CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | |

Internal Assesment Test - III

Sub:	Analysis and Design of Algorithms							Code:	13MCA41
Date:	29 / 05 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	III	Branch:	MCA

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1 (a)	What are decision trees? Explain with example, how decisions trees are used to prove lower bound of sorting problem.	[10]	CO3	L3

Sol:
Definition Decision Trees with uses - 4M
Example+Explanation of decision tree for sorting - 6M

Sol: Decision tree is a mechanism for studying the performance of compari based algorithms such as sorting and searching. For example : for the probl of finding max of 3 numbers the decision tree is a binary tree. Each interna node of a binary decision tree represents a key comparison indicated in the node e.g., k < k . The node's left subtree contains the information about subsequent comparisons made if k' < k , and its right subtree does the same for the cas k' >k. Each leaf represents a possible outcome of the algorithm's run on som input of size n. The number of leaves can be greater than the number of outcomes because, for some algorithms, the same outcome can be arrived at through a different chain of comparisons but the number leav cannot be lesser than the number of possible outcomes because if so, the algorithm is not correct since for a particular combination which should res in the missing outcome, the algorithm would not produce the correct result. algorithm's work on a particular input of size n can be traced by a path fron root to a leaf in its decision tree, and the number of comparisons made by t algorithm on such a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree. The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, hasto be tall enough to have that many leaves. We use the lemma that for any binary tree with l leaves and height h, $h \geq \log_2 l$. Indeed, a binary tree of height h with the largest number of leaves has all its leaves on the last level. Hence, the largest number of leaves in such a tree is 2^h. In other words, $2^h \geq l$, which immediately implies, $h \geq \log_2 l$. Inequality given above puts a lower bound on the heights of binary decision trees and hence the worst-case number of comparisons made by any comparison-based algorithm for the problem in question. Such a bound is called the **information theoretic lower bound.**

Decision trees are an alternate way of representing comparison based algorithms and thus have the following advantages:

1. they provide a visual representation of the algorithm and in understanding the comparisons made.

2. The height of the decision tree automatically conveys the worst case performance of the algorithm.

3. They can be used to prove the lower bound on any problem. This is useful since a person would attempt a problem for which lower bound is proved only if there is a gap between the lower bound and the best solution available. Otherwise a person can give up trying to obtain a better solution.

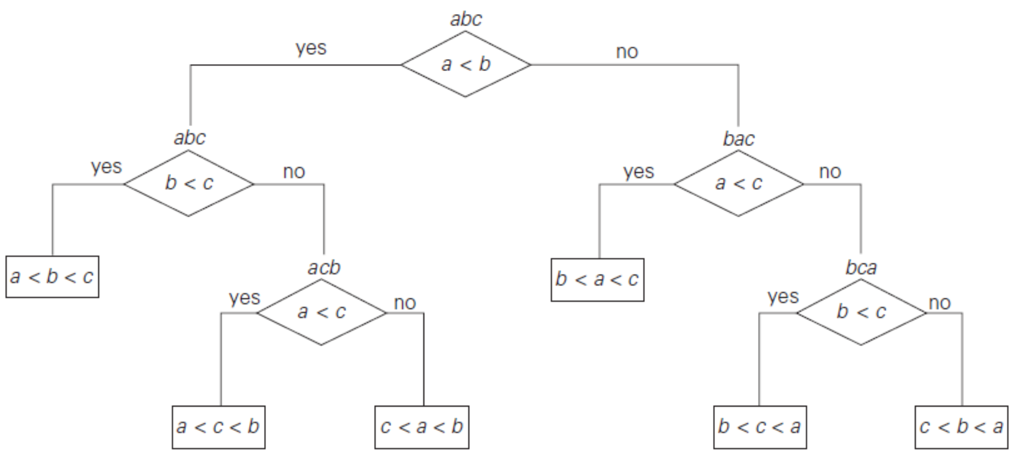
Sorting:
We can interpret an outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list's elements in ascending order. Consider, as an example, a three-element list a, b, c of orderable items such as real numbers or strings. For the outcome a < c<b obtained by sorting this list, the permutation in question is 1, 3, 2. In general, the number of possible outcomes for sorting an arbitrary n-element list is equal to n!.
Inequality above implies that the height of a binary decision tree for any comparison-based sorting algorithm and hence the worst-case number of comparisons made by such an algorithm cannot be less than

$$C_{worst}(n) \ge \lceil \log_2 n! \rceil .$$

Using Stirling's formula for n!, we get:
In other words, about n log2 n comparisons are necessary in the worst case to sort an arbitrary n-element list by any comparison-based sorting algorithm. An example of of decision tree for sorting of 3 numbers is shown.

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n = n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n \log_2 n.$$

In other words, about n log2 n comparisons are necessary in the worst case to sort an arbitrary n-element list by any comparison-based sorting algorithm. An example of of decision tree for sorting of 3 numbers is shown.



2 (a) Define with examples the following classes: P, NP and NP-Complete. [10] CO5 L1

Sol: P,Np,Np-Complete, NP-Hard - 2.5x4=10M

Class P: There are many algorithms for which polynomial time solution exists and thus are tractable(i.e. solvable in a reasonable amount of time). Informally P consists of the set of problems which are tractable. A formal definition of P is:
Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called **polynomial**.The main reason for only considering decision problems is that many naturally occurring problems can be posed as a decision problem and decision problems in general are deemed to be "easier" to solve that their non decision counterparts, i.e. if a natural
version of the problem is tractable then there is a tractable algorithm available for the decision version as well. Examples: Whether an element is present in the array(O(n), element uniqueness problem(O(nlgn). Etc.
Decision problems fall under 3 categories:
1. No solution: For example for the halting problem which is the problem of determining whether a given algorithm would halt on a given input does not have any algorithm for it. Thus it is undecidable.

	2. Decidable but exponential time: There are other decidable problems which are intractable. The natural problems in this category are very rare 3. No polynomial solution till date and nobody has been able to prove a lower bound. Many important problems fall under this category. a. **Hamiltonian circuit problem** Determine whether a given graph has a Hamiltonian circuit b. **Graph coloring: Can a graph be colored with n colors?** **For a majority of these problems the number of choices while constructing a solution rises exponentially but checking** whether a proposed solution actually solves the problem is computationally easy. This observation gives rise to the notion of non deterministic algorithm. A **nondeterministic algorithm** is a two-stage procedure that takes as its input an instance I of a decision problem and does the following. Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (but may be complete gibberish as well). Deterministic ("verification") stage: A deterministic algorithm takes both I and S as its input and outputs yes or no if S represents a solution to instance I or no if it doesn't. We say that a nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution. Class NP is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called **nondeterministic polynomial**. The problems in class P are in NP because the polynomial time solution can be used for guessing and the result of verification can be ignored and hence . But in addition P also contains decision problems which currently don't have a polynomial time solution e.g. Hamiltonian circuit problem, knapsack, graph coloring etc. The question as to whether P = NP remains unanswered. A decision problem D is said to be **NP-complete** if: **1.** it belongs to class NP **2.** every problem in NP(Q) is polynomially reducible to D i.e. it should be possible to change an instance of Q to an instance of D and get the answer of Q from the output of D in polynomial time. The first example of NP complete problem(proved by Cook) is **CNF-satisfiability problem which is to determine given a Boolean expression** in CNF form whether or not one can assign values true and false to variables to make the entire expression true. Other examples of NP-Complete problems are : Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring etc.NP complete problems are very important because even if one of the problems are solvable in polynomial time then a wide variety of important problems would have a polynomial time solution. NP Hard problems: A decision problem D is said to be **NP-Hard** if every problem in NP is polynomially reducible to D In some sense NP- hard problems are harder than NP problems. All problems in NP complete are in NP hard. In addition Halting problem which is not solvable(and hence not in NP- complete) is also NP-Hard which means that if there is ever a polynomial time solution to Halting problem then every problem in NP would be solvable in polynomial time.			
3 (a)	Explain backtracking. Describe the 8-queen's problem and discuss the possible solution. Sol: Explanation backtracking - 4M NQueen's problem description - 2M Solution using backtracking - 4M There are certain problems encountered that require finding an element with a special	[10]	CO4,CO1	L2

property in a domain that grows exponentially fast (or faster) with the size of the problem's input. For such problems the exhaustive-search technique suggests generating all candidate solutions and then identifying the one with a desired property. Backtracking is a more intelligent variation of this approach where the main 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.

This kind of processing can be done by a state-space tree. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution, the

nodes of the second level represent the choices for the second component,

A node in a state-space tree is said to be promising if it can lead to a compl

solution. A DFS is used to implement backtracking.

The n-queens problem. is to place n queens on an n × n chessboard so that n

queens attack each other by being in the same row or in the same column or

same diagonal.

To solve this using backtracing we use the following strategy:

We start with the empty board and then place queen 1 in the first possible

of its row, which is in column 1 of row 1. Then we place queen 2, after trying

unsuccessfully columns 1 and 2, in the first acceptable position for it, which

(2, 3), the square in row 2 and column 3. This proves to be a dead end becau

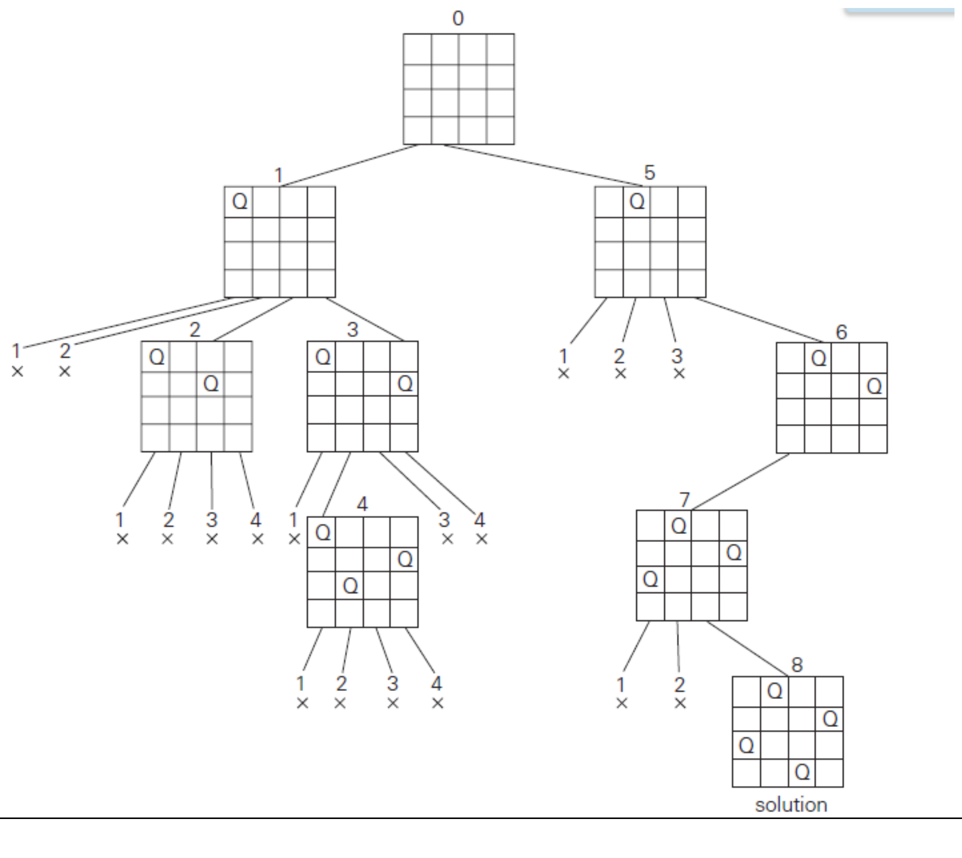
is no acceptable position for queen 3. So, the algorithm backtracks and puts

in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), whic

to be another dead end. The algorithm then backtracks all the way to queen

moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4

which is a solution to the problem. **The state space tree is shown below:**



4 (a) Solve the following assignment problem using branch and bound(Fig 4(a)) [6] CO2 L3

$$\begin{array}{c|c} \text{Job} \rightarrow 1 \quad 2 \quad 3 \quad 4 & \text{person} \downarrow \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} & \begin{array}{c} a \\ b \\ c \\ d \end{array} \end{array}$$

Fig. 4(a)

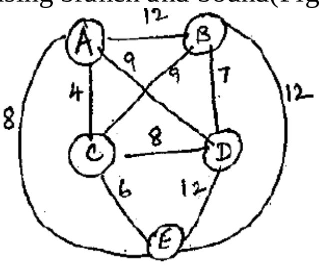
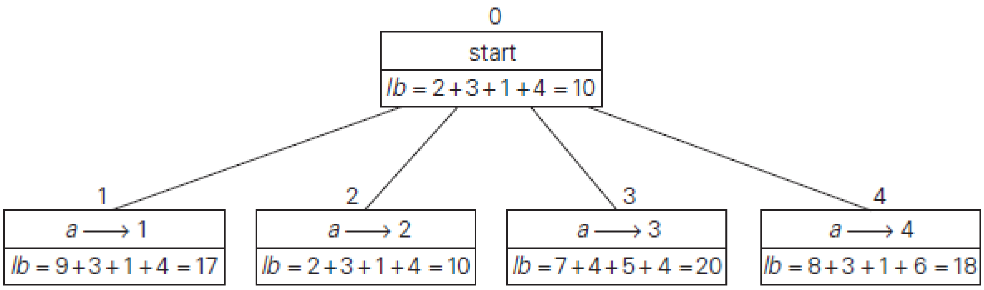


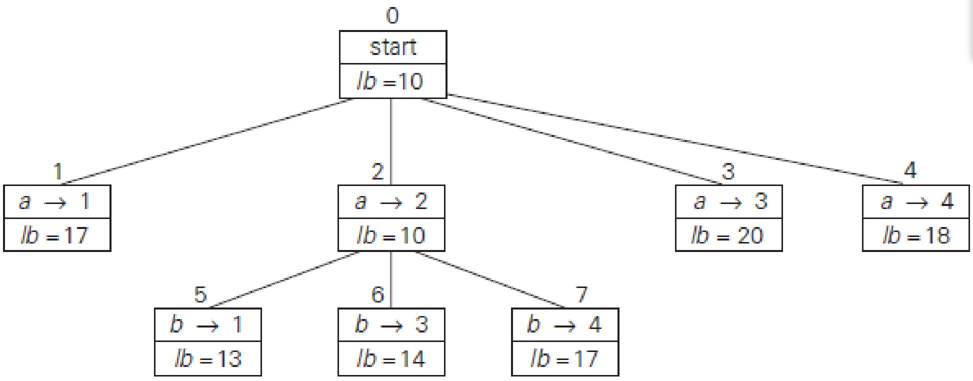
Fig. 4(b)

Sol: Solution 4 levels in the tree - 1.5x4 = 6M

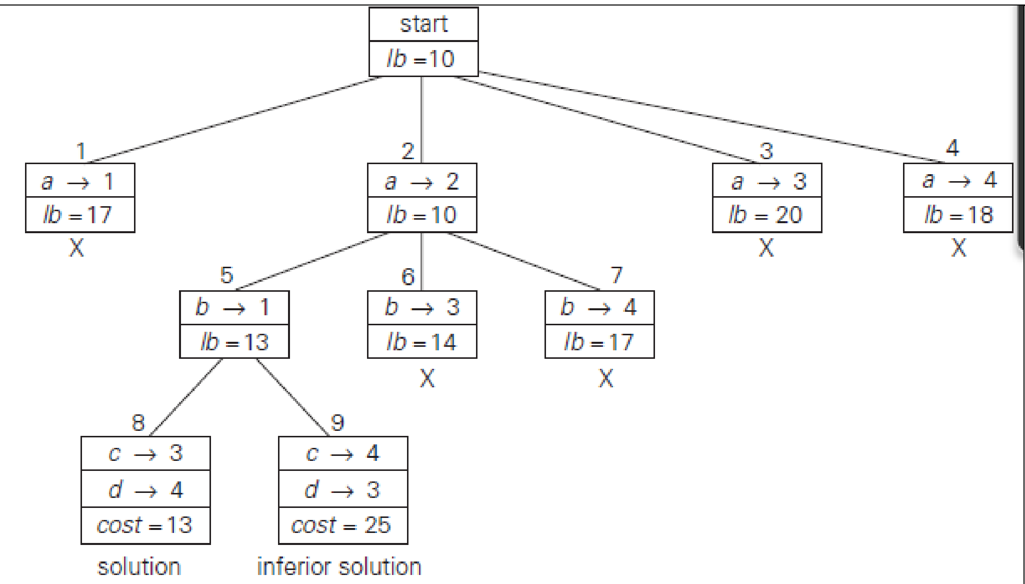
To find the lower bound,the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is 2 + 3+ 1+ 4 = 10. We can apply the same thinking to partially constructed solutions. For example, for anylegitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17. Constructed in this way the search tree is



Expanding the state with the least lb(i.e. state 2)



Now state 5 has the least lb among all states . So expanding it



We find that all the other solutions or partially constructed solutions are inferior to the best solution till now i.e. state 8 since they have a lower bound greater than the best solution till now i.e. 13. So we do not expand any of the other states and the best solution is : a-->2, b-->1 , c-->3, d-->4 with a total cost of 13.

(b) Write the psedocode for sum of subsets problem using backtracking method. [4] CO5 L3

Pseudocode - 4M

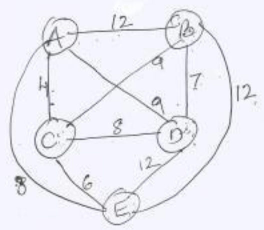
```
//  s - set of numbers
// n- number of items
// N  - sum required
//sum - sum of the partially constructed subset
//i - position of the number to be considered for inclusion/exclusion
//r - sum of the rest of numbers
//x - binary solution - if x[i]= 1 then 'i'th element is included, else it is not included
Algorithm SOS(s[1..n], N , sum, i, r,x[1..n])
{
    If ( i <= n)
    {
        x[i] = 1 // include the ith item in subset
        If ( sum + x[i] = N ) // if we found the solution
            Print  "Solution is ",x
        else if (sum+x[i]+x[i+1]=N
             SOS(s,N,sum+x[i],i+1,r-x[i],x)

        x[i]=0   //case when ith object is not included
       if (sum+r-x[i] >=N and sum+x[i+1] <= N )
           SOS(s,N,sum,i+1,r-x[i],x)
    }
}
```

5 (a) Draw state space tree of branch-and-bound technique to find optimal tour for travelling salesperson for the given graph Fig. 4(b) [10] CO1,CO6 L2,L4

Sol: Explanation of lower bound - 2M

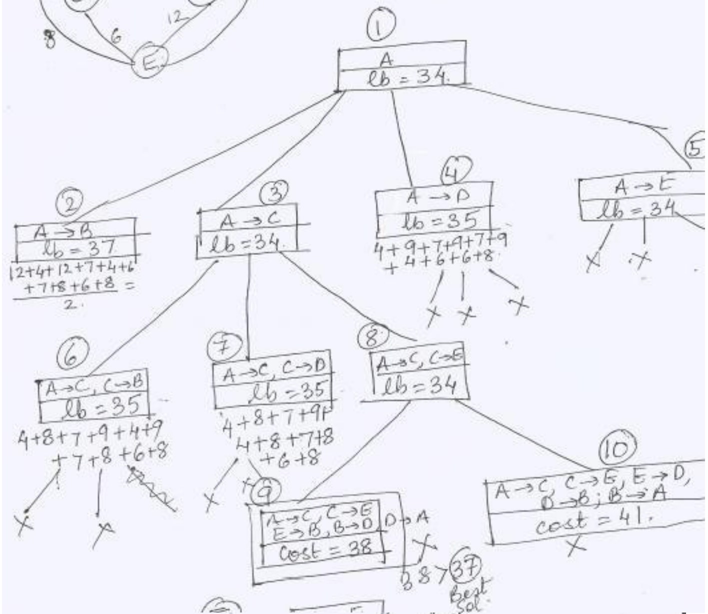
Drawing 4 levels - 2M each - 2x4 = 8M

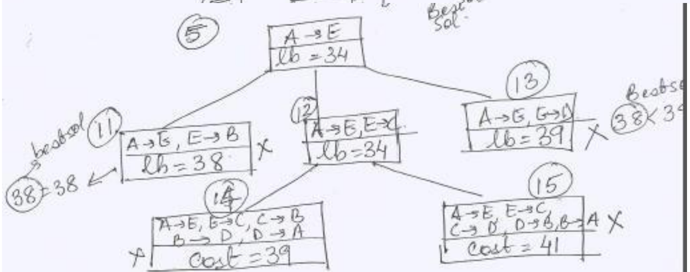


The initial lower bound

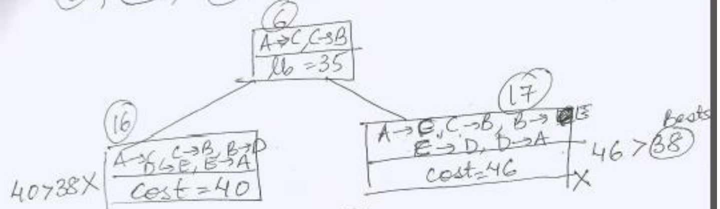
$= \frac{4+8+7+9+4+6+7+8+6+}{2}$

$= \frac{67}{2} = 33.5 = 34.$

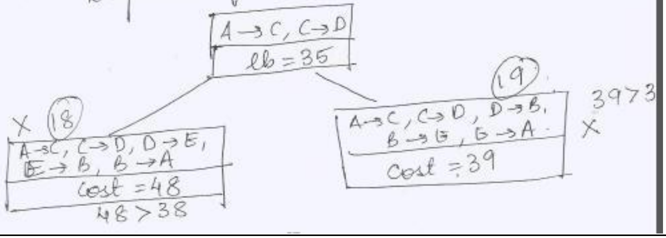


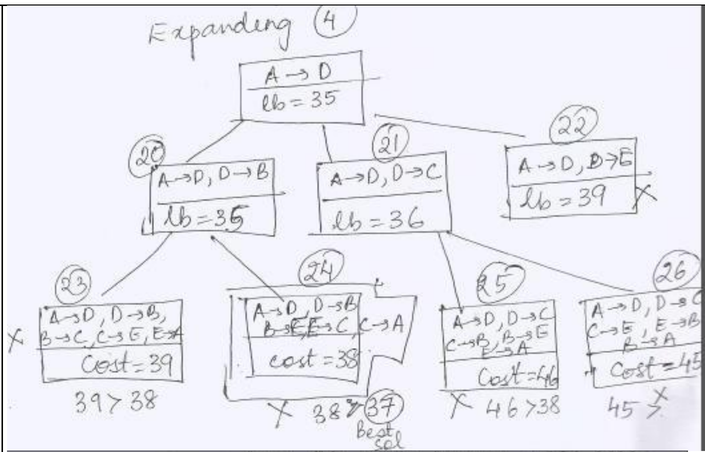
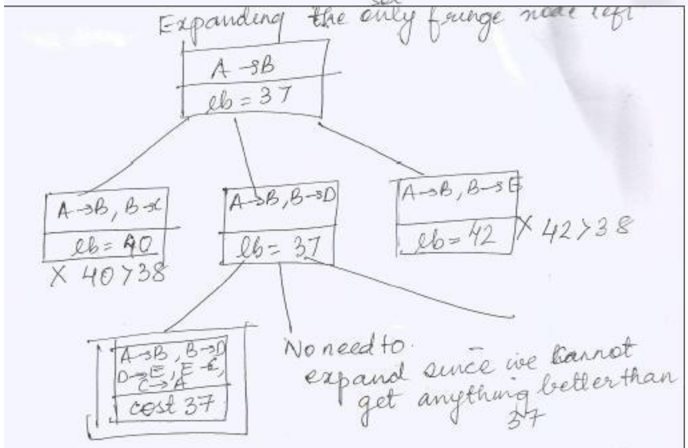


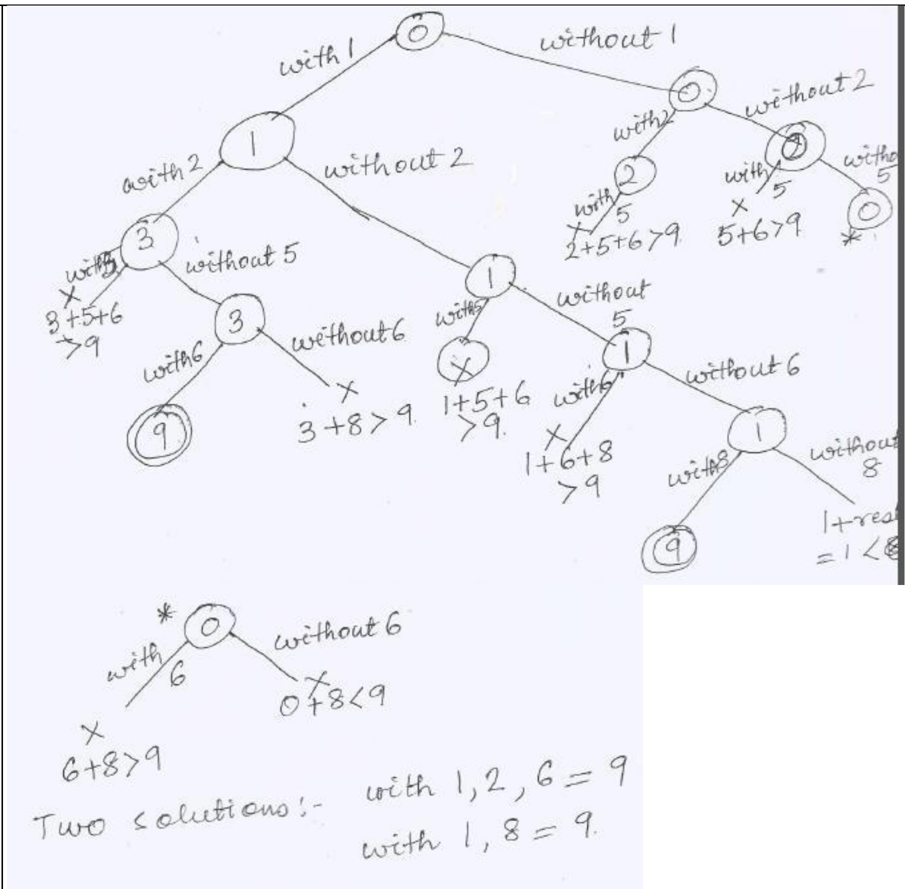
There are 3 fringe states with mincost 6, 7, 4 each with cost 35. Expanding



Expanding 7



	∴ Best solution A — B — D — E — C — A cost = 37			
6 (a)	Using backtracking solve sum of subsets problem for the following instance n=5, d=9, set S= { 1,2,5,6,8}. Sol: 4 levels - 1,5x4 = 6M n=5, d=9.; S = {1, 2, 5, 6, 8} Below is the state space tree applied to S = {1, 2, 5, 6, 8}, d = 9. The number inside a node is the sum of elements already included. The inequality below a node shows reason for termination	[6]	CO3	L1



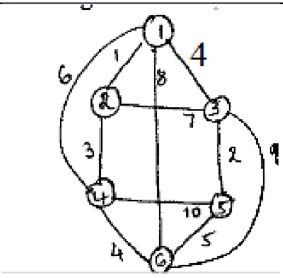
(b) Distinguish between backtracking and branch and bound methods. [4] CO2 L2,L3

Sol: 2 differences in details - 2x2M = 4M

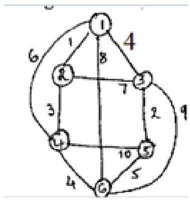
Backtracking and branch and bound are similar to each other because they are normally used to solve problems whose state space grows exponential. To save time in searching in the huge search space a method is employed to prune some states which would not lead to a solution. Backtracking checks for a partially constructed solution satisfying constraints whereas branch and bound also uses an additional bound per state which decides if a state is promising or not. If a state is unpromising then the state is not explored/expanded.

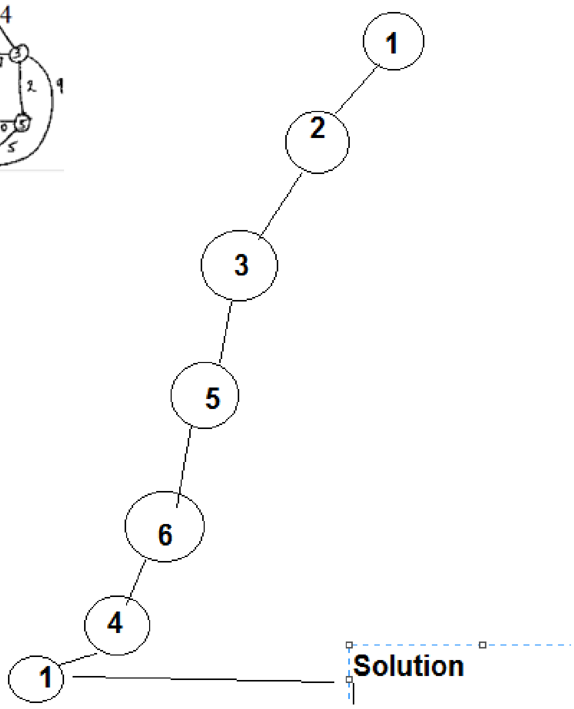
Backtracking	Branch and bound
Used for nonoptimization problems	Used for solving optimization problems
Uses a DFS	Sometimes uses BFS to implement because all states at a level are produced to check which is the most promising.
Does not use any bound. Checks if partial solution follows the constraints	Defines a lower/upper bound for every state which is used for 2 purposes: To determine the next most promising state. To use the bound to prune a state by comparing its lower(upper bound) of the already found solution.

7 (a) Find a Hamiltonian circuit using backtracking method for the graph below ignoring the weights. [5] CO6 L4



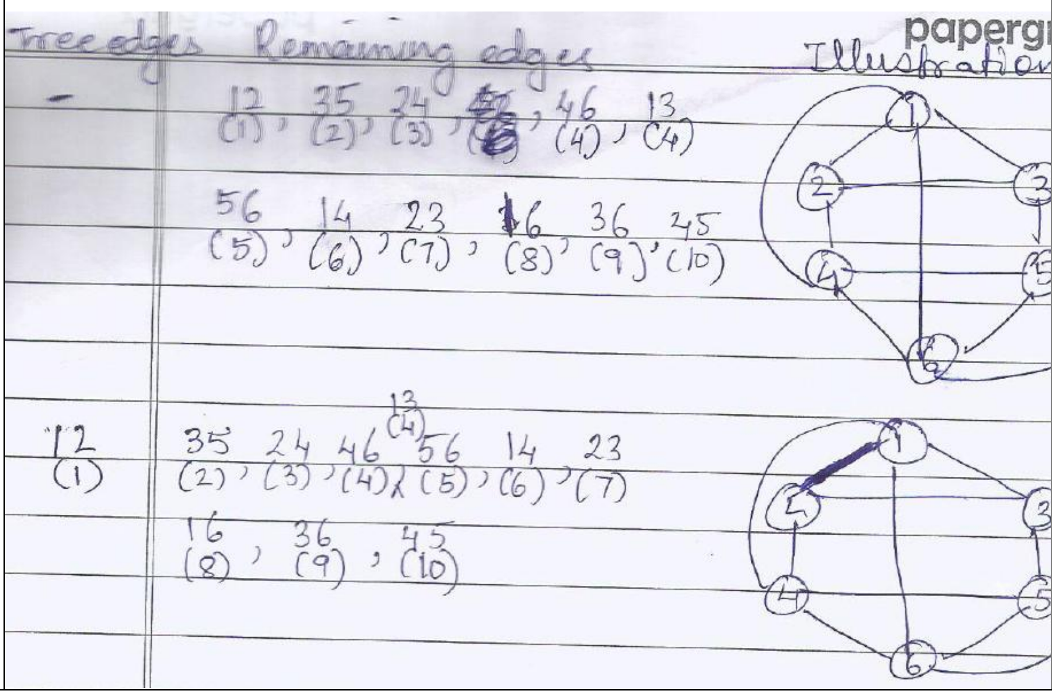
Sol: Steps in the solution. Atleast 3 solutions - One of which is shown below - 5M

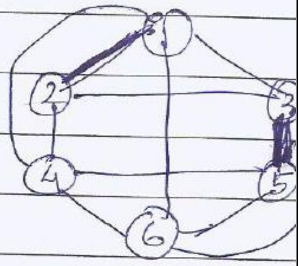
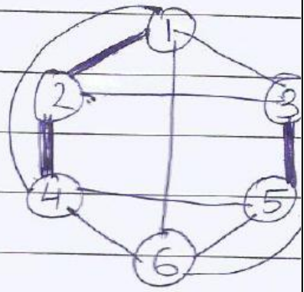
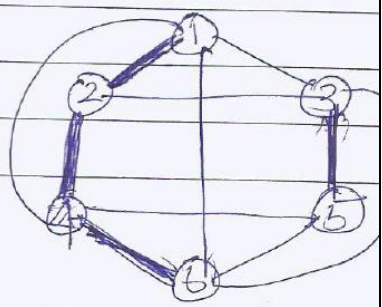
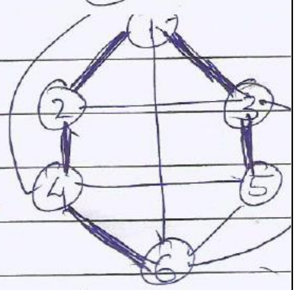




(b) For the above graph find he minimal spanning tree using Kruskal's algorithm. [5] CO3 L3

Sol: 5 edges in the tree - 1x5 = 5M



35 (2)	24 (3), 46 (4), ~~56~~ 13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
24 (3)	46 (4), 13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
46 (4)	13 (4), 56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	
13 (4)	56 (5), 14 (6), 23 (7), 16 (8), 36 (9), 45 (10)	

Hence the edges in the minimal spanning tree using Kruskal's is: 1-2, 2-4, 4-6,3-5 and 1-3, having cost of 1+2+3+4+4 = 14

8(a) Write and Analyze the pseudo code for Kruskal's algorithm for finding spanning tree [5] CO6 L3

Sol: Algorithm - 3M
Analysis - 2M

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 of the algorithm. The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise. Thepseudocode is outlined below: **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 We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges. The initial forest consists of |*V*| trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge *(u, v)* from the sorted list of the graph's edges, finds the trees containing the vertices *u* and *v*, and, if these trees are not the same, unites them in a larger tree by adding the edge *(u, v)*. There are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called ***unionfind*** algorithms which uses two operations: union and find , find to find the representative element and union for combining two disconnected components whdn an edge is added between them. union operation takes O(1) time since a max of 3 operations are performed, whereas find can be performed in time O(lgn). Since the find has to be done every time an edge is considered fpr addition in the tree the time taken for performing the find across all iterations would be atmost Elg|V|.. Across all iterations the union would take O(|E|) time. The time taken for sorting the edges would take O(Elg|E|) time for a total time complesity of : O(|E|lg|E|+|E|lg|V|+|E|) = O(|E|lg|E|) since for a connected graph |V| < |E|.			
(b)	Write and Analyze the pseudo code for Dijkstra's algorithm for finding the single source shortest path.	[5]	CO6	L2

Sol: Algorithm - 3M
Analysis - 2M

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.

- 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 weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by and $d_{u^*} + w(u^*, u)$, respectively.

The psuedocode for Dijkstra's is as given below:

ALGORITHM *Dijkstra(G, s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weight
//         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 $\Theta(|V|)$ 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 $\Theta(1)$ time because the index of the vertex would be known. Thus the total time complexity is $\Theta(|V|^2)$.

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

All heap operations take $\Theta(\lg|V|)$ time. Thus the first loop runs |V| times and each time the Insert would take $\Theta(\lg|V|)$ 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 $\Theta(V\lg|V|)$. 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 $\Theta((\|E\|+\|V\|)\lg\|V\|)$.			