Fourth Semester B.E. Degree Examination, CBCS - Dec 2017 / Jan 2018

Design and Analysis of Algorithms

Time: 3 hrs. Max. Marks: 80

Note : *Answer any FIVE full questions, selecting ONE full question from each module.*

Module - 1

1. a. **Define Algorithm. Discuss the criteria of an algorithm with an example. (06 Marks)**

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

There are many criteria upon which we can judge a program, for instance:

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation which describes how to use it and how it works?
4. Are procedures created in such a way that they perform logical sub-functions?
5. Is the code readable?

Consider the examples below:

```
x:=x+1;
```

We assume that the statement x:=x+1 is not contained within any loop either explicit or implicit. Then its frequency count is one.

```
for i:=1 to n do
  x:=x+1;
```

Now, the same statement will be executed n times.

```
for i:=1 to n do
  for i:=1 to n do
  x:=x+1;
```

It will be executed (n*n) times now.

b. **Prove that if $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ (06 Marks)**

Ans. The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a2 \leq 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c1 and some nonnegative integer n1 such that $t_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$.

Similarly, since $t_2(n) \in O(g_2(n))$, $t_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$.

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n=\}$ so that we can use both inequalities. Adding those yields the following:

$t_1(n)+t_2(n) \leq c_1 g_1(n) + c_2 g_2(n) \leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \leq c_3 2\max\{g_1(n), g_2(n)\}$.

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.

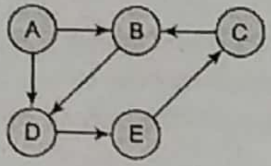
c. Explain the two common ways to represent a graph with an example. **(04 Marks)**

Ans. There are several ways to represent graphs, each with its advantages and disadvantages. Some situations, or algorithms that we want to run with graphs as input, call for one representation, and others call for a different representation.

Adjacency matrices

1. This kind of representation is achieved by using the adjacency matrix.
2. Adjacency matrix is used to represent which nodes are adjacent to each other i.e is there a edge connecting two nodes on a graph.
3. The adjacency matrix is of dimension n×n where n is number of nodes.
4. If an edge is present from i to j then Aij =1 else the value will be set to 0. If it is a weighted graph, we replace 1 with the weight of that edge.

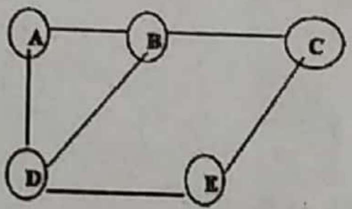
An example of adjacency matrix is shown below

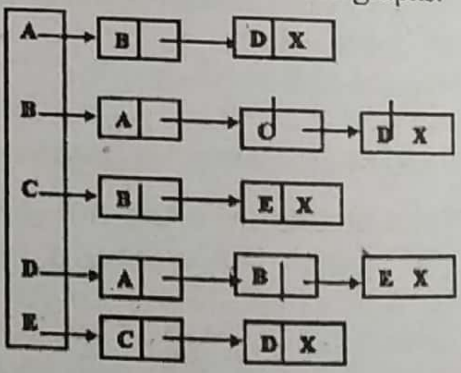


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 0 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

Linked representation

1. It is implemented using a adjacency list
2. It contains a linked list of all nodes in the graph.
3. Moreover, each node is in turn linked to its own list that contains names of all other nodes that are adjacent to it.
4. Such kind of representation is often used for small to moderate sized graphs.





OR

2. a. Consider the following algorithm

```
ALGORITHM GUESS(A[ ][ ])
For i ← 0 to n-1
  For j ← 0 to i
        A[i][j] ← 0
```

i. What does the algorithm computer? **ii. What is basic operation?**
iii. What is the efficiency of this algorithm? **(03 Marks)**

Ans. i) Below and including diagonal element assigned with zero
ii) Assignment is basic operation
iii) O(n) is efficiency of this algorithm

b. List and explain important problem types that are solved by computer (07 Marks)

Ans. Important Problem types.

a. Sorting
b. Searching
c. String processing
d. Graph problems
e. Combinatorial problems
f. Geometric problems
g. Numerical problems

Sorting algorithm is an algorithm that puts elements of a list in a certain order. The mostused orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order)
2. The output is a permutation, or reordering, of the input.

Searching: In computer science, a search algorithm, broadly speaking, is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph. Searching algorithms are closely related to the concept of dictionaries.

String searching algorithms are important in all sorts of applications that we meet every day. In text editors, we might want to search through a very large document (say, a million characters) for the occurrence of a given string (maybe dozens of characters). In text retrieval tools, we might potentially want to search through thousands of such documents (though normally these files would be indexed, making this unnecessary).

Geometric algorithms deal with geometric objects such as points, lines, and polygons. Ancient Greeks were very much interested in developing procedures for solving a variety of geometric problems including problems of constructing simple geometric shapes-triangles.

Numerical problems, another large area of applications are problems that involve mathematical objects of continuous nature: solving equations and system of equation, computing definite integrals, evaluating functions and so on. The majority of such mathematical problems can be solved only approximately.

c. Design an algorithm for checking whether all elements in a given array are distinct or not derive its worst complexity. (06 Marks)

Ans.
```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for i ←0 to n − 2 do
for j ←i + 1 to n − 1 do
if A[i]= A[j ] return false
return true
```

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Applying mathematical induction we get

$$C_{worst}(n) = \frac{1}{2}n^2 \in \text{theta}(n^2)$$

Module - 2

3. a. Explain divide and conquer technique. Write a recursive algorithm for finding maximum and minimum element from a list. **(08 Marks)**

Ans. 1. A problem is divided into several subproblems of the same type, ideally of about equal size.

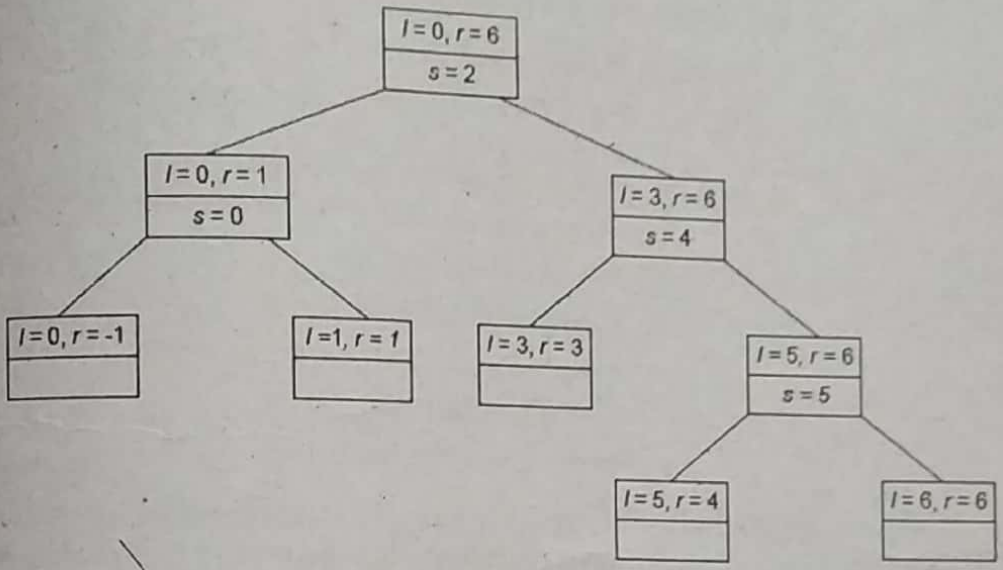
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).

3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

```
Algorithm maximum(int array[], int index, int len)
{
if(index >= len-2)
      {
            if(array[index] > array[index + 1])
         return array[index];
            else
                  return array[index + 1];
      }
max = maximum(array, index + 1, len);
if(array[index] > max)
    return array[index];
        else
                return max;
}
Algorithm minimum(int array[], int index, int len)
{
if(index >= len-2)
      {
if(array[index] < array[index + 1])
                  return array[index];
else
return array[index + 1];
      }
min = minimum(array, index + 1, len);
if(array[index] < min)
return array[index];
else
                return min;
}
```

b. **Apply quick sort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made.** **(08 Marks)**

Ans.



0	1	2	3	4	5	6
E	$\overset{i}{X}$	A	M	P	L	$\overset{j}{E}$
E	E	$\overset{j}{A}$	$\overset{i}{M}$	P	L	X
A	E	E	M	P	L	X
A	$\overset{i\,j}{E}$					
$\overset{j}{A}$	$\overset{i}{E}$					
A	E					
	E					
			M	$\overset{i}{P}$	L	$\overset{j}{X}$
			M	$\overset{i}{P}$	$\overset{j}{L}$	X
			M	$\overset{i}{L}$	$\overset{j}{P}$	X
			M	$\overset{j}{L}$	$\overset{i}{P}$	X
			L	M	P	X
			L			
					P	$\overset{i\,j}{X}$
					$\overset{j}{P}$	$\overset{i}{X}$
					P	X
						X

OR

4. a. **Discuss strassen's matrix multiplication and der ve its time complexity. (08 marks)**

Ans. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2 × 2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm. This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions. These numbers should not lead us to multiplying 2 × 2 matrices by Strassen's algorithm. Its importance stems from its asymptotic superiority as matrix order n goes to infinity. Let A and B be two n × n matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four n/2 × n/2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].$$

Let us evaluate the asymptotic efficiency of this algorithm. If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n × n matrices (where n is a power of 2), we get the following recurrence relation for it:

$M(n) = 7M(n/2)$ for $n > 1$, $M(1) = 1$.

Since $n = 2^k$,

$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \ldots$

$= 7^iM(2^{k-i}) \ldots = 7^kM(2^{k-k}) = 7^k.$

Since $k = \log_2 n$,

$M(n) = 7^{\log_2 n} = n^{\log 2\ 7} \approx n^{2.807}$, which is smaller than n3 required by the brute-force algorithm.

b. Design merge sort algorithm and discuss its best-case, average case and worst case efficiency (08 Marks)

Ans.
```
ALGORITHM    Mergesort(A[0..n - 1])
//Sorts array A[0..n – 1] by recursive mergesort
//Input: An array A[0..n – 1] of orderable elements
//Output: Array A[0..n – 1] sorted in non dèer easing order
if n > 1
copy A[0.. |n/2| – 1] to B[0.. |n/2| – 1]
copy A[|n/2|..n − 1] to C[0..|n/2| − 1]
Mergesort(B[0..|n/2| − 1])
Mergesort(C[0.. |n/2| −1])
Merge(B, C, A)
```

```
ALGORITHM   Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
//Merges Two sorted arrays into one sorted array
//Input: Arrays B[0..p - 1] and C[0..q - 1] both sorted
//Output: Sorted array A[0..p + q — 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
if B[i] ≤ [j]
A[k] ← B[i]; i ← i + 1
else A[k] ← C[i]; j ← j + 1
if i = p
copy C[j..q − 1]to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```

Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is $C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$, $C(1) = 0$.

Let us analyze Cmerge(n), the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case,

$C_{merge}(n) = n - 1$, and we have the recurrence $C_{worst}(n) = 2C_{worst}(n/2) + n - 1$ for $n > 1$, $C_{worst}(1) = 0$. Hence, according to the Master Theorem, Cworst(n) □ Theta(n log n). In fact, it is easy to find the exact solution to the worst-case recurrence for $n = 2^k$:

$C_{worst}(n) = n \log2 n - n + 1$.

Module - 3

5. a. **Solve the greedy knapsack problem where m=10, n=4, p=(40, 42, 25, 12), w=(4, 7, 5, 3).** **(06 Marks)**

Ans.

object	1	2	3	4
Profit	40	42	25	12
Weight	4	7	5	3
pi//wi	10	6	5	4

Object	1	2	3	4
pi/wi	10	6	5	4

object	1	2
Weight	4	6/7*7=6

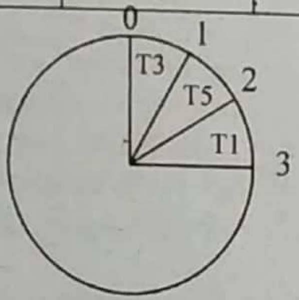
Cummulative weight after including each object	4	10
Profit pi	40	6/7*42=36
Cummulative profit after including each object	40	76

Total Profit =76, items selected=(1,6/7, 0, 0)

b. What is job sequencing with deadlines problem? Let n=5, profits [10, 3, 33, 11, 40] and deadlines [3, 1, 1, 2, 2] respectively. Find the optimal solution using greedy algorithm **(05 Marks)**

Ans.

Task	Deadline	profit
t5	2	40
t3	1	33
t4	2	11
t1	3	10
t2	1	3



Sequence={T3, T5, T1} Profit= 83

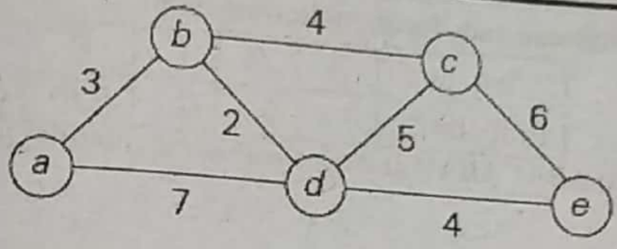
c. Define minimum cost spanning tree (MST). Write prim's algorithm to construct minimum cost spanning tree. **(05 Marks)**

Ans. The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

```
ALGORITHM   Prim(G)
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = (V, E)
//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
$E_T \leftarrow \theta$
for i ← 1 to |V| – 1 do
find a mini mum-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 $E_T$
```

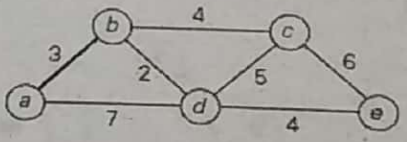
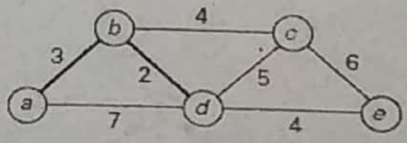
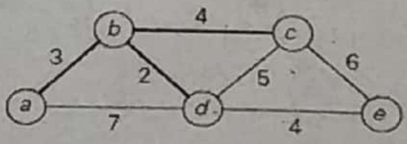
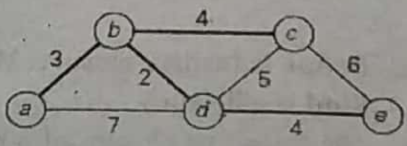
OR

6. a. Design Dijkstra's algorithm and apply the same to find the single source shortest path for graph taking vertex 'a' as source of Fig. Q6 (a). **(08 Marks)**



Ans. 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 dv of a shortest path from s to v
//          and its penultimate vertex pv for every vertex v in V
Inhialize(Q)   //initialize priority queue to empty
for every vertex v in V
    $d_v \leftarrow 00$;   $p_v \leftarrow$ null
    Insert(Q, v, $d_v$)   //initialize vertex priority in the priority queue
ds ← 0;   Decrease (Q, s, ds)   //update priority of s with $d_s$
for i ← 0 to |V| −1 do
    u* ← DeleteMin(Q)   //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in V — $V_T$ that is adjacent to u* do
        if $d_u^* + w(u^*, u) < d_u$
            $d_u \leftarrow d_u^* + w(u^*, u)$;   $p_u \leftarrow u^*$
            Decrease(Q, u, $d_u$)
```

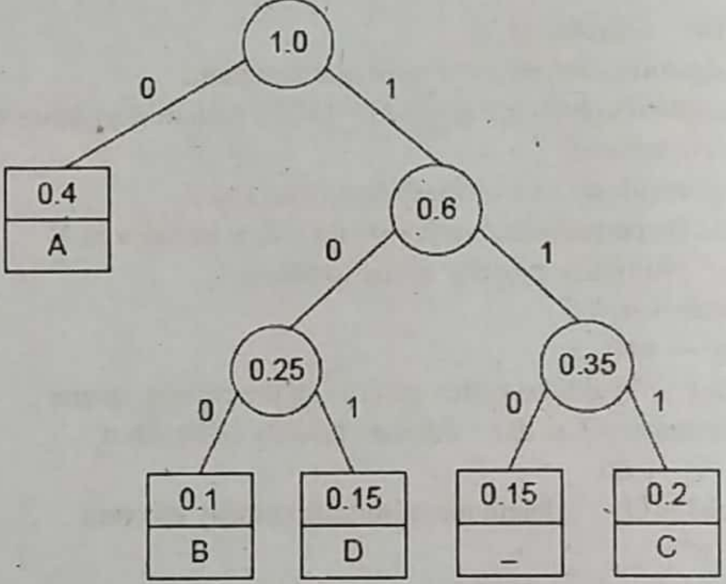
Tree vertices	Remaining vertices	Illustration
a(−, 0)	b(a, 3) c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) d(b, 3 + 2) e(−, ∞)	
d(b, 5)	c(b, 7) e(d, 5 + 4)	
c(b, 7)	e(d, 9)	
e(d, 9)		

b. Construct a Huffman code for the following data:

Character	A	B	C	D	-
Probability	0.4	0.1	0.2	0.15	0.15

Encode the text ABACABAD and decode the text 100010111001010, using the above code. **(04 Marks)**

Ans.



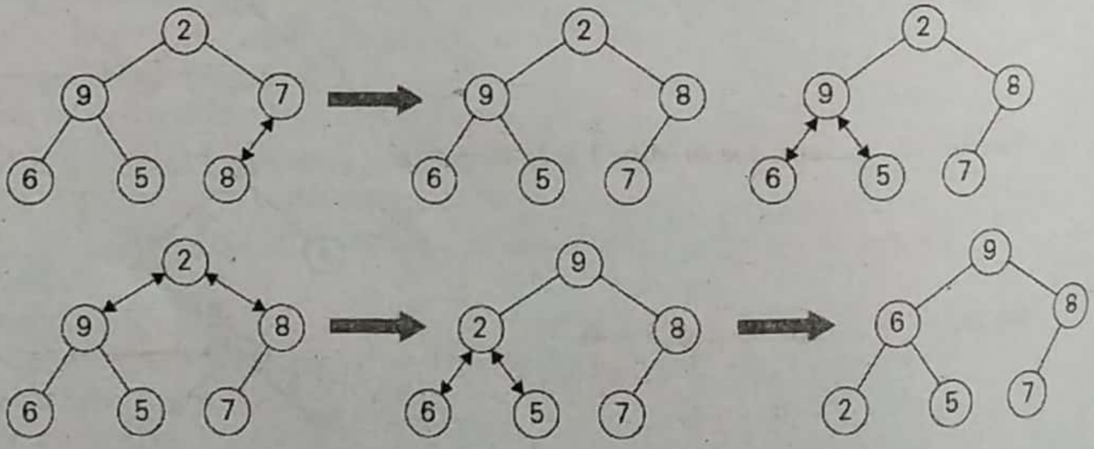
The text ABACABAD will be encoded as 0100011101000101.

With the code of part a, 100010111001010 will be decoded as

100	0	101	110	0	101	0
B	A	D	_	A	D	A

c. Construct the heap for the list 2, 9, 7, 6, 5, 8 by bottom-up algorithm. (04 Marks)

Ans.



Module - 4

7. a. Define transitive closure. Write warshall's algorithm to compute transitive closure. Find is efficiency. **(08 Marks)**

Ans. The transitive closure of a directed graph with n vertices can be defined as the n × n Boolean matrix T = {t_{ij}}, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

```
ALGORITHM   Warshall{A[l.n, l..n])
//Implements Wars hall'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 ← 1 to n do
  for i ← 1 to n do
    for j ← 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 $R^{(n)}$
```

Time efficiency of this algorithm is theta(n^3)

b. Apply floyd's algorithm to find all pair shortest path for the graph **(08 Marks)**

0 0 3 0 0
0 2 0 0 4
0 7 0 1 0
6 0 0 0 0
0 0 0 2 0

Ans. R(0)=

∞	∞	3	∞	∞
∞	2	∞	∞	4
∞	7	∞	1	∞
6	∞	9	∞	∞
∞	∞	∞	2	∞

R(1)=

∞	∞	3	∞	∞
∞	2	∞	∞	4
∞	7	∞	1	11
6	∞	9	∞	∞
∞	∞	∞	2	∞

R(2)=

∞	10	3	4	14
∞	2	∞	∞	4
∞	7	∞	1	11
6	16	9	10	20
∞	∞	∞	2	∞

R(3)=

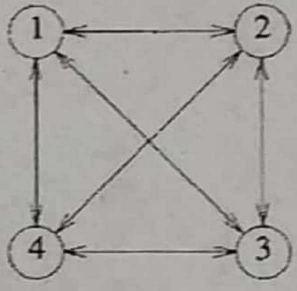
10	10	3	4	14
∞	2	∞	∞	4
7	7	10	1	11
6	16	9	10	20
8	18	11	2	22

R(4)=

10	10	3	4	14
12	2	15	6	4
7	7	10	1	11
6	16	9	10	20
8	18	11	2	22

OR

8. a. For the given cost matrix obtain optimal cost tour using dynamic programming. **(08 Marks)**



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Ans. Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

$g(2,\{3\}) = c_{23} + g(3,\phi) = 15$ $\quad$ $g(2,\{4\}) = 18$

$g(3,\{2\}) = 18$ $\quad$ $g(3,\{4\}) = 20$

$g(4,\{2\}) = 13$ $\quad$ $g(4,\{3\}) = 15$

Next, we compute g(i, S) with $|S| = 2, i \neq 1, 1 \notin S$ and $i \notin S$.

$g(2,\{3,4\}) = \min \{c_{23}+g(3, \{4\}), c_{24} + g(4,(3\})\} = 25$

$g(3,\{2,4\}) = \min \{c_{32}+3(2, \{4\}), c_{34} + g(4,(2\})\} = 25$

$g(4,\{2,3\}) = \min \{c_{42}+3(2, \{3\}), c_{43} + g(3,\{2\})\} = 23$

Finally, from (5.20) we obtain

$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}), C_{13}+g(3,\{2,4\}), C_{14}+g(4,\{2,3\})\}$

$= \min\{35,40,43\} = 35$

b. Write a pseudo code to find an optimal binary search tree by dynamic programming. **(08 Marks)**

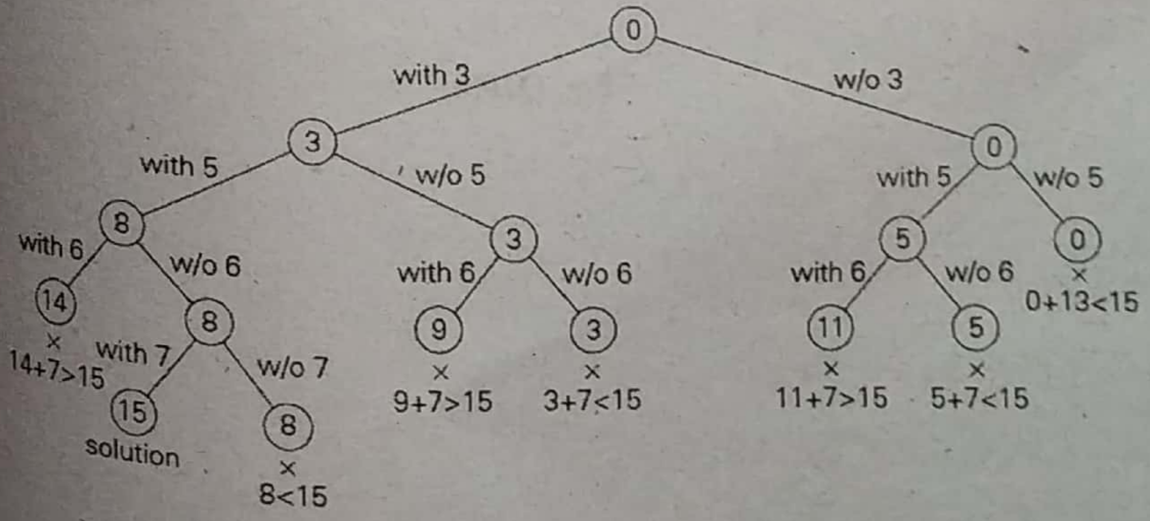
Ans.
```
ALGORITHM    OptimalBST{P[1..n])
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[l..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
```

```
//  optimal EST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i,i −1] ← 0
    C[i,i] ← P[i]
    R[i, i] ← i
C[n + 1,n] ← 0
for d ← 1 to n − 1 do    //diagonal count
    for i ← 1 to n − d do
        j ← 1 + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j];   kmin ← k
        R[i, j] ← kmin
        sum ← P[i];   for s − i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

Module - 5

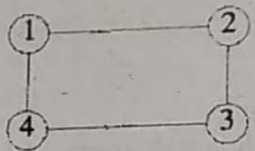
9. a. **Write the pseudo code for backtracking algorithm. Let w= [3, 5, 6, 7} and m=15. Find all possible subsets of w that sum to m. Draw the state space tree that is generated.** **(09 Marks)**

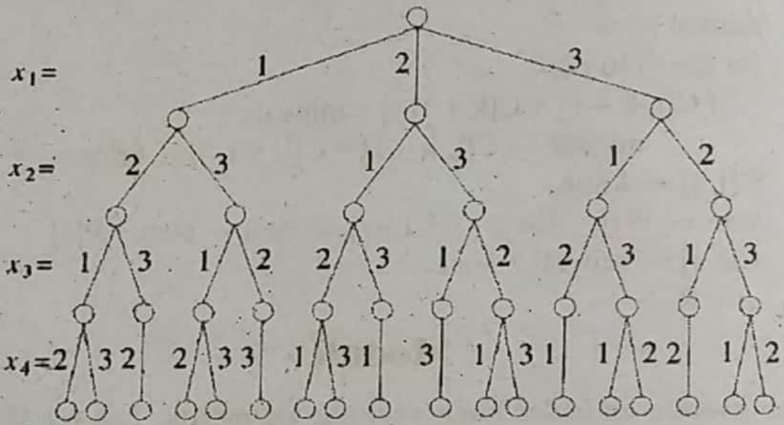
Ans.
```
ALGORITHM Backtrack(X[1..i])
//Gives a template of a generic backtracking algorithm
//Input: X[1..i] specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if X[1..i] is a solution write X[1..i]
else
for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
X[i + 1]←x
Backtrack(X[1..i + 1])
```



b. Draw the portion of the state space tree for m-colorings of a graph when n=4 and m=3 **(07 Marks)**

Ans.





OR

10. a. with a help of a state space tree. Solve travelling salesman problem (TSP) of fig. Q10(a), using branch-and-bound algorithm. **(08 Marks)**

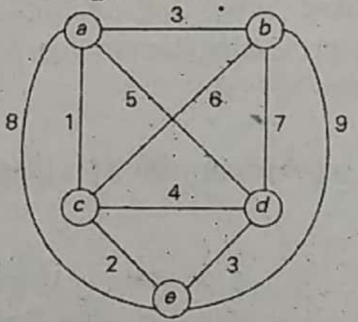
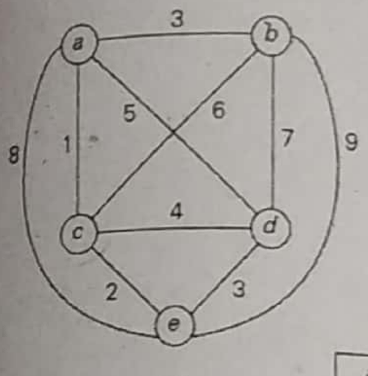
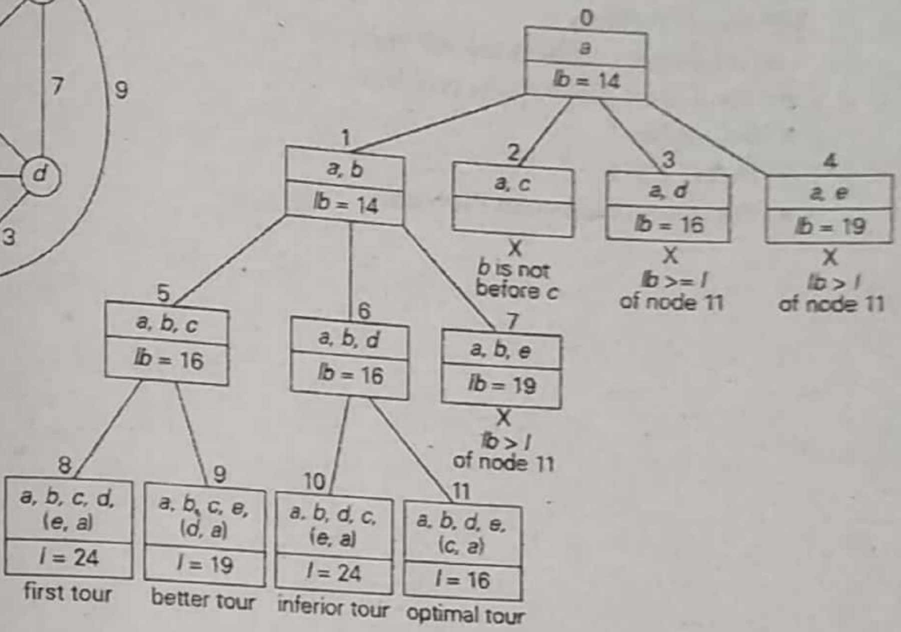


Fig. Q10(a)

Ans.

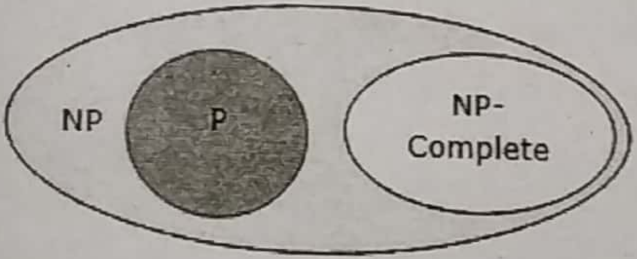


(a)



b. **Explain the classes of NP-Hard and NP-Complete.** **(08 Marks)**

Ans. A problem is in the class NPC if it is in NP and is as hard as any problem in NP. A problem is NP hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called NP-complete. The phenomenon of NP completeness is important for both theoretical and practical reasons.

A language B is NP-complete if it satisfies two conditions

- B is in NP
- Every A in NP is polynomial time reducible to B.

If a language satisfies the second property, but not necessarily the first one, the language B is known as NP-Hard. Informally, a search problem B is NP-Hard if there exists some NP-Complete problem A that Turing reduces to B.

The problem in NP-Hard cannot be solved in polynomial time, until P = NP. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem