted by ‘O’ is a method of representing the upper bound of algorithm’s running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.
Definition:

Let, f(n) and g(n) are two non-negative functions. And if there exists an integer n_0 and constant C such that C > 0 and for all integers $n > n_0$, $f(n) \leq c*g(n)$, then
$f(n) = Og(n)$.

Example:

Consider the function $f(n) = 2n+2$ and $g(n)=n^2$, find constant C so that $f(n)\leq g(n)$, in other words $2n+2\leq n^2$ then find that for C = 1 or 2 f(n) is greater than g(n). That means when n=1 f(n)=4 and g(n)=1 for n=2, f(n)=6 and g(n)=4. When $n > 2$ we obtain $f(n) < g(n)$. Then we obtain $O(n^2)$ for $n > 2$.

Various meanings associated with big-oh are

O(1)	constant computing time
O(n)	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
O(logn)	logarithmic

The relationship among these computing time is
$O(1)< O(\log n)< O(n)< O(n\log n)< O(n^2)< O(2^n)$

Omega Notation

Omega notation denoted 'Ω' is a method of representing the lower bound of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm to complete.

Definition:

Let, f(n) and g(n) are two non-negative functions. And if there exists an integer n_0 and constant C such that $C > 0$ and for all integers $n > n_0$, $f(n) >c*g(n)$, then
$f(n) = \Omega\ g(n)$.

Example:

Consider $f(n) = 2n+5$ and $g(n) = 2(n)$ then $2n+5\geq 2n$ for $n>1$. Hence $2n+5=\Omega(n)$.

Theta Notation

Theta notation denoted as 'θ' is a method of representing running time between upper bound and lower bound.

Definition:

Let, f(n) and g(n) are two non-negative functions. There exists positive constants C_1 and C_2 such that C_1 g(n) ≤ f(n) ≤ C_2 g(n) and f(n) = θ g(n)

Example:

If f(n)=2n+8>5n where n≥2; 2n+8≥6n where n≥2 and 2n+8<7n where n≥2. Hence 2n+8 = θ(n) such that constants C_1 = 5, C_2 = 7 and n_0 = 2.

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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

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)

1.5 RECURENCES

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.
There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

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

<= cn/2Log(n/2) + n

= cnLogn - cnLog2 + n

= cnLogn - cn + n

<= cnLogn

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

For example consider the recurrence relation

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

```
        cn^2
      /      \
  T(n/4)     T(n/2)
```

If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.

```
            cn^2
        /          \
      c(n^2)/16      c(n^2)/4
     /     \         /     \
  T(n/16)     T(n/8)  T(n/8)    T(n/4)
```
Breaking down further gives us following
```
               cn^2
           /           \
      c(n^2)/16         c(n^2)/4
       /     \           /      \
c(n^2)/256   c(n^2)/64  c(n^2)/64    c(n^2)/16
 /    \      /    \    /    \       /    \
```

To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

1.6 PROBABILISTIC ANALYSIS

In analysis of algorithms, probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or a computational problem. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm.

This approach is not the same as that of probabilistic algorithms, but the two may be combined.

For non-probabilistic, more specifically, for deterministic algorithms, the most common types of complexity estimates are the average-case complexity (expected time complexityand the almost always complexity. To obtain the average-case complexity, given an input distribution, the expected time of an algorithm is evaluated, whereas for the almost always complexity estimate, it is evaluated that the algorithm admits a given complexity estimate that almost surely holds.

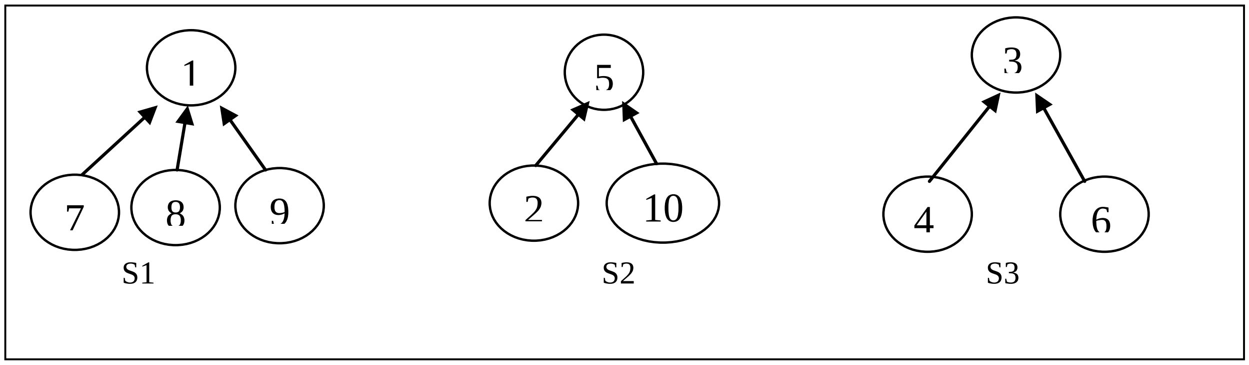
In probabilistic analysis of probabilistic (randomized) algorithms, the distributions or averaging for all possible choices in randomized steps are also taken into an account, in addition to the input distributions.

1.7 DISJOINT SET OPERATIONS(UNION AND FIND ALGORITHMS)

Disjoint Sets: If S_i and S_j, $i \neq j$ are two sets, then there is no element that is in both S_i and S_j..
For example: n=10 elements can be partitioned into three disjoint sets,

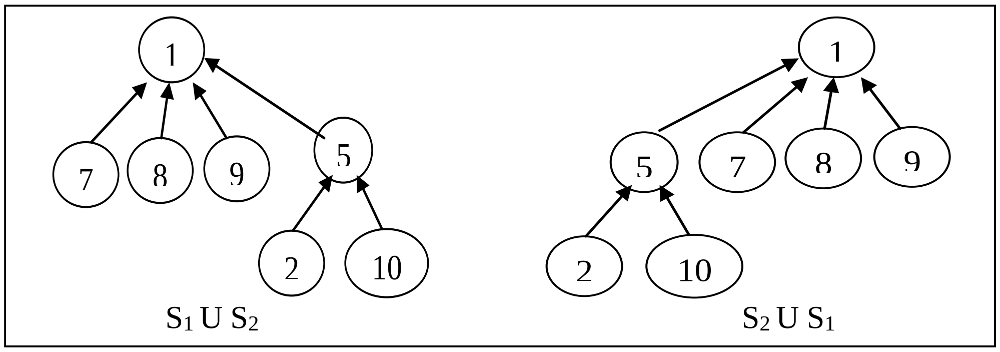
$$\boxed{\begin{array}{c} \mathbf{S_1 = \{1, 7, 8, 9\}} \\ \mathbf{S_2 = \{2, 5, 10\}} \\ \mathbf{S_3 = \{3, 4, 6\}} \end{array}}$$

Tree representation of sets:



Disjoint set Operations:
- Disjoint set Union
- Find(i)

Disjoint set Union: Means Combination of two disjoint sets elements. Form above example S_1 U S_2 ={1,7,8,9,5,2,10 }
For S_1 U S_2 tree representation, simply make one of the tree is a subtree of the other.

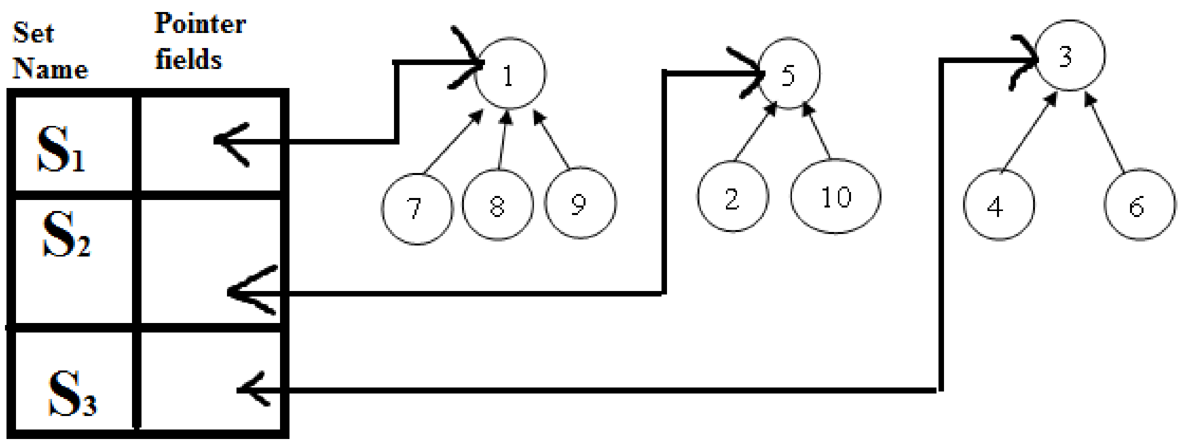


Find: Given element i, find the set containing i.
Form above example:

Find(4)→S_3
Find(1)→S_1
Find(10)→S_2

Data representation of sets:

Tress can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.



For presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them.
For example: if we determine that element ‘i’ is in a tree with root ‘j’ has a pointer to entry ‘k’ in the set name table, then the set name is just **name[k]**

For unite (**adding or combine**) to a particular set we use FindPointer function.
Example: If you wish to unite to S_i and S_j then we wish to unite the tree with roots FindPointer (S_i) and FindPointer (S_j)
FindPointer→ is a function that takes a set name and determines the root of the tree that represents it.
For determining operations:
Find(i)→ 1St determine the root of the tree and find its pointer to entry in setname table.
Union(i, j)→ Means union of two trees whose roots are i and j.

If set contains numbers 1 through n, we represents tree node
P[1:n].
n→Maximum number of elements.

Each node represent in array

i	1	2	3	4	5	6	7	8	9	10
P	-1	5	-1	3	-1	3	1	1	1	5

Find(i) by following the indices, starting at i until we reach a node with parent value -1.
Example: Find(6) start at 6 and then moves to 6’s parent. Since P[3] is negative, we reached the root.

Algorithm for finding Union(i, j):	Algorithm for find(i)

Algorithm Simple union(i, j)
{
P[i]:=j; // Accomplishes the union
}
	Algorithm SimpleFind(i)
	{
	While(P[i]≥0) do i:=P[i];
	return i;
	}

If n numbers of roots are there then the above algorithms are not useful for union and find.
For union of n trees→ Union(1,2), Union(2,3), Union(3,4),…..Union(n-1,n).
For Find i in n trees→ Find(1), Find(2),….Find(n).

Time taken for the union (simple union) is → O(1) (constant).
For the n-1 unions→ O(n).

Time taken for the find for an element at level i of a tree is → O(i).
For n finds → O(n^2).

To improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. For this we use a weighting rule for union(i, j)