CMR
INSTITUTE OF
TECHNOLOGY

Internal Assessment Test 2 – November 2019

Sub:	Design and Analysis of Algorithms							Code:	18MCA33
Date:	*18-11-2019*	*Duration:*	*90 mins*	*Max Marks:*	*50*	*Sem:*	IIIA	*Branch:*	*MCA*

Dr. Vakula Rani

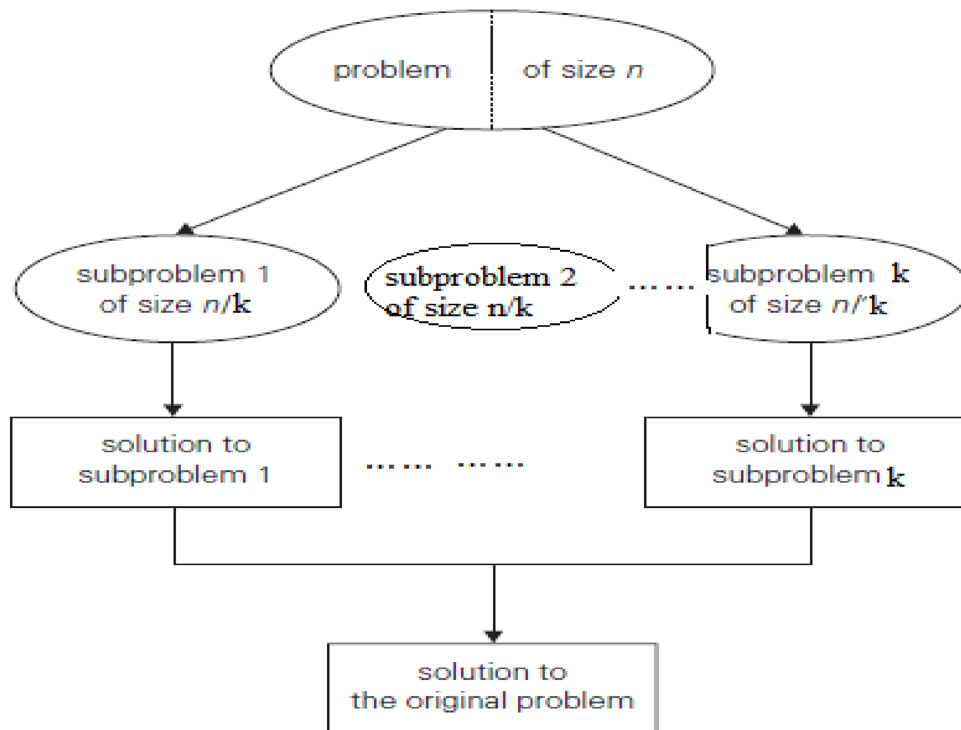
Answer any five of the following **10 Marks**

Q1 Describe the general method for divide and conquer. State the Masters Theorem

Sol : Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The general method is shown diagrammatically as below:



The pseudo code for the same is given by:

```
Algorithm DivideAndConquer(P,S)
    Divide the problem P into k subproblems P1,P2…Pk
    For each i in [1..k]
      / /solve each of the problem recursively by using the same technique
      Si← DivideAndConquer(Pi)
    Combine the solutions to the subproblems P1,P2… Pk i.e. S1, S2…, Sk to form the solution S
  Return S
```

A recurrence is a recursive description of a function, or in other words, a description of a function in terms of itself.

Divide and Conquer Recurrences (Recursion Trees)
Many divide and conquer algorithms give us running-time recurrences of the form

$$T(n) = a\,T(n/b) + f(n)$$

where a and b are constants and f (n)= n^d is some other function.

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$
$$T(1) = c$$

where $a \ge 1$, $b \ge 2$, $c>0$. If f(n) is $\Theta(n^d)$ where $d \ge 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

(2) Explain and write algorithm for the brute force string matching process and Apply it to search for ABABC using the above algorithm in the text : BAABABABCCA.

Solution

Given a string of n characters called the text and a string of m characters ($m \le n$) called the pattern, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that
$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$\begin{array}{ccccccccccc} t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & \\ & & p_0 & \ldots & p_j & \ldots & p_{m-1} & & \text{pattern } P & \end{array}$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted. align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

```
Algorithm Brute Force string match (T[0..,n-1], P[0..m-1])
// Input: An array T [0..n-1] of n chars, text
//            An array P [0..m-1] of m chars , a pattern.
// Output: The position of the first character in the text that starts the first
//            matching substring if the search is successful and -1 otherwise.

  for i ⟵ 0 to n-m do
         j ⟵ 0
         while j < m and P[j] = T[i+j] do
                 j ⟵ j+1
         if j = m return i
  return -1
```

Example

Text String = { BAABABABCCA **}**

Pattern String ={ ABABC **}**

B	A	A	B	A	B	A	B	C	C	A	
A	B	A	B	C							
	A	B	A	B	C						
		A	B	A	B	C					
			A	B	A	B	C				
				A	B	A	B	C			

String is matched return the starting Index -4

B	A	A	B	A	B	A	B	C	C	A	
				A	B	A	B	C			

The time complexity would be analyzed by finding the number of times the basic operation j=j+1 is executed.
The inner loop will be executed a maximum of m times (j=0 to m-1).
Therefore

T(n)= $\sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1 = \sum_{i=0}^{n-m} m$ = (n-m)*m = θ(mn).

Where m is the length of pattern and n is the length of text.

Q3. Write an algorithm to multiply two large integers using divide and conquer and analyze its efficiency. Use the divide and conquer strategy to multiply 6721 and 3032.

T he conventional algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.)
By using divide-and-conquer method , it would be possible to design an algorithm with fewer than n^2 digit multiplications,

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$
\begin{aligned}
c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\
&= c_2 10^n + c_1 10^{n/2} + c_0,
\end{aligned}
$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Solving it by backward substitutions for $n = 2^k$ yields

$$
\begin{aligned}
M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\
&= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k.
\end{aligned}
$$

Example :

To demonstrate the basic idea of the algorithm, let us start with a case of

Four-digit integers –6721 and 3032 . These numbers can be represented as follows:

X= 3421 = 67 * 10^2 + 21 Let A = 67 ; B = 21

Y=3032 = 30 * 10^2 + 32 Let C = 30; D = 32

Now let us multiply them:

X* Y = AC * 10^4 + [AC + (A – B)* (D – C) + BD] * 10^2 + BD

= 67 * 60 * 10^4 +[(67 * 60) +(67 – 21) * (32 -30) * 10^2 + (21 * 32)

= 4020 * 10000 + [4020 +92 + 672] * 100 + 672

= 40200000 + 478400 +672

= 40679072

Algorithm multi(X, Y, n)

```
//Input : X & Y two long integers; n – no.of digits of X
// Output : Product of two long integers
Begin
   If ( n == 1)
       Return( X * Y)
   Else
        A= Left   n/2  bits of  X
        B = Right   n/2  bits of   X
        C= Left   n/2  bits of   Y
        D = Right   n/2  bits of   Y
        m1 = multi( A,C)
        m2 = multi( A-B, D-C)
        m3  = multi( B,D)
        return (m1 * 10^n  +( m1 + m2  +m3) * 10^(n/2) +m^3
 End
```

Analysis

Analysis

since multiplication of n-digit number required three multiplications of n/2 digit no's the no of recurrance for th no of multiplications T(n) will be.

$$T(n) = 3T(n/2) + Cn \quad \text{(ignore c)}$$

solve using master's theorem

$$T(n) = aT(n/2) + f(n)$$

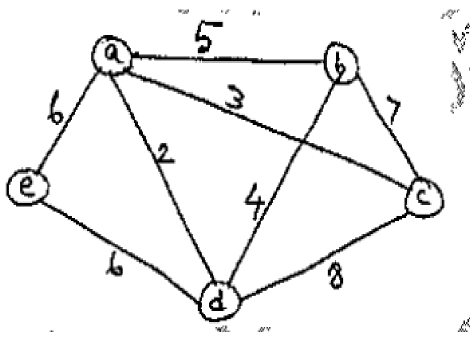
here $a>3$ $b=2$ $d=1$.

$a > b^d$.

$3 > 2$.

Hence T(n) = $\Theta(n^{\log_2 3})$ = $\Theta(n^{1.58})$. This time complexity is much better than the brute force multiplication which takes $\Theta(n^2)$ time for n digit multiplication.

Q4. Explain and design Prim"s algorithm and apply it for the given graph to find minimum cost spanning tree .



Sol : Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.
Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n − 1, where n is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.
Here is pseudocode of this algorithm.

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
  find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
  such that v is in V_T and u is in $V - V_T$
  $V_T \leftarrow V_T \cup \{u^*\}$
  $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Min cost = 2+3+4+6 = 15

Q5.a) Explain the greedy method and write the general algorithm for the method

Sol:

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step—and this is the central point of this technique—the choice made must be:

- Feasible, i.e., it has to satisfy the problem's constraints
- Locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
- Irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm

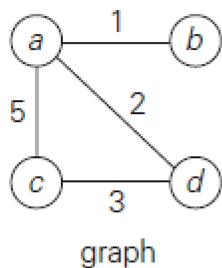
The function Select selects an input from $a[\,]$ and removes it. The selected input's value is assigned to x. Feasible is a Boolean-valued function that determines whether x can be included into the solution vector. The function Union combines x with the solution and updates the objective function. The

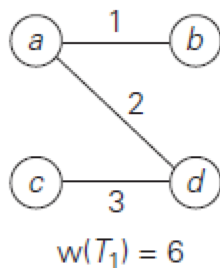
```
1    Algorithm Greedy(a, n)
2    // a[1 : n] contains the n inputs.
3    {
4        solution := ∅; // Initialize the solution.
5        for i := 1 to n do
6        {
7            x := Select(a);
8            if Feasible(solution, x) then
9                solution := Union(solution, x);
10       }
11       return solution;
12   }
```

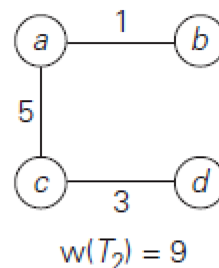
Q5.b) Define Minimum spanning tree and mention its applications

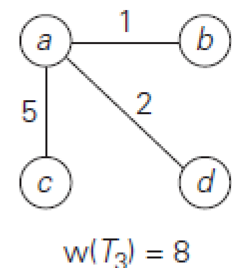
Sol:

A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.









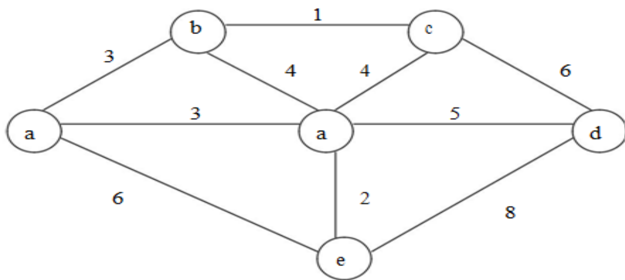
Graph and its spanning trees, with T1 being the minimum spanning tree.

Spanning trees have many applications. For example, they can be used to obtain an independent set of circuit equations for an electric network. First, a spanning tree for the electric network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to

the spanning tree creates a cycle. Kirchoff's second law is used on each cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B that is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it can be shown that the cycles obtained by introducing the edges of B one at a time into the resulting spanning tree form a cycle basis, and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis.

Another application of spanning trees arises from the property that a spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected. (A minimal subgraph is one with the fewest number of edges.) Any connected graph with n vertices must have at least $n-1$ edges and all connected graphs with $n-1$ edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is $n-1$. The spanning trees of G represent all feasible choices.

Q6. Find the minimum cost spanning tree for the given graph below by applying Kruskal's algorithm. Write the algorithm and compute minimum cost .



Kruskal's algorithm is used for solving the minimal spanning tree problem. ***Spanning tree*** of an undirected connected graph is its connected acyclic subgraph(tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.
Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages

Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

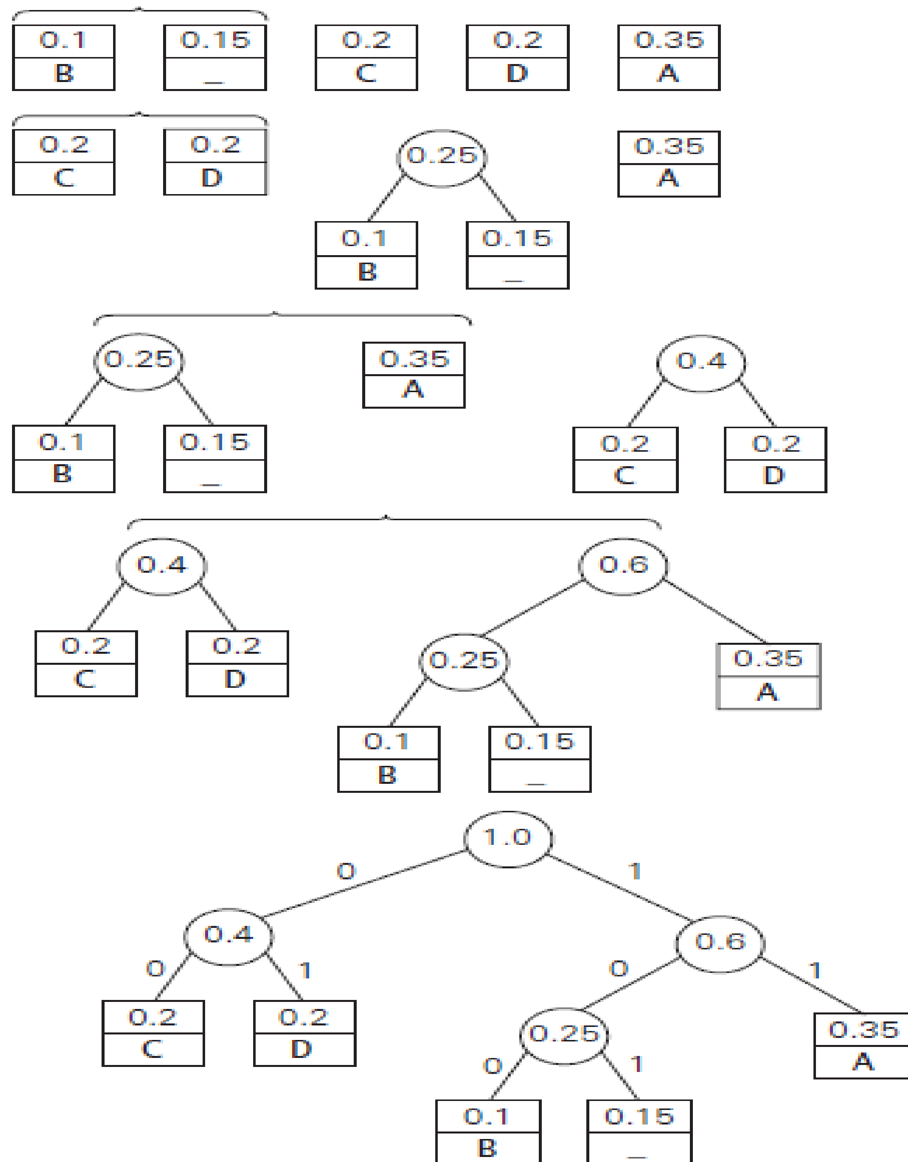
ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
  $k \leftarrow k + 1$
  if $E_T \cup \{e_{i_k}\}$ is acyclic
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

Q 7 Write and analyze the Huffman coding algorithm. Show the tree and code for the set of symbols given below along with their relative frequency.

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

Huffman's algorithm
Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)
Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.
A tree constructed by the above algorithm is called a ***Huffman tree***. It defines—in the manner described above—a ***Huffman code***.

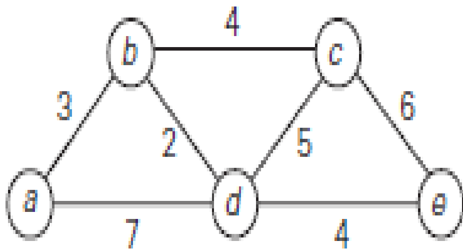


The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, **DAD** is encoded as **011101**, and
10011011011101 is decoded as **BAD_AD.**

Q8. Explain the Dijkstra's single source shortest path algorithms and analyze its time complexity. Source vertex 'a'.



Dijkstra's algorithm is an algorithm for solving the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph with non negative edges, find shortest paths to all its other vertices. Some of the applications of the problem are transportation planning, packet routing in communication networks finding shortest paths in social networks, etc. First, it finds the shortest path from the source. to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration starts, the algorithm has already identified the shortest paths to i − 1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree Ti of the given graph. The set of vertices adjacent to the vertices in T called "fringe vertices"; are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v and the length dv of the shortest path from the source to v and then selects the vertex with the smallest such d value. d indicates the length of the shortest path from the source to that vertex till that point. We also associate a value p with each vertex which indicates the name of the next-to-last vertex on such a path, . After we have identified a vertex u* to be added to the tree, we need to perform two operations.
The psuedocode for Dijkstra's is as given below:

- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

```
ALGORITHM Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
    //        and its vertex s
    //Output: The length d_v of a shortest path from s to v
    //         and its penultimate vertex p_v for every vertex v in V
    Initialize(Q)  //initialize priority queue to empty
    for every vertex v in V
        d_v ← ∞;   p_v ← null
        Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
    d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
    V_T ← ∅
    for i ← 0 to |V| − 1 do
        u* ← DeleteMin(Q)  //delete the minimum priority element
        V_T ← V_T ∪ {u*}
        for every vertex u in V − V_T that is adjacent to u* do
            if d_u* + w(u*, u) < d_u
                d_u ← d_u* + w(u*, u);   p_u ← u*
                Decrease(Q, u, d_u)
```

Analysis:

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.

Graph represented by adjacency matrix and priority queue by array:

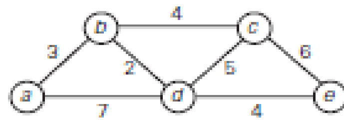
In loop for initialization takes time |V| since the insertion into the queue would just involve appending the vertices at the end(since it is an array implementation). For the second loop, the loop runs |V| times. Each time the DeleteMin operation would take a maximum of θ(|V|) time since it would involve finding the vertex in the array with min d value, for a total time of |V|2. The for loop (for iupdating the neighbor vetices) would run |V| times again. However the Decrease would take θ(1) time because the index of the vertex would be known. Thus the total time complexity is θ(|V|2).

Graph represented by adjacency list and priority queue by binary heap:

All heap operations take θ(lg|V|) time. Thus the first loop runs |V| times and each time the Insert would take θ(lg|V|) time. The second loop runs |V| times and the DeleteMin would again take lg|V| time. Thus the total number of time DecreaseMin would run across all iterations is θ(Vlg|V|). In the second loop the basic operation is Decrease(Q,u,du) whoch is run the maximum number of times. Across all iterations using adjacency list, since for each vertex Decrease is called for a maximum of all its adjacent vertices, the number of times Decrease is invoked |E| times. For each time it is onvoked , it takes O(lg|V|) time to execute. Thus the total time complexity is θ((|E|+|V|)lg|V|).

Graph represented by adjacency list and priority queue by fibonacci heap:

The time taken in this case θ(|E|+|V|lg|V|).



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

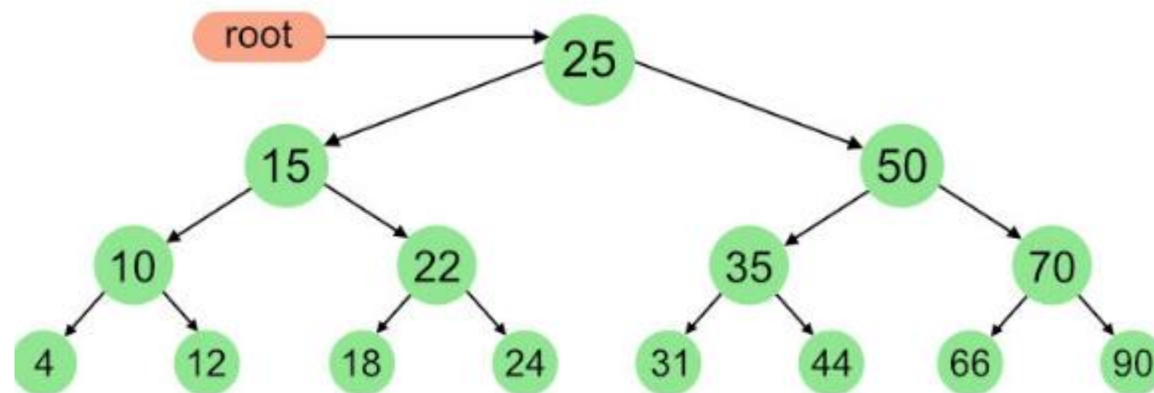
The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b : $a - b$ of length 3
from a to d : $a - b - d$ of length 5
from a to c : $a - b - c$ of length 7
from a to e : $a - b - d - e$ of length 9

Q9. Define Binary Tree .Write algorithms for inorder, preorder and postorder traversal of a binary tree. Give examples for all three .

Sol:

Def : A binary tree is a special type of tree in which every node or vertex has at most two children that is either no child node or one child node or two child nodes



Tree traversal techniques

- Inoder Traversal
- Pre-order Traversal
- Post orderTraversal

InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Pre-order

- If the tree T consists on only the root r, then r is the preorder traversal of T.
- If $T_1, T_2, \ldots, T_n$ are the subtrees rooted at r from left to right, then the preorder traversal comprises of visiting r, followed by T_1 in pre-order, then T_2 in preorder and lastly T_n in pre-order.

- **InOrder:**
  - If the tree T consists on only the root r, then r is the inorder traversal of T.
  - If $T_1, T_2, \ldots, T_n$ are the subtrees rooted at r from left to right, then the preorder traversal comprises of visiting T_1 in inorder, followed by r in inorder, then T_2 in inorder and lastly T_n in inorder.
- **PostOrder:**
  - If the tree T consists on only the root r, then r is the postorder traversal of T.
  - If $T_1, T_2, \ldots, T_n$ are the subtrees rooted at r from left to right, then the preorder traversal comprises of visiting T_1 in postorder, followed

Algorithm inorder(T)

```
Begin
      if T not empty
           inorder(TL)
           print root(T)
           inorder(TR)
End
```

Algorithm preorder(T)

```
Begin
       if T not empty
            print root(T)
            preorder(TL)
            preorder(TR)
End
```

Algorithm postorder(T)

```
Begin
   if T not empty
           postorder(TL)
           postorder(TR)
           print root(T)
End
```

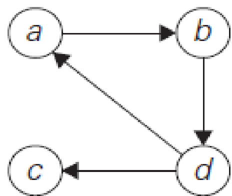
Q10. Explain Warshall's algorithm for the finding the transitive closure of a graph.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0. \end{pmatrix}$$

ALGORITHM *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).