# Fourth Semester B.E. Degree Examination, CBCS - June / July 2017
## 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. Explain asymptotic notations, Big O, Big Omega, Big Theta notations.** **(08 Marks)**

**Ans.** An algorithm is a procedure or formula for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm. In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem.

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate.

**Big-O Notation:**

A function t (n) is said to be in O(g(n)), denoted t (n) $\in$ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer $n_0$ such that t (n) $\leq$ cg(n) for all n $\geq$ $n_0$.

**Ω-notation**

A function t (n) is said to be in Ω(g(n)), denoted t(n) $\in$ Ω(g(n)), if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer $n_0$ such that t (n) $\geq$ cg(n) for all n $\geq$ $n_0$.

**Theta-notation**

A function t (n) is said to be in Theta (g(n)), denoted t (n) $\in$ Theta(g(n)), If t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer $n_0$ such that $c_2g(n) \leq t(n) \leq c_1g(n)$ for all $n \geq n_0$.

b. **Explain general plan of mathematical analysis of non-recursive algorithms with example.** **(08 Marks)**

**Ans.** General Plan for Analyzing the Time Efficiency of Non-recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

```
ALGORITHM MaxElement(A[0..n − 1])
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
```

```
maxval ←A[0]
for i ←1 to n − 1 do
if A[i]>maxval
maxval←A[i]
return maxval
```

The obvious measure of an input's size here is the number of elements in the array, i.e., n. The operations that are going to be executed most often are in the algorithms for loop. There are two operations in the loop's body: the comparison A[i] > maxval and the assignment maxval←A[i]. Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. Note that the number of comparisons will be the same for all arrays of size n; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.

$C(n)= \sum_{i=1}^{n-1} 1$

This is an easy sum to compute because it is nothing other than 1 repeated n − 1 times. Thus, n − 1 ∈ Theta(n).

**OR**

2. a. **Define time and space complexity. Explain important problem types. (08 Marks)**

Ans. Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

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 most used 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 (reordering) of the input

[illegible] algorithm, broadly speaking, is an algorithm [illegible] collection of items. The items may [illegible]

[illegible]

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

**b. Illustrate mathematical analysis of recursive algorithm for tower of Hanoi.**

(08 Marks)

**Ans.** The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves M(n) depends on n only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \text{ for } n > 1.$$

With the obvious initial condition M(1) = 1, we have the following recurrence relation for the number of moves M(n):

$$M(n) = 2M(n-1) + 1 \text{ for } n > 1, M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$

$= 2[2M(n-2) + 1] + 1 = 22M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$

$= 2^2[2M(n-3) - 1] + 2 + 1 = 23M(n-3) + 22 + 2 - 1.$

The pattern of the first three sums on the left suggests that the next one will be

$2^4M(n-4) + 23 + 22 + 2 + 1$, and generally, after i substitutions, we get

$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$

Since the initial condition is specified for n = 1, which is achieved for i = n − 1, we get the following formula for the solution to recurrence

M(n) $= 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1$

$= 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$

## Module - 2

**3. a. Explain concept of divide and conquer. Write merge sort algorithm.**

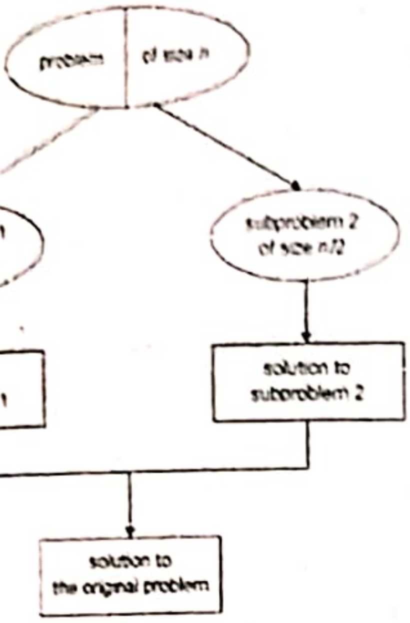
(08 Marks)

**Ans.** 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 [illegible] size.

2. The [illegible] problems are solved (typically recursively, [illegible]) algorithm is employed, [illegible]

3. If necessary, the solutions to the subproblems [illegible] original problem.



```
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 nondecreasing 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]≤ C[j ]
        A[k]←B[i]; i ←i + 1
    else A[k]←C[j ]; j ←j + 1
    k←k + 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]
```

**b. Write a recursive algorithm for binary search and also bring out its efficiency** **(08 Marks)**

**Ans.**

```
ALGORITHM BinRec(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if n = 1 return 1
```

se return BinRec(n/2) + 1

nalysis of recursive binary search

.(n) = A(n/2) + 1 for n > 1.

Since the recursive calls end when n is equal to 1 and there are no additions made then, he initial condition is A(1) = 0.

let us apply this recipe to our recurrence, which for $n = 2^k$ takes the form

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

ow backward substitutions encounter no problems:

$A(2^k) = A(2^{k-1}) + 1$ substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$ substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 \ldots$

. .

.

$= A(2^{k-i}) + i$

. . .

$= A(2^{k-k}) + k$.

Thus, we end up with

$A(2^k) = A(1) + k = k$, or, after returning to the original variable $n = 2^k$

and hence $k = \log_2 n$, $A(n) = \log_2 n \in \text{Theta}(\log n)$.

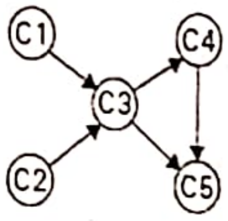
**OR**

**4. a. Illustrate the tracing of quick sort algorithm for the following set of numbers: 25, 10, 72, 18, 40, 11, 64, 58, 32, 9** **(08 Marks)**

**Ans.**

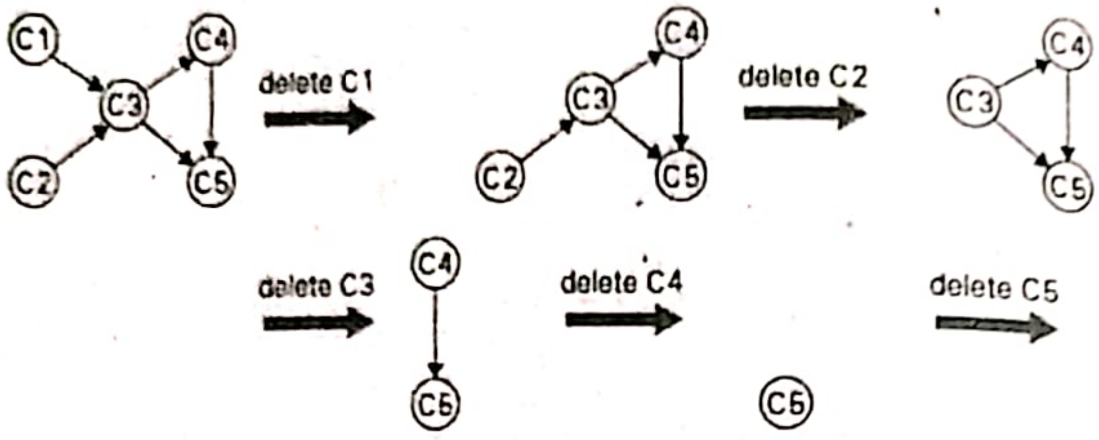
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 25 | 10 | 72 | 18 | 40 | 11 | 64 | 58 | 32 | 9 |
| 25 | 10 | 72 | 18 | 40 | 11 | 64 | 58 | 32 | 9 |
| 25 | 10 | 9 | 18 | 40 | 11 | 64 | 58 | 32 | 72 |
| 25 | 10 | 9 | 18 | 11 | 40 | 64 | 58 | 32 | 72 |
| 11 | 10 | 9 | 18 | 25 | 40 | 64 | 58 | 32 | 72 |
| 9 | 10 | 11 | 18 | 25 | | | | | |
| | | | | | 40 | 32 | 58 | 64 | 72 |
| | | | | | 32 | 40 | 58 | 64 | 72 |

**b. List out the advantages and disadvantages of divide and conquer methods and illustrate the topological sorting for the following graph.**



**(08 Marks)**

**Ans.**



The solution obtained is C1, C2, C3, C4, C5

## Module - 3

**5. a. Explain greedy criterion. Write a prim's algorithm to find minimum cost spanning tree. (08 Marks)**

**Ans.** A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

```
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
  VT ← {$V_0$} the set of tree vertices can be initialized with any vertex
  ET ← ∅
  for i ← 1 to |V| - 1do
          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 $E_T$
```

**b. Sort the given list of numbers using heap sort: 2, 9, 7, 6, 5, 8 (08 Marks)**

**Ans.**

```
Stage 1 (heap construction)      Stage 2 (maximum deletions)
   2  9  7  6  5  8               9  6  8  2  5  7
   2  9  8  6  5  7               7  6  8  2  5 | 9
   2  9  8  6  5  7               8  6  7  2  5
   9  2  8  6  5  7               5  6  7  2 | 8
   9  6  8  2  5  7               7  6  5  2
                                  2  6  5 | 7
                                  6  2  5
                                  5  2 | 6
                                  5  2
                                  2 | 5
                                  2
```

**OR**

**6. a. Write an algorithm to find single source shortest path.** **(08 Marks)**

**Ans.**

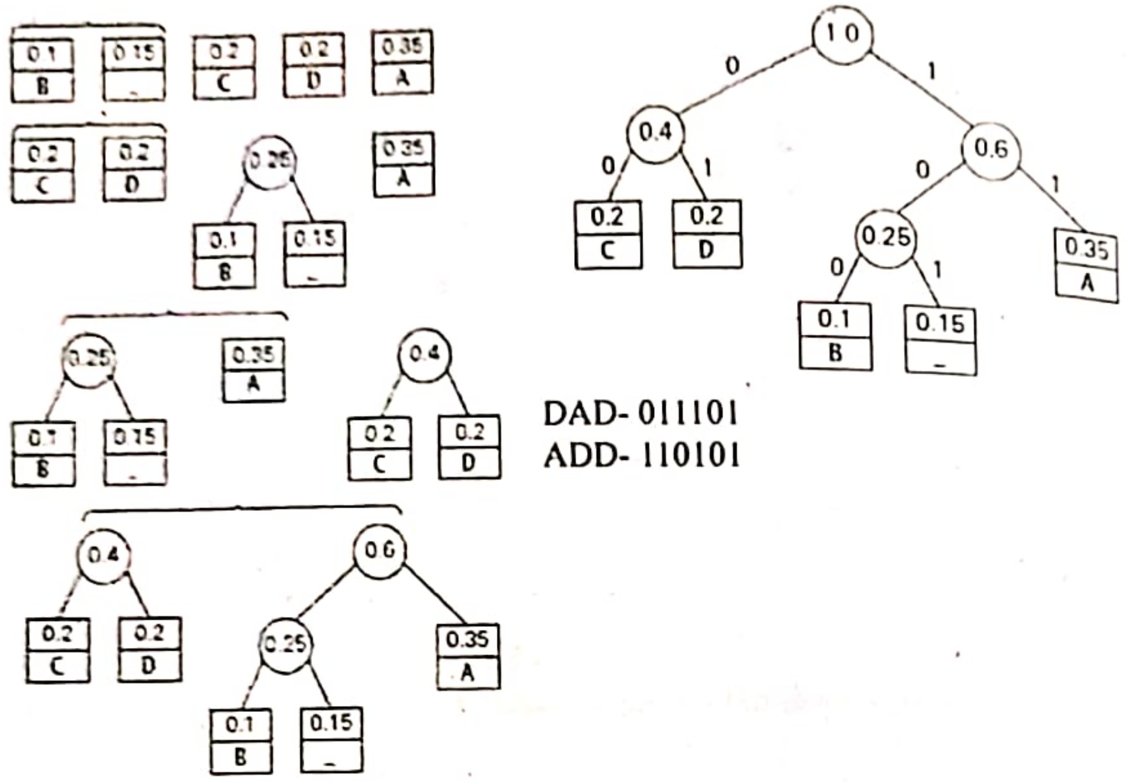
```
ALGORITHM    Dijkstra(G, s)
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = (V, E) with nonnegative weights
//       and its vertex s
//Output: The length dv of a shortest path from s to v
//        and its penultimate vertex pv for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    $d_v \leftarrow \infty$;   $p_v \leftarrow$ null
Insert(Q, v, $d_v$)  //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$;   Decrease(Q, s, $d_s$)  //update priority of s with $d_s$
$V_T \leftarrow \varnothing$
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, $d_u$)
```

b. **Construct a Huffman tree and resulting code word for the following:**

| Character | A | B | C | D | - |
|---|---|---|---|---|---|
| Probability | 0.35 | 0.1 | 0.2 | 0.2 | 0.15 |

**Encode the words DAD and ADD** **(08 Marks)**

**Ans.**



DAD- 011101
ADD- 110101

## Module - 4

7. a. **Explain the concept of dynamic programming with example.** **(08 Marks)**

**Ans.** Dynamic programming (usually referred to as DP ) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again..Shortly 'Remember your Past' :) . If the given problem can be broken up in to smaller sub-problems and these smaller sub-problems are in turn divided in to still-smaller ones, and in this process, if you observe some overlapping sub-problems, then its a big hint for DP.

There are two ways of doing this.

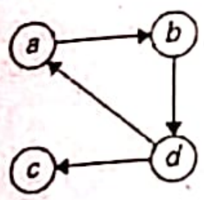
1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive This is referred to as Memorization.

2.) Bottom-Up: Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial sub-problem, up towards the given problem. In this process, it is guaranteed that the sub-problems are solved before solving the problem This is referred to as Dynamic Programming.

Example: Coin-rowproblem There is a row of n coins whose values are some positive integers c1, c2, . . . , cn, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up. Let F(n) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F(n), we partition all the allowed coin selections into two groups: those that include the last coin and those without it. The largest amount we can get from the first group is equal to cn + F(n − 2)—the value of the nth coin plus the maximum amount we can pick up from the first n − 2 coins. The maximum amount we can get from the second group is equal to F(n − 1) by the definition of F(n). Thus, we have the following recurrence subject to the obvious initial conditions: F(n) = max{cn + F(n − 2), F(n − 1)} for n > 1, F(0) = 0, F(1) = c1.

**b. Trace the following graph using warshall's algorithm.** **(08 Marks)**

**Ans.**



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

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 $R^{(1)}$.

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

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 $R^{(2)}$.

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

1's reflect the existence of paths with intermediate vertices numbered not higher than 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 $R^{(4)}$.

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

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).

**OR**

**8. a. Explain multistage graphs with example. Write multistage graph algorithm to forward approach.** **(08 Marks)**

**Ans.** A multistage graph G=(V,E) which is a directed graph. In this graph all the vertices are partitioned into the k stages where k>=2. In multistage graph problem we have to find the shortest path from source to sink. The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path. In multistage graph problem we have to find the path from S to T. there is set of vertices in each stage.

The multistage graph can be solved using forward and backward approach. Let us solve multistage problem for both the approaches with the help of example.
Consider the graph G as shown in the figure.

```
int[] MStageForward(Graph G)
{
  // returns vector of vertices to follow through the graph
  // let c[i][j] be the cost matrix of G
  int n = G.n (number of nodes);
  int k = G.k (number of stages);
  float[] C = new float[n];
  int[]  D = new int[n];
  int[]  P = new int[k];
  for (i = 1 to n) C[i] = 0.0;
  for j = n-1 to 1 by -1 {
    r = vertex such that (j,r) in G.E and c(j,r)+C(r) is minimum
    C[j] = c(j,r)+C(r);
    D[j] = r;      }
  P[1] = 1; P[k] = n;
  for j = 2 to k-1 {
    P[j] = D[P[j-1]];    }
  return P;      }
```

**b. Solve the following instance of knapsack problem using dynamic programming. Knapsack capacity is 5** **(08 Marks)**

| Item | Weight | Value |
|---|---|---|
| 1 | 2 | $12 |
| 2 | 1 | $10 |
| 3 | 3 | $20 |
| 4 | 2 | $15 |

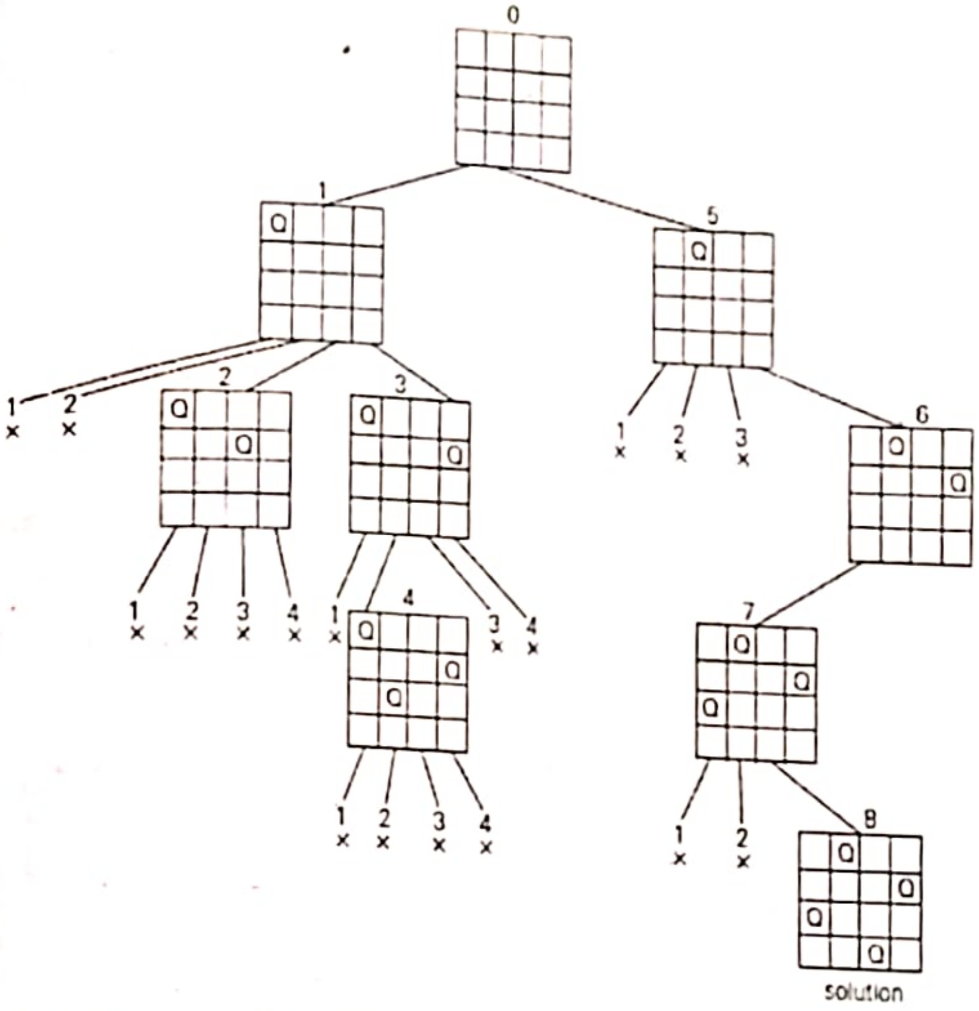
**Ans.**

| | | capacity *j* | | | | | |
|---|---|---|---|---|---|---|---|
| | *i* | 0 | 1 | 2 | 3 | 4 | 5 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1 = 2, v_1 = 12$ | 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| $w_2 = 1, v_2 = 10$ | 2 | 0 | 10 | 12 | 22 | 22 | 22 |
| $w_3 = 3, v_3 = 20$ | 3 | 0 | 10 | 12 | 22 | 30 | 32 |
| $w_4 = 2, v_4 = 15$ | 4 | 0 | 10 | 15 | 25 | 30 | 37 |

## Module - 5

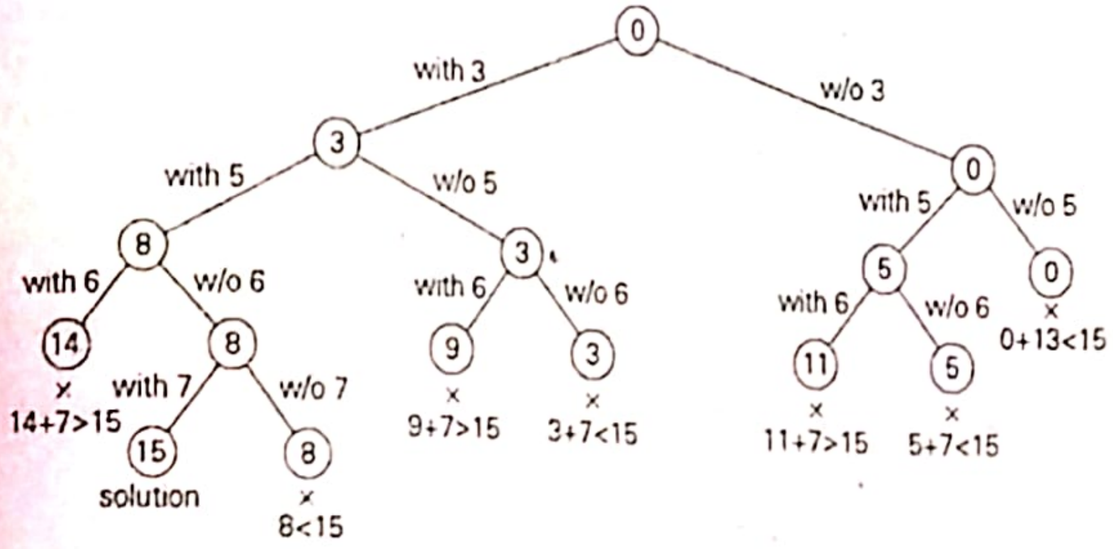
**9. a. Explain backtracking concept. Illustrate N queens problem using backtracking to solve 4-queens problem.** **(08 Marks)**

**Ans.** The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows. If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.



**b. Solve subset sum problem for the following example, S={3, 5, 6, 7} and d=15. Construct a state space tree.** **(08 Marks)**

**Ans.**

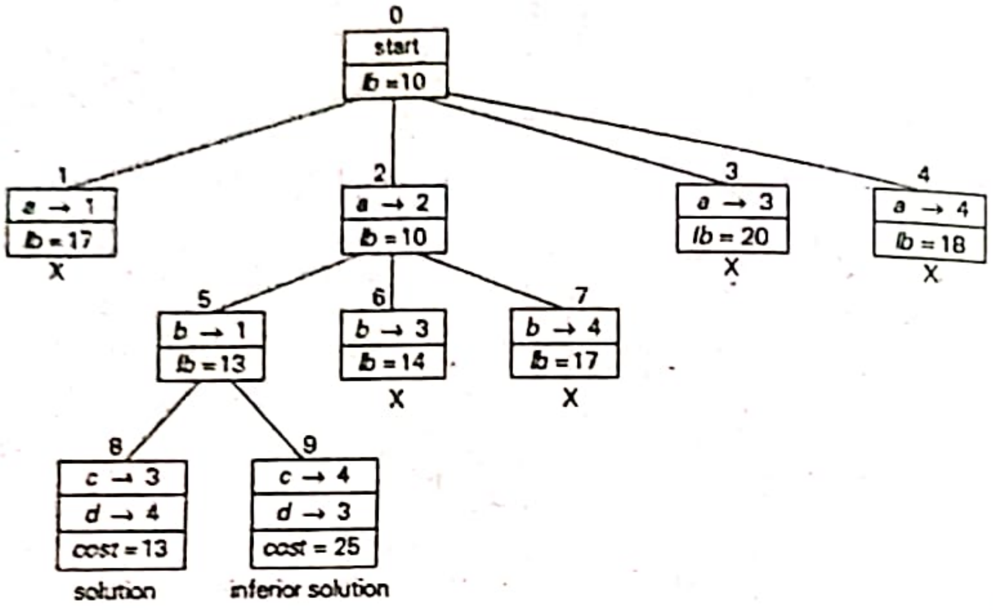


**OR**

**10 a. Explain the concept of branch and bound solve assignment problem for the following and obtain optimal solution.** **(08 Marks)**

$$C = \begin{array}{ccccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ \left[\begin{matrix} 9 \\ 6 \\ 5 \\ 7 \end{matrix}\right. & \begin{matrix} 2 \\ 4 \\ 8 \\ 6 \end{matrix} & \begin{matrix} 7 \\ 3 \\ 1 \\ 9 \end{matrix} & \left.\begin{matrix} 8 \\ 7 \\ 8 \\ 4 \end{matrix}\right] & \begin{matrix} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix} \end{array}$$

**Ans.**



**b. Explain LC Branch and bound and FIFO branch and bound.** **(08 Marks)**

**Ans. FIFO branch-and-bound algorithm**

- Initially, there is only one live node; no queen has been placed on the chessboard
- The only live node becomes E -node
- Expand and generate all its children; children being a queen in column 1, 2, 3, and 4 of row 1 (only live nodes left)
- Next E -node is the node with queen in row 1 and column 1
- Expand this node, and add the possible nodes to the queue of live nodes
- Bound the nodes that become dead nodes

Compare with backtracking algorithm

- Backtracking is superior method for this search problem

**Least Cost ( LC ) search**

– Selection rule does not give preference to nodes that will lead to answer quickly but just queues those behind the current live nodes

- In 4-queen problem, if three queens have been placed on the board, it is obvious that the answer may be reached in one more move
- The rigid selection rule requires that other live nodes be expanded and then, the current node be tested

– Rank the live nodes by using a heuristic c(·)
The next E -node is selected on the basis of this ranking function
– Heuristic is based on the expected additional computational effort (cost) to reach a solution from the current live node