Design & Analysis of Algorithms.

→ Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran. "Fundamentals of Computer Algorithms", 2nd Edition.

Syllabus

unit - I :

Introduction : 1. Algorithm Definition, 2. Algorithm Specification, 3. performance Analysis, 4. performance measurement, 5. Asymptotic notation, 6. Randomized Algorithms.

Sets & Disjoint set union : 1. Introduction, 2. union and find operations.

Basic Traversal & search Techniques : 1. Techniques for Graphs, 2. Connected components and spanning trees, 3. Bi-Connected components and DFS.

unit - II :

Divide and conquer : 1. General Method, 2. Defective chessboard, 3. Binary search, 4. finding the maximum and minimum, 5. merge sort, 6. Quick sort.

The Greedy method : 1. The General Method, 2. container loading, 3. knapsack problem, 4. Job sequencing with dead lines, 5. Minimum cost spanning trees.

unit - III :

Dynamic programming : 1. The general method, 2. multistage graphs, 3. All pairs - shortest paths, 4. single source shortest paths : general weights, 5. Optimal binary search trees,

6. 0/1 knapsack, 7. Reliability design, 8. The travelling sales person problem, 9. matrix chain multiplication.

unit - IV :

Backtracking : 1. The General method, 2. The 8 - Queens problem, 3. Sum of subsets, 4. Graph coloring, 5. Hamiltonian cycles, 6. knapsack problem.

Branch and Bound : 1. FIFO Branch & Bound, 2. LC Branch & Bound, 3. 0/1 knapsack problem, 4. Travelling salesman person problem.

Unit - V :

Np-Hard and Np - complete problems : 1. Basic Concepts, 2. Cook's Theorem.

String Matching : 1. Introduction, 2. String matching - Meaning and Application 3. Naive string Matching Algorithm, 4. Rabin Karp algorithm, 5. Knuth - Morris - pratt Automata, 6. Tries, 7. Suffix Tree.

Unit-I

Introduction - chapter-1

* An algorithm, that word comes from the name of a "persian Author, Abu Ja'far Mohammed ibn al Musa al-Khowarizmi" - 9th century mathematician.
* 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. This is what makes algorithm different from words such as process, technique, or method.

1. Algorithm Definition:

Def 1: Basically algorithm is a finite set of instructions that can be used to perform certain task.

Def 2: The algorithm is defined as a collection of unambiguous instructions occurring in some specific sequence and such an algorithm should produce output for given set of input in finite amount of time.

*Notational representation:

→ After understanding the problem statement we have to create an algorithm carefully for the given problem.
→ The algorithm is then converted into some programming language and then given to some computing device (computer).
→ The computer then executes this algorithm which is actually submitted in the form of source program.
→ During the process of execution it requires certain set of input.

→ with the help of algorithm (in the form of program) and input set, the result is produced as an output.

→ If the given input is invalid then it should raise appropriate error message; otherwise correct result will be produced as an output.

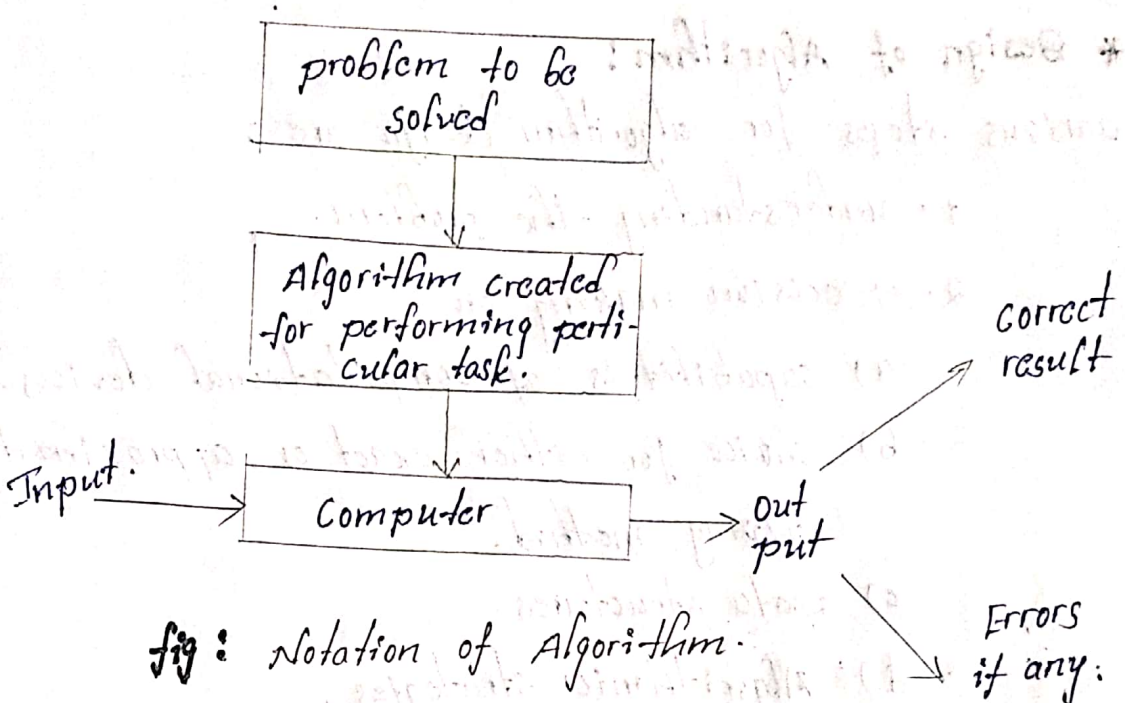


fig: Notation of Algorithm.

* properties of Algorithm :

Algorithm should posses following properties:

1) **Input** : The input of zero or more number of quantities should be given to the algorithm.

2) **Output** : The algorithm should produce at least one quantity, as an output.

3) **Definiteness** : Each instruction is clear and unambiguous.

4) **Finiteness** : An algorithm should be finite. That means after finite number of steps it should be terminate.

5) **Effectiveness** : Every step of algorithm should be feasible, and it must be very basic so that it can be carried out, in principle, by a person using only pencil & paper. (practical)

The implementation of such algorithms can be done by programming language. Hence the term program can be defined as -

"program is the expression of algorithm using some programming language."

* Design of Algorithm:

various steps for algorithm design are:

1. understanding the problem.
2. Decision making on
   a) capabilities of computational devices.
   b) choice for either exact or approximate problem solving method.
   c) Data structures.
   d) Algorithmic strategies.
3. Specification of Algorithm.
4. Algorithmic Verification.
5. Analysis of Algorithm.
6. Implementation or coding of algorithm.

Let us now discuss each step in detail.

1) understanding the problem:

→ This is the very first step in designing of algorithm. In this step first of all you need to understand the problem statement completely.

→ while understanding the problem statements, read the problem description carefully and ask questions for clarifying the doubts about the problem.

→ But there are some types of problems that are commonly occuring and to solve such problems there are typical algorithms which are already available. Hence if the given problem is common type of problem, then already existing algorithms acts as a solution to that problem.

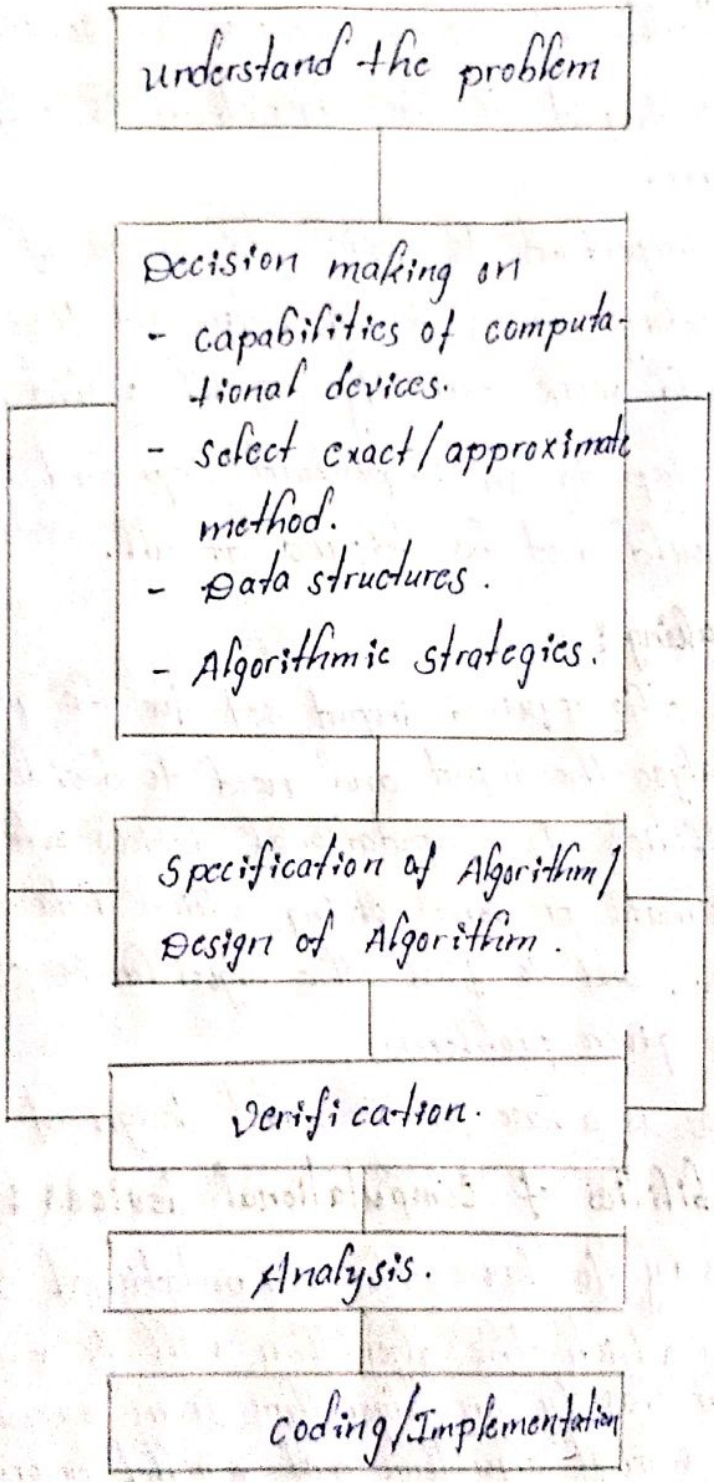


Fig: Algorithm design steps.

→ After applying such an existing algorithm, it is necessary to find its strength and weakness (For example, efficiency, memory utilization), but it is very rare to have such a ready made algorithm. Normally you have to design an algorithm on your own.

→ After carefully understanding the problem statements find out what are the necessary inputs for solving that problem. The input to the algorithm is called instance of the problem.

→ It is very important to decide the range of inputs so that the boundary values of algorithm get fixed. The algorithm should work correctly for all valid inputs.

∴ This step is an important step and in algorithmic solving it should not be skipped at all.

2) Decision Making:

→ After finding the required input set for the given problem we have to analyze the input and need to decide certain issues such as capabilities of computational devices whether to use exact or approximate problem solving, which data structures has to be used, and to find the algorithmic technique for solving the given problem.

→ This step serve as a base for the actual design of algorithm.

a) Capabilities of computational devices:

→ It is necessary to know the computational capabilities of devices on which the algorithm will be running.

→ Globally we can classify an algorithm from execution point of view as sequential algorithm and parallel algorithm.

→ The sequential algorithm specifically runs on the machine in which the instructions are executed one after another. such a machine is called as Random Access Machine (RAM).

→ the parallel algorithms are run on the machine in which the instructions are executed in parallel.
→ There are certain complex problems require huge amount of memory, so the execution time is an important factor
→ For solving such problems it is essential to have proper choice of a computational device which is space and time efficient.

b) choice for either exact or approximate problem solving method:

→ The next important decision is to decide whether the problem is to be solved exactly or approximately.
→ If the problem needs to be solved correctly then we need exact algorithm.
→ If the problem is so complex that we won't get the exact solution then in that situation we need to choose approximation algorithm, the typical example of approximation algorithm is travelling salesperson problem.

c) Data structures:

→ Data structure and algorithm work together and these and independent.
→ Hence choice of proper data structure is required before designing the actual algorithm. The implementation of algorithm (program) is possible with the help of algorithm and data structure.

d) Algorithmic strategies:

→ Algorithmic strategy is a general approach by which many problems can be solved algorithmically.
→ These problems may be belong to different areas of computing. Algorithmic strategies are also called as algorithmic techniques or algorithmic paradigm.

Algorithm design Techniques:

Brute force: This is straight forward technique with naive approach.

Devide and conquer: The problem is divided into smaller instances.

Dynamic programming: The results of smaller, recounting instances are obtained to solve the problem.

Greedy technique: To solve the problem locally optimal decisions are made.

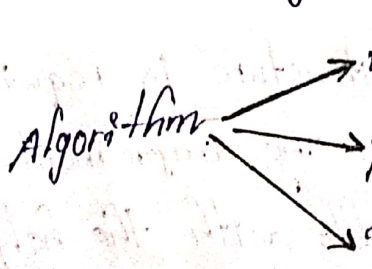
Back tracking: This method is based on the trial and error. if we want to solve some problem then desired solution is chosen from the finite set S.

→ various algorithmic strategies are classified according the design idea that the particular algorithm adopts.

→ If using certain design idea the particular problem is getting solved then that problem belongs to corresponding algorithmic strategy.

3) Specification of algorithm:

→ There are various ways by which we can specify an algorith



→ using natural language:

→ It is very simple to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear, and there by we get brief specification.

Ex: write an algorithm to perform addition of two numbers.

step 1 : Read the first number say a.
step 2 : Read the second number say b.
step 3 : Add the two numbers and store the result in a variable c.
step 4 : Display the result.

→ such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of pseudo code.

→ **using pseudo code :**

→ pseudo code is nothing but a combination of natural language and programming language constructs. A pseudo code is usually more precise than a natural language.

Ex : write an algorithm for performing addition of two numbers.

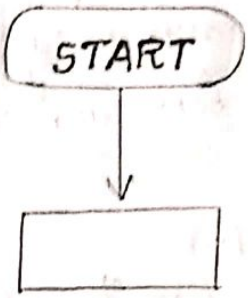
```
Algorithm sum(a,b)
// problem Description - This algorithm performs addition
   of two integers
// Input : two integers a and b.
// Output : addition of two integers.
c ← a+b
write(c).
```

→ This specification is more useful in implemention point of view.

→ **using Flowchart :**

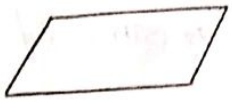
→ Another way of representing the algorithm is by flowchart. Flowchart is a graphical representation of an algorithm.

→ Typical symbols used in flowchart are -

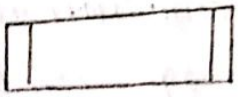


start state.

Transition.



progressing or assignment statements.

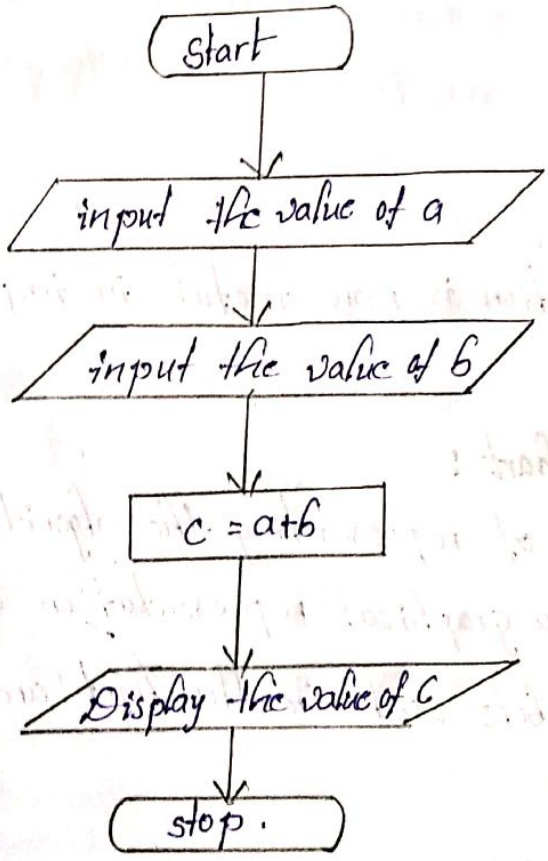


input - output statement.

Conditional statement.

stop state.

Ex: Addition of two numbers.



4) **Algorithmic Verification :**

→ Algorithmic verification means checking correctness of an algorithm.

→ After specifying an algorithm we go for checking its correctness, we normally check whether the algorithm gives correct output in finite amount of time for a valid set of input.

→ The proof of correctness of an algorithm can be complex sometimes. A common method of proving the correctness of an algorithm can be is by using mathematical induction.

→ But to show that an algorithm works incorrectly we have to show that at least for one instance of valid input, the algorithm gives wrong result.

5) **Analysis of Algorithm :**

→ while analyzing an algorithm we should consider following factors -
- Time efficiency of an algorithm.
- Space efficiency of an algorithm
- Simplicity of an algorithm.
- Generality of an algorithm.
- Range of input.

→ **Time complexity** of an algorithm means the amount of time taken by an algorithm to run. By computing time complexity we come to know whether the algorithm is slow or fast.

→ **Space complexity** of an algorithm means the amount of space (memory) taken by an algorithm. By computing space complexity we can analyze whether an algorithm requires more or less space.

→ **Simplicity** of an algorithm means generating sequence of instructions which are easy to understand. This is an important characteristic of an algorithm because simple algorithms can be understood quickly and one can then write simpler programs for such algorithms. while simplifying an algorithm we have to compute any predefined computations or some complex,

mathematical derivation. Finding out bugs from algorithms or debugging the program becomes easy when an algorithm is simple. Sometimes simpler algorithms are more efficient than complex algorithms. But it is not always possible that the algorithm is simple.

→ **Generality** shows that sometimes it becomes easier to design an algorithm in more general way rather than designing it for particular set of input. Hence we should write general algorithms always. For example designing an algorithm for finding GCD of any two numbers is more appealing than that of particular two values. But some times it is not at all required to design a generalized algorithm. For example, an algorithm for finding roots of quadratic equations can be designed to handle a polynomial of arbitrary degree.

→ **Range of inputs** comes in picture when we execute an algorithm. The design of an algorithm should be such that it should handle the range of input in which is the most natural to corresponding problem.

∴ Analysis of an algorithm means checking the characteristics such as : Time complexity, space complexity, simplicity, generality and range of input. If these factors are not satisfactory then we must redesign the algorithm.

* Testing a program :

→ Testing a program is an activity carried out to expose as many errors as possible and to correct them.

→ There are two phases for testing a program :

1. Debugging.
2. profiling.

→ Debugging is a technique in which a simple set of data is tested to see whether faulty results occur or not. If any faulty result occurs then those results are corrected.
→ But in debugging technique only presence of error is pointed out. Any hidden error can not be identified. Thus in debugging we cannot verify correctness of output on sample data. Hence profiling concept is introduced.
→ profiling or performance measurement is the process of executing a correct program on a sample set of data. Then the time and space required by the program to execute is measured.
→ Then analysis is made on program for further optimization.

2. Algorithm Specification :

- pseudocode conventions.
- Recursive Algorithms.

* pseudo code Conventions :

→ Algorithm is basically a sequence of instructions written in simple English language.
→ The algorithm is broadly divided into two sections -

Algorithm heading
It consists of name of algorithm, problem description, input and output.
Algorithm body
It consists of logical body of the algorithm by making use of various programming constructs and assignment statement.

Fig : structure of Algorithm.

Rules: let us understand some rules for writing the algorithm.

1. Algorithm is procedure consisting of heading and body. The heading consists of keyword Algorithm and name of the algorithm and parameter list. The syntax is

Algorithm name ($p_1, p_2, \cdots, p_n$)

This keyword should be written first

Here write the name of an algorithm

Write parameters (if any).

2. Then in heading section we should write following things:

// problem Description:
// Input:
// output:

3. Then body of an algorithm is written, in which various programming constructs like if, for, while, or some assignment statements may be written.

4. The compound statements should be enclosed within { and } braces.

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

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

It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself. Basic data types used are integer, float, char, boolean and so on. The pointer type is also used to point memory location. The compound data type such as structure or record can also be used.

7. Using assignment operator ← an assignment statement can be given.
For instance : Variable ← Expression.

8. 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 <, <=, >, >=, =, ≠.

9. The array indices are stored with in square brackets '[' ']'. The index of array usually start at zero. The multidimensional arrays can also be done using read and write used in algorithm.

10. The inputting and outputting can be done using read and write.
For Example :

```
write ("This msg will be displayed on console");
read (val);
```

11. 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.

12. 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.

13. The general form for writing for loop is :

```
for variable ← value1 to valuen do
{
    statement 1
    statement 2
      :
    statement n.
}
```

Here value1 is initialization condition and valuen is a terminating condition.

Sometime a keyword step is used to denote increament the value of variable for example.

```
for i ← 1 to n step 1
{
    write (i)
}
```

Here variable i is incremented by 1 at each iteration.

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

```
repeat
    statement 1
    statement 2
      :
    statement n.
until (condition).
```

15. 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 existing from function.

Note that statements in an algorithm executes in sequential order i.e. in the same order as they appear - one after the other.

Examples:

1. write an algorithm to count the sum of n numbers.

```
1. Algorithm sum (1, n)
2. // problem Description: This algorithm is for finding the
3. // sum of given n numbers.
4. // Input : 1 to n numbers.
5. // Output : The sum of n numbers.
1       return ← 0
6.      result ← 0
7.      for i ← 1 to n do i ← i+1
8.          result ← result+i
9.      return result:
```

2. Selection sort algorithm.

```
1. for i:= 1 to n do
2. {
3. Examine a[i] to a[n] and suppose
4. the smallest element is at a[j];
5. Interchange a[i] and a[j];

6. }
```

To turn algorithm into a pseudocode program, two clearly defined subtasks remain: finding the smallest element (say a[j]) and interchanging it with a[i].

```
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];
10.        a[i] := a[j];
11.        a[j] := t;
12.    }
13. }
```

Recursive Algorithms :

Def : A recursive function is a function that is defined in terms of itself.

→ we have two types of Recursive algorithms.

- Direct recursive.
- Indirect recursive.

Direct recursive - An algorithm is said to be recursive if the same algorithm is invoked in the body. An algorithm that calls itself is "direct recursive."

Indirect recursive - Algorithm A is said to be "indirect recursive" if it calls another algorithm which in turn calls A.

→ These recursive mechanisms are extremely powerful, but even more importantly, many times they can express an otherwise complex problems very clearly.

→ The following ~~three~~ examples show how to develop a recursive algorithm.

Ex: [Towers of Honoi]

→ The towers of Honoi puzzle is fashioned after the ancient Tower of Brahma ritual.

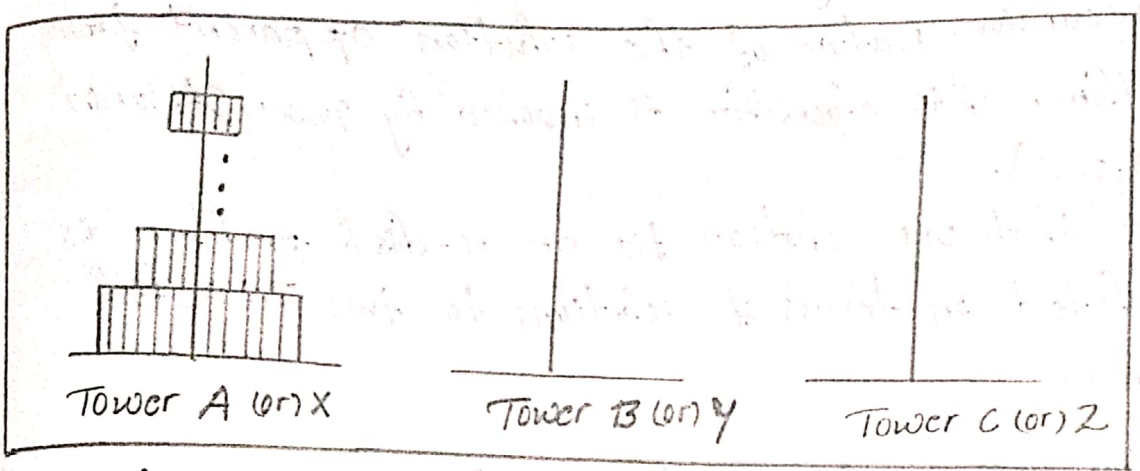


Fig: Towers of Honoi

→ According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.

→ The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.

→ Besides this tower there were two other diamond towers (labeled B and C)

→ Since the time of creation, Brahman priests have been attempting to move the disks from tower A to tower B using tower C for intermediate storage.

→ As the disks are very heavy, they can be moved only one at a time. In addition, at no time can a disk be on top of a smaller disk.

→ According to legend, the world will come to an end when the priests have completed their task.

→ A very elegant solutions results from the use of recursion.

→ Assume that the no. of disks is n. To get the largest disk to the bottom of tower B, we move the remaining n-1 disks to the ~~bottom~~ tower C and then move the largest to tower B.

→ Now we left with the task of moving the disks from tower C to tower B, To do this we have towers A and B available.

→ The fact that tower B has a disk on it can be ignored as the disk is larger than the disks being moved from tower C and so any disk can be placed on top of it.

→ The recursive nature of the solution apparent from Algorithm. This algorithm is invoked by TowersOfHanoi (n, A, B, C).

→ Observe that our solution for an n-disk problem is formulated in terms of solutions to two (n-1)-disk problems.

```
1. Algorithm TowersOfHanoi (n, x, y, z)
2. // Move the top n disks from tower x to tower y.
3. {
4.    if (n ≥ 1) then :
5.    {
6.       TowersOfHanoi (n-1, x, z, y);
7.       write ("move top disk from tower", x,
8.          " to top of tower", y) ;
9.       TowersOfHanoi (n-1, z, y, x);
10.   }
11. }
```

Towers of Hanoi.

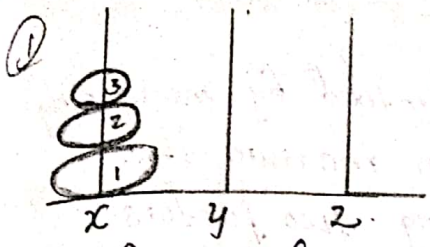
→ Let us take three disks to see the funtionality of Towers of Hanoi algorithm.

Here n = 3 (no of disks).

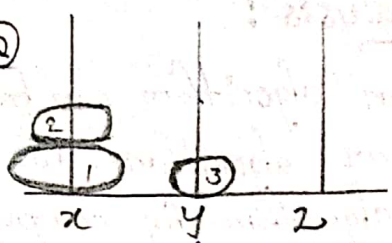
Source tower = x.

Destination tower = y

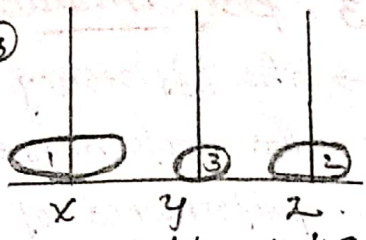
using tower = z.



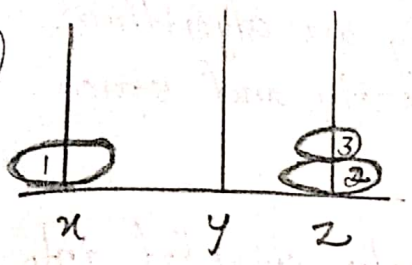
three disks.



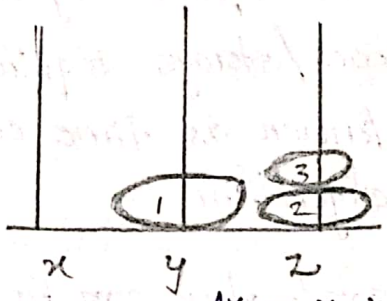
move from x-y.



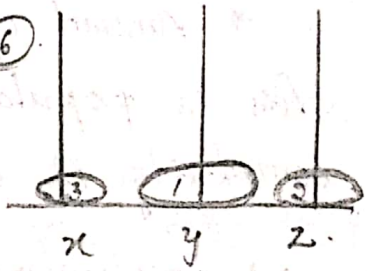
move from x to z.



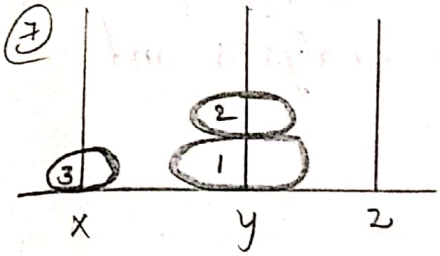
move from y-z



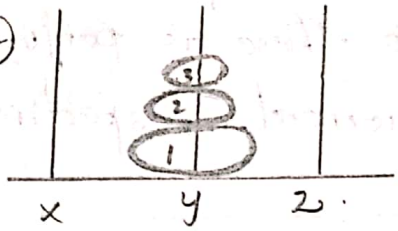
move from x-y.



move from z-x.



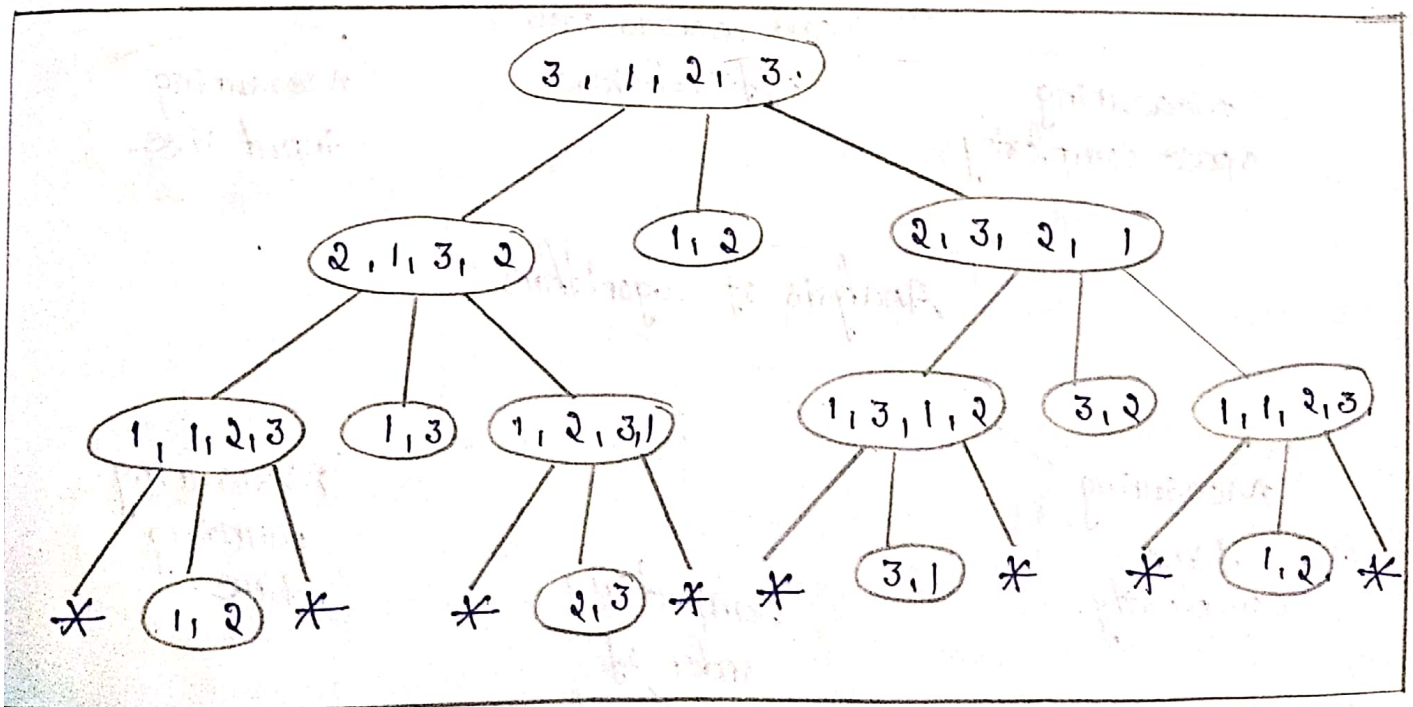
move from z-y.



move from x-y.

simple Graphical representation.

→ simple flow chart representation.



(1,2), (1,3), (2,3), (1,2), (3,1), (3,2), (1,2)

are the moves for given example.

3. performance Analysis :

→ The efficiency of an algorithm can be decided by measuring the performance of an algorithm. we can measure the performance of an algorithm by computing two factors.

* Amount of time required by an algorithm to execute.
* Amount of space/storage required by an algorithm.

This is popularly known as time complexity and space complexity of an algorithm.

> **Note:** performance evaluation can be loosely divided into two major phases : (1) a priori estimates and (2) a posteriori testing. we refer to these as performance analysis and performance measurement, respectively.

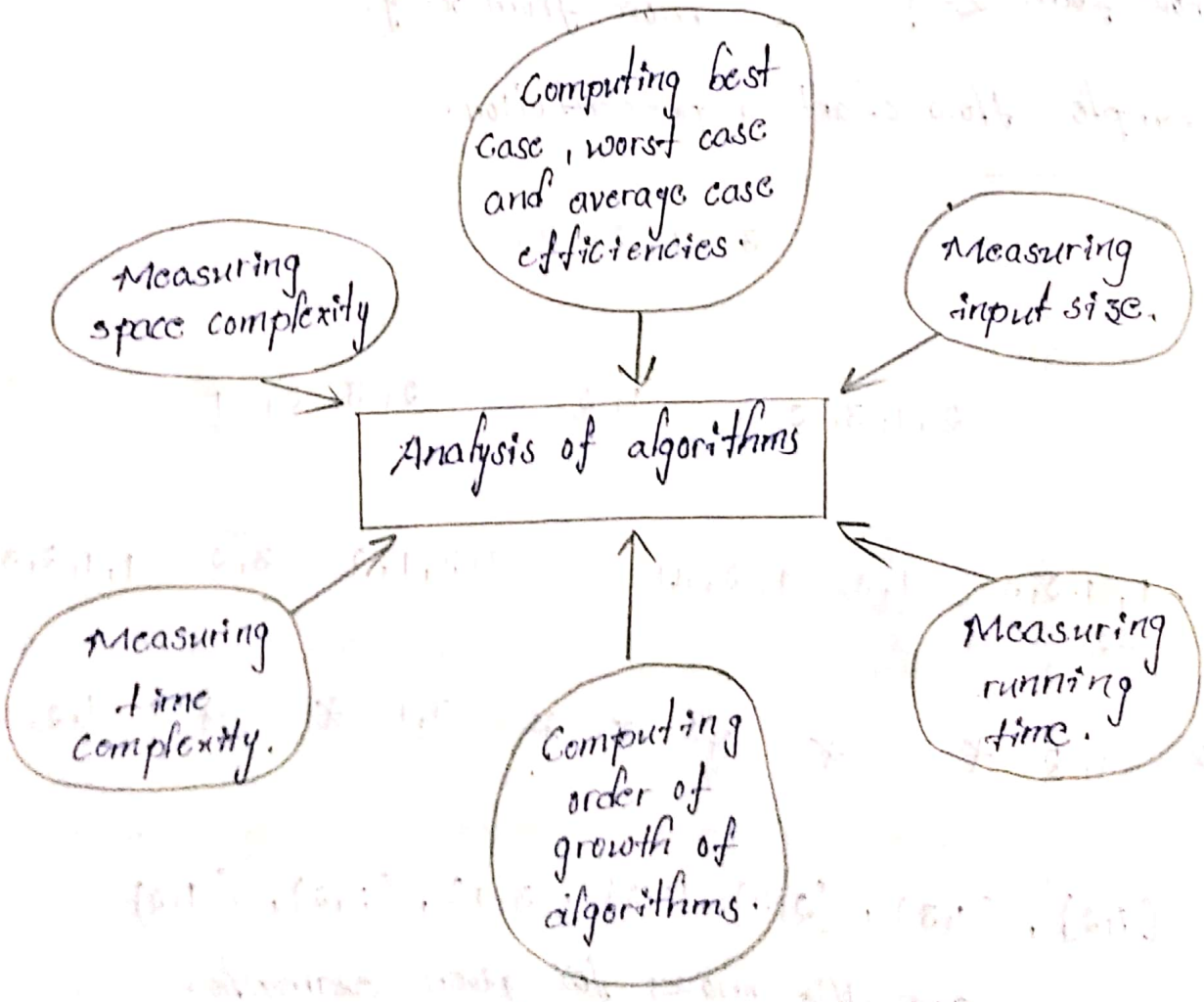


Fig : Analysis of Algorithms.

space complexity :

→ The space complexity can be devided as amount of memory required by an algorithm to run.

→ To compute the space complexity we use two factors :
- constant characteristics.
- instance characteristics.

→ The space requirement S(P) can be given as :

$$S(P) = C + Sp.$$

where C = 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, and

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

Ex-1 : Algorithm Add (a,b,c)
// problem Description : This algorithm computes the addition.
// of three elements.
// Input : a,b and c are of floating type.
// Output : The addition is returned.
return a+b+c

The space requirement for algorithm given in example is

$$S(P) = C \quad \Theta(Sp) = 0.$$

If we assume that a,b and c occupy one word size then total size comes to be 3.

Ex-2 : Algorithm Add (x,n)
// problem Description : The algorithm performs addition of
// all the elements in an array. Array is of floating type.
// Input : An array x and n is total no.of elements in array.
// Output : returns sum which is one of data type float.

```
sum ← 0.0
for ← i ← 1 to n do
    sum ← sum + x[i]
return sum.
```

The space requirement to the above algorithm is -

$S(p) \geq (n+3)$.

The 'n' space required for x[ ], one unit space for n, one unit for i and one unit for sum.

Time complexity :

→ The time complexity of an algorithm is the amount of time required by an algorithm to run to completion.

→ "It is difficult to compute the time complexity in terms of physically clocked time."

→ For instance in multiuser system, executing time depends on many factors such as - System load.
- No. of other programs running.
- Instruction set used.
- Speed of underlying hardware.

→ The time compexity is therefore given in terms of "frequency count."

→ Frequency count is a count denoting no. of times of execution of statement.

[redacted] **Ex :** The algorithm for swapping of two numbers.

```
Algorithm swap(a,b) ---> 0
{                   ---> 0
    temp = a;       ---> 1
    a = b;          ---> 1
    b = temp;       ---> 1
}                   ---> 0
```

Frequency count.

a → 1
b → 1
temp → 1

$s(p) = 3+0 = 3$
$s(p) = O(1)$

$f(n) = 0+0+1+1+1+0 = 3 = O(1)$

Note : Every simple statement will take one unit of time.

→ we have different types in time complexity calculation.

* Best case.
* worst case.
* Average case.

Best case: It is the minimum no of steps that can be executed for a given parameter.

worst case: It is the maximum no of steps that can be executed for a given parameter.

Average case: It is the ~~Average~~ no of steps that can be executed for a given parameter.

Measuring an input size:

→ It is observed that if the input size is longer, then usually algorithm runs for a longer time. Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter.

Ex: while performing multiplication of two matrices we should know order of these matrices. then only we can enter the elements of matrices.

Measuring Running time:

→ we have already discussed that the time complexity is measured in terms of a unit called frequency count. The time which is measured for analyzing an algorithm is generally running time.

→ From an algorithm:

we first identify the important operation (core logic) of an algorithm. this operation is called the basic operation.

It is not difficult to identify basis operation from

an algorithm. Generally the operation which is more time consuming is a basic operation in the algorithm. normally such basic operation is located in inner loop.

problem statement	Input size	Basic operation.
Searching a key element from the list of n elements.	List of n elements.	Comparison of key with every element of list.
performing matrix multiplication.	The two matrix with order nxn	Actual multiplication of the elements in the matrices.
computing GCD of two numbers.	Two numbers	Division.

Basic operations from input.

→ Then we compute total no of time taken by this basic operation. we can compute the running time of basic operation by following formula.

$$T(n) = C_{op} C(n).$$

- $T(n)$: Running time of basic operation
- C_{op}: Time taken by the basic operation to execute.
- $C(n)$: No of times the operation needs to be executed.

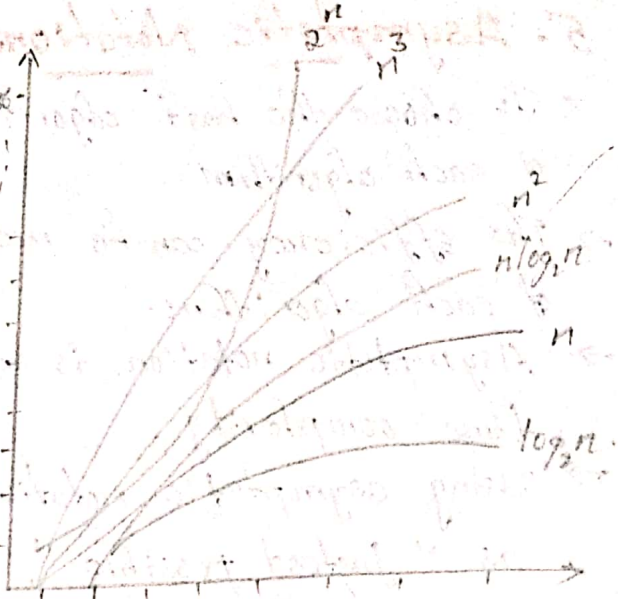
Order of Growth :

→ Measuring the performance of an algorithm in relation with the input size n is called order of growth.

→ For example the order of growth for varying input size of n is as given below.

order of Growth.

n	log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296



Rate of growth of common computing time function.

→ From the above drawn graph it is clear that the logarithmic function is the slowest growing function.
→ The exponential function 2^n is fastest and grows rapidly with varying input size n.

4. performance Measurement:

* performance analysis estimates space and time complexity in advance, "while performance measurement measures the space and time taken in actual runs."
* In this while the algorithm is executed we measure the execution time. it gives accurate values and it is very costly. using Asymptotic Notations we can measure it.
* we can also call this one as posterior Analysis.

Ex: Sequential search.

```
Algorithm SeqSearch (a, x, n)
{ // search for x in a[1:n],
      a[0] is used additional space.
    i = n;
    a[0] = x;
    while (a[i] ≠ x) do i = i - 1;
    return n;
}
```

Explanation:

5, 4, 6, 8
x = 4

i = 4
a[0] = 4
a[4] = x
8 ≠ 4
i = 4 - 1 = 3
a[3] = x
6 ≠ 4

i = 3 - 1 = 2.
a[2] = x
4 = 4.

5- 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 standard way to represent the time complexity.

→ Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "Average time".

* Big-oh Notation. (O) (upper bound)
* Omega Notation. (Ω) (lower bound)
* Theta Notation. (Θ) (Average bound).

Big-Oh notation :

→ Big-oh notation is indicated by "O"

→ Big-oh notation is used to represent worst case / Best case / Avg case.

→ Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Def: The function $f(n) = O(g(n))$ iff there exists positive constants c and n_0 such that

$$f(n) \leq c * g(n), \forall n, n \geq n_0.$$

Here $F(n)$ and $g(n)$ be two non-negative functions.

n_0 and c two integers.

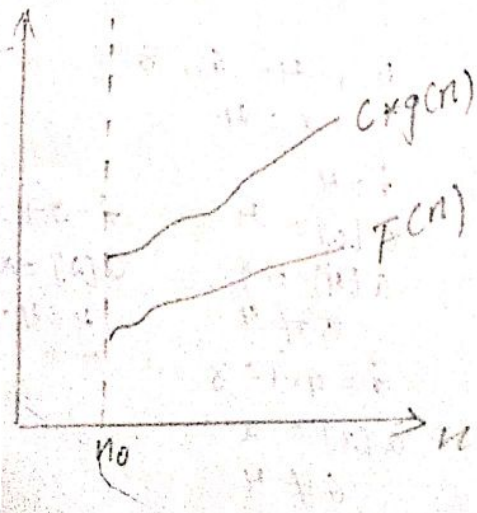


Fig : $f(n) \in O(g(n))$

Note: procedure of precedence

$$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n \cdots < n^n$$

Omega notation :

→ Omega notation is indicated by "Ω"

→ Omega notation is used to represent best case / worst case / Avg case.

→ Using Omega notation we can give shortest amount of time taken by the algorithm to complete.

Def : The function $f(n) = \Omega(g(n))$ iff there exists positives constants c and n_0 such that

$$f(n) \geq c * g(n), \forall n, n \geq n_0$$

Here $F(n)$ and $g(n)$ be two non-negative functions. n_0, c two integers.

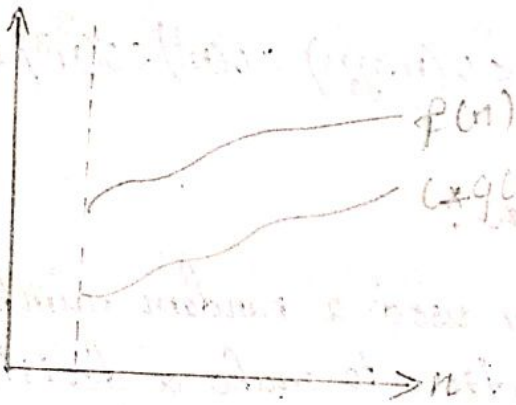


Fig: $f(n) \in \Omega(g(n))$.

[Θ] Theta Notation :

→ The Theta notation is indicated by "Θ"

→ Theta notation is used to represent Average / Best / worst cases.

→ Using Theta notation we can denote Average amount of time taken by algorithm.

Def : The function $f(n) = \Theta(g(n))$ iff there exists positive constants c_1, c_2 and n_0 such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n), \forall n, n \geq n_0$$

Here $f(n)$ and $g(n)$ be two non-negative functions. n_0, c_1 & c_2 are integers.

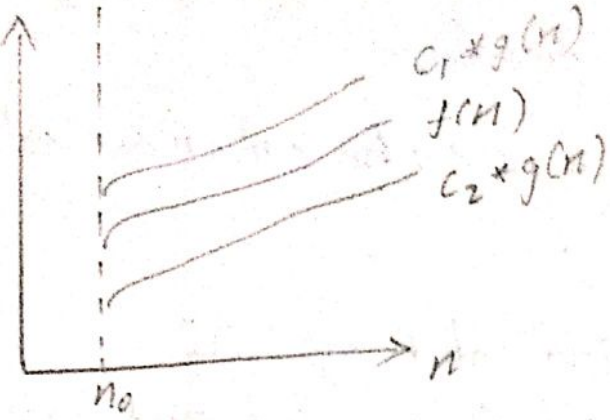


Fig: $f(n) \in \Theta(g(n))$.

→ The no of times a statement is executed is usually referred as frequency count.

→ we can take the frequency count of each operation to calculate the time complexity.

→ Among all frequency counts we will take the maximum frequency count of any operation which will be the order of algorithm.

$$O(1) < O(\log_2^n) < O(n) < O(n\log_2^n) < O(n^2) < O(n^3) < O(2^n)$$

6. Randomized Algorithms:

Def: A randomized algorithm used a random number generator during the compution to make a decision at least once.

→ For a randomized algorithm, execution time may vary from run to run.

→ This algorithms actully used in case of solving Np-hard algorithms.

→ Np-hard problem algorithms are nothing but which are having non-deterministic polynomial time or can't able to run the algorithm with in the polynomial time and having more time complexity algorithms.

→ so to reduce the computing resources and time complexity we introduced this randomness concept.

→ There are two types of randomized algorithms.

1. Las Vegas algorithms.
2. Monte Carlo algorithms.

Las Vegas algorithms: These are the type of randomized algorithms which produce same outcome on each execution for same input.

Monte Carlo algorithms: These are the type of randomized algorithms which produce different outcomes on each execution for same input.

Ex: Randomized Quick sort.

→ In Randomized Quick sort, we use a random number to pick the pivot element, or we randomly shuffle the array.

Advantages: These are two major advantages of randomized algorithms.

1. These algorithms are simple to implement.
2. These algorithms are many times efficient than traditional algorithms.

Dis advantages: However randomized algorithms may have some drawbacks.

1. The small degree of error may be dangerous for some applications.
2. It is not always possible to obtain better results using randomized algorithm.

→ Now we will see some example algorithms.

1. Las Vegas algorithm :

```
Algorithm LasVegas ( )
{
    while (true) do
    {
        i := Random ( ) mod 2;
        if (i = 1) then return;
    }
}
```

2. Identifying the repeated array number.

```
Algorithm RepeatedElement (a, n)
{ // finds the repeated element from a[1:n].
  while (true) do                    choosing some random number
  {
     i := Random ( ) mod n+1 ;
     j := Random ( ) mod n+1 ;
        // i and j are random numbers in the
                                  range [1, n].
     if ((i ≠ j) and (a[i] = a[j])) then
           return i;
  }
}
```

Different positional elements with same value

This is a simple and efficient las vegas algorithm then it may take $O(\log n)$ time. The notation used to denote running time of las vegas is $\tilde{O}( ) - \tilde{O}(\log n)$.

chapter - 2.

Sets & Disjoint set union

1. Introduction :

→ A disjoint set is a kind of data structure that contains partitioned sets.

These partitioned sets are separate non overlapping sets.

Eg : If there are $n = 10$ elements that can be partitioned into three disjoint sets S_1, S_2 and S_3 such that no element is common among these sets.

Let $S_1 = \{5, 4, 7, 9\}$, $S_2 = \{10, 12, 14\}$, $S_3 = \{2, 3, 6\}$, then each set can be represented as a tree.

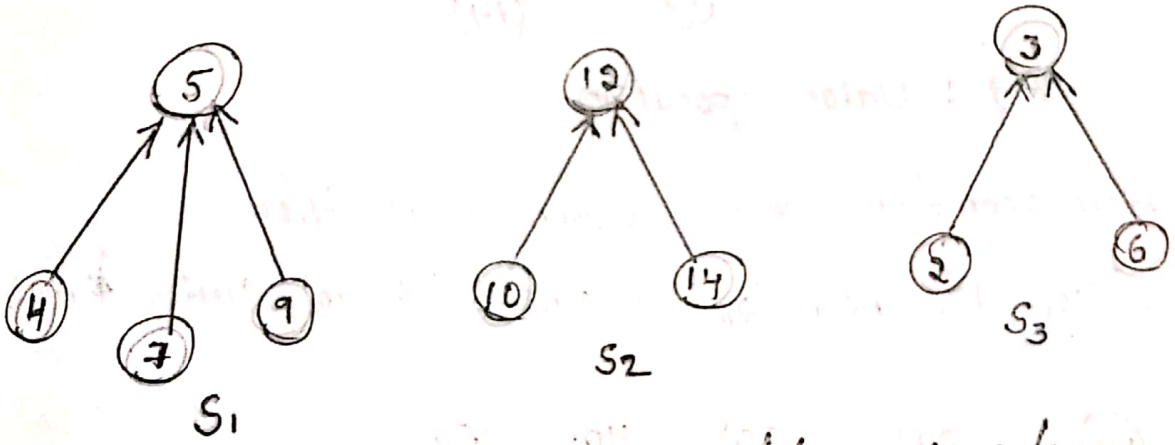


Fig : Tree representation of sets.

→ There are two operations that can be performed on the data structure : disjoint set.

* union and find.

Disjoint set union : If there exists two sets S_i and S_j then $S_i \cup S_j$ = { all the elements from set S_i and S_j }.

In above example $S_1 \cup S_2 = \{5, 4, 7, 9, 10, 12, 14\}$.

Find (i) : For finding out the element i from given set, is done by this operation.

for example - Element 7 is in S_1, element 14 is is S_2.

2. Union and Find Oporations :

Union operation : The union operation combines the elements from two sets.

→ consider $S_1 = \{5, 4, 7, 9\}$ and $S_2 = \{10, 12, 14\}$ then

$S_1 \cup S_2$ is

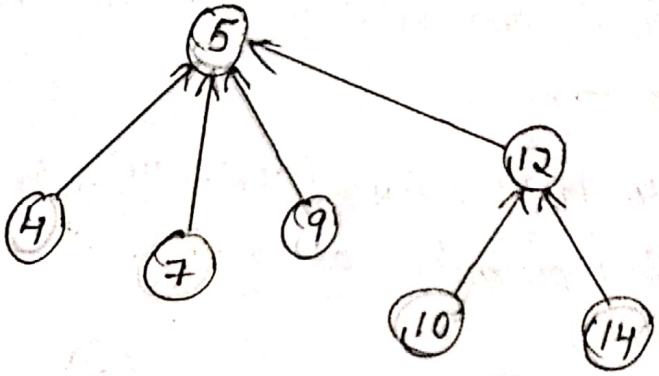


Fig : Union operation.

→ The union operation can be performed as follows -

Union (10,20), union (20,30), union (30,40), Union (40,50).

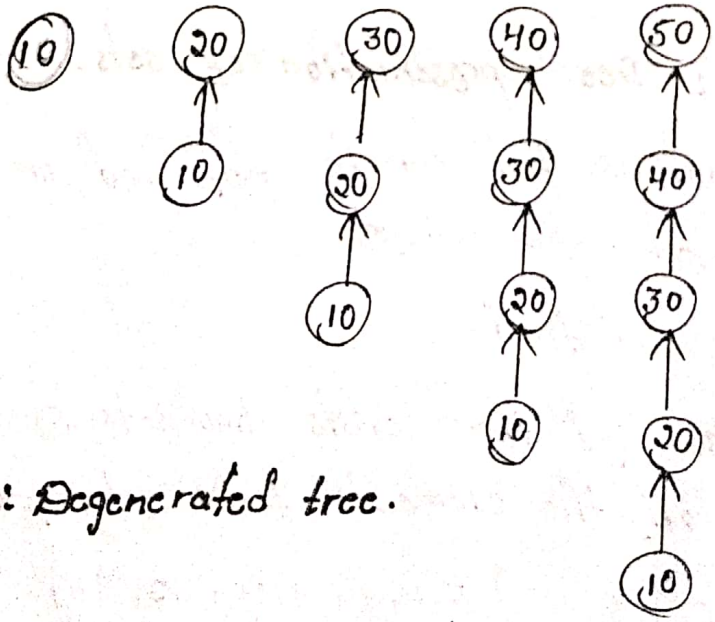


Fig : Degenerated tree.

→ This sequence of union operations generate degenerated tree.

Algorithm for union :

```
Algorithm union (x1 , x2)
{
    A[x1] = x2 ;
}
```

Analysis : The time required by (n-1) unions is O(n).

Find operation :

→ To perform union or find operations efficiently on sets, it is necessary to represent the set elements in a proper manner.

Data representation of sets - The data representation of sets is illustrated by following example -

Consider $S_1 = \{5, 4, 7, 9\}$, $S_2 = \{10, 12, 14\}$ and $S_3 = \{2, 3, 6\}$.

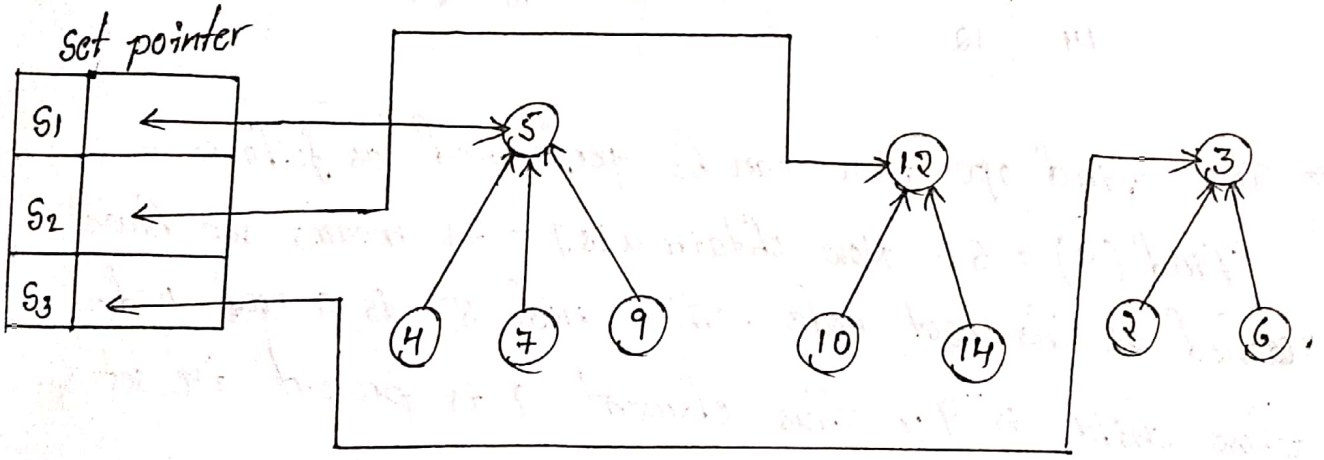


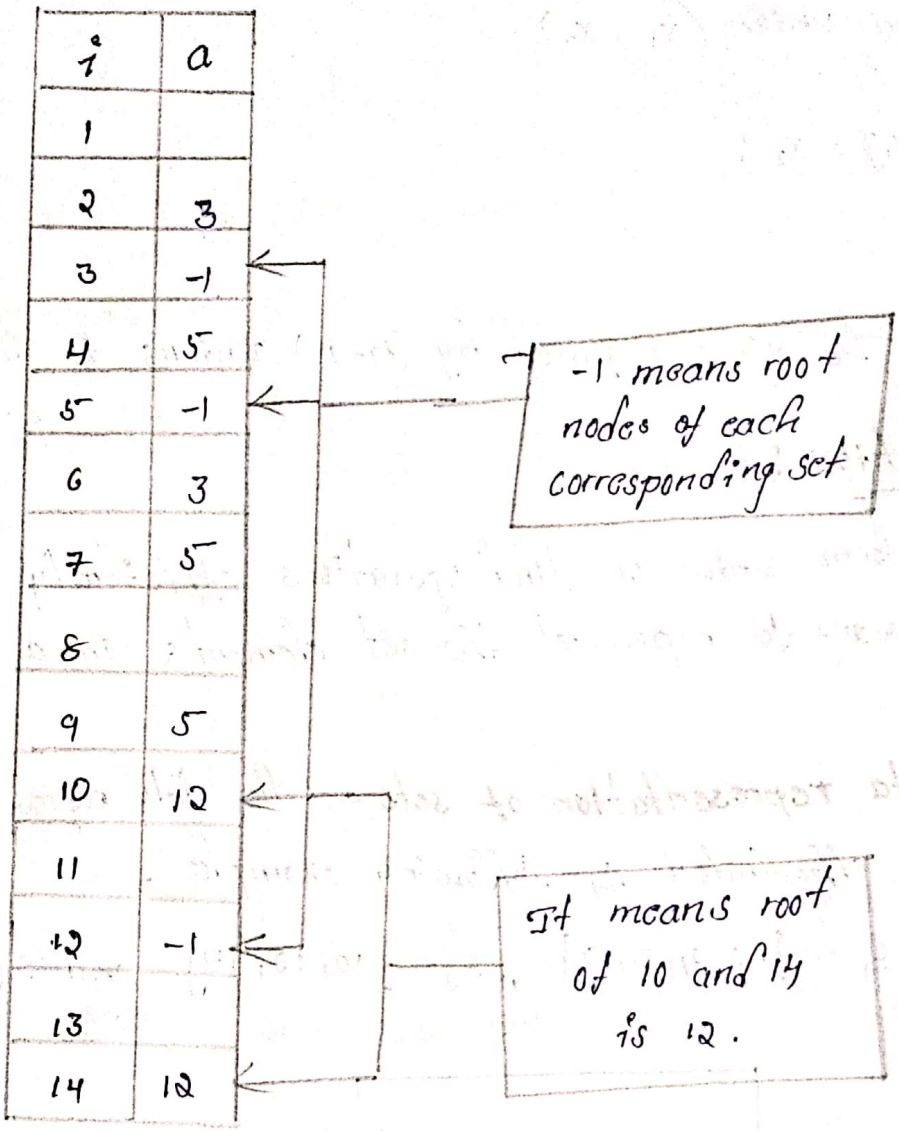
Fig : Data representation.

→ For finding out any desired element,

i) Find the pointer of the root node of corresponding set.

ii) Then find the desired element from that set.

→ The sets can be represented using arrays as follows -



→ The find operation can be performed as follows -

Find (9) = 5. Now obtain a[5] = -1 means we have reached at the root node. Then node 5 is a root node whose child is 9., thus element 9 is present in set S_1.

Algorithm for Find :

```
Algorithm Find (x)
{
    while (a[x] ≥ 0) do
        x = a[x];
    return x;
}
```

Analysis : The time taken to process a find for an element at

level i of tree is $O(i)$.

→ Hence total time required by find operation is $O(n^2)$.

chapter - 3.

Basic Traversal & Search Techniques

1. Techniques for Graphs :

Graphs :

Def : A graph G consists of two sets V and E. The set V is a finite, non empty set of vertices. The set E is a set of pairs of vertices ; these pairs are called edges.

→ The notations V(G) and E(G) represent the sets of vertices and edges, respectively, of graph G.

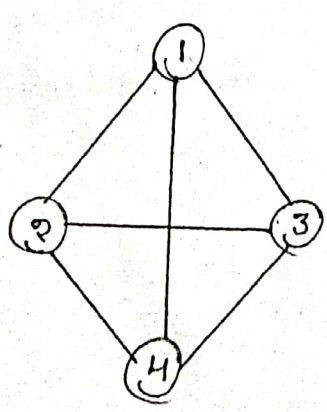
We also represent a graph like - G = (V, E).

Undirected Graph : In an undirected graph the pair of vertices representing any edge is unordered.

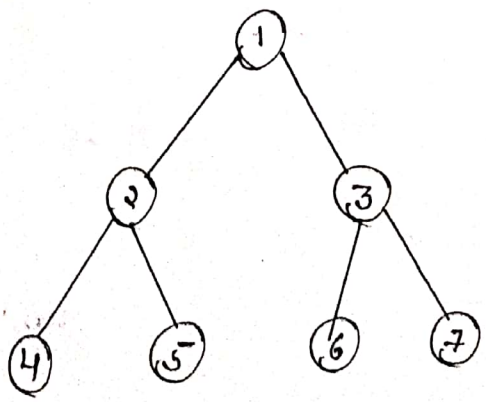
Thus, the pair (u, v) and (v, u) represent the same edge.

Directed Graph : In a directed graph each edge is represented by a directed pair (u, v) ; u is the tail and v the head of the edge.

Therefore, (u, v) and (v, u) represent two different edges.



G_1



G_2

G_3.

Three sample Graphs.

→ The Graphs G_1 and G_2 are undirected; G_3 is directed.

→ The set representations of three graphs are

$V(G_1) = \{1, 2, 3, 4\}$ $\quad E(G_1) = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$

$V(G_2) = \{1, 2, 3, 4, 5, 6, 7\}$ $\quad E(G_2) = \{(1,2), (1,3), (2,4), (2,5), (3,6), (3,7)\}$.

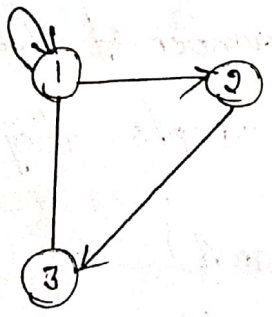
$V(G_3) = \{1, 2, 3\}$ $\quad E(G_3) = \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle\}$.

→ The edges of a directed graph are drawn with an arrow from the tail to the head.

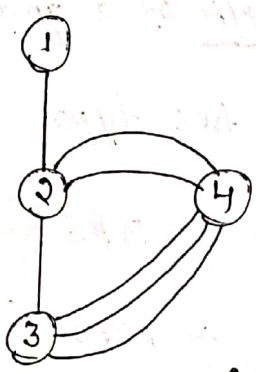
→ The graphs G_1 and G_3 are not trees, but graph G_2 is a tree.

→ A graph may not have an edge from a vertex v back to itself. That is, edges of the form (v,v) and $\langle v,v\rangle$ are not legal. Such edges are known as self-edges (or) self loops. We obtain a data object referred to as a graph with self-edges.

→ A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a multi-graph.



Graph with a self edge.



multi graph.

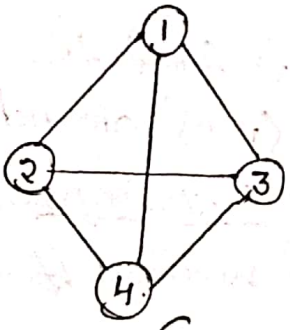
→ $\frac{n(n-1)}{2}$ is the maximum no of edges in any n-vertex, undirected Graph.

→ An n-vertex, undirected graph with exactly $\frac{n(n-1)}{2}$ edges is said to be complete graph.

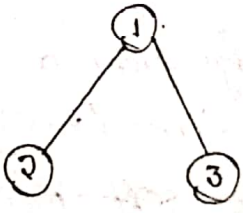
→ From before examples Graph G_1 is the complete Graph, where as G_2 and G_3 are not complete graphs.

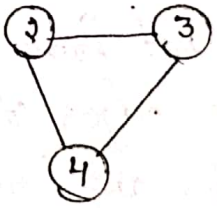
→ A directed graph on n vertices, the maximum number of edges is $n(n-1)$.

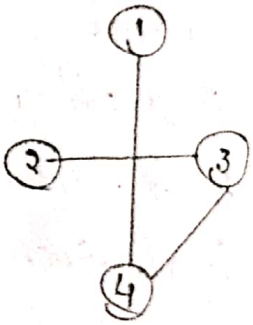
→ A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



G_1







Some of the subgraphs of G_1.

→ A path from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \ldots i_k, v$ such that (u, i_1), $(i_1, i_2), \ldots, (i_k, v)$ are edges in $E(G)$.

→ The Length of a graph path is the number of edges on it.

→ we have two types of techniques for Graphs. They are:

- BFS (Breadth First Search)
- DFS (Depth First search).

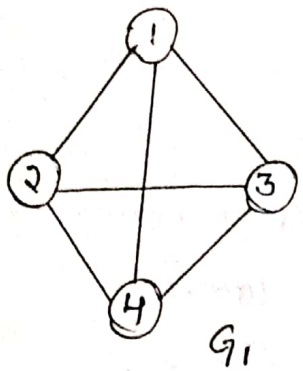
→ we have two types of Graph representations.

- Adjacency Matrix.
- Adjacency Lists.

Graph Representations :

Adjacency Matrix :

→ Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a, with the property that $a[i,j] = 1$ iff the edge (i,j) is in $E(G)$. The element $a[i,j] = 0$ if there is no such edge in G.



G_1

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Adjacency matrix.

$O(n \times n)$

$O(n^2)$

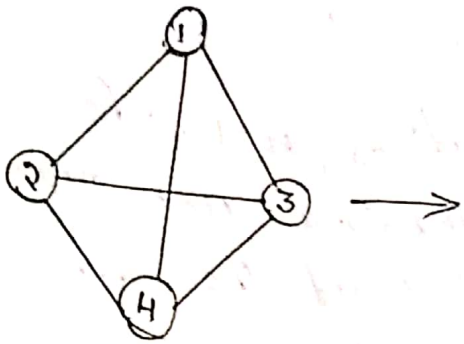
→ From the adjacency matrix, we can readily determine whether there is an edge connecting any two vertices i and j. For an undirected graph the degree of any vertex is its row sum: $\sum_{j=1}^{n} a[i,j]$.

For a directed graph the row sum is the out degree, and the column sum is the in-degree.

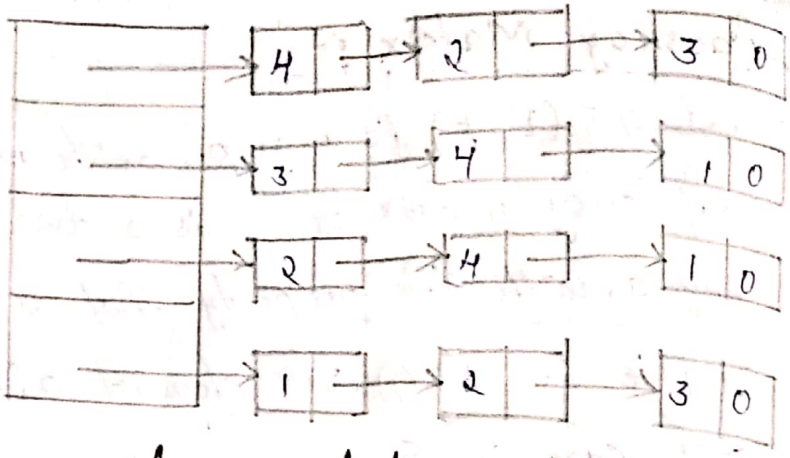
Adjacency Lists :

→ In this representation of graphs, the n-rows of the adjacency matrix are represented as n linked lists.

→ There is one list for each vertex in G. The nodes in list i represent the vertices that are adjacent from vertex i. each node has at least two fields: vertex and link.



G1



$O(n+2E)$ Adjacency lists.

Techniques for Graphs :

BFS :

→ BFS stands for Breadth first search is a vertex based technique for finding a shortest path in graph.

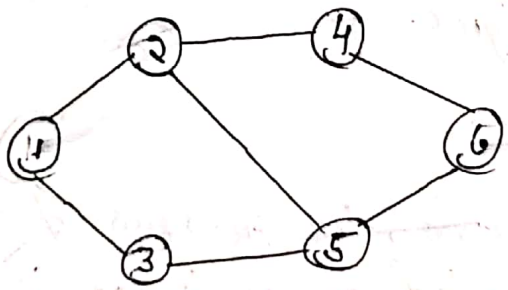
→ we can call BFS is a level based technique.

→ It uses a Queue data structure which follows first in first out.

→ In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the Queue.

→ It is slower than DFS.

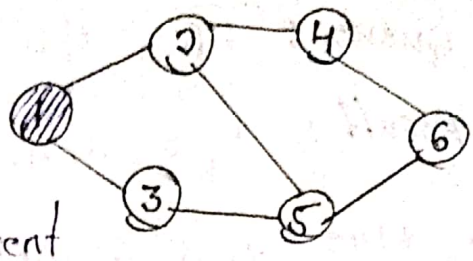
Ex :



Here first we need to set or select one vertex as a root - any one vertex out of all.

Here selecting ① as a root then we need to explore ①.

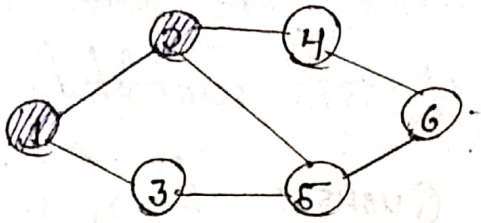
Queue : | 1 | | | | | |



Result : 1

→ so for ① vertex ② & ③ are adjacent vertices we can visit and explore in any order.

Here I am taking and sending those vertices into Queue as 2, 3.

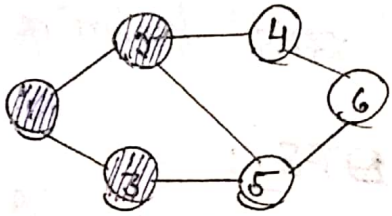


Queue : | 1 | 2 | 3 | 4 | 5 | |

Result : 1, 2

→ Here we need explore ② vertex in Queue order.

For ② vertex 1, 4, 5 are adjacent vertices. so ① already visited, so we have to push 4, 5 into Queue in any order. next we need to explore ③ vertex.

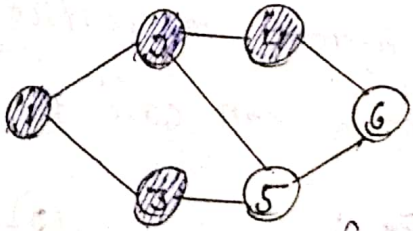


Queue : | 1 | 2 | 3 | 4 | 5 | |

Result : 1, 2, 3

→ Here 1, 5 are adjacent vertices of ③, and they are already visited vertices so we no need to push any thing (vertex).

now we need to select ④ vertex in Queue order.



Queue : | 1 | 2 | 3 | 4 | 5 | 6 |

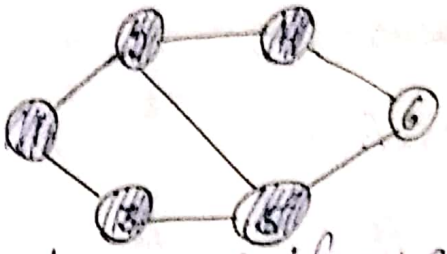
Result : 1 2 3 4

→ Here 2, 6 are adjacent nodes for ④, but ② already visited. so we just push ⑥ into the Queue.

now we need to explore ⑤ vertex.

Queue : | 1 | 2 | 3 | 4 | 5 | 6 |

Result : 1, 2, 3, 4, 5

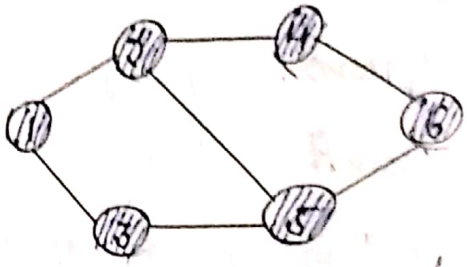


→ Here 2, 3, 6 are adjacent vertices and they are already visited vertices.

Now select (6) vertex and explore. But for (6) vertex there were no unvisited adjacent vertices then proceduce of BFS completed.

Queue : | 1 | 2 | 3 | 4 | 5 | 6 |

Result : | 1, 2, 3, 4, 5, 6 |



All nodes are visited.

∴ Hence this is the over all procedure for BFS technique.

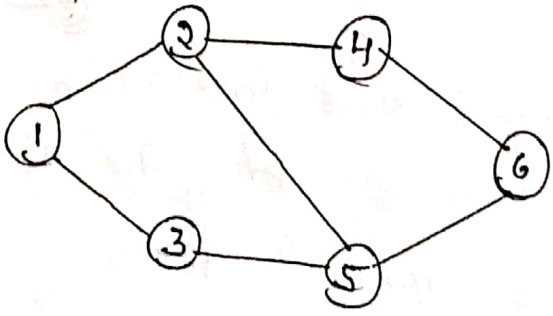
DFS :

→ DFS stands for Depth first search is a edge based technique.

→ It uses the stack data structure, performs two stages, first visited vertices are pushed into stack and second if there is no vertices then visited vertices are popped.

→ we can call DFS as a pre order technique.

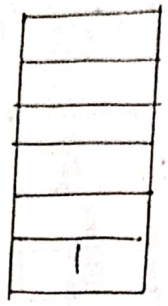
Ex :

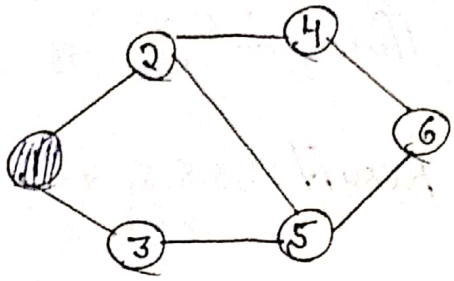


→ Here we need to select any vertex as a root vertex and have to push that into stack.

Result : 1

stack:



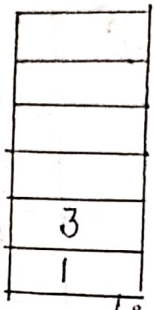


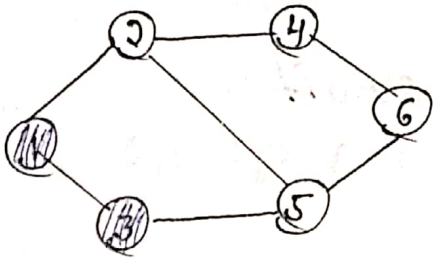
→ Now any one of adjacent nodes (or) vertices of (1) we can take.

Here (2) & (3) are adjacent vertices and here selecting (3) vertex, push that into stack.

Result : 1,3

stack:

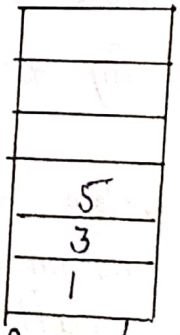


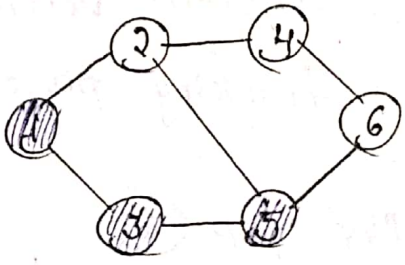


→ Now any one of adjacent vertices of (3) we can take. Here (5) is adjacent one then push that into stack.

Result : 1,3,5

stack:

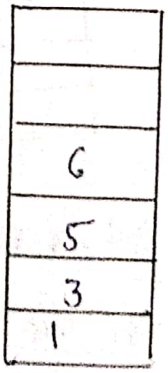


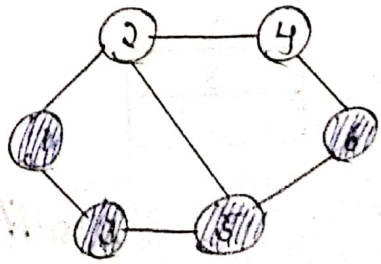


→ Now explore (5) and adjacent nodes (or) vertices are (2), (6), any one of these pick and push into stack.

Result : 1,3,5,6

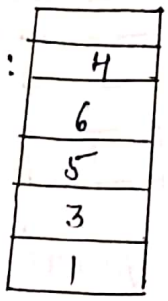
stack:

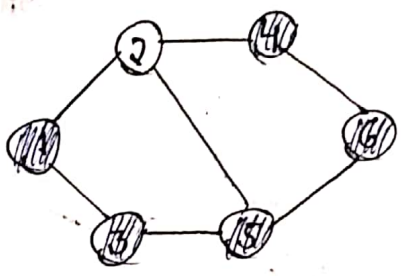




→ Now explore (6) vertex, any one of the un visited vertex we can push on to the stack. Here (4), (5) there but (5) is visited one so push (4) into stack.

Result: 1, 3, 5, 6, 4 stack:

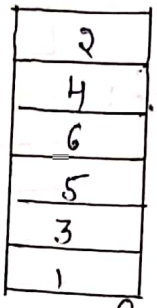


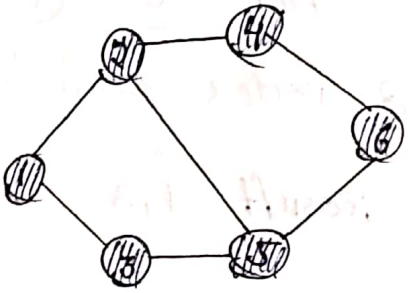


→ Now explore (4) and push adjacent vertex (2) into stack.

Result:
1, 3, 5, 6, 4, 2

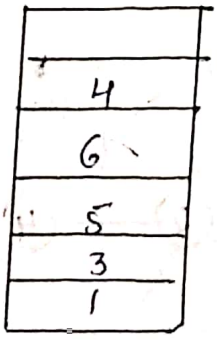
stack:



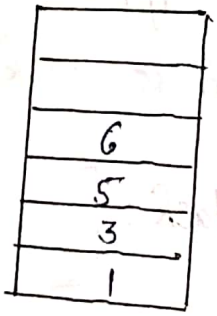


→ Now explore (2) but there were no un visited adjacent vertices now we have to start the back tracking procedure by popping up nodes one by one.

→ first pop (2) no unvisited vertices.



→ Now pop (4), no un visited vertices.



→ Continue this process upto the (1) node if any un visited vertices there include those to result and push onto stack, or else our procedure completed.

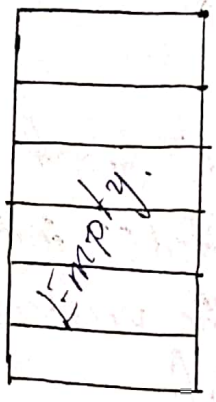
Result : 1, 3, 5, 6, 4, 2

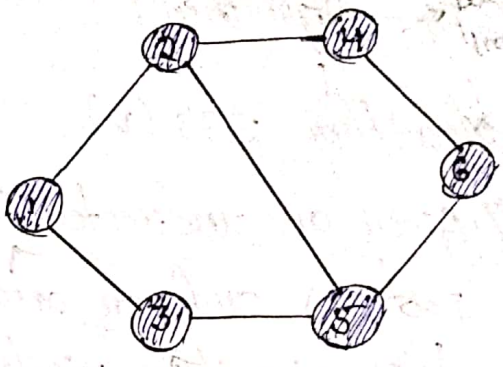
∴ Hence this is the over all procedure for DFS technique.

Result :

1, 3, 5, 6, 4, 2.

stack :





BFS - Algorithm :

```
Algorithm BFS(v)
// A breadth first search of G is carried out beginning at
   vertex v. For any node i, visited[i] = 1 if i has already
   been visited. The graph G and array visited[] are
   global; visited[] is initialized to zero.
{
   u := v ; // q is queue of unexplored vertices.
   visited[v] := 1;
   repeat
   {  for all vertices w adjacent from u do
      {  if (visited[w] = 0) then
         {  Add w to q ; // w is unexplored.
            visited[w] := 1;
         }
      } if q is empty then return ;
        // nominal unexplored vertex.
        Delete the next element, u from q ;
        // Get first unexplored vertex.
   } until (false);
}
```

DFS - Algorithm:

```
Algorithm DFS(v)
// Given an undirected (directed) Graph G = (V, E) with n
   vertices and an array visited[] initially set to zero,
   this algorithm visits all the vertices reachable from v.
   G and visited[] are global.
{  visited[v] := 1;
   for each vertex w adjacent from v do
   {  if (visited[w] = 0) then DFS(w);
   }
}
```

2. Connected components and spanning trees:

Spanning trees:

Def: A spanning of a graph G is a subgraph which is basically a tree and it contains all the vertices of G containing no circuit.

(or)

A spanning tree is a subset of an undirected graph that has all the vertices connected by minimum number of edges.

→ If all the vertices are connected in a graph, then there exists atleast one spanning tree.

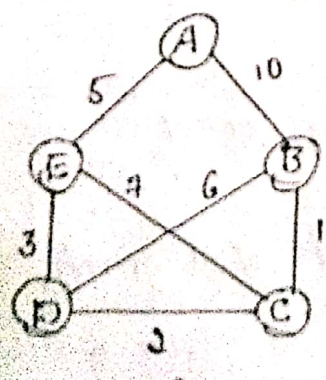
→ In a graph, there may exist more than one spanning tree.

Weight of the tree: A weight of the tree is defined as the sum of weights of all its edges.

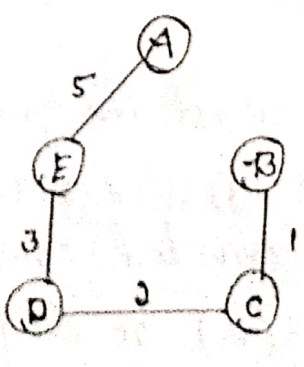
Minimum spanning Tree: A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight.

Ex: consider a graph G as given below. This graph is called weighted connected graph because some weights are given along every edge and the graph is a connected graph.

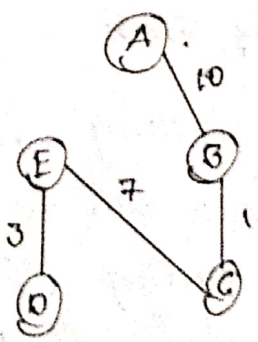
Graph and two spanning trees out of which (c) is a minimum spanning Tree.



Graph G



wt(T_1) = 11



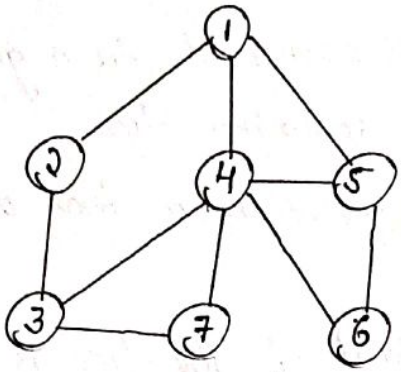
wt(T_2) = 21.

Connected components :

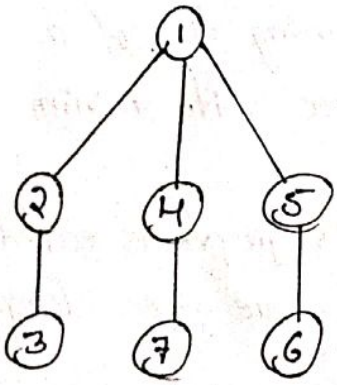
→ A graph is said to be connected if there exists a path between any two vertices.

→ If the graph G is connected undirected graph, then we can visit all the vertices of the graph in first call to Breadth First Search (BFS) or Depth First Search (DFS). The subgraph which we obtain after traversing the graph using BFS or DFS represents the connected component of the graph.

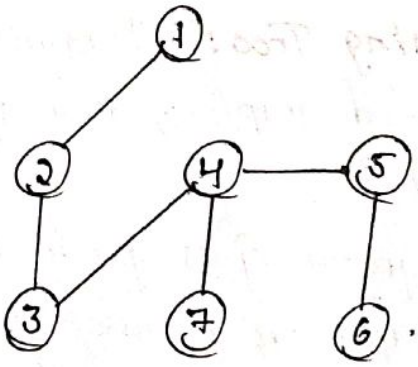
For Ex : Consider the graph G whose connected component can be given as below -



Graph G.



connected components by BFS.



connected component by DFS.

→ For checking whether or not the graph is connected we make a call to either BFS or DFS.

→ For finding all the connected components of a graph, a repeated call to DFS(v_1) or BFS(v_1) is given with vertex v_1 which is not yet visited.

Algorithm for finding connected components:

```
Algorithm components (G, n)
{
  for (i ← 1 to n)
  {
     visited[i] ← 0;
  }
  for ( i ← 1 to n)
  {
    if (visited[i] == 0)
        DFS (i)
     output the newly visited vertices
     with adjacent edges.
  }
}
```

Time complexity: The time complexity of above algorithm is $O(n+E)$. The total time taken by DFS is $O(E)$. And the for loop in which DFS routine is called takes $O(n)$ time. Hence all the connected components get generated in $O(n+E)$ time.

Def: The connected components of a graph is a collection of vertices such that there is a path between every pair of vertices in the same component.

That means there exists a path between v_i and v_j if v_i and v_j are in the same component and there is no path between v_i and v_k if v_i and v_k are in two different components.

For obtaining path between any two vertices, the transitive closure is obtained.

3. Bi-connected components :- DFS :

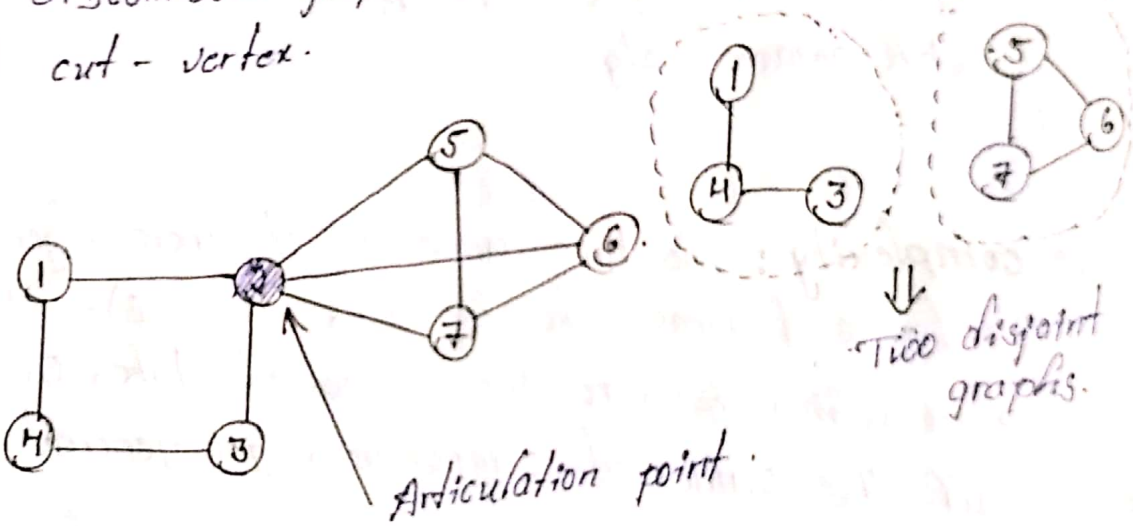
→ In this section we will understand two important concepts those are articulation point and bi-connected components.

→ we will also learn how depth first search helps in finding an articulation point and bi-connected components

Articulation point :

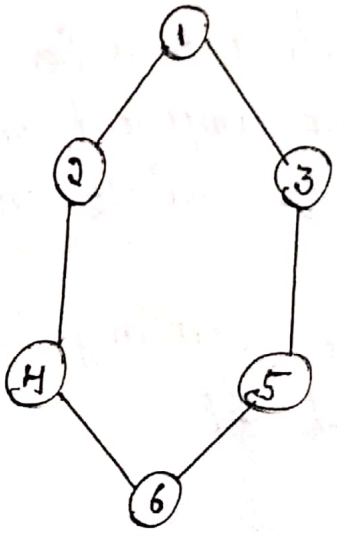
Def: Let $G = (V, E_0)$ be a connected undirected graph, then an articulation point of graph G is a vertex whose removal disconnects graph G. This articulation point is a kind of cut-vertex.

Eg:



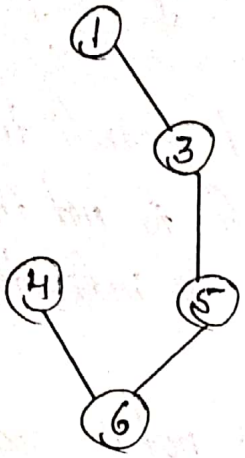
→ A graph G is said to be bi-connected if it contains no articulation point.

Eg:



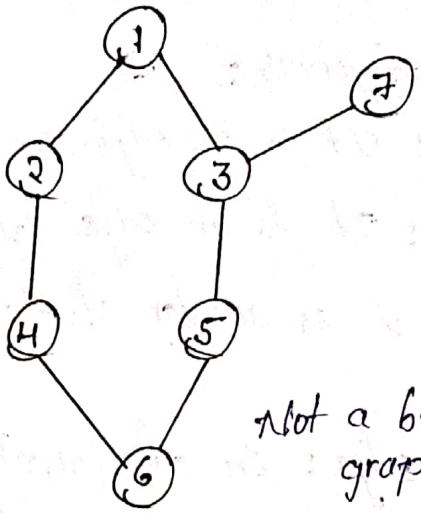
eventhough we remove any single vertex we do not get disjoint graphs.

Let us remove vertex 2 and we will get



Bi - Connectivity

but



Not a bi-connective graphs.

→ If there exists any articulation point in the given graph then it is an undesirable feature of that graph.

→ For instance in communication network the nodes or vertices represent the communication station x.

→ If this communication station x is an articulation point then failure of this station makes the entire communication system down! And this is surely not desirable feature.

Identification of Articulation point:

→ The easiest method is to remove a vertex and its corresponding edges one by one from graph G and test whether the resulting graph is still disconnected or not. The time complexity of this activity will be $O(V(V+E))$.

→ Another method is to use depth first search in order to find the articulation point. After performing depth first search on the given graph we get a 'DFS tree'.

→ While building the DFS tree we number each vertex. These numbers indicate the order in which a depth first search visits the vertices. These numbers are called as depth first search numbers (dfn) of the corresponding vertex.

→ While building the DFS tree we can classify the graph into four categories:

i) Tree edge: It is an edge in depth first search tree.

ii) Back edge: It is an edge (u,v) which is not in DFS tree and v is ancestor of u. It basically indicates a loop.

iii) Forward edge: An edge (u,v) which is not in search tree and u is an ancestor of v.

iv) Cross edge: An edge (u,v) not in search tree and v is neither ancestor nor a descendant of u.

→ To identify articulation points following observations can be made.

i) The root of the DFS tree is an articulation if it has two or more children.

ii) A leaf node of DFS tree is not an articulation point.

iii) If u is any internal node then it is not an articulation point if and only if from every child w of u it is possible to each an ancestor of u using only a path made up of descendents of w and back edge.

This observation leads to a simple rule as,

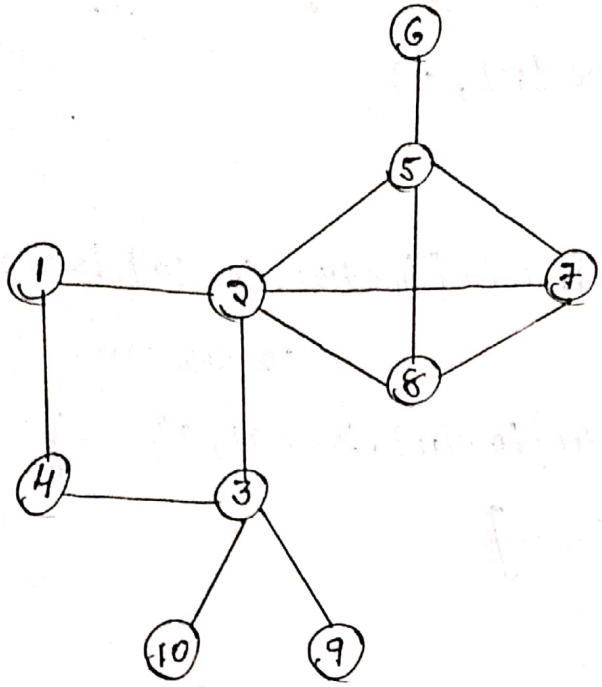
$$low[u] = \min\{dfn[u], \min\{low[w] / w \text{ is child of } u\}, \min\{dfn[w] / (u,w) \text{ is a cross edge}\}\}.$$

where $low[w]$ is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge.

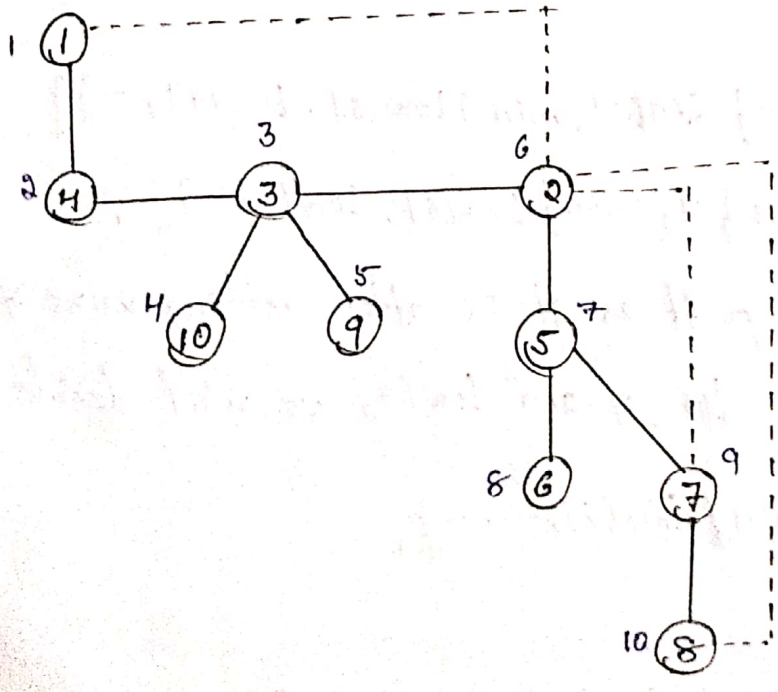
The vertex u is an articulation point if u is child of w such that.

$$L[w] \geq dfn[u].$$

Eg: obtain the articulation point for following graph.



The DFS tree can be drawn as follows.



Let us compute Low[u] using the formula.

$$Low[u] = min\{ dfn[u], min\{low[w] / w \text{ is a child of } u\}, min\{dfn[w] / (u,w) \text{ is backedge}\}\}$$

$Low[1] = min\{ dfn[1], min\{ low[4] \}, dfn[2] \}$.

$= min\{1, Low[4], 6\}$.

$\therefore$ $Low[1] = 1$ $\because$ nothing is less than 1.

$Low[2] = min\{ dfn[2], min\{ low[5] \}, dfn[1] \}$.

$= min\{6, Low[5], 1\}$

$\therefore$ $Low[2] = 1$

$Low[3] = min\{ dfn[3], min\{ low[10], low[9], low[2] \}, \}$.

no back edge

$= min\{3, min\{ low[10], low[9], 1 \}, - \}$.

$= min\{3, 1, - \}$

$\therefore$ $Low[3] = 1$

$Low[4] = min\{ dfn[4], min\{ low[3] \}, - \}$

$= min\{2, 1, - \}$.

$\therefore$ $Low[4] = 1$

$Low[5] = min\{ dfn[5], min\{ Low[6], Low[7], - \} \}$

$= min\{7, min\{ Low[6], Low[7] - \} \}$.

$\therefore$ $Low[5]$ = keep it as it is after getting value of $Low[6]$ and $low[7]$ we will decide $low[5]$.

$low[6] = min\{ dfn[6], -, - \}$.

$\therefore$ $low[6] = 8$.

$Low[7] = min\{ dfn[8], -, dfn[2] \}$.

$= min\{10, -16\}$.

$\therefore$ $Low[7] = 6$

As we have got low[6] = 8 and low[7] = 6. we will compute our incomplete computation Low[5].

$\therefore$ Low[5] = min{ 7, min[8, 6], - }.

= min{ 7, 6, - }.

$\therefore$ Low[5] = 6.

Now Low[8] = min{ dfn[8], -, dfn[2] }.

= min{ 10, 6 }.

$\therefore$ Low[8] = 6.

Low[9] = min{ dfn[9], -, - }.

$\therefore$ Low[9] = 5.

Low[10] = min{ dfn[10], -, - }.

= min{ 4, -, - }.

$\therefore$ Low[10] = 4.

Hence low values are low[1:10] = { 1, 1, 1, 1, 6, 8, 6, 6, 5, 4 }.

Here vertex (3) is articulation point because, child is 10

Low[10] = 4.

dfn[3] = 3.

Low[10] $\geq$ dfn[3]. $\because$ Low[w] $\geq$ dfn[u].

4 $\geq$ 3 (True)

similarly vertex (2) is an articulation point. because. child is 5

Low[5] = 6.

dfn[2] = 6.

Low[5] $\geq$ dfn[2].

6 $\geq$ 6. (True).

vertex (5) is articulation point because child of 5 is 6

Low[6] = 8

dfn[5] = 7.

low[6] ≥ dfn[5].

8 ≥ 7. (True).

Hence in above given graph vertex 2, 3, and 5 are articulation points.

Algorithm:

The algorithm for obtaing the articulation point is a given below.

```
Algorithm DFS_Art (u,v)
// the vertex u is a starting vertex for depth first traversal.
// In depth first tree v is a parent (ancestor) of u.
// Initially an array dfn[] is initialized to 0, the
   dfn[ ] stores the depth first search numbers.
// Array low[ ] is used to gives the lowest depth first
   number that can be reached from u.
{
  // put the dfn number for u in dfn array.
   dfn[u] := dfn_numb;
   low[u] := dfn_num;
   dfn_num := dfn_num + 1;
   for (each vertex w adjacent to u) do
   {
      // w is child of u and if w is not visited.
      if (dfn[w] = 0) then
      {
         DFS_Art (w,u); // finding the dfn of w
         low[u] := min(low[u], low[w]);
      }
```

```
else if (w != u) then // the edge u w can be back
                        edge then update Low[u].
    Low[u] := min(Low[u], dfn(w));
  }
}
```

Analysis: The DFS_Art has a complexity O(n+E) where E is number of edges in graph G and n is total no. of nodes. thus the articulation point can be determined in O(n+E) time.

Identification of Bi-Connected components:

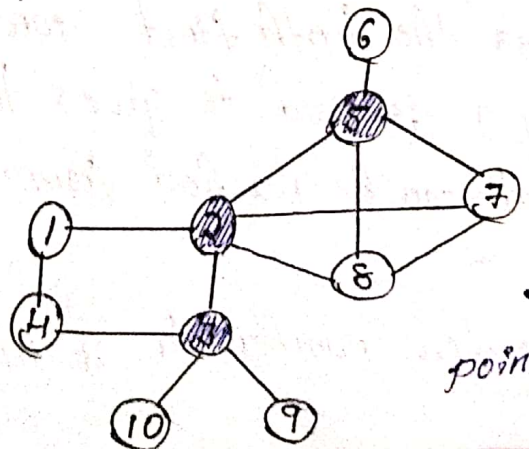
→ A bi-connected graph G = (V, E) is a connected graph which has no articulation points.

→ A bi-connected component of a Graph G is maximal bi-connected subgraph. That means it is not contained in any larger bi-connected subgraph of G.

→ Some key observations can be made in regard to bi-Connected components of graph.

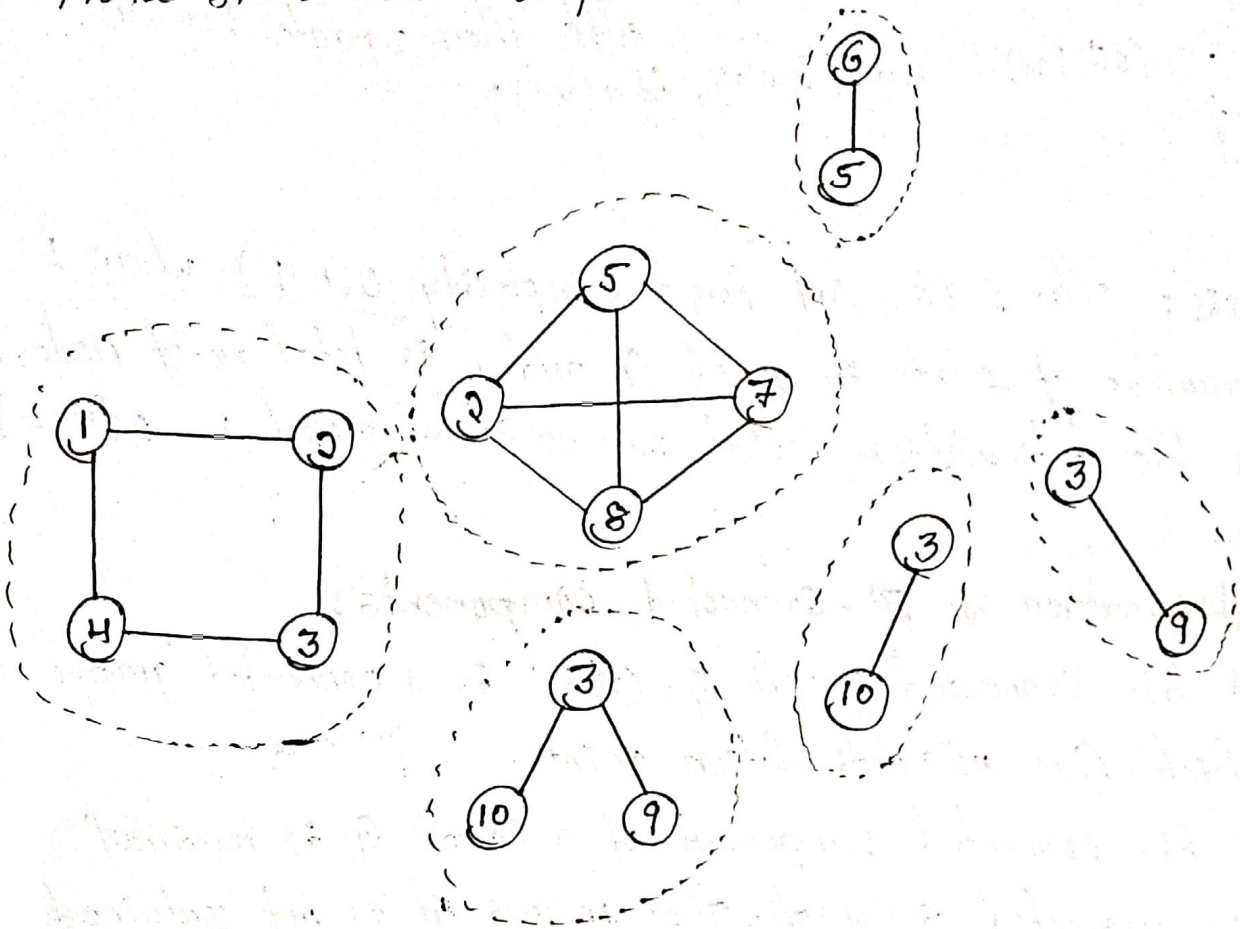
1) Two different bi-connected components should not have any common edges.
2) Two different bi-connected components can have common vertex.
3) The common vertex which is attaching two (or more) bi-connected components must be an articulation point of G.

Eg: In following graph.



The articulation points are: 2, 3, 5

Hence bi-connected components are :



Bi-connected components of G.

Algorithm:

The algorithm for obtaining bi-connected components is as given below.

Algorithm Bi-Connect (u,v)

// The vertex u is a starting vertex for depth first traversal.

// In depth first tree v is a parent (ancestor) of u.

// Initially an array dfn[] is initialized to 0. The dfn[] stores the depth first search numbers.

// Array Low[] is used to gives the lowest depth first number that can be reached from u.

{

// put the dfn number for u in dfn array.

```
dfn[u] ← dfn_num;
Low[u] ← dfn_num;
def_num ← dfn_num+1;
for(each vertex w adjacent to u) do
{ // w is child of u and if w is not visited.
  if ((v!=w) and (dfn[w]<dfn[u])) then.
      push(u,w) onto the stack st;
  if (dfn[w]=0) then
  { if (Low[w] >= dfn[u]) then
    { write ("Obtained Articulation point:");
      write ("The new Bi-connected component :");
      repeat
      { edge (x,y) ← pop edge from the top of the stack;
        write (x,y);
      } until ((x,y) = (u,w)) OR ((x,y) = (w,u)));
    }
  // if w is unvisited then.
  Bi-connect (w,u)
  // update Low[u]
  Low[u] ← min (Low[u], Low[w]);
}
else if (w!=u) then // the edge u w can be a back edge
                    then update Low[u].
Low[u] ← min(Low[u], dfn[w]);
}
```

For the algorithm the time complexity remains $O(n+E)$.