DESIGN AND ANALYSIS OF ALGORITHM-7MCE1C2

UNIT-1

1.1 INTRODUCTION

1.1.1 ALGORITHM DEFINITION

1.1.2 ALGORITHM SPECIFICATIONS

1.1.3 PERFORMANCE ANALYSIS

1.2. ELEMENTARY DATA STRUCTURES

1.2.1. STACK

1.2.2. QUEUE

1.2.3. TREES

1.2.4. DICTIONARIES

1.2.5. PRIORITY QUEUES

1.2.6. SETS AND DISJOINT SETS

1.2.7. GRAPHS

1.1. INTRODUCTION

1.1.1 ALGORITHM DEFINITION

What is an Algorithm?

An *algorithm* is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

1. input: there are zero or more quantities which are externally supplied;
2. output: at least one quantity is produced;
3. definiteness: each instruction must be clear and unambiguous;
4. finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
5. effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

1.1.2. ALGORITHM SPECIFICATIONS

Recursive Algorithms

- Recursion is similar to the method of induction which is often used to prove mathematical statements.
- In mathematical induction, a statement about integers (e.g., the sum of the first n positive integers is n (n +1)12) is proved by showing that the statement can be proved for integer k if it is assumed to be true for integer k -1.

To understand a recursive function, you must.

(1) Formulate in your mind a statement of what it is that the function is supposed to do, for a given input.

(2) Verify that the function does achieve its goal if the recursive invocations to itself do what they are supposed to.

(3) Ensure that a finite number of recursive invocations of the function eventually lead to an invocation which satisfies the terminating condition (otherwise, the function will keep calling itself and not terminate!).·

(4) The function should perform the correct computations if the terminating condition is encountered. .

Example 1.5 [Permutation generator]:·

- Given a set of n > 1 elements, the problem is to print all possible permutations of this set.
- For example if the set is {a, b, c}, then the set of permutations is {(a, b, c), (a, c, b),(b, a, c),(b, c, a),(c, a, b),(c, b, a)}. It is easy to see that given n elements, there are n ! different permutations.

- A simple algorithm can be obtained by looking at the the case of four elements (a,b,c,d). The answer can be constructed by writing.

  - a followed by all permutations of (b,c,d)
  (2) b followed by all permutations of (a,c,d)
  (3) c followed by all permutations of (a,b,d)
  (4) d followed by all permutations of (a,b,c)

- The expression “followed by all permutations” is the clue to ‘recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on n – 1 elements. These observations lead to Program 1.11, which is invoked by perm (a, 0, n).

1.1.3. PERFORMANCE ANALYSIS

Space complexity:

Amount of memory space required to solve the algorithm.

- i)Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- ii)Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
    - recursive stack space, formal parameters, local variables, return address

Space Complexity

$S(P)=C+S_P(I)$

*Program 1.10: Iterative function for summing a list of numbers (p.20)

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
inti;
  for (i = 0; i<n; i++)
tempsum += list [i];
```

```
  return tempsum;
}
```

*Program 1.11: Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
 }
```

Time Complexity :

- Amount of compilation time and run time to execute algorithm
- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

$$T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)$$

*Program 1.12: Program 1.10 with count statements (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
inti;
   for (i = 0; i< n; i++) {
       count++;           /*for the for loop */
tempsum += list[i]; count++;  /* for assignment */
   }
   count++;       /* last execution of for */
   return tempsum;
   count++;       /* for return */
}
```

*Program 1.13: Simplified version of Program 1.12 (p.23)

```
float sum(float list[ ], int n)
{
```

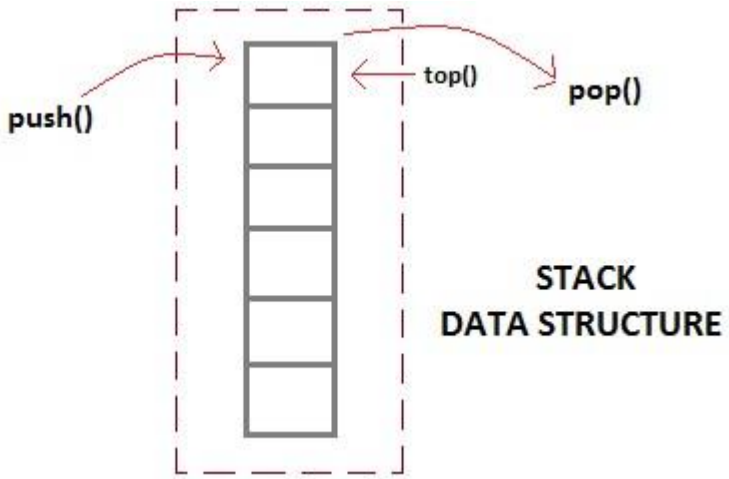
```
    float tempsum = 0;
inti;
    for (i = 0; i< n; i++)
        count += 2;
    count += 3;
    return 0;
}
```

1.2. ELEMENTARY DATA STRUCTURES

1.2.1. STACK

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out)
- **Stack** is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order.
- Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.

Basic features of Stack



1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Algorithm for PUSH operation

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

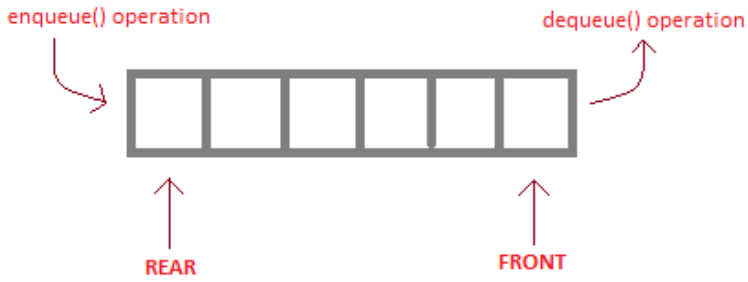
There are other uses also like:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

1.2.2. QUEUE

What is a Queue Data Structure?

- **Queue** is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).
- This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.
- The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

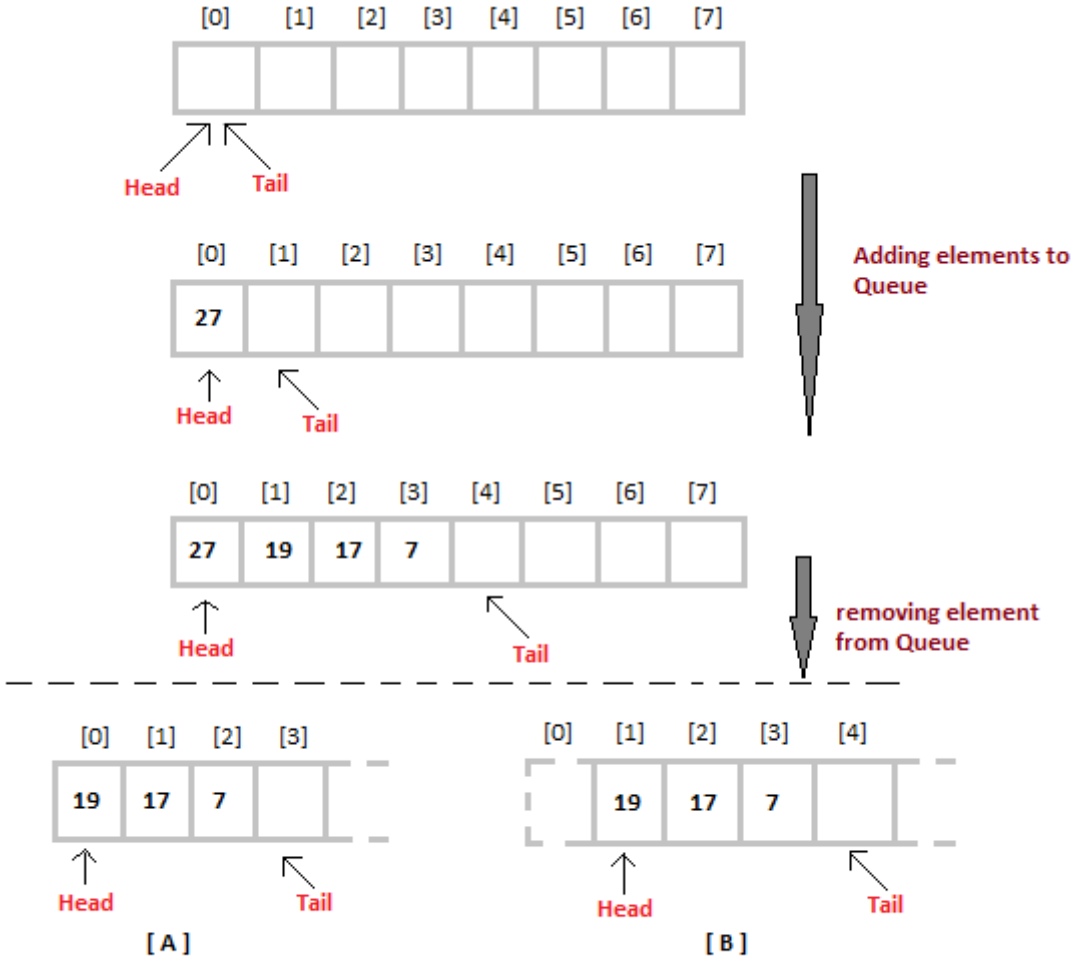
QUEUE DATA STRUCTURE

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

Implementation of Queue Data Structure

- Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.
- Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

1.2.3. TREES

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

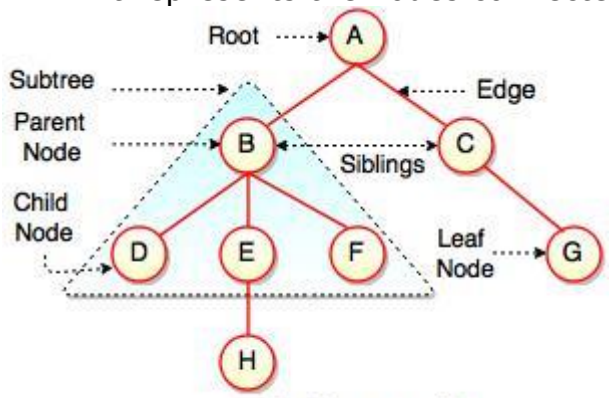


Fig. Structure of Tree

The above figure represents structure of a tree. Tree has 2 subtrees.
A is a parent of B and C.
B is called a child of A and also parent of D, E, F.

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.

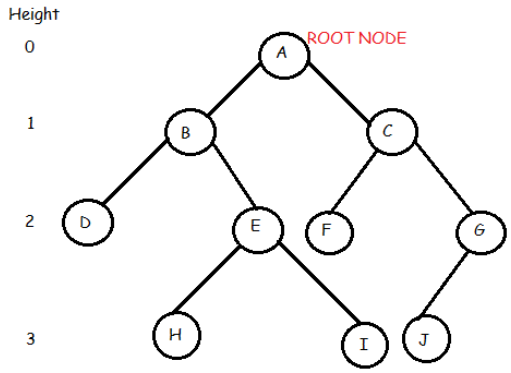
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

BINARY TREE

- A binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.
- Each node contains three components:

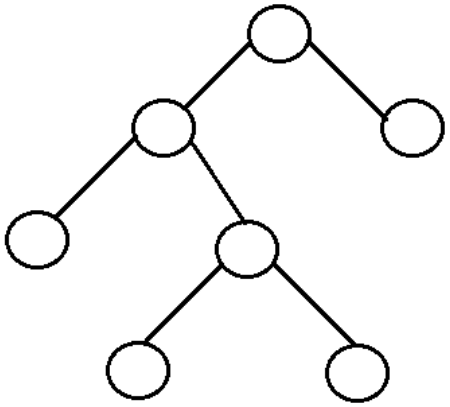
1. Pointer to left subtree
2. Pointer to right subtree
3. Data element

The topmost node in the tree is called the root. An empty tree is represented by **NULL** pointer.

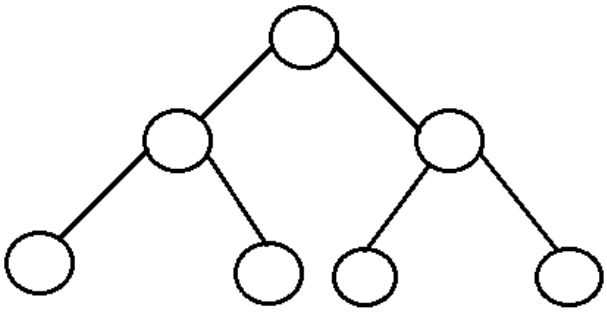


Types of Binary Trees (Based on Structure)

- **Rooted binary tree:** It has a root node and every node has atmost two children.
- **Full binary tree:** It is a tree in which every node in the tree has either 0 or 2 children.

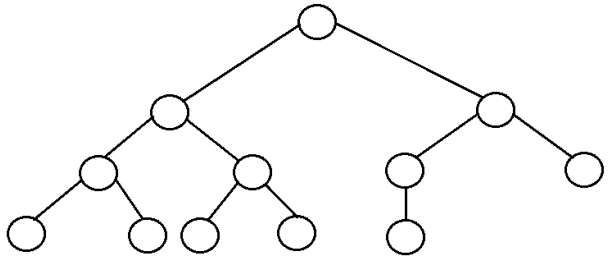


  - The number of nodes, *n*, in a full binary tree is atleast n = 2h – 1, and atmost $n = 2^{h+1} - 1$, where *h* is the height of the tree.
  - The number of leaf nodes *l*, in a full binary tree is number, *L* of internal nodes + 1, i.e, $l = L+1$.
- **Perfect binary tree:** It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



- A perfect binary tree with *l* leaves has $n = 2l-1$ nodes.
- In perfect full binary tree, $l = 2h$ and $n = 2h+1 - 1$ where, *n* is number of nodes, *h* is height of tree and *l* is number of leaf nodes

- **Complete binary tree:** It is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



The number of internal nodes in a complete binary tree of n nodes is *floor(n/2)*.

1.2.4. DICTIONARIES

- A *dictionary* is a general-purpose data structure for storing a group of objects. A dictionary has a set of *keys* and each key has a single associated *value*.
- When presented with a key, the dictionary will return the associated value.
- For example, the results of a classroom test could be represented as a dictionary with pupil's names as keys and their scores as the values:

```
results={'Detra':17,
'Nova':84,
'Charlie':22,
'Henry':75,
'Roxanne':92,
'Elsa':29}
```

- The concept of a *key-value store* is widely used in various computing systems, such as caches and high-performance databases.
- Dictionaries are often implemented as hash tables.
- Keys in a dictionary must be unique; an attempt to create a duplicate key will typically overwrite the existing value for that key.

Dictionaries typically support several operations:

- retrieve a value (depending on language, attempting to retrieve a missing key may give a default value or throw an exception)

- insert or update a value (typically, if the key does not exist in the dictionary, the key-value pair is inserted; if the key already exists, its corresponding value is overwritten with the new one)
- remove a key-value pair
- test for existence of a key

1.2.5. Priority Queues

- **Priority queue** data structure is an abstract data type that provides a way to maintain a set of elements, each with an associated value called key.
- There are two kinds of priority queues: a **max-priority queue** and a **min-priority queue**. In both kinds, the priority queue stores a collection of elements and is always able to provide the most **"extreme"** element, which is the only way to interact with the priority queue. For the remainder of this section, we will discuss max-priority queues. Min-priority queues are analogous.

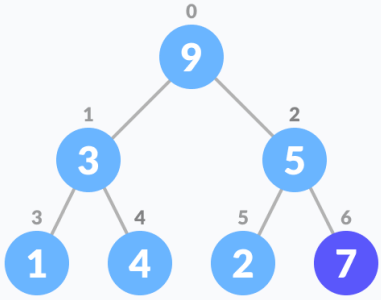
Operations

- A max-priority queue provides the following operations:
  - **insert:** add an element to the priority queue.
  - **maxElement:** return the largest element in the priority queue.
  - **removeMaxElement:** remove the largest element from the priority queue.

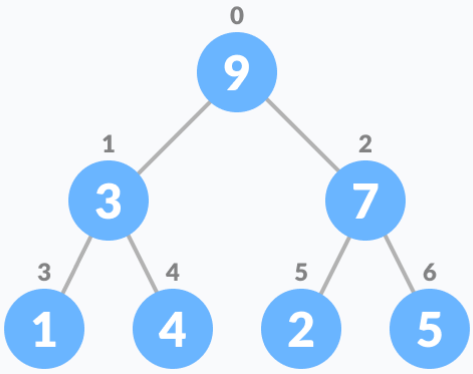
. Inserting an Element into the Priority Queue

Inserting an element into a priority queue (max-heap) is done by the following steps.

- Insert the new element at the end of the tree.



- Insert an element at the end of the queue



- Heapify the tree.
- Heapify after insertion

Algorithm for insertion of an element into priority queue (max-heap)

If there is no node,

create a newNode.

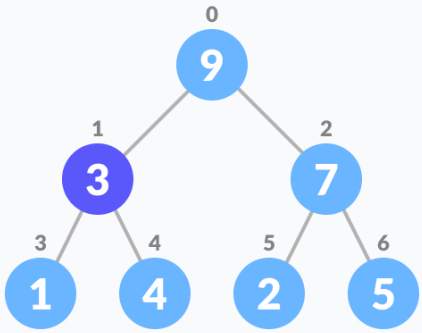
else (a node is already present)

insert the newNode at the end (last node from left to right.)

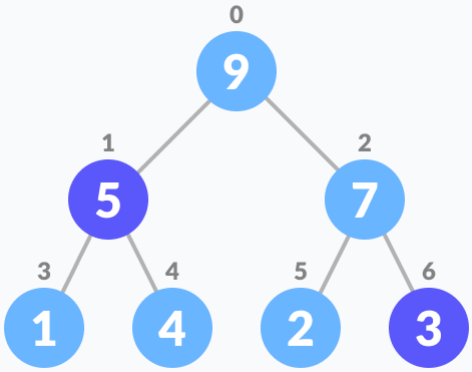
heapify the array

Deleting an Element from the Priority Queue

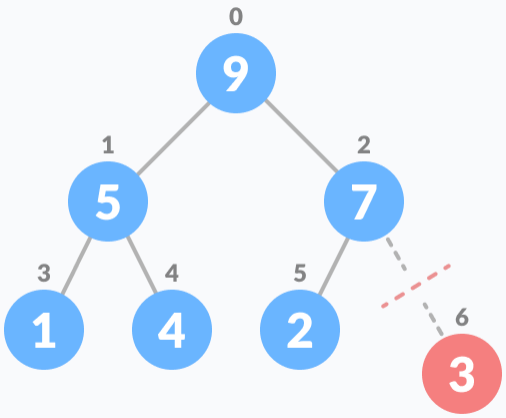
Deleting an element from a priority queue (max-heap) is done as follows:



- Select the element to be deleted.



Swap it with the last element. Swap with the last leaf node element



- Remove the last element.

Remove the last element leaf

- Algorithm for deletion of an element in the priority queue (max-heap)

```
If nodeToBeDeleted is the leafNode

  remove the node
```

```
Else swap nodeToBeDeleted with the lastLeafNode

  remove noteToBeDeleted



heapify the array
```

Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

1.2.6. SETS AND DISJOINT SETS UNION

- **A *disjoint-set data structure* is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.**
- **A *union-find algorithm* is an algorithm that performs two useful operations on such a data structure:**
  Find: **Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.**
  Union: **Join two subsets into a single subset.**

Set:

A set is a collection of distinct elements. The Set can be represented, for examples, as S1={1,2,5,10}.

Disjoint Sets:

The disjoints sets are those do not have any common element.

For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2 are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are

1. Union
2. Find

Disjoint set Union:

If Si and Sj are tow disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:

S1={1,7,8,9} S2={2,5,10}

S1 U S2={1,2,5,7,8,9,10}

Find:

Given the element I, find the set containing i.

Example:

S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

Then,

Find(4)= S3 Find(5)=S2 Find97)=S1

Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:

S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

Then these sets can be represented as

```
2.     bool find(intArr[],int A,int B)
3.     {
4.     if(Arr[ A ]==Arr[ B ])
5.     returntrue;
6.     else
7.     returnfalse;
8.     }
9.     //change all entries from Arr[ A ] to Arr[ B ].
10.    voidunion(intArr[],int N,int A,int B)
                                11.    {
12.    int TEMP =Arr[ A ];
13.    for(inti=0;i<N;i++)
14.    {
15.    if(Arr[i]== TEMP)
16.    Arr[i]=Arr[ B ];
17.    }
```

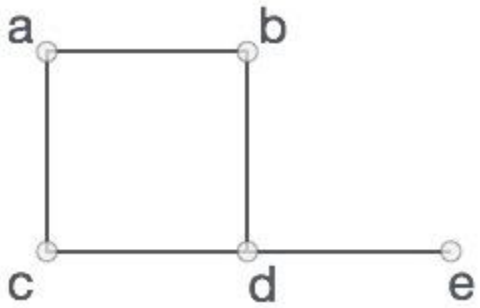
```
18.    }
```

1.2.7. GRAPHS

Definition

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

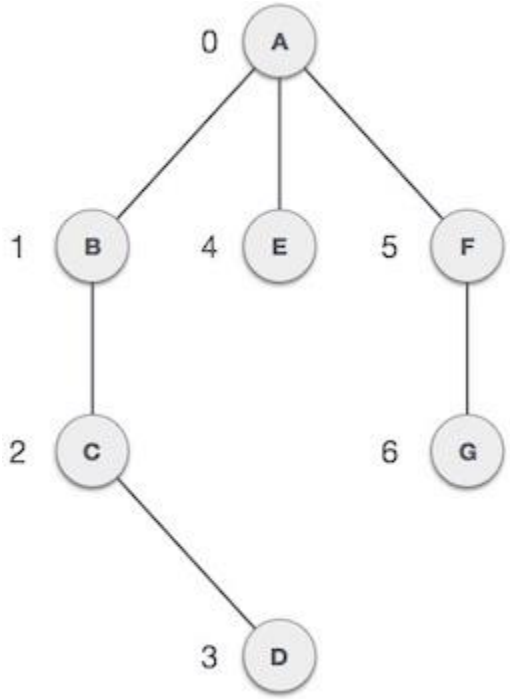
E = {ab, ac, bd, cd, de}

Graph Terminology

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph.

Adjacency Matrix

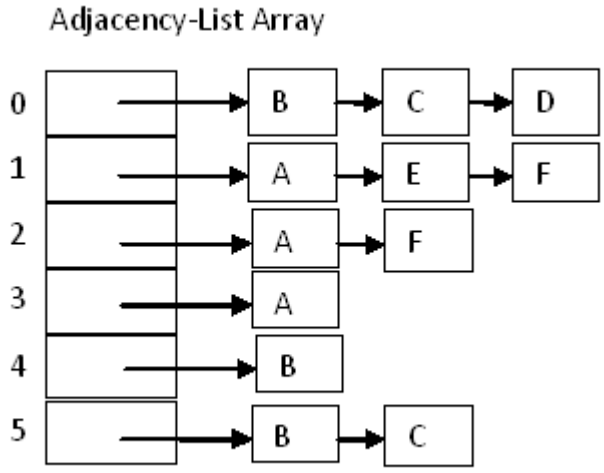
It is a two dimensional array with Boolean flags. As an example, we can represent the edges for the above graph using the following adjacency matrix

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	0	1
D	1	0	0	0	0	0
E	0	1	0	0	0	0
F	0	1	1	0	0	0

In the given graph, A is connected with B, C and D nodes, so adjacency matrix will have 1s in the 'A' row for the 'B', 'C' and 'D' column

Adjacency List

It is an array of linked list nodes. In other words, it is like a list whose elements are a linked list. For the given graph example, the edges will be represented by the below adjacency list:

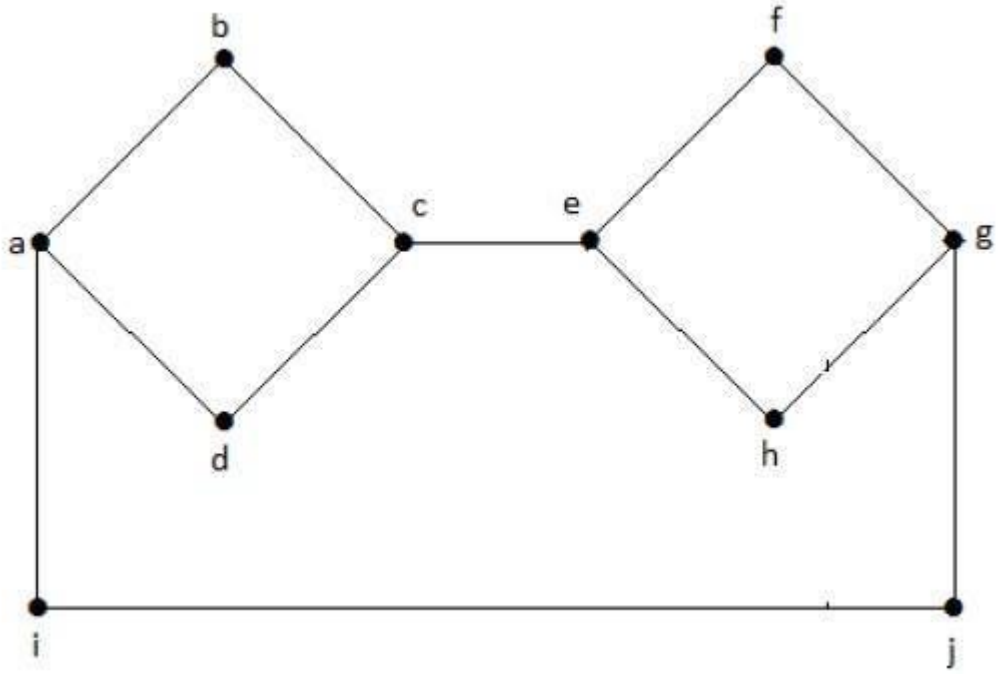


Connected Graph

A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Example

In the following graph, each vertex has its own edge connected to other edge. Hence it is a connected graph.

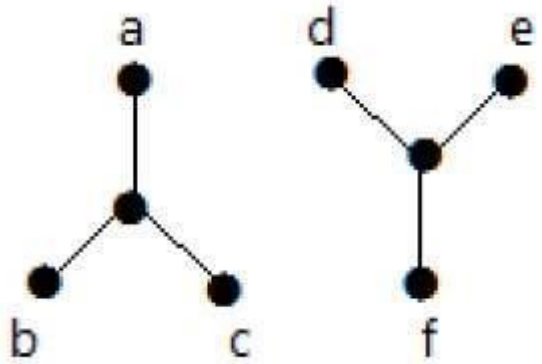


Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

Example 1

The following graph is an example of a Disconnected Graph, where there are two components, one with ‘a’, ‘b’, ‘c’, ‘d’ vertices and another with ‘e’, ’f’, ‘g’, ‘h’ vertices.



The two components are independent and not connected to each other. Hence it is called disconnected graph.

DESIGN AND ANALYSIS OF ALGORITHM-7MCE1C2

UNIT-2

2.1. DIVIDE AND CONQUER

2.1.1. GENERAL METHOD

2.1.2. DEFECTIVE CHESSBOARD

2.1.3. BINARY SEARCH

2.1.4. FINDING MAXIMUM AND MINIMUM

2.1.5. MERGE SORT

2.1.6. QUICK SORT

2.1.7. SELECTION SORT

2.1.8. STRASSEN'S MATRIX MULTIPLICATION

2.1. DIVIDE AND CONQUER

2.1.1. THE GENERAL METHOD

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide-and-conquer is a very powerful use of recursion.

```
DANDC (P)
{
if SMALL (P) then
return S (p);
else
{
divide p into smaller instances p1, p2, …. Pk, k 1; apply DANDC
to each of these sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),….DANDC (pk));
}
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function „S“ is invoked otherwise, the problem „p“ into smaller sub problems. These sub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

2.1.2. THE DEFECTIVE CHESSBOARD

- Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1).

- Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

- This problem can be solved using Divide and Conquer. Below is the recursive algorithm.

```
// n is size of given square, p is location of missing cell
Tile(int n, Point p)

1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing
   but a tile and can be filled with a single tile.

2) Place a L shaped tile at the center such that it does not cover
   the n/2 * n/2 subsquare that has a missing square. Now all four
   subsquares of size n/2 x n/2 have a missing cell (a cell that doesn't
   need to be filled).  See figure 2 below.

3) Solve the problem recursively for following four. Let p1, p2, p3 and
   p4 be positions of the 4 missing cells in 4 squares.
   a) Tile(n/2, p1)
   b) Tile(n/2, p2)
   c) Tile(n/2, p3)
   d) Tile(n/2, p3)
```

The below diagrams show working of above algorithm

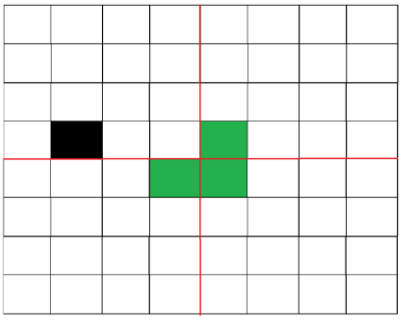


Figure 2: After placing first tile

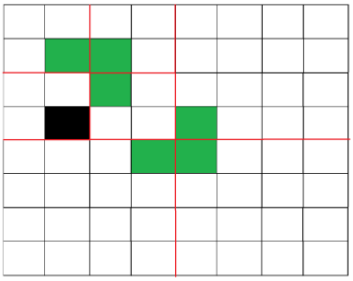


Figure 3: Recurring for first subsquare.

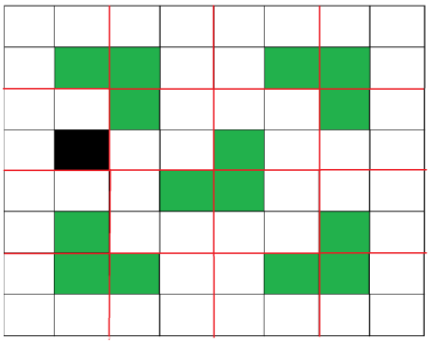


Figure 4: Shows first step in all four subsquares.

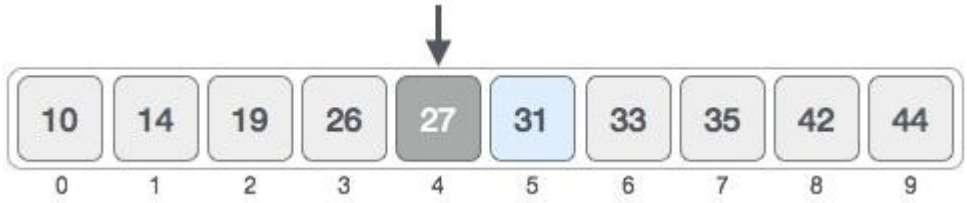
2.1.3 BINARY SEARCH

- Binary search looks for a particular item by comparing the middle most item of the collection. ... If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example.
- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



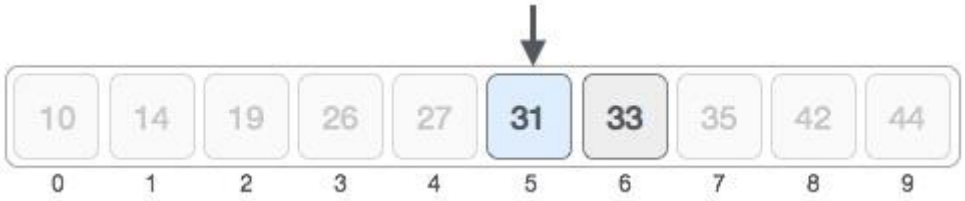
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.


```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ←value to be searched

Set lowerBound =1
Set upperBound = n

while x not found
```

```
if upperBound < lowerBound
        EXIT: x does not exists.

set midPoint = lowerBound +( upperBound - lowerBound )/2

if A[midPoint]< x
set lowerBound = midPoint +1

if A[midPoint]> x
set upperBound = midPoint -1

if A[midPoint]= x
        EXIT: x found at location midPoint
endwhile

end procedure
```

2.1.4. FINDING MAXIMUM AND MINIMUM

1. Let us consider simple problem that can be solved by the divide-and conquer technique.
2. The problem is to find the maximum and minimum value in a set of 'n' elements.
3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.
4. Hence, the time is determined mainly by the total cost of the element comparison.
5. comparison.

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

Explanation:

a. Straight MaxMin requires 2(n-1) element comparisons in the best, average & worst cases.

b. By realizing the comparison of a [i]max is false, improvement in a algorithm can be done.

c. Hence we can replace the contents of the for loop by, If (a [i]> Max) then Max = a [i]; Else if (a [i]< 2(n-1)

d. On the average a[i] is > max half the time, and so, the avg. no. of comparison is 3n/2-1.

A Divide and Conquer Algorithm for this problem would proceed as follows:

a. Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.

b. Here 'n' is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.

c. If the list has more than 2 elements, P has to be divided into smaller instances.

d. For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm:

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

Example:

A	1	2	3	4	5	6	7	8	9
Values	22	13	-5	-8	15	60	17	31	47

Tree Diagram:

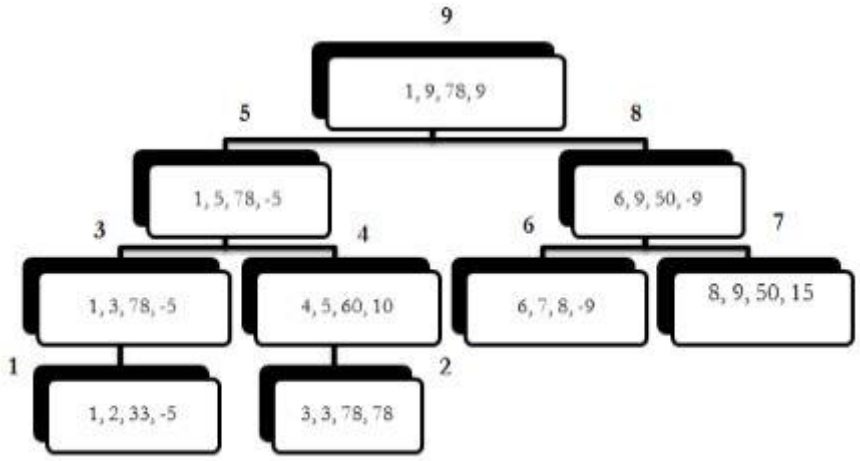
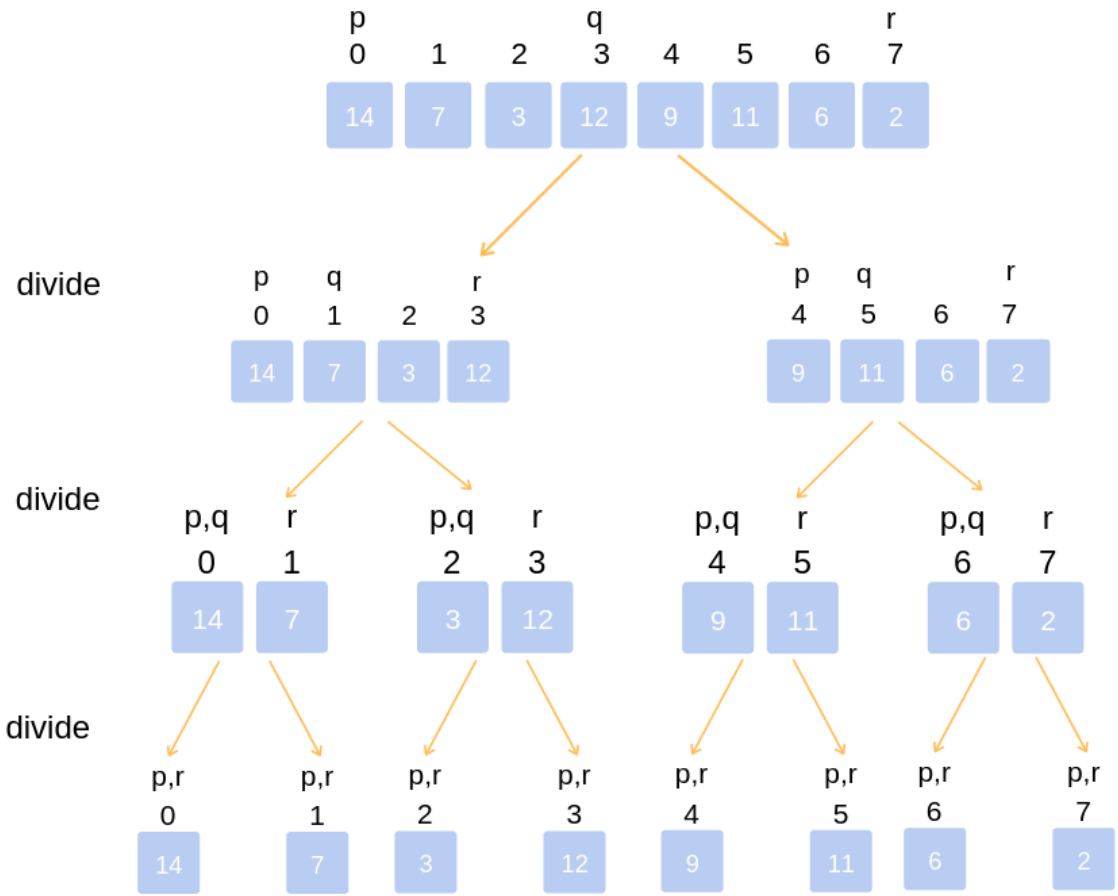


Figure 4

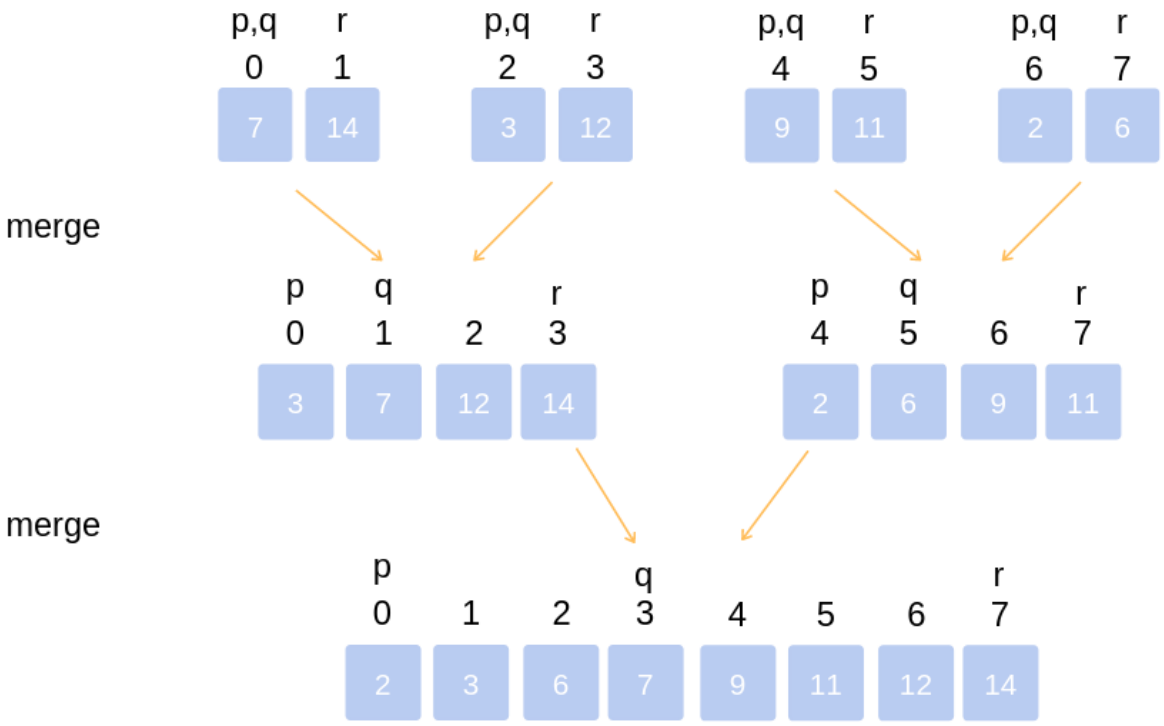
2.1.5. MERGE SORT

- Merge sort is one of the most efficient sorting algorithms.
- It works on the principle of Divide and Conquer.
- Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.
- Example: Let us consider an example to understand the approach better.
    1. Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).



- Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

- The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list.
- This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.




```
voidmerge(int*Arr,intstart,intmid,intend){
        // create a temp array
        inttemp[end-start+1];

        // crawlers for both intervals and for temp
        inti=start,j=mid+1,k=0;

        // traverse both arrays and in each iteration add smaller of both elements in temp
        while(i<=mid&&j<=end){
                if(Arr[i]<=Arr[j]){
                        temp[k]=Arr[i];
                        k+=1;i+=1;
                }
                else{
                        temp[k]=Arr[j];
                        k+=1;j+=1;
                }
        }

        // add elements left in the first interval
        while(i<=mid){
                temp[k]=Arr[i];
                k+=1;i+=1;
        }

        // add elements left in the second interval
        while(j<=end){
```

```
                temp[k]=Arr[j];
                k+=1;j+=1;
        }

        // copy temp to original interval
        for(i=start;i<=end;i+=1){
                Arr[i]=temp[i-start]
        }
}

// Arr is an array of integer type
// start and end are the starting and ending index of current interval of Arr

voidmergeSort(int*Arr,intstart,intend)
{

        if(start<end){
                intmid=(start+end)/2;
                mergeSort(Arr,start,mid);
                mergeSort(Arr,mid+1,end);
                merge(Arr,start,mid,end);
        }
}
```

2.1.6. QUICK SORT:

- Technically, quick sort follows the below steps:
  Step 1 − Make any element as pivot
  Step 2 − Partition the array on the basis of pivot
  Step 3 − Apply quick sort on left partition recursively
  Step 4 − Apply quick sort on right partition recursively

- Consider the following array: 50, 23, 9, 18, 61, 32.
- the pivot (32) comes at its actual position and all elements to its left are lesser, and all elements to the right are greater than itself.
- **Step 2**: The main array after the first step becomes

```
23, 9, 18, 32, 61, 50
```

Step 3: Now the list is divided into two parts:

1. Sublist before pivot element
2. Sublist after pivot element

Step 4: Repeat the steps for the left and right sublists recursively. The final array thus becomes
9, 18, 23, 32, 50, 61.

```
void swap(int *a, int *b)
{
```

```
	int temp;
	temp = *a;
	*a = *b;
	*b = temp;
}

// Partitioning the array on the basis of values at high as pivot value.
int Partition(int a[], int low, int high)
{
	int pivot, index, i;
	index = low;
	pivot = high;


	for(i=low; i < high; i++)
	{
		if(a[i] < a[pivot])
		{


	return index;
}



int QuickSort(int a[], int low, int high)
{
	int pindex;
	if(low < high)
		pindex = RandomPivotPartition(a, low, high);

		QuickSort(a, low, pindex-1);
		QuickSort(a, pindex+1, high);
	}
	return 0;
}

int main()
{
	int n, i;
	cout<<"\nEnter the number of data elements to be sorted: ";
	cin>>n;

	int arr[n];
	for(i = 0; i < n; i++)
	{
		cout<<"Enter element "<<i+1<<": ";
		cin>>arr[i];
	}
```

```
        QuickSort(arr, 0, n-1);


        cout<<"\nSorted Data ";
        for (i = 0; i < n; i++)
        cout<<"->"<<arr[i];

        return 0;
}
```

2.1.7. SELECTION SORT:

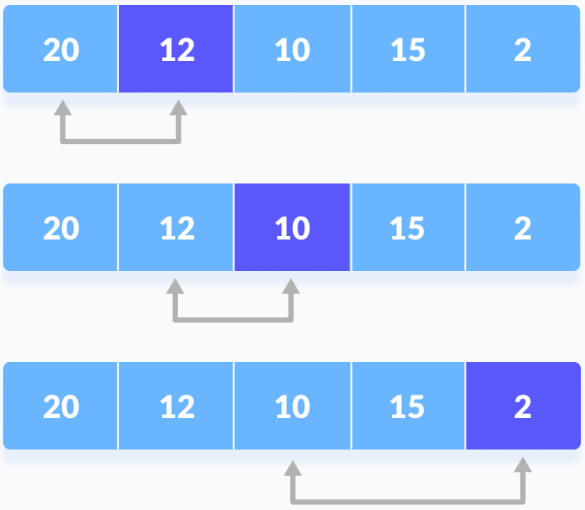
- Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

How Selection Sort Works?

1. Set the first element as minimum.

   Select first element as minimum
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until



the last element. Compare minimum with the remaining elements

3. After each iteration, minimum is placed in the front of the unsorted list.

Swap the first with minimum

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

```
voidselectionSort(intarray[], int size){
for (int step = 0; step < size - 1; step++) {
int min_idx = step;
for (int i = step + 1; i < size; i++) {

// To sort in descending order, change > to < in this line.
// Select the minimum element in each loop.
if (array[i] <array[min_idx])
        min_idx = i;
    }

// put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}
voidswap(int *a, int *b){
```

```
int temp = *a;
  *a = *b;
  *b = temp;
}

// function to print an array
voidprintArray(intarray[], int size){
for (int i = 0; i < size; i++) {
cout<<array[i] <<" ";
  }
cout<<endl;
}
```

2.1.8. STRASSEN'S MATRIX MULTIPLICATION

- For multiplying the two 2*2 dimension matrices **Strassen's** used some formulas in which there are seven multiplication and eighteen addition, subtraction, and in brute force algorithm, there is eight multiplication and four addition.
- The utility of Strassen's formula is shown by its asymptotic superiority when order **n** of matrix reaches infinity. Let us consider two matrices **A** and **B**, **n*n** dimension, where **n** is a power of two.
- It can be observed that we can contain four **n/2*n/2** submatrices from **A**, **B** and their product **C**. **C** is the resultant matrix of **A** and **B**.

Procedure of Strassen matrix multiplication

There are some procedures:

1. Divide a matrix of order of 2*2 recursively till we get the matrix of 2*2.
2. Use the previous set of formulas to carry out 2*2 matrix multiplication.
3. In this eight multiplication and four additions, subtraction are performed.
4. Combine the result of two matrixes to find the final product or final matrix.

Formulas for Stassen's matrix multiplication

In **Strassen's matrix multiplication** there are seven multiplication and four addition, subtraction in total.

1. D1 = (a11 + a22) (b11 + b22)
2. D2 = (a21 + a22).b11
3. D3 = (b12 – b22).a11
4. D4 = (b21 – b11).a22
5. D5 = (a11 + a12).b22
6. D6 = (a21 – a11) . (b11 + b12)
7. D7 = (a12 – a22) . (b21 + b22)

C11 = d1 + d4 – d5 + d7

```
C12 = d3 + d5
C21 = d2 + d4
C22 = d1 + d3 – d2 – d6
```

Algorithm for Strassen's matrix multiplication

Algorithm Strassen(n, a, b, d)

```
begin
        If n = threshold then compute
                C = a * b is a conventional matrix.
        Else
                Partition a into four sub matrices  a11, a12, a21, a22.
                Partition b into four sub matrices b11, b12, b21, b22.
                Strassen ( n/2, a11 + a22, b11 + b22, d1)
                Strassen ( n/2, a21 + a22, b11, d2)
                Strassen ( n/2, a11, b12 – b22, d3)
                Strassen ( n/2, a22, b21 – b11, d4)
                Strassen ( n/2, a11 + a12, b22, d5)
                Strassen (n/2, a21 – a11, b11 + b22, d6)
                Strassen (n/2, a12 – a22, b21 + b22, d7)

                C = d1+d4-d5+d7     d3+d5
                d2+d4          d1+d3-d2-d6

        end if

        return (C)
end.
```

DESIGN ANALYSIS OF ALGORITHM-7MCE1C2

UNIT-3

3.1. GREEDY METHOD

3.1.1. INTRODUCTION

3.1.2. CONTAINER LOADING

3.1.3. KNAPSACK PROBLEM

3.1.4. TREE VERTEX SPLITTING

3.1.5. JOB SEQUENCING WITH DEADLINES

3.1.6. MINIMUM COST SPANNING TREES

3.1.7. OPTIMAL STORAGE ON TAPES

3.1.8. OPTIMAL MERGE PATTERNS

3.1.9. SINGLE SOURCE SHORTEST PATH

3.1. GREEDY METHOD :

3.1.1. GENERAL METHOD

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

- 1 − Select one ₹ 10 coin, the remaining count is 8
- 2 − Then select one ₹ 5 coin, the remaining count is 3
- 3 − Then select one ₹ 2 coin, the remaining count is 1
- 4 − And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

3.1.2. CONTAINER LOADING

- A **container** is a class, a **data structure**, or an abstract **data** type (ADT) whose instances are collections of other objects.
- In other words, they store objects in an organized way that follows specific access rules. The size of the **container** depends on the number of objects (elements) it contains.
- Ship has capacity c.
- m containers are available for loading.

- Weight of container i is w_i.
- Each weight is a positive number.
- Sum of container weight < c.
- Load as many containers as possible without sinking the ship
- Can all containers be loaded into 2 ships whose capacity is c each?
- Same as bin packing with 2 bins
  (Are 2 bins sufficient for all items?)
- Same as machine scheduling with 2 machines
  (Can all jobs be completed by 2 machines in c time units?)
- NP-hard
  .

3.1.3. KNAPSACK PROBLEM

The **knapsack problem** is a **problem** in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

According to the problem statement,

- There are n items in the store
- Weight of ith item wi>0wi>0
- Profit for ith item pi>0pi>0 and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x i of i thth item.
0⩽xi⩽10⩽xi⩽1

The i thth item contributes the weight xi.wixi.wi to the total weight in the knapsack an profit xi.pixi.pi to the total profit.
Hence, the objective of this algorithm is to

maximize∑n=1n(xi.pi)maximize∑n=1n(xi.pi)

subject to constraint,

∑n=1n(xi.wi)⩽W∑n=1n(xi.wi)⩽W

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

∑n=1n(xi.wi)=W∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, x is an array to store the fraction of items.
Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

for i = 1 to n

do x[i] = 0

weight = 0

for i = 1 to n

if weight + w[i] ≤ W then

x[i] = 1

weight = weight + w[i]

else

x[i] = (W - weight) / w[i]

weight = W

break

return x

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Example

Let us consider that the capacity of the knapsack W = 60 and the list of provided items are shown in the following table −

Item	**B**	**A**	**C**	**D**
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (piwi)(piwi)	7	10	6	5

Solution

After sorting all the items according to piwipiwi. First all of B is chosen as weight of B is less than the capacity of the knapsack. Next, item A is chosen, as the available capacity of the

knapsack is greater than the weight of A. Now, C is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of C.

Hence, fraction of C (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is 10 + 40 + 20 * (10/20) = 60

And the total profit is 100 + 280 + 120 * (10/20) = 380 + 60 = 440

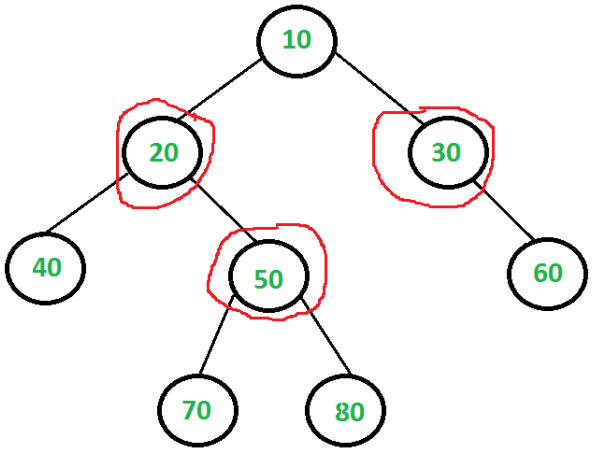
This is the optimal solution. We cannot gain more profit selecting any different combination of items.

3.1.4. TREE VERTEX SPLITTING

- Let G= (V,E,w) be a weighted directed acyclic graph (wdag) with vertex set V, edge set E, and edge weighting funtionw. w(i,j) is the weight of the edge <i,j>∈E. w(i,j) is a positive integer for <i,j>∈E and w(i,j) is undefined if <i,j>∉E.
- A source vetexis a vertex with zero indegree while a sink vetexis a vertex with zero out-degree. The delay, d(P), of the path P is the sum of the weights of the edges on that path. The delay, d (G), of the graph G is the maximum path delay in the graph, i.e.,
    - d(G) = max { d(P) }
        - PinG
- Let G/X be the wdag that results when each vertex v in X is split into two v^i and v^osuch that all edges <v,j>∈E are replaced by edges of the form <v^o,j>and all edges <i,v>∈E are replaced by edges of the form <i,v^i>. I.e., outbound edges of v now leave vertex v^owhile the inbound edges of v now enter vertex v^i. Figure 3 shows the result, G/X, of splitting the vertex 5 of the dag of Figure 2.
- The dag vertex splitting problem (DVSP) is to find a least cardinality vertex set X such that d(G/X) ≤δ , where δ is a prespecified delay. For the dag of Figure 2 and δ = 3, X = {5} is a solution to the DVSP problem.
- **Lemma 1:** Let G= (V,E,w) be a weighted dag and let δ be a prespecified delay value. Let Max-
- EdgeDelay = max { w(i,j) }. Then the DVSP has a solution iffδ≥ MaxEdgeDelay.
    - <i,j>∈E

- **Proof:** Vertex splitting does not eliminate any edges. So, there is no X such that d(G/X) <MaxEdgeDelay.
- Further, d(G/V) = MaxEdgeDelay. So, for every δ≥ MaxEdgeDelay, there is a least cardinality set X such that d(G/X) ≤δ. □

- For example, consider the following binary tree. The smallest vertex cover is {20, 50, 30} and size of the vertex cover is 3.



3.1.5. JOB SEQUENCING WITH DEADLINES

- Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline.
- These jobs need to be ordered in such a way that there is maximum profit.
- It may happen that all of the given jobs may not be completed within their deadlines.
- Assume, deadline of $\mathbf{i^{th}}$ job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.
- Thus, D(i)>0D(i)>0 for 1⩽i⩽n1⩽i⩽n.
- Initially, these jobs are ordered according to profit, i.e. p1⩾p2⩾p3⩾...⩾pnp1⩾p2⩾p3⩾...⩾pn.

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
  r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r do
    r := r – 1
```

```
if D(J(r)) ≤ D(i) and D(i) > r then
  for l = k … r + 1 by -1 do
    J(l + 1) := J(l)
    J(r + 1) := i
    k := k + 1
```

- Let us consider a set of given jobs as shown in the following table.
- We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	**J_1**	**J_2**	**J_3**	**J_4**	**J_5**
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J_2**	**J_1**	**J_4**	**J_3**	**J_5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.

- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

3.1.6. MINIMUM COST SPANNING TREE

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.

If there are **n** number of vertices, the spanning tree should have ***n* - 1** number of edges.

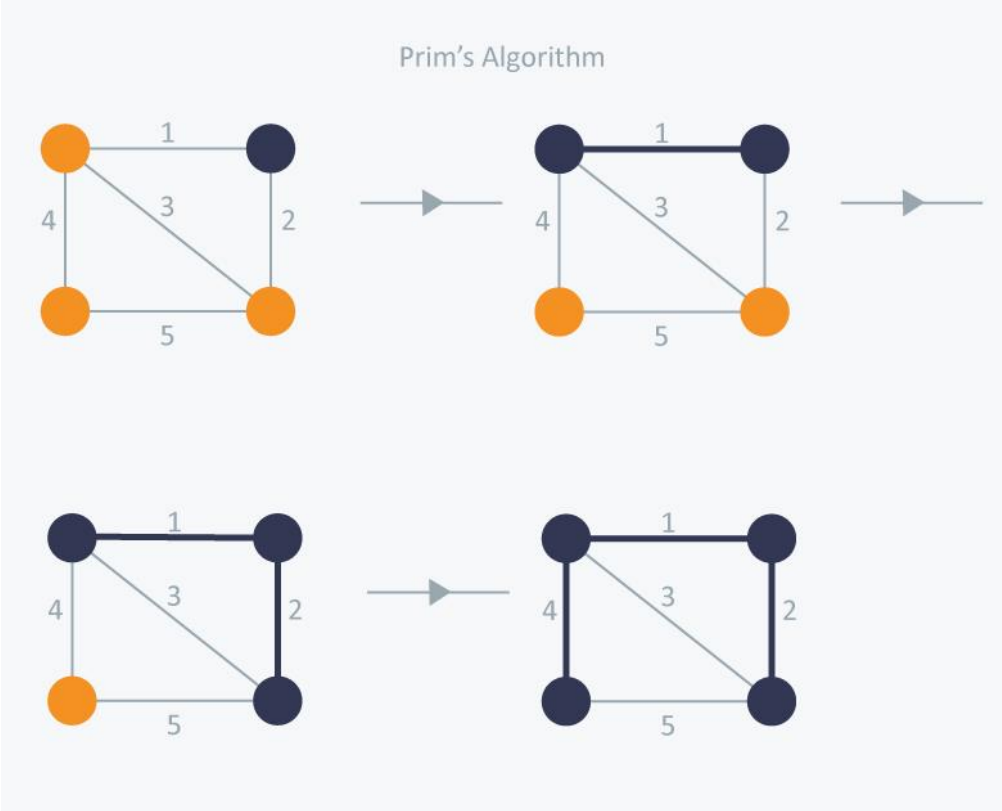
Prim's Algorithm :

In Prim's Algorithm we grow the spanning tree from a starting position.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:



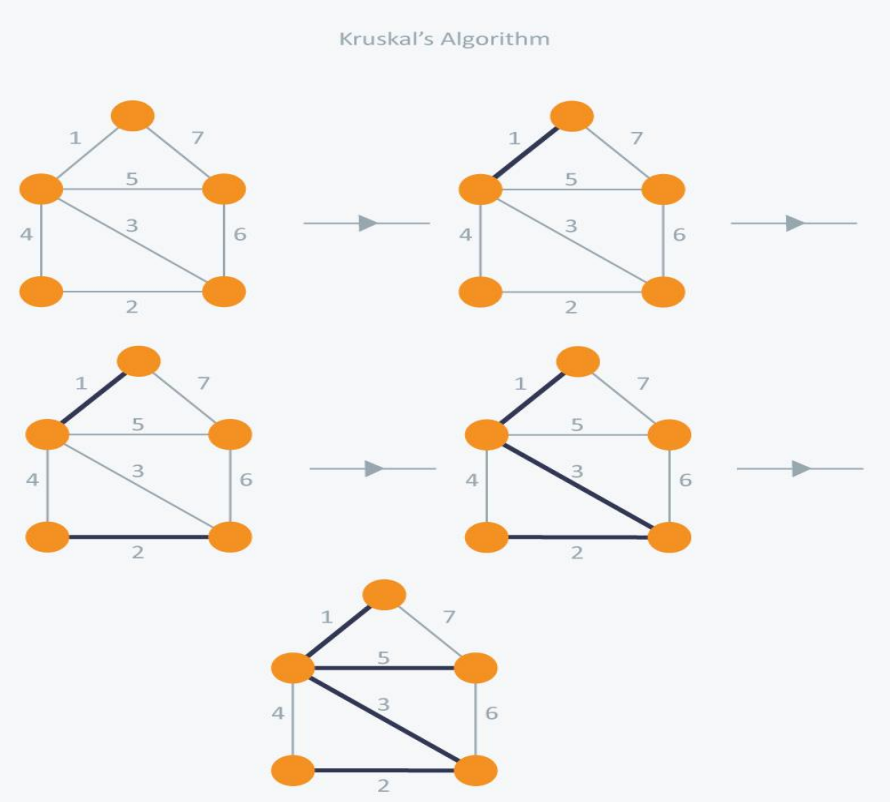
Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

- Consider following example:



3.1.7. OPTIMAL STORAGE ON TAPES

Input: We are given 'n' problem that are to be stored on computer tape of length L and the length of program i is Li

Such that $1 \leq i \leq n$ and $\Sigma\ 1 \leq k \leq j\ Li \leq 1$

Output: A permutation from all n! For the n programs so that when they are stored on tape in the order the MRT is minimized.

Example:

Let n = 3, (l1, l2, l3) = (8, 12, 2). As n = 3, there are 3! =6 possible ordering.

All these orderings and their respective d value are given below:

Ordering	**d (i)**	**Value**
1, 2, 3	8 + (8+12) + (8+12+2)	50
1, 3, 2	8 + 8 + 2 + 8 + 2 + 12	40
2, 1, 3	12 + 12 + 8 +12 + 8 + 2	54
2, 3, 1	12 + 12 +2 +12 + 2 + 8	48
3, 1, 2	2 + 2 + 8 + 2 + 8+ 12	34
3, 2, 1	2 + 2 +12 + 2 + 12 + 8	38

The optimal ordering is 3, 1, 2.

The greedy method is now applied to solve this problem. It requires that the programs are stored in non-decreasing order which can be done in O (nlogn) time.

Greedy solution:

i. Make tape empty

ii. Fori = 1 to n do;

iii. Grab the next shortest path

iv. Put it on next tape.

The algorithm takes the best shortest term choice without checking to see whether it is a big long term decision.

Algorithm:

```
Algorithm Optimal_Storage (n, m)
{
K = 0; // Next tape to be stored.
For i = 1 to n do
        {
        Write (i, k); // "Assign program", j, "to tape", k;
        k = (k +1) mod m;
        }
}
```

3.1.8. OPTIMAL MERGE PATTERNS

- **Optimal merge pattern** is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.
- If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.
- There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time.The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, f (i) represents the number of records in each file and d (i) represents the depth.

Algorithm for optimal merge pattern

```
Algorithm Tree(n)
//list is a global list of n single node
{
        For  i=1 to i= n-1 do
        {
                // get a new tree node
                Pt: new treenode;
                // merge two trees with smallest length
                (Pt = lchild) = least(list);
                (Pt = rchild) = least(list);
                (Pt =weight) = ((Pt = lchild) = weight) = ((Pt = rchild) = weight);
```

```
                    Insert (list , Pt);
            }
            // tree left in list
            Return least(list);
}
```

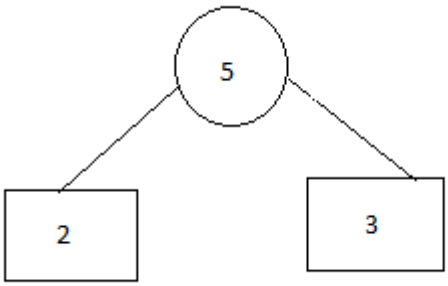
Example:

- Given a set of unsorted files: 5, 3, 2, 7, 9, 13
- Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13
- After this, pick two smallest numbers and repeat this until we left with only one number.

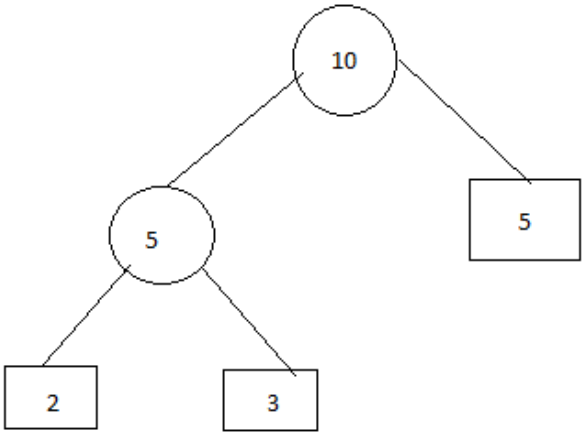
Now follow following steps:

Step 1: Insert 2, 3

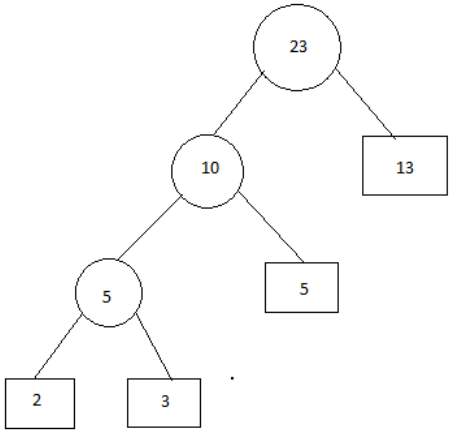
Step 2:



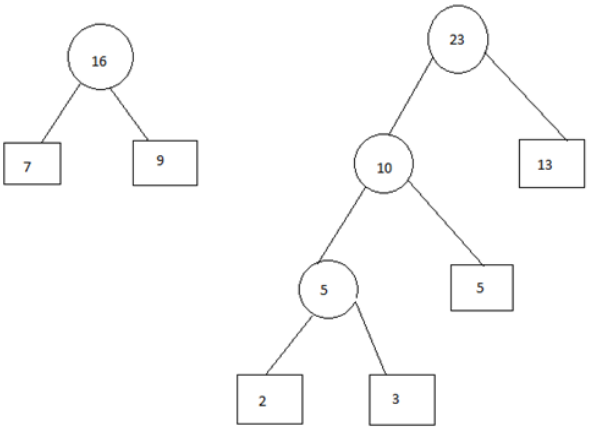
Step 3: Insert 5



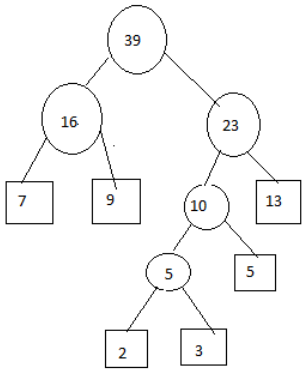
Step 4: Insert 13



Step 5: Insert 7 and 9



Step 6:



So, The merging cost = 5 + 10 + 16 + 23 + 39 = 93

3.1.9. SINGLE SOURCE SHORTEST PATH

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$).

Algorithm: Dijkstra's-Algorithm (G, w, s)

```
for each vertex v Є G.V
v.d := ∞
   v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v Є G.adj[u]
      if v.d>u.d + w(u, v)
v.d := u.d + w(u, v)
         v.∏ := u
```

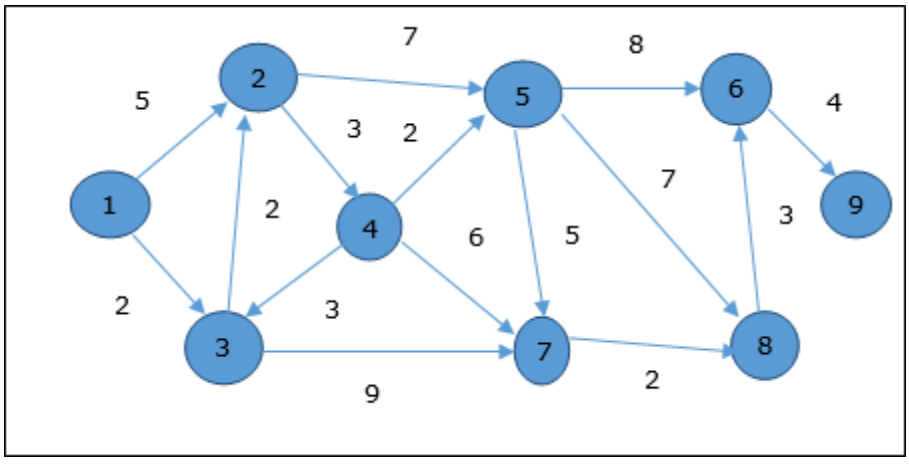
Example

Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	**Initial**	**Step1 V_1**	**Step2 V_3**	**Step3 V_2**	**Step4 V_4**	**Step5 V_5**	**Step6 V_7**	**Step7 V_8**	**Step8 V_6**
1	0	0	0	0	0	0	0	0	0

2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

- Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is
- 1→ 3→ 7→ 8→ 6→ 9
- This path is determined based on predecessor information.



DESIGN AND ANALYSIS OF ALGORITHM-7MCE1C2

UNIT-4

4.1. DYNAMIC PROGRAMMING

4.1.1. THE GENERAL METHOD

4.1.2. MULTISTAGE GRAPHS

4.1.3. ALL PAIRS SHORTEST PATH

4.1.4. SINGLE SOURCE SHORTEST PATH

4.1.5. OPTIMAL BINARY SEARCH TREES

4.1.6. STRING EDITING

4.1.7. 0/1 KNAPSACK

4.1.8. RELIABLITY DESIGN

4.1.9. TRAVELLING SALESMAN PROBLEM

4.1.10. FLOWSHOP SCHEDULING

4.2. BASIC TRAVERSAL AND SEARCH TECHNIQUES

4.2.1. TRAVERSAL TECHNIQUES FOR TREES

4.2.2. TRAVERSAL TECHNIQUES FOR GRAPHS

4.2.3. CONNECTED COMPONENT AND SPANNING TREES

4.2.4. BICONNECTED COMPONENTS AND DFS

4.1. DYNAMIC PROBLEM

4.1.1. GENERAL METHOD

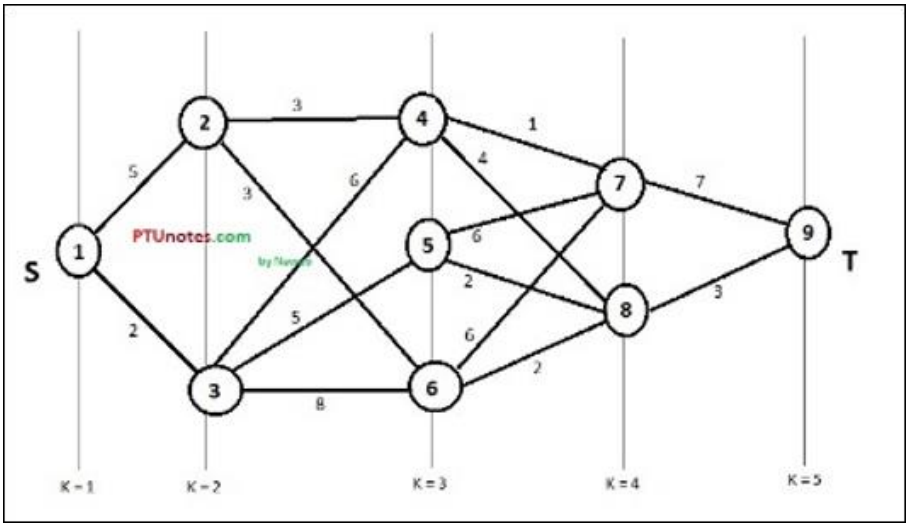
- Dynamic Programming solves problems by combining the solutions of subproblems.
- Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

4.1.2. MULTISTAGE GRAPHS

- A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where $k > 1$) number of disjoint subsets $S = \{s_1, s_2, \ldots, s_k\}$ such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{i+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.
- The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**.
- ***G*** is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.
- The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.
- **Example**
- Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

- In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

$$Cost(3, 4) = min \{c(4, 7) + Cost(7, 9), c(4, 8) + Cost(8, 9)\} = 7$$

$$Cost(3, 5) = min \{c(5, 7) + Cost(7, 9), c(5, 8) + Cost(8, 9)\} = 5$$

$$Cost(3, 6) = min \{c(6, 7) + Cost(7, 9), c(6, 8) + Cost(8, 9)\} = 5$$

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

$$Cost(2, 2) = min \{c(2, 4) + Cost(4, 8) + Cost(8, 9), c(2, 6) +$$

$$Cost(6, 8) + Cost(8, 9)\} = 8$$

$$Cost(2, 3) = \{c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8) + Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)\} = 10$$

Step-3: Cost (K-4, j)

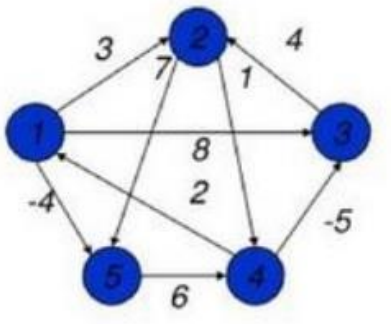
$$Cost\ (1, 1) = \{c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))\} = 12$$

$$c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))\} = 13$$

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

4.1.3. ALL PAIRS SHORTEST PATH

- The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

- At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.
- The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.
- Input − The cost matrix of the graph.

```
0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0
```

Output − Matrix of all pair shortest path.

```
0 3 4 5 6 7 7
3 0 2 1 3 4 4
4 2 0 1 3 2 3
5 1 1 0 2 3 3
6 3 3 2 0 2 1
7 4 2 3 2 0 1
7 4 3 3 1 1 0
```

Algorithm
floydWarshal(cost)

Input − The cost matrix of given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
   for k := 0 to n, do
```

```
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

4.1.4. SINGLE SOURCE SHORTEST PATH

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph $G = (V, E)$, where all the edges are non-negative (i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$).

Algorithm: Dijkstra's-Algorithm (G, w, s)

```
for each vertex v Є G.V
v.d := ∞
  v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
  u := Extract-Min (Q)
  S := S U {u}
  for each vertex v Є G.adj[u]
    if v.d>u.d + w(u, v)
v.d := u.d + w(u, v)
      v.∏ := u
```

Example

Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

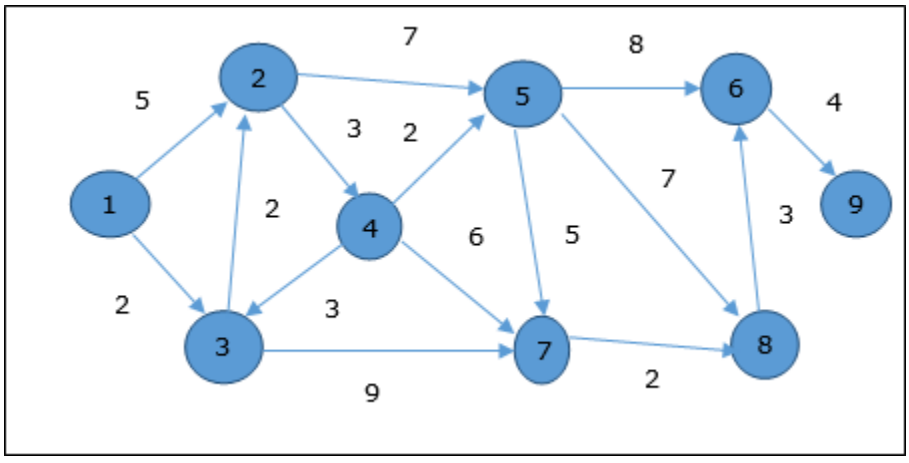
Vertex	**Initial**	**Step1 V_1**	**Step2 V_3**	**Step3 V_2**	**Step4 V_4**	**Step5 V_5**	**Step6 V_7**	**Step7 V_8**	**Step8 V_6**
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2

4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



4.1.5. OPTIMAL BINARY SEARCH TREES

- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes.
- The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.
- When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree.

- An optimal binary search tree is a BST, which has minimal expected cost of locating each node

Optimal Binary Search Tree

- Given a sorted array *keys[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts, where *freq[i]* is the number of searches to *keys[i]*.
- Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.
- Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

Examples:

```
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                   12
          \                 /
           12              10
          I                II
Frequency of searches of 10 and 12 are 34 and 50 respectively.
The cost of tree I is 34*1 + 50*2 = 134
The cost of tree II is 50*1 + 34*2 = 118


Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10              12               20        10            20
      \           /   \            /           \          /
      12         10    20         12             20        10
        \                       /              /         \
         20                   10              12          12
    I             II          III            IV           V
Among all possible BSTs, cost of the fifth BST is minimum.
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

4.1.6. STRING EDITING

- There are two strings given.
- The first string is the source string and the second string is the target string.
- In this program, we have to find how many possible edits are needed to convert first string to the second string.
- The edit of strings can be either Insert some elements, delete something from the first string or modify something to convert into the second string.

Input and Output

```
Input:
Two strings to compare.
string 1: Programming
string 2: Programs
Output:
Enter the initial string: Programming
Enter the final string: Programs
The number of changes required to convert Programming to Programs is 4
```

Algorithm

```
editCount(initStr, finalStr, initLen, finalLen)
```

Input − The initial and final string and their lengths.
Output − Number of edits are required to make initStr to finalStr.

```
Begin
  if initLen = 0, then
    return finalLen
  if finalLen := 0, then
    return initLen

  if initStr[initLen - 1] = finalStr[finalLen - 1], then
    return editCount(initStr, finalStr, initLen – 1, finalLen - 1)
  answer := 1 + min of (editCount(initStr, finalStr, initLen , finalLen - 1)),
    (editCount(initStr, finalStr, initLen – 1, finalLen ),
    (editCount(initStr, finalStr, initLen – 1, finalLen - 1)
  return answer
End
```

4.1.7. 0/1 KNAPSACK

- In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it.
- This is reason behind calling it as 0-1 Knapsack.
- Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.
    - Knapsack cannot be solved by Greedy approach.
- Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D

Profit	24	18	18	10
Weight	24	10	10	7

- Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.
- After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

The algorithm takes the following inputs

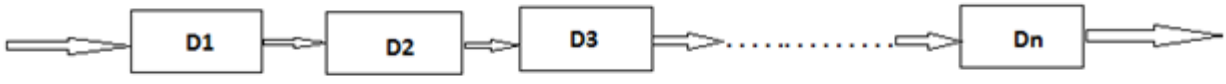
- The max$_i$mum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
      if vi + c[i-1, w-wi] then
        c[i, w] = vi + c[i-1, w-wi]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

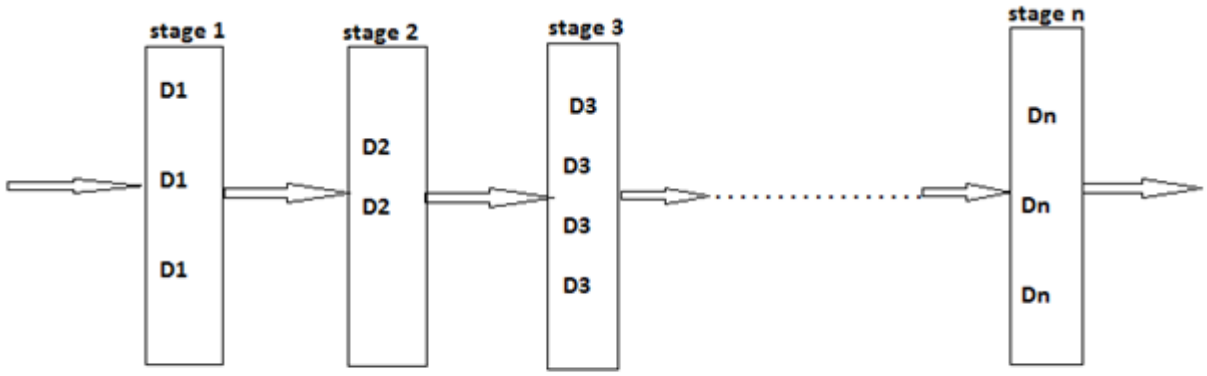
4.1.8. RELIABILITY DESIGN

- In **reliability design**, the problem is to design a system that is composed of several devices connected in series.



- If we imagine that r1 is the reliability of the device.
- Then the reliability of the function can be given by **πr1**.
- If **r1 = 0.99** and **n = 10** that **n** devices are set in a series, **1 <= i<= 10**, then reliability of the whole system **πri** can be given as: **Πri = 0.904**

- So, if we duplicate the devices at each stage then the reliability of the system can be increased.
- they make use of such devices at each stage, that result is increase in reliability at each stage. If at each stage, there are **mi** similar types of devices **Di**, then the probability that all **mi** have a malfunction is **(1 - ri)^mi**, which is very less.
- And the reliability of the stage **I** becomes **(1 – (1 - ri) ^mi)**. Thus, if **ri = 0.99** and **mi = 2**, then the stage reliability becomes **0.9999** which is almost equal to **1**. Which is much better than that of the previous case or we can say the reliability is little less than **1 - (1 - ri) ^mi** because of less reliability of switching circuits.



Multiple Devices Connected in Parallel in Each Stage

Maximize π Øi (mi) for 1 <= I <= n

Subject to:

$$\sum_{i=1}^{n} ci^{\square} mi^{\square} <= C$$

mi>= 1 and integer 1 <= i<= n

could not Here, **Øi (mi)** denotes the reliability of the stage **i**.

The reliability of the system can be given as follows:

Π Øi (mi) for 1 <= i<= n

If we increase the number of devices at any stage beyond the certain limit, then also only the cost will increase but the reliability increase.

4.1.9. TRAVELLING SALESMAN PROBLEM

- In the traveling salesman Problem, a salesman must visits n cities.
- We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from.
- There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost.
- We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).
- We can model the cities as a complete graph of n vertices, where each vertex represents a city.
- Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

- When $|S| > 1$, we define $C(S, 1) = \propto$ since the path cannot start and end at **1**.

- Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$$C(S,j)=\min C(S-\{j\},i)+d(i,j) \text{ where } i \in S \text{ and } i \neq j$$

$$c(S,j)=\min C(s-\{j\},i)+d(i,j)$$

$$\text{where } i \in S \text{ and } i \neq j$$

$$C(S,j)=\min C(S-\{j\},i)+d(i,j)$$

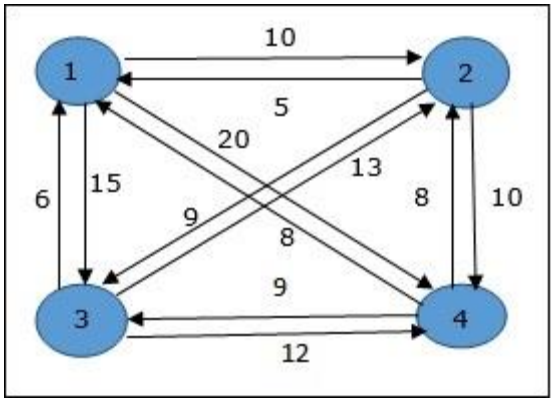
$$\text{where } i \in S \text{ and } i \neq j c(S,j)=\min C(s-\{j\},i)+d(i,j)$$

$$\text{where } i \in S \text{ and } i \neq j$$

Algorithm: Traveling-Salesman-Problem

C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**

4.1.10.FLOW SHOP SCHEDULING

- There are *n* machines and *m* jobs. Each job contains exactly *n* operations.
- The *i*-th operation of the job must be executed on the *i*-th machine.
- No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.
- Operations within one job must be performed in the specified order.

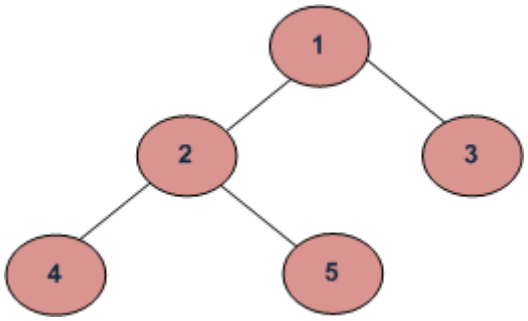
- The first operation gets executed on the first machine, then (as the first operation is finished) the second operation on the second machine, and so on until the *n*-th operation.
- Jobs can be executed in any order, however. Problem definition implies that this job order is exactly the same for each machine.
- The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan

- In an optimal schedule, job i precedes job j if $min\{p_{1i},p_{2j}\} < min\{p_{1j},p_{2i}\}$.
- Where as, p_{1i} is the processing time of job i on machine 1 and p_{2i} is the processing time of job i on machine 2.
- Similarly, p_{1j} and p_{2j} are processing times of job j on machine 1 and machine 2 respectively.
  The steps are summarized below for Johnson's algorithms:
  let, p_{1j}=processing time of job j on machine 1
  p_{2j}=processing time of job j on machine 2
  Johnson's Algorithm
  Step 1:Form set1 containing all the jobs with $p_{1j} < p_{2j}$
  Step 2:Form set2 containing all the jobs with $p_{1j} > p_{2j}$, the jobs with $p_{1j}=p_{2j}$ may be put in either set.
  Step 3: Form the sequence as follows:
  (i) The job in set1 go first in the sequence and they go in increasing order of p_{1j}(SPT)
  (ii) The jobs in set2 follow in decreasing order of p_{2j} (LPT). Ties are broken arbitrarily.
  This type schedule is referred as SPT(1)-LPT(2) schedule.

4.2. BASIC TRAVERSAL AND SEARCH TECHNIQUES

4.2.1.TREE TRAVERSAL TECHNIQUES

Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Inorder Traversal (Practice):

Algorithm Inorder(tree)
1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

- In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.
- To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal (Practice):

Algorithm Preorder(tree)
1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expression on of an expression tree.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal (Practice):

Algorithm Postorder(tree)
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

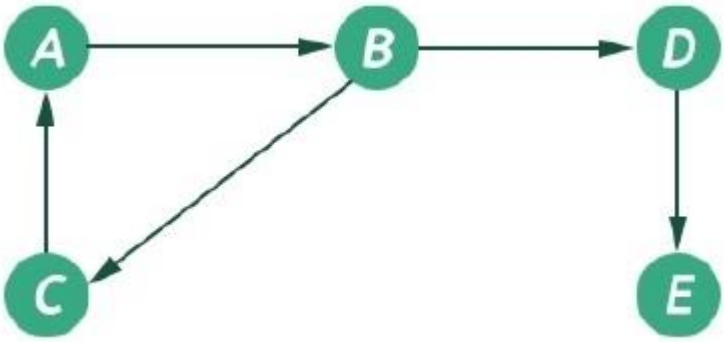
- **Uses of Postorder**
  Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details.

- Postorder traversal is also useful to get the postfix expression of an expression tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

4.2.2. TRAVERSAL TECHNIQUES FOR GRAPHS

- The graph is one non-linear data structure.
- That is consists of some nodes and their connected edges.
- The edges may be director or undirected.
- This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



- The graph has two types of traversal algorithms.
- These are called the Breadth First Search and Depth First Search.

Breadth First Search (BFS)

- The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph.
- In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.
- After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

Algorithm

```
bfs(vertices, start)
Input: The list of vertices, and the start vertex.
Output: Traverse all of the nodes, if the graph is connected.
Begin
   define an empty queue que
   at first mark all nodes status as unvisited
   add the start vertex into the que
   while que is not empty, do
      delete item from que and set to u
      display the vertex u
```

```
    for all vertices 1 adjacent with u, do
      if vertices[i] is unvisited, then
        mark vertices[i] as temporarily visited
        add v into the queue
      mark
    done
    mark u as completely visited
  done
End
```

Depth First Search (DFS)

- The Depth First Search (DFS) is a graph traversal algorithm.
- In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

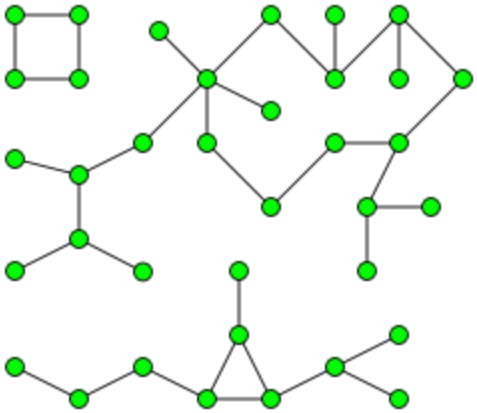
Algorithm

```
dfs(vertices, start)
Input: The list of all vertices, and the start node.
Output: Traverse all nodes in the graph.
Begin
  initially make the state to unvisited for all nodes
  push start into the stack
  while stack is not empty, do
    pop element from stack and set to u
    display the node u
    if u is not visited, then
      mark u as visited
      for all nodes i connected to u, do
        if ith vertex is unvisited, then
          push ith vertex into the stack
          mark ith vertex as visited
      done
  done
End
```

4.2.3. CONNECTED COMPONENTS

- In graph theory, a **component**, sometimes called a **connected component**, of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

- For example, the graph shown in the illustration has three components. A vertex with no incident edges is itself a component.
- A graph that is itself connected has exactly one component, consisting of the whole graph.



A graph with three components.

SPANNING TREE

The **spanning tree** of a graph (*G*) is a subset of *G* that covers all of its vertices using the minimum number of edges.

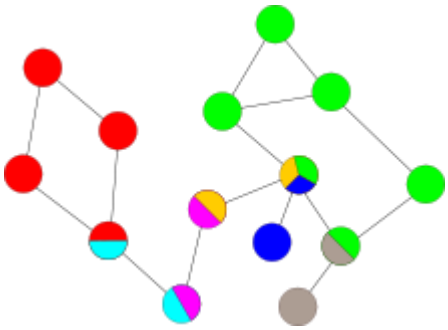
Some properties of a spanning tree can be deduced from this definition:

1. Since "a spanning tree covers all of the vertices", it cannot be disconnected.

2. A spanning tree cannot have any cycles *and* consist of (n-1)(*n*−1) edges (where n*n* is the number of vertices of the graph) because "it uses the minimum number of edges".

4.2.4. BICONNECTED COMPONENT AND DFS

- In graph theory, a **biconnected component** (sometimes known as a **2-connected component**) is a maximal biconnected subgraph.
  Any connected graph decomposes into a tree of biconnected components called the **block-cut tree** of the graph.
- The blocks are attached to each other at shared vertices called **cut vertices** or **articulation points**.
- Specifically, a **cut vertex** is any vertex whose removal increases the number of connected components.



Each color corresponds to a biconnected component. Multi-colored vertices are cut vertices, and thus belong to multiple biconnected components.

DESIGN AND ANALYSIS OF ALGORITHM-7MCE1C2

UNIT-5

5.1. BACK TRACKING

5.1.1. GENERAL METHOD

5.1.2. 8-QUEENS PROBLEM

5.1.3. SUM OF SUBSETS

5.1.4. GRAPH COLORING

5.1.5. HAMILTONIAN CYCLES

5.2. BRANCH AND BOUND

5.2.1. GENERAL METHOD

5.2.2. 0/1 KNAPSACK

5.1. BACK TRACKING

5.1.1. GENERAL METHOD:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, ... , x_n) where each x_i Є S, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x_1, ... , x_n). Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i"s must relate to each other.

5.1.2. 8-QUEENS PROBLEM:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples (x_1, ... , x_8), where x_i is the column of the i^{th} row where the i^{th} queen is placed.

The explicit constraints using this formulation are S_i = {1, 2, 3, 4, 5, 6, 7, 8}, $1 \leq i \leq 8$.Therefore the solution space consists of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i"s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 8^8 tuples to 8! Tuples.

The promising function must check whether two queens are in the same column or diagonal:

Suppose two queens are placed at positions (i, j) and (k, l) Then:

- Column Conflicts: Two queens conflict if their x_i values areidentical.
- Diag 45 conflict: Two queens i and j are on the same 45^0 diagonal if:

  $$i - j = k - l.$$

  This implies, j - l = i - k
- Diag 135 conflict:

  $$i + j = k + l.$$

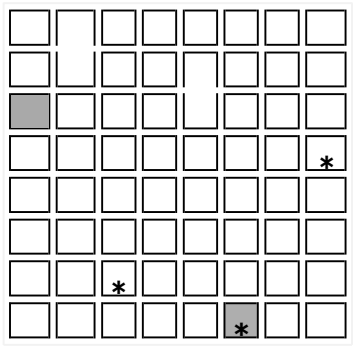
  This implies, j - l = k - i

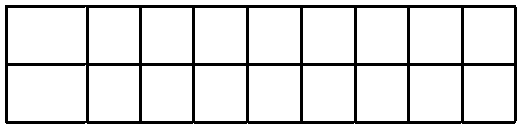
Therefore, two queens lie on the same diagonal if and only if:

□j - l□= □i - k □□

□

Where, j be the column of object in row i for the i^{th} queen and l be the column of object in row „k“ for the k^{th} queen.



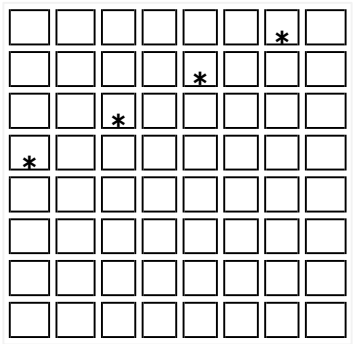


Step 1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.



Step 3:

If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

Program for N-Queens Problem:

```
# include <stdio.h>
#  include  <conio.h>
# include <stdlib.h>

int x[10] = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5};

place (int k)
{
        int i;
```

```
        for (i=1; i < k; i++)
        {
                if ((x [i] == x [k]) || (abs (x [i] - x [k]) == abs (i - k)))
                return (0);
        }
        return  (1);
}
nqueen  (int  n)
{
        int m, k, i = 0;
        x [1] = 0;
        k = 1;
        while (k > 0)
        {
                x [k] = x [k] + 1;
                while ((x [k] <= n) && (!place (k)))
                        x [k] = x [k] +1;
 ________       if(x  [k]  <=  n)
                {
                        if  (k  ==  n)
                        {
                                i++;
                                printf  ("\ncombination;  %d\n",i);
                                for (m=1;m<=n; m++)
                                printf("row = %3d\t column=%3d\n", m, x[m]);
                                getch();
                        }
                        else
                        {
                                k++;
                                x   [k]=0;
                        }
                }

                else
                        k--;
        }
        return (0);


}
  main   ()
  {
        int n;
        clrscr ();
        printf ("enter value for N: ");
        scanf ("%d", &n);
        nqueen (n);
   }
```

Output:

Enter the value for N: 4

Combination: 1		Combination: 2
Row = 1	column = 2	3
Row = 2	column = 4	1
Row = 3	column = 1	4
Row = 4	column = 3	2

For N = 8, there will be 92 combinations.

5.1.3.SUM OF SUBSETS:

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of w_i whose sums are „m".

All solutions are k-tuples, $1 \le k \le n$.

Explicit constraints:

$x_i \in \{j \mid j$ is an integer and $1 \le j \le n\}$.

Implicit constraints:

No two x_i can be the same.

The sum of the corresponding w_i"s be m.

$x_i < x_{i+1}$, $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

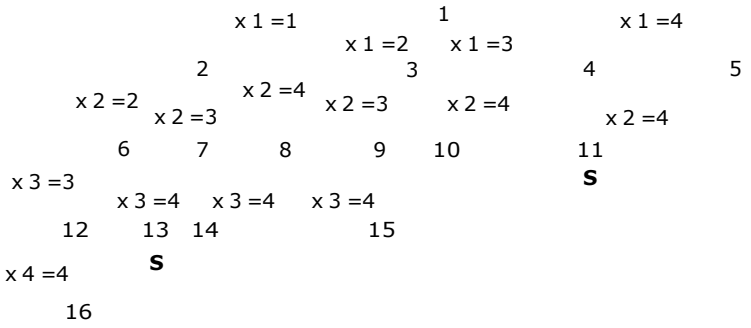
A better formulation of the problem is where the solution subset is represented by an n-tuple (x_1, ... , x_n) such that $x_i \in \{0, 1\}$.

The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1).

For both the above formulations, the solution space is 2^n distinct tuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11, 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.



The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left mot sub-tree defines all subsets containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, and so on.

5.1.4.GRAPH COLORING

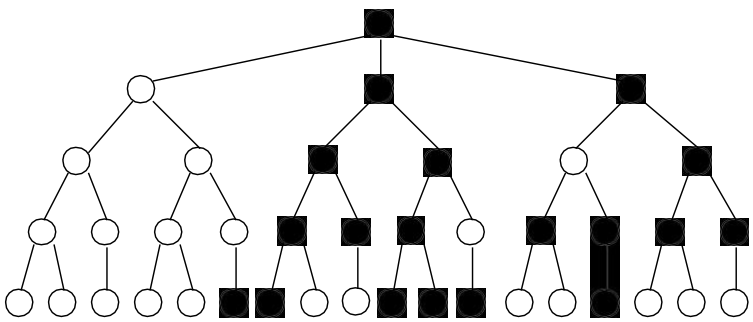
Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

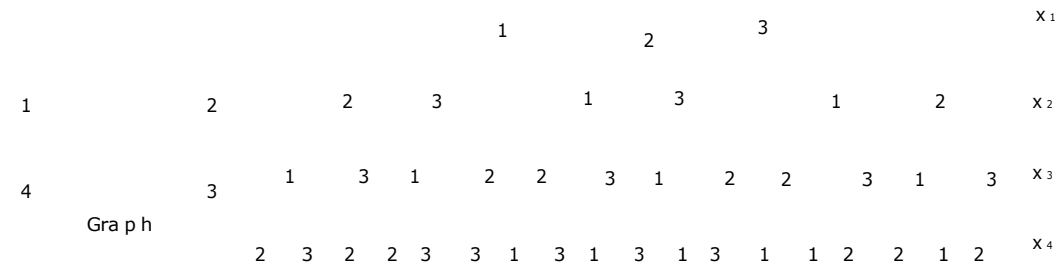


```
Algorithm mcoloring (k)
{
        repeat
        {
                NextValue  (k);
                If (x [k] = 0)  then return;
                If (k =  n) then
                        write (x [1: n]);
                        else  mcoloring  (k+1);
                } until (false);
        }
```

```
Algorithm NextValue (k)
{
        repeat
        {
                x [k]: = (x [k] +1)  mod (m+1)
                If (x [k] = 0)  then return;
                for j := 1 to n do
                {
                        if ((G [k, j] □0) and (x [k] = x [j]))
                        then break;
                }
                if (j = n+1)  then return;
        } until  (false);
}
```

Example:

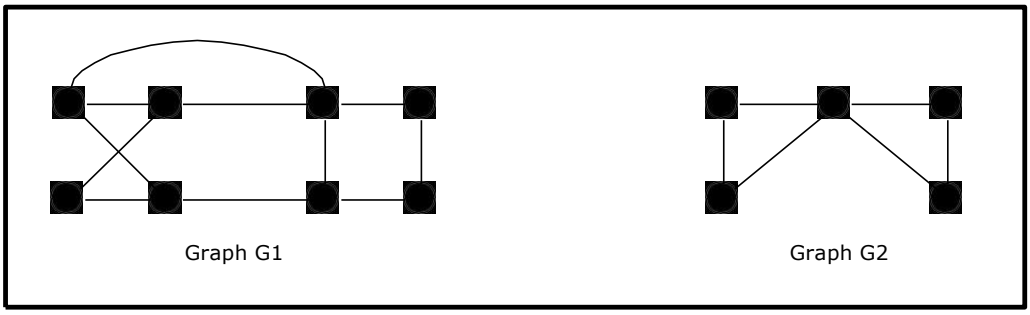
Color the graph given below with minimum number of colors by backtracking using state space tree.



A 4- node graph and all possible 3-colorings

5.1.5. HAMILTONIAN CYCLES:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order $v_1, v_2, \ldots\ldots\ldots\ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.



Two graphs to illustrate Hamiltonian cycle

The backtracking solution vector $(x_1, \ldots\ldots\ldots x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If k = 1, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to k_{x-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1: n, 1: n], then setting x[2: n] to zero and x[1] to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

Algorithm NextValue (k)

```
// x [1: k-1] is a path of k - 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
// assigned to x [k]. After execution, x[k] is assigned to the next highest numbered vertex
// which does not already appear in x [1 : k - 1] and is connected by an edge to x [k - 1].
// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
{
        repeat
        {
                x [k] := (x [k] +1)  mod (n+1);          // Next vertex.
                If (x [k] = 0) then return;
                If (G [x [k - 1], x [k]] □0) then
                {                                        // Is there an edge?
                        for j := 1 to k - 1 do if (x [j] = x [k]) then break;
                                                         // check for distinctness.
                        If (j =  k) then                 // If true, then the vertex is distinct.
                        If ((k < n) or ((k = n) and G [x [n], x [1]] □0))
                        then return;
                }
        } until  (false);
}
```

Algorithm Hamiltonian (k)

```
// This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles  begin
// at node 1.
{
        repeat
        {
                NextValue (k);
        if (x [k] = 0) then return;
                if (k = n) then write (x [1: n]);
                else Hamiltonian (k + 1) } until (false);
}
```

5.2. BRANCH AND BOUND

5.2.1. THE GENERAL METHOD

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.

2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node other live node becomes the E-node

Branch and Bound is the generalisation of both graph search strate search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

- A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

5.2.2. 0/1 KNAPSACK:

Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

$$\sum_{1 \le i \le n} w_i x_i \le m \qquad \text{and} \qquad \sum_{1 \le i \le n} p_i x_i \text{ is maximized.}$$

The x_i"s constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i"s.

Bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far, than that live node can be killed.

If at node Z the values of x_i, $1 \le i \le k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements $x_i = 0$ or 1.

The i thth item contributes the weight xi.wixi.wi to the total weight in the knapsack an profit xi.pixi.pi to the total profit.
Hence, the objective of this algorithm is to

maximize∑n=1n(xi.pi)maximize∑n=1n(xi.pi)

subject to constraint,

∑n=1n(xi.wi)⩽W∑n=1n(xi.wi)⩽W

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

∑n=1n(xi.wi)=W∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, x is an array to store the fraction of items. Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

for i = 1 to n

do x[i] = 0

weight = 0

for i = 1 to n

if weight + w[i] ≤ W then

x[i] = 1

weight = weight + w[i]

else

x[i] = (W - weight) / w[i]

weight = W

break

return x