20CB/CS/DS/IT404

Hall Ticket Number:

II/IV B.Tech (Regular\Supplementary) DEGREE EXAMINATION

July/August, 2023 — **Common to CB,CS,DS & IT Branches**

Fourth Semester — **Design And Analysis of Algorithms**

Time: Three Hours — **Maximum:** 70 Marks

Answer question 1 compulsory. **(14X1 = 14Marks)**

Answer one question from each unit. **(4X14=56 Marks)**

			CO	BL	M
1	a)	Define an algorithm.	CO1	L1	1
	b)	Define pseudocode.	CO1	L2	1
	c)	Name the criteria to be satisfied by an algorithm.	CO1	L1	1
	d)	How to calculate the time complexity?	CO1	L2	1
	e)	Identify the difference between divide and conquer and greedy method.	CO2	L2	1
	f)	Label the time complexity of quick sort.	CO2	L1	1
	g)	Why is it necessary to have auxiliary array in merge function?	CO2	L2	1
	h)	What is job sequencing with deadlines?	CO2	L1	1
	i)	Differentiate connected and biconnected components.	CO3	L2	1
	j)	Where do we use dynamic programming?	CO3	L2	1
	k)	State the principal of optimality.	CO3	L1	1
	l)	Summarise the advantages of backtracking.	CO4	L2	1
	m)	When is a problem considered to be NP-hard?	CO4	L1	1
	n)	What is backtracking?	CO4	L1	1

Unit-I

2	a)	What is asymptotic notation? Explain the usage of different types of notations.	CO1	L2	7M
	b)	Label the master theorem and inspect the implementation of the master theorem.	CO1	L4	7M

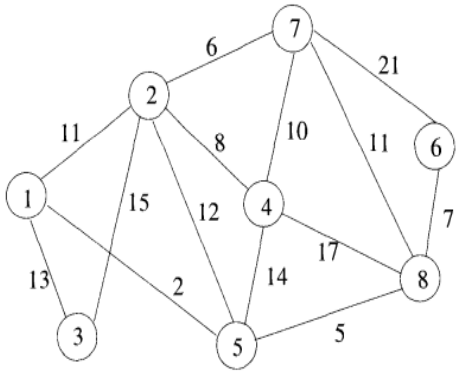
(OR)

3	a)	Show and describe the Pseudocode conventions with syntax and examples.	CO1	L2	7M
	b)	How to devise, validate, analyze, and test an algorithm? Discuss in detail.	CO1	L3	7M

Unit-II

4	a)	Illustrate the solution for executing the single source shortest path problem.	CO2	L2	7M
	b)	Consider the array, a[l:10] = (310,285,179,652,351,423,861,254,450,520). Show the merge sort algorithm and sort the above array using merge sort.	CO2	L3	7M

(OR)

5	a)	 Compute a minimum cost spanning tree for the graph using Kruskal's algorithm.	CO2	L3	7M
	b)	Demonstrate the strategy followed by the divide and conquer approach.	CO2	L2	7M

Unit-III

6	a)	Elucidate the mechanism for solving the 0/1 knapsack problem using dynamic programming.	CO3	L2	7M
	b)	Differentiate the implementation of depth first traversal and breadth first traversal.	CO3	L3	7M
		(OR)			
7	a)	Find the longest common subsequence of X and Y using dynamic programming. X=<ABCDAF>, Y=<ACBCF>	CO3	L2	7M
	b)	Inspect the traveling salesperson problem to find a tour of minimum cost.	CO3	L4	7M
		Unit-IV			
8	a)	Elucidate the N queens problem using backtracking to solve 4-Queens problem.	CO4	L2	7M
	b)	Illustrate the following sum of subsets problem instance where n=4, m=31, w(1:4)=(7,11,13,24).	CO4	L2	7M
		(OR)			
9	a)	Construct the portion of the state space tree generated by LC branch and bound of 0/1 knapsack problem for instance n = 4, (p1,p2,p3,p4) = (10,10,12,18), (w1,w2,w3,w4)=(2,4,6,9) and m=15.	CO4	L2	7M
	b)	How are P and NP problems related? With a neat diagram explain the relevance of NP-hard and NP-complete problems.	CO4	L3	7M

II/IV B.Tech (Regular\Supplementary) DEGREE EXAMINATION

July/August, 2023 — **Common to CB,CS,DS & IT Branches**

Fourth Semester — **Design And Analysis of Algorithms**

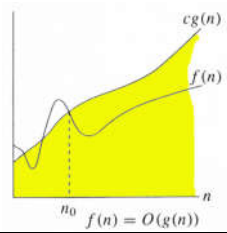
Time: Three Hours — **Maximum:** 70 Marks

Answer question 1 compulsory. **(14X1 = 14Marks)**

Answer one question from each unit. **(4X14=56 Marks)**

			CO	BL	M
1	a)	Define an algorithm. An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task.	CO1	L1	1
	b)	Define pseudocode. Pseudo code is a compact and informal high-level description of a computer programming language. In this method, we should typically describe algorithms as program, which resembles language like Pascal & C.	CO1	L2	1
	c)	Name the criteria to be satisfied by an algorithm. 1. Input 2. Output 3. Definiteness 4. Finiteness 5. Effectiveness	CO1	L1	1
	d)	How to calculate the time complexity? The time complexity of an algorithm is the amount of computer time it needs to run to compilation. The time $t(P)$ taken by a program P is the sum of the compile time and the run time(execution time).	CO1	L2	1
	e)	Identify the difference between divide and conquer and greedy method. In summary, the main difference between greedy algorithms and divide and conquer algorithms is in their approach to solving problems. Greedy algorithms make locally optimal choices at each step, while divide and conquer algorithms divide a problem into smaller subproblems and solve each subproblem independently.	CO2	L2	1
	f)	Label the time complexity of quick sort. Worst Case Analysis: $T(n) = O(n^2)$ Best Case Analysis: $T(n) = O (n \log n)$	CO2	L1	1

	g)	Why is it necessary to have auxiliary array in merge function? It is possible to merge the subarrays without using a second array, but this is extremely difficult to do efficiently and is not really practical. Merging the two subarrays into a second array, while simple to implement, presents another difficulty. The merge process ends with the sorted list in the auxiliary array.	CO2	L2	1
	h)	What is job sequencing with deadlines? We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned iff the job is completed by its deadline. To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing Jobs.	CO2	L1	1
	i)	Differentiate connected and biconnected components. Connected component (graph theory), a set of vertices in a graph that are linked to each other by paths. Connected component (topology), a maximal subset of a topological space that cannot be covered by the union of two disjoint open sets. A graph is Biconnected if it has no vertex such that its removal increases the number of connected components in the graph. And if there exists such a vertex then it is not Biconnected. (or) An articulation point of a graph is a vertex v such that when we remove v and all edges incident upon v , we break a connected component of the graph into two or more pieces. A connected graph with no articulation points is said to be biconnected.	CO3	L2	1
	j)	Where do we use dynamic programming? Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.	CO3	L2	1
	k)	State the principal of optimality. Principle of Optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.	CO3	L1	1

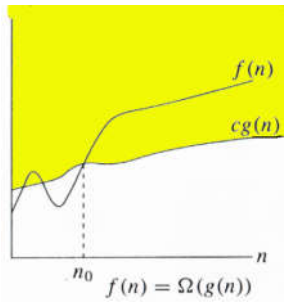
	l)	Summarise the advantages of backtracking. Backtracking has a brute-force nature; due to this reason, it can solve maximum problems. Backtracking problems are very intuitive to code. The step-by-step representation of the backtracking solution is straightforward to understand. You can easily debug backtracking code.	CO4	L2	1
	m)	When is a problem considered to be NP-hard? A problem L is NP-hard if and only if satisfiabiliy reduces to L (satisfiability α L). (or) A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.	CO4	L1	1
	n)	What is backtracking? Backtracking is one of the most general algorithm design techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. (or) Depth first node generation with bounding function is called backtracking.	CO4	L1	1
		Unit-I			
2	a)	What is asymptotic notation? Explain the usage of different types of notations. Algorithms perform **f(n)** basic operations to accomplish task. **Asymptotic** refers to study of function f as n approaches infinity. Asymptotic Analysis is used to compare two algorithms with running times f(n) and g(n), we need a rough measure that characterizes how fast each function grows. **1. Big-oh Notation (O) :** ***O*(*g*(*n*)) = { *f*(*n*) : there exists positive constants *c* and n_0, such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ }** • O(g(n)) is the set of functions with smaller or same order of growth as g(n). • g(n) is an **asymptotic upper bound** for f(n). Consider relavent examples. **Big-oh Visualization**  	CO1	L2	7M

2. Omega Notation (Ω) :

Ω(*g*(*n*)) = { *f* (*n*) : there exists positive constants *c* and n_0, such that 0 ≤ *cg*(*n*) ≤ *f*(*n*) for all *n* ≥ n_0 }

- Ω(g(n)) is the set of functions with larger or same order of growth as g(n).
- g(n) is an **asymptotic lower bound** for f(n).

Consider relavent examples.

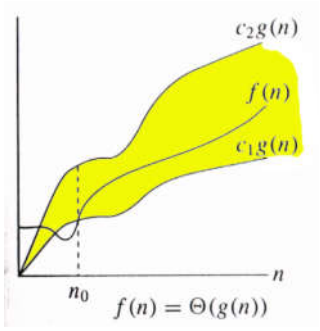


3. Theta Notation (Θ) :

Θ(*g*(*n*)) = { *f*(*n*) : there exists positive constants c_1, c_2, and n_0, such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \}$$

- Θ(g(n)) is the set of functions with the same order of growth as g(n).
- *g*(*n*) is an ***asymptotically tight bound*** for *f*(*n*).



For any two functions *g*(*n*) and *f*(*n*), *f*(*n*) = Θ(*g*(*n*)) iff *f*(*n*) = *O*(*g*(*n*)) and *f*(*n*) = Ω(*g*(*n*)).

Consider relavent examples.

4. Little-oh Notation (o) :

***o*(*g*(*n*)) = { *f*(*n*) : ∀ *c* > 0, ∃ n_0 > 0 such that ∀ *n* ≥ n_0, we have 0 ≤ *f*(*n*) < *cg*(*n*) }.**

f(*n*) becomes insignificant relative to *g*(*n*) as *n* approaches infinity:

$$\lim_{n \to \infty} [\, f(n) / g(n) \,] = 0$$

- *g*(*n*) is an ***upper bound*** for *f*(*n*) that is **not asymptotically tight.**

		5. Little omega Notation (*w*) : **$w(g(n)) = \{ f(n) : \forall c > 0, \exists n_0 > 0$ such that $\forall n \geq n_0$, we have $0 \leq cg(n) < f(n) \}$.** $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity: $\lim_{n \to \infty} [f(n) / g(n)] = \infty$. • $g(n)$ is a ***lower bound*** for $f(n)$ that is **not asymptotically tight.** Consider relavent examples.			
	b)	Label the master theorem and inspect the implementation of the master theorem. The Master Theorem applies to recurrences of the following form: **$T(n) = a\,T(n/b) + f(n)$** where **$a \geq 1$** and **$b > 1$** are constants and **f(n)** is an asymptotically positive function. There are 3 cases: 1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. 2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. 3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n. Consider relavent examples. **Master's Theorem for Decreasing Functions** **Highlights:** • Used to directly calculate the time complexity function of 'decreasing' recurrence relations of the form: ◦ $T(n) = aT(n-b) + f(n)$ • $f(n) = \theta(n^k)$ • The value of 'a' will decide the time complexity function for the 'decreasing' recurrence relation. For decreasing functions of the form $T(n) = aT(n-b) + f(n)$, where $f(n) = \theta(n^k)$ for example: • $T(n) = T(n-2) + 1$	CO1	L4	7M

- $T(n) = 2T(n-1) + n^2$

where:
n = input size (or the size of the problem)
a = count of subproblems in the decreasing recursive function
n-b = size of each subproblem (Assuming size of each subproblem is same)

Here a, b, and k are constants that satisfy the following conditions:

- a>0, b>0
- k>=0

Case 1) If a<1, then $T(n) = \theta(n^k)$

Case 2) If a=1, then $T(n) = \theta(n^{k+1})$

Case 3) If a>1, then $T(n) = \theta(n^{\frac{n}{b}} * f(n))$

Consider relavent examples.

(OR)

3 a) Show and describe the Pseudocode conventions with syntax and examples. CO1 L2 7M

Pseudo-code: Pseudo code is a compact and informal high-level description of a computer programming language. In this method, we should typically describe algorithms as program, which resembles language like Pascal & C.

Pseudo-Code Conventions:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces { and }.
3. An identifier begins with a **letter**. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

```
Node Record
{
  data type – 1   data-1;
          .
          .
          .
  data type – n  data – n;
  node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

   <Variable> := <expression>;

6. There are two Boolean values TRUE and FALSE. In order to produce these values we need

   Logical Operators AND, OR, NOT

   Relational Operators **<, <=, >, >=, =, !=**

7. The following looping statements are employed.

While Loop:

```
while < condition > do
{
        <statement-1>
                .
                .
        <statement-n>
}
```

For Loop:

```
for variable := value-1 to value-2 step step do
{
        <statement-1>
                .
                .
        <statement-n>
}
```

repeat-until:

```
repeat
{
        <statement-1>
                .
                .
        <statement-n>
}until<condition>
```

8. A conditional statement has the following forms.

```
if <condition> then <statement-1>

if <condition> then <statement-1>
else <statement-2>
```

Case statement:

```
case
{
        : <condition-1> : <statement-1>
                        .
                        .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

		9. Input and output are done using the instructions read & write. 10. There is only one type of procedure: Algorithm, the heading takes the form, **Algorithm Name (Parameter lists)**			
	b)	How to devise, validate, analyze, and test an algorithm? Discuss in detail. **Issues or study of Algorithm:** The study of algorithm includes many important and active areas of research. These are, • How to devise algorithms : creating an algorithm. • How to validate algorithms : checking correctness. • How to analyze algorithms : time and space complexity. • How to Testing a program : checking for error. Consider explanation of each one with relevant examples. After devising algorithm we need to convert it into program using programming language. A **Program** is the expression of an algorithm in a programming language.	CO1	L3	7M
		Unit-II			
4	a)	Illustrate the solution for executing the single source shortest path problem. Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions: 1. Is there a path from A to B? 2. If there is more than one path from A to B? Which is the shortest path? The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination.	CO2	L2	7M

In the problem we consider, we are given a directed graph G = (V, E), a weighting function **cost** for the edges of G, and a source vertex $\mathbf{v_0}$. The problem is to determine the shortest path from $\mathbf{v_0}$ to all the remaining vertices of G.

- It is assumed that all the weights associated with the edges are positive.
- The shortest path between $\mathbf{v_0}$ and some other node **v** is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure.

- One possibility is to build the shortest paths one by one.
- As an optimization measure we can use the sum of the lengths of all paths so far generated.
- For this measure to be minimized, each individual path must be of minimum length.
- If we have already constructed **i** shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.
- The greedy way to generate the shortest paths from $\mathbf{v_0}$ to the remaining vertices is to generate these paths in non-decreasing order of path length.
- First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.

In order to generate the shortest paths, we need to be able to determine,

1. The next vertex to which a shortest path must be generated and
2. A shortest path to this vertex.

Let **S** denotes the set of vertices (including $\mathbf{v_0}$ **)** to which the shortest paths have already been generated. For **w** not in **S**, let **dist[w]** be the length of the shortest path staring from $\mathbf{v_0}$**,** going through only those vertices that are in **S**, and ending at **w**.

Algorithm ShortestPaths(v, cost, dist, n)

// dist[j], $1 \leq j \leq n$, is set to the length of the shortest path from vertex v to
// vertex j in a digraph G with n vertices. dist[v] is set to zero. G is represented
// by its cost adjacency matrix, cost[1 : n, 1 : n]

{

```
for i := 1 to n do  //initialize set S
{
        S[ i ] := false; dist[ i ] := cost[ v, i ];
}
S[ v ] := true;  dist[ v ] := 0.0;          // put vertex v in set S
for num := 2 to n - 1 do
{       // Determine n - 1 paths from vertex v
        choose u from among those vertices not in S such that
        dist[ u ] is minimum;
        S[ u ] := true;             // put vertex u in set S
        for( each w adjacent to u with S[ w ] = false ) do
        {       //update distances
                if(  dist[ w ]  >  dist[ u ] + cost[ u, w ] ) then
                        dist[ w ]  =  dist[ u ] + cost[ u, w ];
        }
}
}
```

The set S is maintained as a bit array with S[i] $= \begin{cases} \mathbf{0} & if\ vertex\ i\ is\ not\ in\ S \\ \mathbf{1} & if\ vertex\ i\ is\ in\ S \end{cases}$

cost[i, j] $= \begin{cases} \textbf{weight of the edge} < i, j > & if\ i \neq j\ and\ < i,j > \in E(G) \\ +\infty & if\ i \neq j\ and\ < i,j > \notin E(G) \\ \textbf{non negitive number} & if\ i = j \end{cases}$

	b)	Consider the array, a[l:10] = (310,285,179,652,351,423,861,254,450,520). Show the merge sort algorithm and sort the above array using merge sort. Consider the array of ten elements a [1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520) The merge sort algorithm first divides the array as follows a[1:5] and a[6:10] and then a[1:3] and a[4:5] and a[6:8] and a[9:10] and then into a[1:2] and a[3:3] and a[6:7] and a[8:8] and finally it will look like a[1:1],a[2:2],a[3:3],a[4:4],a[5:5],a[6:6],a[7:7],a[8:8],a[9:9],a[10:10] Pictorially the file can now be viewed as (310|285|179|652,351|423,861,254,450,520) Elements a[1] and a[2] are merged as (285,310|179|652,351|423,861,254,450,520) And then a[3] is merged with a[1:2]	CO2	L3	7M

(179,285, 310 |652,351|423,861,254,450,520)next a[4] and a[5]

(179,285, 310 |351,652|423,861,254,450,520)and then a[1:3] and a[4:5]

(179,285, 310,351,652|423,861,254,450,520)

Repeated recursive calls are invoked producing the following array

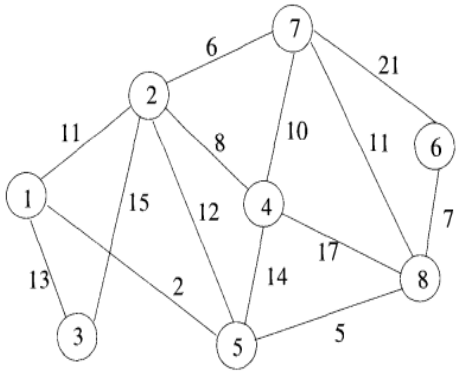
(179,285, 310 ,351,652|254,423, 450,520,861) and finally

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

```
Algorithm MergeSort(low,high)
//a[low:high] is a global array to be sorted
//b[low:high] is a temporary array used for merging
{
    if(low<high)then
    {
        mid:=floor(low+high)/2);
        MergeSort(low,mid);
        MergeSort(mid+1,high);
        Merge(low,mid,high);
    }
}

 Algorithm Merge(low,mid,high)
 //a[low:mid] and a[mid+1:high] are two sorted logical
 //partitions of global array a
 //These two partitions are merged to array b[low:high]
 //The elements are copied from b[low:high] to a[low:high]
 {
     h:=low;i:=low;j:=mid+1;
     while(h<=mid and j<=high) do
     {
         if(a[h]<=a[j])then
         {
            b[i]:=a[h];h:=h+1;
         }
         else
         {
            b[i]:=a[j];j:=j+1;
         }
         i:=i+1;
     }


     for k:= h to mid  do
     {
        b[i]:=a[k];i:=i+1;
     }
     for k:= j to high  do
     {
        b[i]:=a[k];i:=i+1;
     }
     for k:= low to high  do
        a[k]:=b[k];
     }

  }
```

(OR)

5 a) CO2 L3 7M



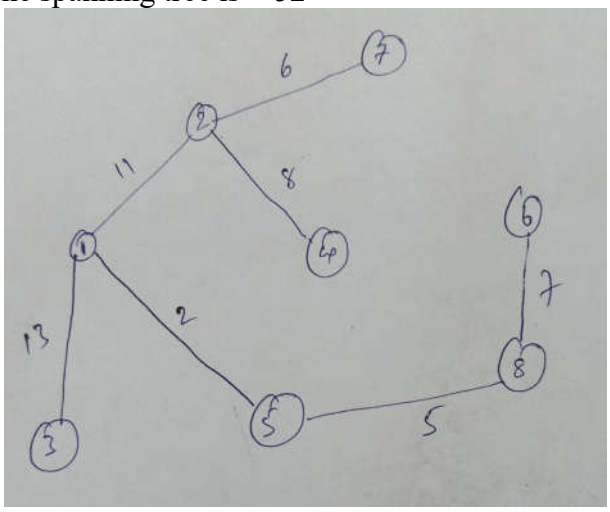
Compute a minimum cost spanning tree for the graph using Kruskal's algorithm.

Consider the edges in non-decreasing order of the edge costs.

E (G) = { (1, 5), (5, 8), (2, 7), (6, 8), (2, 4), (4, 7), (1, 2), (7, 8), (2, 5), (1, 3), (4, 5), (2, 3), (4, 8) }

Minimum-Cost Edge Selected	Action	Disjoint Sets	Cost Of the Spanning Tree
Initial	--	{ 1 } { 2 } { 3 } { 4 } { 5 } { 6 } { 7 } { 8 }	0.0
(1, 5)	Add	{ 1, 5 } { 2 } { 3 } { 4 } { 6 } { 7 } { 8 }	2
(5, 8)	Add	{ 1, 5, 8 } { 2 } { 3 } { 4 } { 6 } { 7 }	7
(2, 7)	Add	{ 1, 5, 8 } { 2, 7 } { 3 } { 4 } { 6 }	13
(6, 8)	Add	{ 1, 5, 6, 8 } { 2, 7 } { 3 } { 4 }	20
(2, 4)	Add	{ 1, 5, 6, 8 } { 2, 4, 7 } { 3 }	28
(4, 7)	Discard	{ 1, 5, 6, 8 } { 2, 4, 7 } { 3 }	28
(1, 2)	Add	{ 1, 2, 4, 5, 6, 7, 8 } { 3 }	39
(7, 8)	Discard	{ 1, 2, 4, 5, 6, 7, 8 } { 3 }	39
(2, 5)	Discard	{ 1, 2, 4, 5, 6, 7, 8 } { 3 }	39
(1, 3)	Add	{ 1, 2, 3, 4, 5, 6, 7, 8 }	52
(4, 5)	Discard	{ 1, 2, 3, 4, 5, 6, 7, 8 }	52
(2, 3)	Discard	{ 1, 2, 3, 4, 5, 6, 7, 8 }	52
(4, 8)	Discard	{ 1, 2, 3, 4, 5, 6, 7, 8 }	52

minimum cost of the spanning tree is = 52



	b)	Demonstrate the strategy followed by the divide and conquer approach.	CO2	L2	7M

GENERAL METHOD:

- Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, **1 < k ≤ n**, yielding '**k**' sub problems.
- These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.
- For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
- DAndC (Algorithm) is initially invoked as DandC(P), where 'P' is the problem to be solved.
- Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting.
- If this so, the function 'S' is invoked.
- Otherwise, the problem P is divided into smaller sub problems.
- These sub problems P_1, P_2 ,…, P_k are solved by recursive application of DAndC.
- Combine is a function that determines the solution to P using the solutions to the 'k' sub problems.
- If the size of '**P**' is **n** and the sizes of the '**k**' sub problems are $\mathbf{n_1, n_2,\ldots,n_k}$, respectively, then the computing time of D And C is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where T(n) is the time for DAndC on any input of size 'n'.

g(n) is the time of compute the answer directly for small inputs.

f(n) is the time for dividing P & combining the solution to sub problems.

Control Abstraction for DAndC Algorithm:

We can write a control abstraction that mirrors the way an algorithm based on DAndC will look. By a control abstraction we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

```
Algorithm DAndC( P )
{
    if small( P ) then  return S( P );
    else
    {
        divide P into smaller instances P1, P2,…., Pk,  k ≥ 1;
        apply DAndC to each of these sub problems;
        return combine( DAndC( P1 ), DAndC( P2 ) ,……., DAndC( Pk ) );
    }
}
```

The complexity of many divide-and-conquer algorithms is given by recurrences of the form,

$$T(n) = \begin{cases} T(1) & if\ n = 1 \\ a\,T\left(n/b\right) + f(n) & if\ n > 1 \end{cases}$$

Where a & b are known constants.

We assume that T(1) is known & 'n' is a power of b i.e., $n = b^k$.

Unit-III

6	a)	Elucidate the mechanism for solving the 0/1 knapsack problem using dynamic programming.	CO3	L2	7M

The terminology and notation used in this section is the same as in section 5.1. A solution to the knapsack problem may be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves deciding which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \ldots, x_1$. Following a decision on x_n we may be in one of two possible states: the capacity remaining in the knapsack is M and no profit has accrued or the capacity remaining is $M - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principle of optimality holds.

Let $f_j(X)$ be the value of an optimal solution to KNAP$(1, j, X)$. Since the principle of optimality holds, we obtain

$$f_n(M) = \max\{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\} \quad (5.13)$$

For arbitrary $f_i(X)$, $i > 0$, Equation (5.13) generalizes to

$$f_i(X) = \max\{f_{i-1}(X), f_{i-1}(X - w_i) + p_i\} \quad (5.14)$$

Equation (5.14) may be solved for $f_n(M)$ by beginning with the knowledge $f_0(X) = 0$ for all X and $f_i(x) = -\infty, x < 0$. $f_1, f_2, \ldots, f_n$ may be successively computed using (5.14).

(P_j, W_j) where W_j is a value of X at which f_i takes a jump. $P_j = f_i(W_j)$. If there are r jumps then we need to know r pairs (P_j, W_j), $1 \le j \le r$. For convenience we introduce the pair $(P_0, W_0) = (0, 0)$. If we assume $W_j < W_{j+1}$, $0 \le j < r$ then from (5.14) it follows that $P_j < P_{j+1}$. Further, $f_i(X) = f_i(W_i)$ for all X such that $W_j \le X < W_{j+1}$, $0 \le j < r$. $f_i(X) = f_i(W_r)$ for all X, $X \ge W_r$. If S^{i-1} is the set of all pairs for f_{i-1} (including(0, 0)) then the set S_1^i of all pairs for $g_i(X) = f_{i-1}(X - w_i) + p_i$ is obtained by adding to each pair in S^{i-1} the pair (p_i, w_i).

$$S_1^i = \{(P, W) \mid (P - p_i, W - w_i) \in S^{i-1}\} \quad (5.15)$$

S^i may now be obtained by merging together S^{i-1} and S_1^i. This merge corresponds to taking the maximum of the two functions $f_{i-1}(X)$ and $f_{i-1}(X - w_i) + p_i$ in Equation (5.14). Thus, if one of S^{i-1} and S_1^i has a pair (P_j, W_j) and the other has a pair (P_k, W_k) and $P_j \le P_k$ while $W_j \ge W_k$ then the pair (P_j, W_j) is discarded. This is required by (5.14). $f_i(W_j) = \max\{P_j, P_k\} = P_k$.

S^{i+1} contains two tuples (P_j, W_j) and (P_k, W_k) with the property that $P_j \le P_k$ and $W_j \ge W_k$ then the tuple (P_j, W_j) may be discarded. This is so because for any decision sequence $x_{i+2}, \ldots, x_n$ with the property $W_j + \sum_{l=i+2}^{n} w_l x_l \le M$, it is the case that $W_k + \sum_{l=i+2}^{n} w_l x_l \le M$ and $P_k + \sum_{l=i+2}^{n} p_l x_l \ge P_j + \sum_{l=i+2}^{n} p_l x_l$. Hence, (P_j, W_j) cannot lead to a solution better than the best obtainable from (P_k, W_k). This discarding rule is identical to the purging rule stated above. Discarding or purging rules are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

If $(P1, W1)$ is the last tuple in S^n, a set of 0/1 values for the x_is such that $\sum p_i x_i = P1$ and $\sum w_i x_i = W1$ may be determined by carrying out a search through the S^is. We may set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$ then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we may set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1}. This may be done by using the argument used to determine x_n.

```
1  Algorithm DKP(p, w, n, m)
2  {
3      S^0 := {(0, 0)};
4      for i := 1 to n - 1 do
5      {
6          S_1^{i-1} := {(P, W) | (P - p_i, W - w_i) ∈ S^{i-1} and W ≤ m};
7          S^i := MergePurge(S^{i-1}, S_1^{i-1});
8      }
9      (PX, WX) :=last pair in S^{n-1};
10     (PY, WY) := (P' + p_n, W' + w_n) where W' is the largest W in
11         any pair in S^{n-1} such that W + w_n ≤ m;
12     // Trace back for x_n, x_{n-1}, ..., x_1.
13     if (PX > PY) then x_n := 0;
14     else x_n := 1;
15     TraceBackFor(x_{n-1}, ..., x_1);
16 }
```

Algorithm 5.6 Informal knapsack algorithm

	b)	Differentiate the implementation of depth first traversal and breadth first traversal. **BFS:** • In BFS, start at vertex **v** and mark it as having been reached (visited). The vertex at this time said to be unexplored. • A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. • All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been explored. • The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. • The list of unexplored vertices operates as a queue.	CO3	L3	7M

```
1   Algorithm BFS(v)
2   // A breadth first search of G is carried out beginning
3   // at vertex v. For any node i, visited[i] = 1 if i has
4   // already been visited. The graph G and array visited[ ]
5   // are global; visited[ ] is initialized to zero.
6   {
7       u := v; // q is a queue of unexplored vertices.
8       visited[v] := 1;
9       repeat
10      {
11          for all vertices w adjacent from u do
12          {
13              if (visited[w] = 0) then
14              {
15                  Add w to q; // w is unexplored.
16                  visited[w] := 1;
17              }
18          }
19          if q is empty then return; // No unexplored vertex.
20          Delete u from q; // Get first unexplored vertex.
21      } until(false);
22  }
```

Algorithm 6.5 Pseudocode for breadth first search

DFS:

- A DFS of a graph differs from a BFS in that the exploration of a vertex **v** is suspended as soon as a new vertex is reached.
- At this time the exploration of the new vertex **u** begins.
- When this new vertex has been explored, the exploration of **v** continues.
- The search terminates when all reached vertices have been fully explored.

```
1   Algorithm DFS(v)
2   // Given an undirected (directed) graph G = (V, E) with
3   // n vertices and an array visited[ ] initially set
4   // to zero, this algorithm visits all vertices
5   // reachable from v. G and visited[ ] are global.
6   {
7       visited[v] := 1;
8       for each vertex w adjacent from v do
9       {
10          if (visited[w] = 0) then DFS(w);
11      }
12  }
```

Algorithm 6.7 Depth first search of a graph

		(OR)			
7	a)	Find the longest common subsequence of X and Y using dynamic programming. X=<ABCDAF>, Y=<ACBCF>	CO3	L2	7M

$$c[\,i,j\,] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ c[\,i-1, j-1\,] + 1 & if\ i,j > 0\ and\ x_i = y_j \\ \max(\,c[\,i-1, j\,], c[\,i.j-1\,]) & if\ i,j > 0\ and\ x_i \neq y_j \end{cases}$$

X = ⟨ ABCDAF ⟩ and Y = ⟨ ACBCF ⟩

m = 6 and n = 5

C[i, 0] = 0 for all $0 \le i \le 6$

C[0, j] = 0 for all $0 \le j \le 5$

Compare X_i and Y_j where i = 1 and $1 \le j \le 5$

X[1] = Y[1] then, C[1, 1] = C[0, 0] + 1 = 0 + 1 = 1

X[1] ≠ Y[2] and max(C[0, 2], C[1, 1]) is C[1, 1] then, C[1, 2] = C[1, 1] = 1

X[1] ≠ Y[3] and max(C[0, 3], C[1, 2]) is C[1, 2] then C[1, 3] = C[1, 2] = 1

X[1] ≠ Y[4] and max(C[0, 4], C[1, 3]) is C[1, 3] then C[1, 4] = C[1, 3] = 1

X[1] ≠ Y[3] and max(C[0, 5], C[1, 4]) is C[1, 4] then C[1, 5] = C[1, 4] = 1

Similarly compare X_i and Y_j ,

When i = 2 and $1 \le j \le 5$

When i = 3 and $1 \le j \le 5$

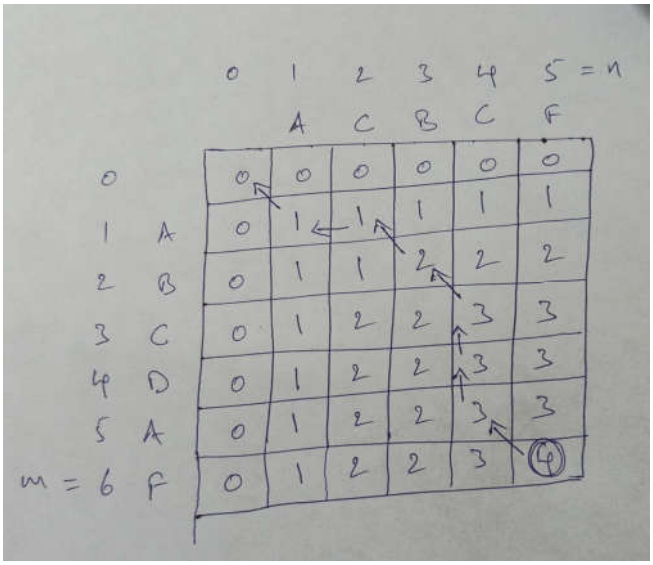
When i = 4 and $1 \le j \le 5$

When i = 5 and $1 \le j \le 5$

When i = 6 and $1 \le j \le 5$

Extracting the Actual Sequence:

Extracting the final LCS is done by using the back pointers stored in b[0 : m, 0 : n]. Intuitively b[i, j] = ADDXY means that X[I] and Y[j] together form the last character of the LCS. So we take this common character, and continue with entry b[i−1, j−1] to the northwest (↖). If b[i, j] = SKIPX, then we know that X[i] is not in the LCS, and so we skip it and go to b[i−1, j] above us (↑). Similarly, if b[i, j] = SKIPY, then we know that Y[j] is not in the LCS, and so we skip it and go to b[i, j−1]to the left (←). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

		Finally we get C[6, 5] = 4 as the length of the LCS and LCS = ‹ ABCF › 			
	b)	Inspect the traveling salesperson problem to find a tour of minimum cost. Let G(V, E) be a directed graph with edge cost c_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$, if < i, j > $\notin$ E. Let $\|V\| = n$ and assume n > 1. • A tour of G is a directed simple cycle that includes every vertex in V. • The cost of a tour is the sum of the costs of the edges on the tour. • The traveling salesman problem is to find a tour of minimum cost. Without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. • Every tour consists of an edge < 1, k > for some k $\in$ V - { 1 } and a path from vertex k to vertex 1. • The path from vertex k to vertex 1 goes through each vertex in V - { 1, k } exactly once. Let **g**(i, S) be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function **g**(1, V - { 1 }) is the length of an optimal sales person tour. From the principle of optimality it follows that, $$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{----(1)}$$	CO3	L4	7M

		Generalizing equation (1) , we obtain (for $i \notin S$) $g(i, S) = \min_{j \in S}\{c_{ij} + g(j, S - \{j\})\}$ ----(2) Equation (1) can be solved for **g**(1, V - { 1 }) if we know **g**(k, V - { 1, k }) for all choices of k. The **g** values can be obtained by using equation (2). Clearly, **g**(i, Ø) = $c_{i1}, 1 \le i \le n$. Hence we can use equation (2) to obtain **g(i, S)** for all S of size 1. Then we can obtain **g(i, S)** for $\|S\| = 2$, and so on. • When $\|S\| < n - 1$, the values of i and S for which **g(i, S)** is needed are such that $i \neq 1$, $1 \notin S$ and $i \notin S$.			
		Unit-IV			
8	a)	Elucidate the N queens problem using backtracking to solve 4-Queens problem. The n-queens problem is place n-queens on an n x n chessboard so that no two queens attack i.e., no two queens are on the same row, or column, or diagonal. If we imagine the squares of the chessboard being numbered as the indices of the two dimensional array a[1 : n, 1 : n], then we observe that every element on the same diagonal which runs from the upper left to the lower right has the same "row – column" value. Also, every element on the same diagonal which goes from the upper right to the lower left has the same "row + column" value. Suppose two queens are placed at positions (i, j) and (k, *l*). Then by the above they are on the same diagonal only if $i - j = k - l$ **or** $i + j = k + l$ The first equation implies $j - l = i - k$ The second equation implies $j - l = k - i$ Therefore two queens lie on the same diagonal if and only if $\|j - l\| = \|i - k\|$. The total number of nodes in the 8-queens state space tree is $1 + \sum_{j=0}^{7}\left[\prod_{i=0}^{j}(8 - i)\right] = 69,281$	CO4	L2	7M

- All solutions to n-queens problem can therefore be represented as n-tuples (x_1, …., x_n), where x_i is the column on which queen I is placed.
- Explicit constraints S_i = { 1, 2, 3, 4, ……, n }, $1 \leq i \leq n$
- Implicit constraints for this problem are that
    - No two x_i's can be the same and
    - No two queens can be on the same diagonal

Algorithm place (k, i)

// Return true if a queen can be placed in k^{th} row and i^{th} column. Otherwise // it returns false. x[ ] is a global array whose first (k-1) values have been // set. abs(r) returns the absolute value of r.

```
{
        for j := 1 to k-1 do
        {
                if ( ( x [ j ] = i )          // two in the same column.
                        or ( abs( x [ j ] – i ) = abs( j – k ) ) ) then
                                              // or in the same diagonal
                 return false;
        }
        return true;
}
```

Algorithm Nqueens(k, n)

// Using backtracking, this procedure prints all possible placements of n

// queens on an n x n chessboard so that they are non-tracking.

```
{
        for i :=1 to n do
        {
                If(  place ( k, i ) ) then
                  {
                        x[ k ] := i;
                        if ( k = n ) then write( x[ 1 : n ] );
                        else Nqueens(k+1,n);
                }
        }
}
```

Example: 4-queens.

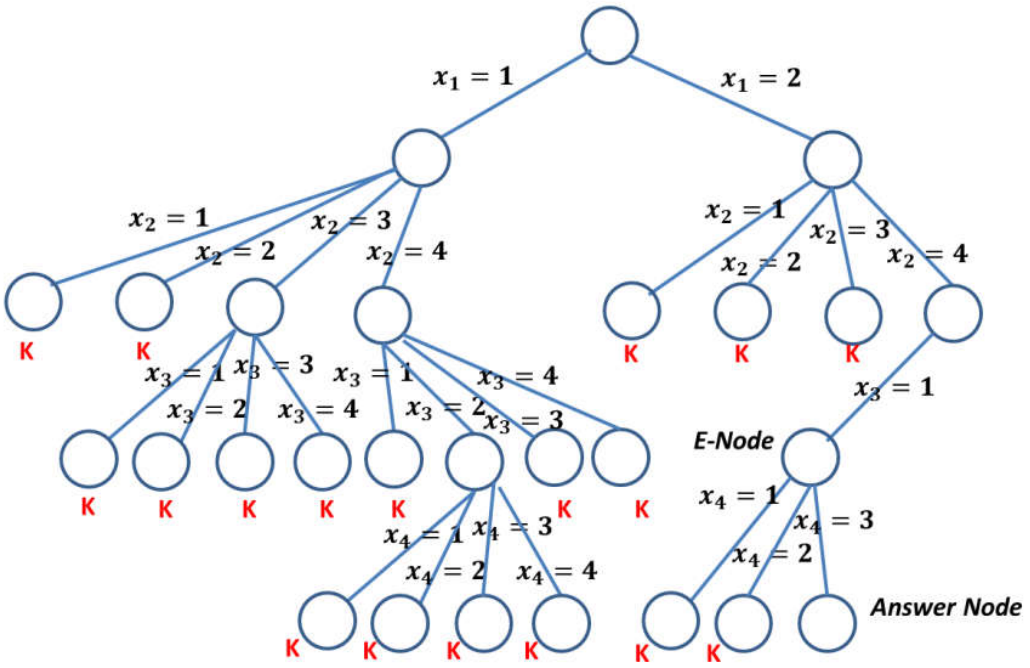
Two possible solutions are

	Q		
			Q
Q			
		Q	

Solutin-1
(2, 4, 1, 3)

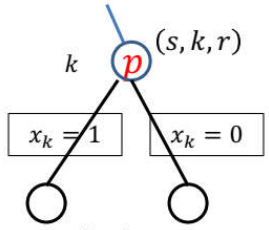
		Q	
Q			
			Q
	Q		

Solution 2
(3, 1, 4, 2)



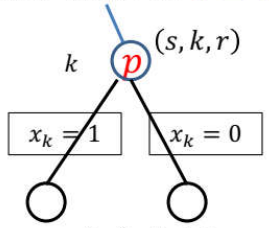
	b)	Illustrate the following sum of subsets problem instance where n=4, m=31, w(1:4)=(7,11,13,24). We can design the state space tree in such a way that: For a node at level k we create the children with $x_k = 1$ *and* $x_k = 0$, which indicate that weight wk is included or not in the solution. • To avoid summation of weights at each node, in the state space tree, we maintain a three tuple (s, k, r) associated with each node. • k is the level of the node. • s is the sum of integers/weights included in the partial solution represented by the node. • *At root node* $s = 0$.	CO4	L2	7M

- r is the sum of the remaining integers(weights) to be considered in building the solution.
- At the root node, $r = \sum_{i=1}^{n} w_i$



$(s = s + w_k, k + 1, r = r - w_k)$ $(s, k + 1, r = r - w_k)$

- A node defined by $(x_1, x_2, \ldots x_k)$ is an answer node iff $s = m$
- $B(x_1, x_2, \ldots x_k) = true\ iff$
  - $s \leq m\ for\ x_k = 1$
  - $s + r \geq m\ for\ x_k = 0$



$(s = s + w_k, k + 1, r = r - w_k)$ $(s, k + 1, r = r - w_k)$

$s \leq m$ $s + r \geq m$

Step-1

$n = 4, (w_1, w_2, w_3, w_4) = (7,11,13,24), m = 31$

0,1,55

Step-2

$n = 4, (w_1, w_2, w_3, w_4) = (7,11,13,24), m = 31$

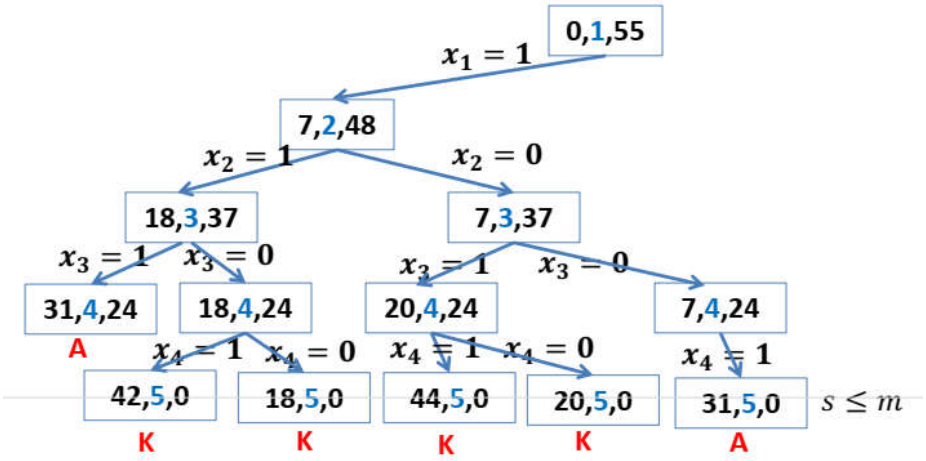
0,1,55

$x_1 = 1$

$s \leq m$ **7,2,48**

And soon finally we get,

$n = 4, (w_1, w_2, w_3, w_4) = (7,11,13,24), m = 31$

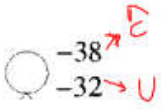


Solution vectors are (1, 1, 1, 0) and (1,0, 0, 1)

(OR)

9	a)	Construct the portion of the state space tree generated by LC branch and bound of 0/1 knapsack problem for instance n = 4, (p1,p2,p3,p4) = (10,10,12,18), (w1,w2,w3,w4)=(2,4,6,9) and m=15.	CO4	L2	7M

Step-1 : C(1) and U(1) calculation



Upper = -32

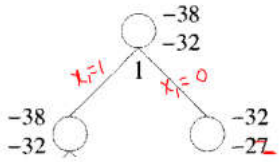
$\min\{ U, U(X) + \epsilon \}$

$\hat{c}(1) = -(10+10+12+(3*(18/9))) = -38$

$U(1) = -(10+10+12) = -32$

Upper number = $\hat{c}$
Lower number = u

Step-2 : C(2) and U(2) & C(3) and U(4) calculation



Upper = -32

$\min\{ U, U(X) + \epsilon \}$

Upper number = $\hat{c}$
Lower number = u

and soon finally we get,

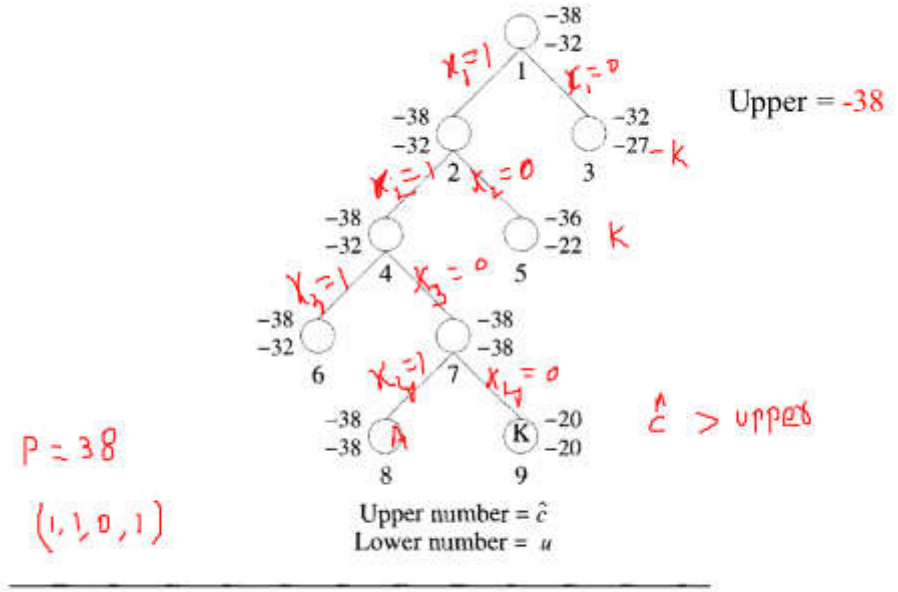


Figure 8.8 LC branch-and-bound tree for Example 8.2

Solution vector is (1, 1, 0, 1)
Maximum profit = 38

	b)	How are P and NP problems related? With a neat diagram explain the relevance of NP-hard and NP-complete problems.	CO4	L3	7M

An algorithm A is of polynomial complexity if there exists a polynomial P( ) such that the computing time of A is O(P(n)) for every input of size n.

- The P in the P class stands for **Polynomial Time**.
- It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

- The NP in NP class stands for **Non-deterministic Polynomial Time**.
- It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

P problems : Problems whose solutions are bounded by polynomials of small degree.

- P Problems can be solved and verified in polynomial time.
- For input size ***n***, if worst-case time complexity of an algorithm is $O(n^k)$, where ***k*** is a constant, the algorithm is a polynomial time algorithm.
- **Examples of P Problems:** Insertion sort, Merge sort, Linear search, Matrix multiplication, Finding minimum and maximum elements from the array, Single Source Shortest Path, Minimum Spanning Tree etc.
- Ex : O(logn), O(n), O(nlogn), O(mn)

NP problems : Problems whose best-known algorithms are non-polynomial.

- The solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be **verified** in polynomial time.
- **Examples of NP problems:** Knapsack problem ($O(2^n)$), Travelling salesman problem (O(n!)), Tower of Hanoi ($O(2^n - 1)$), Hamiltonian cycle (O(n!)).
- Ex : O($n^2\ 2^n$), O($2^{n/2}$)

The classes NP-hard and NP- complete

Let L1 and L2 be problems. Problem L1 **reduces to** L2 (also written L1 α L2) if and only if there is a way to solve L1 by a deterministic polynomial time using a deterministic algorithm that solves L2 in polynomial time.

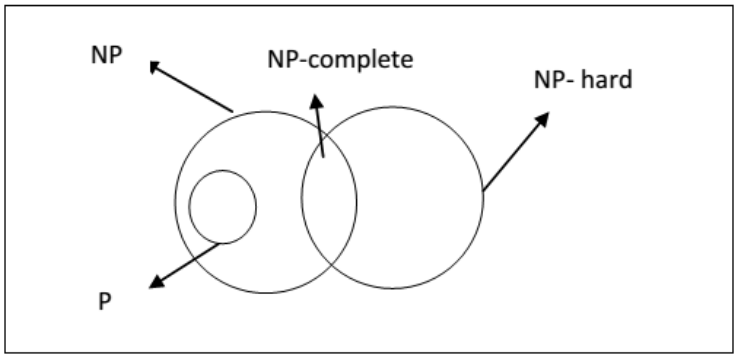
Two problems L1 and L2 are said to be **polynomial equivalent** if and only if L1 α L2 and L2 α L1

- A problem L is **NP-hard** if and only if satisfiabiliy reduces to L (satisfiability α L).

  Ex : Optimization problems.
- A problem L is **NP-complete** if and only if L is NP-hard and L Є NP.

  Ex : Decision problems

Optimization problems can't be NP- complete whereas decision problems can.



A problem that is NP-complete has the property that it can be solved in polynomial time if and only if all other NP-complete problems can also be solved in polynomial time.

If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.