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

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

1. a) **What are the characteristics of an algorithm?**

Input, Output, Definiteness, Finiteness, Effectiveness

b) **Define order of an algorithm and the need to analyze the algorithm?**

In general the order of an algorithm translates to the efficiency of an algorithm. Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

c) **Define time complexity.**

The time complexity of an algorithm is the amount of computer time it needs to run to Compilation.

d) **What is average case efficiency of an algorithm?**

In computational complexity theory, the average-case complexity of an algorithm is the amount of some computational resource (typically time) used by the algorithm, averaged over all possible inputs. Average Case Efficiency - average comparisons between minimum no. of comparisons and maximum no.

e) **Give an example showing that Quicksort is not a stable sorting algorithm.**

Quick Sort is not stable because it swaps non-adjacent elements. The most succinct example: Given [2, 2, 1], the '2' values will not retain their initial order.

Quick Sort is an unstable algorithm because we do swapping of elements according to pivot's position (without considering their original positions).

f) **Differentiate between greedy method and Dynamic programming.**

The essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated.

g) **What is divide and conquer method?**

Given a function to compute on “n” inputs the divide-and-conquer strategy suggests splitting the inputs into “k” distinct subsets, $1 < k \leq n$, 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.

h) **List the features of dynamic programming.**

1. Sub problems overlap
2. Substructure has optimal property

i) **What is a spanning tree?**

Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together.

j) **What is branch and bound?**

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.

k) **Define 0/1 knapsack problem.**

Given n objects and a knapsack. Object i has weight wi and profit pi and the knapsack has a capacity m. The objective is to maximize the total profit by selecting objects (no fractions allowed) without exceeding the total capacity of the knapsack.

$maximize\ \Sigma_{1\le i\le n}\ p_i x_i$

- $Subject\ to\ \Sigma_{1\le i\le n}\ w_i x_i \le m$
- $x_i = 0\ or\ 1\ and\ 1 \le i \le n$
- x_i is 0, the object i is not selected.
- x_i is 1, the object i is selected.

l) **What are the two types of constraints used in backtracking?**

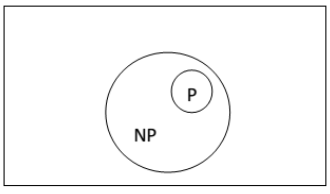
1. Explicit constraints are rules that restrict each xi to take on values only from a given set.
2. Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.

m) **Define Cook's Theorem.**

Satisfiability is in P if and only if P = NP

n) **Explain the relation between P and NP problems.**

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.



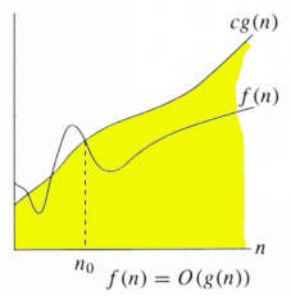
2. a) **Explain the various Asymptotic notations used in algorithmic analysis with examples.** CO1 7M

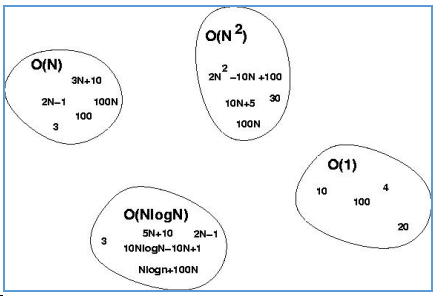
1. Big-oh Notation (O) :

$O(g(n)) = \{ f(n)$: 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).

Big-oh Visualization

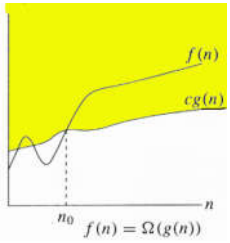




2. Omega Notation (Ω) :

$\Omega(g(n)) = \{ f(n)$: there exists positive constants c and n_0, such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

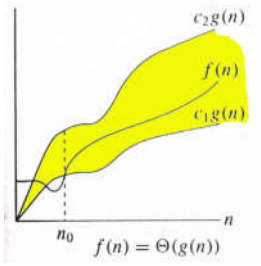
- Ω(g(n)) is the set of functions with larger or same order of growth as g(n).
- g(n) is an asymptotic lower bound for f(n).



3. Theta Notation (Θ) :

$\Theta(g(n)) = \{ f(n)$: there exists positive constants c_1, c_2, and n_0, such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ 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 $f(n)$, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

4. Little-oh Notation (o) :

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) < cg(n)\}.$$

$f(n)$ 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*) :

$$w(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq cg(n) < f(n)\}.$$

$f(n)$ 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.

b) **Write an algorithm to find the factorial of a number and find the time complexity of the algorithm** CO1 7M

The factorial of a number is defined as:

f(n) = n * f(n-1) → for all n >0
f(0) = 1 → for n = 0

```
Algorithm factorial(n) {
   if n is 0
       return 1
   return n * factorial(n-1)
}
```

Time complexity

If we look at the pseudo-code again, added below for convenience. Then we notice that:

factorial(0) is only comparison (1 unit of time), factorial(n) is 1 comparison, 1 multiplication, 1 subtraction and time for factorial(n-1)

From the above analysis we can write:

T(n) = T(n — 1) + 3
T(0) = 1
T(n) = T(n-1) + 3
= T(n-2) + 6
= T(n-3) + 9
= T(n-4) + 12
= ...
= T(n-k) + 3k

as we know T(0) = 1

we need to find the value of k for which n - k = 0, k = n

T(n) = T(0) + 3n , k = n

= 1 + 3n

that gives us a time complexity of O(n)

(OR)

3. a) **What is the difference between Big 'O' notation and little 'o' notation? When do we use Theta (Θ) notation. Explain with examples.** CO1 7M

they are both asymptotic notations that specify upper-bounds for functions and running times of algorithms.

However, the difference is that **big-O may be asymptotically tight** while **little-o makes sure that the upper bound isn't asymptotically tight**.

Let's read on to understand what exactly it means to be asymptotically tight.

2. Mathematical Definition

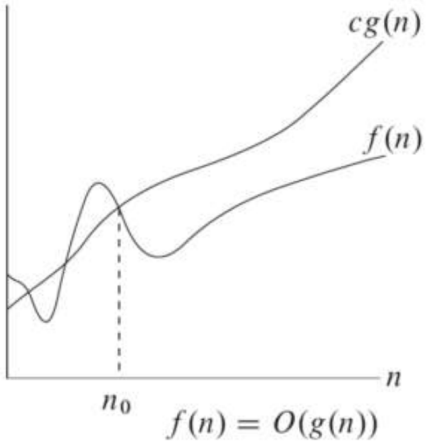
Big-O and little-o notations have very similar definitions, and **their difference lies in how strict they are regarding the upper bound** they represent.

2.1. Big-O

For a given function $g(n)$, $O(g(n))$ is defined as:

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.

So $O(g(n))$ is a set of functions that are, after n_0, smaller than or equal to $g(n)$. The function's behavior before n_0 is unimportant since big-O notation (also little-o notation) analyzes the function for huge numbers. As an example, let's have a look at the following figure:



Here, $f(n)$ is only one of the possible functions that belong to $O(g(n))$. Before n_0, $f(n)$ is not always smaller than or equal to $g(n)$, but after n_0, it never goes above $g(n)$.

The equal sign in the definition represents the concept of asymptotical tightness, meaning that when n gets very large, $f(n)$ and $g(n)$ grow at the same rate. For instance, $3n^3 = O(n^3)$ satisfies the equal sign, hence it is asymptotically tight, while $3n = O(n^3)$ is not.

For more explanation on this notation, look at an introduction to the theory of big-O notation (/cs/big-o-notation).

2.2. Little-o

Little-o notation is used to denote an upper-bound that is not asymptotically tight. It is formally defined as:

$o(g(n)) = \{f(n)$: for any positive constant c, there exists positive constant n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Note that in this definition, the set of functions $f(n)$ are strictly smaller than $cg(n)$, meaning that little-o notation is a stronger upper bound than big-O notation. In other words, the little-o notation does not allow the function $f(n)$ to have the same growth rate as $g(n)$.

Intuitively, this means that as the n approaches infinity, $f(n)$ becomes insignificant compared to $g(n)$. In mathematical terms:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

In addition, the inequality in the definition of little-o should hold for any constant c, whereas for big-O, it is enough to find some c that satisfies the inequality.

If we drew an analogy (https://mitpress.mit.edu/books/introduction-algorithms-third-edition) between asymptotic comparison of $f(n)$ and $g(n)$ and the comparison of real numbers a and b, we would have $f(n) = O(g(n)) \approx a \leq b$ while $f(n) = o(g(n)) \approx a < b$.

3. Examples

Let's have a look at some examples to make things clearer.

For $f(n) = 2n + 3$, we have:

- $\boldsymbol{f(n) = O(n)}$ but $\boldsymbol{f(n) \neq o(n)}$
- $f(n) = O(n^2)$ and $f(n) = o(n^2)$
- $f(n) = O(n^3)$ and $f(n) = o(n^3)$

In general, for $f(n) = a_x n^x + a_{x-1} n^{x-1} + ... + a_0$, we will have:

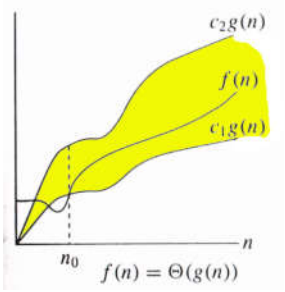
- $\boldsymbol{f(n) = O(n^x)}$ but $\boldsymbol{f(n) \neq o(n^x)}$
- $f(n) = O(n^{x+1})$ and $f(n) = o(n^{x+1})$
- $f(n) = O(n^{x+2})$ and $f(n) = o(n^{x+2})$

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Θ(g(n)) = { f(n) : there exists positive constants c_1, c_2, and n_0, such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }

- 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 $f(n)$, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

b) **How many cases are there under Master's Theorem? Explain any two of them with example.** CO1 7M

The Master Theorem applies to recurrences of the following form:

T (n) = a T (n/b) + f (n)

where **a ≥ 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.

Example-Case 1

- $T(n) = 4T\left(\frac{n}{2}\right) + n$
- $a = 4, b = 2, k = 1$
- $\log_b a > k$
- $T(n) = \theta(n^2)$

Example-Case 2

- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
- $a = 4, b = 2, k = 2$
- $\log_b a = k, p = 0$
- $T(n) = \theta(n^2 \log n)$

Example-Case 3

- $T(n) = 3T\left(\frac{n}{2}\right) + n^2$
- $a = 3, b = 2, k = 2$
- $\log_b a > k$
- $T(n) = \theta(n^2)$

4. a) **Explain Quick sort algorithm and simulate it for the following data: 20, 35, 10, 16, 54, 21, 25** CO2 7M

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
- In merge sort, the file a[1 : n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
- In Quick sort, the division into two sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by rearranging the elements in a[1 : n] such that a[i] ≤ a[j] for all i between 1 & m and all j between (m+1) & n for some m, 1 ≤ m ≤ n.
- Thus the elements in a[1 : m] & a[m+1 : n] can be independently sorted.
- No merge is needed. This rearranging is referred to as partitioning.
- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m : p-1]
- It is assumed that a[p] >= a[m] and that a[m] is the partitioning element. If m=1 & p-1 = n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1 : n]
- The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in the set are better in practice.
- The function interchange (a, i, j) exchanges a[i] with a[j].

Algorithm QuickSort(p, q)

//Sort the elements a[p],…..,a[q] which resides in the global array a[1 : n] into ascending //order; a[n+1] is considered to be defined and must be >= all the elements in a[1 : n]

```
{
    if( p < q ) then                        // If there are more than one element
    {
       j = Partition( a, p, q+1 );      //'j' is the position of the partitioning element.
       Quicksort( p, j-1 );
       Quicksort( j+1, q );
    }
}
```

Algorithm Partition(a, m, p)

//within a[m], a[m+1],…..,a[p-1] the elements are rearranged in such a manner that if

//initially t = a[m], then after completion a[q] = t for some q between m and p-1, a[k] <= t //for m <= k < q, and a[k] >= t for q < k < p , q is returned. Set a[p] = infinite.

```
{
        v = a[ m ]; i = m; j = p;
        repeat
        {
            repeat
                i = i + 1;
            }until( a[ i ] ≥ v);
            repeat
                j = j - 1;
            }until( a[ j ] ≤ v );
            if ( i < j ) then interchange(a, i. j );
        }until( i ≥ j );
        a[ m ] = a[ j ]; a[ j ] = v;
        retun j;
}
```

Algorithm Interchange(a, i, j)

```
//Exchange a[ i ] with a[ j ]
{
        temp  =  a[ i ];
        a[ i ]  =  a[ j ];
        a[ j ]  =  temp;
}
```

(1) (2) (3) (4) (5) (6) (7) (8)
20, 35, 10, 16, 54, 21, 25 +∞

20, 16, 10, 35, 54, 21, 25 +∞

10, 16, **20**, 35, 54, 21, 25 +∞

(4) (5) (6) (7) (8)
35, 54, 21, 25 +∞

35, 25, 21, 54 +∞

21, 25, **35**, 54 +∞

(1) (2) (3) (4) (5) (6) (7)

10, 16, 20, 21, 25, 35, 54

b) **Illustrate Merge sort algorithm and discuss Time complexity in both worst case and average case.** CO2 7M

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Algorithm 3.7 Merge sort

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

(OR)

5. a) **Solve the following greedy fractional knapsack problem.** CO2 7M

Knapsack problem instance n = 4, Knapsack capacity m = 15, (P1, P2, P3, P4) = (10, 10, 12, 18) and (W1, W2, W3, W4) = (2, 4, 6, 9).

When one applies the greedy method to the solution of the knapsack problem, there are different measures one can attempt to optimize when determining which object to include next. Once an optimization measure has been chosen, the greedy method suggests choosing objects for inclusion into the solution in such away that each choice optimizes the measure at that time.

We design an algorithm to achieve a balance between the rate at which profit and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit capacity used. This means that objects are considered in order of the ratio $\mathbf{P_i / W_i}$.

If $\mathbf{P_1 / W_1 \geq P_2 / W_2 \geq \ldots\ldots \geq P_n / W_n}$, then greedy knapsack generates an optimal solution to the given instance of the knapsack problem.

$P_1 / W_1 = 10/2 = 5$

$P_2 / W_2 = 10/4 = 2.5$

$P_3 / W_3 = 12/6 = 2$

$P_4 / W_4 = 18/9 = 2$

Arranging the object in descending order of unit cost i.e; P_1 / W_1 , P_2 / W_2 , P_3 / W_3 , P_4 / W_4 (5 , 2.5, 2, 2) then,

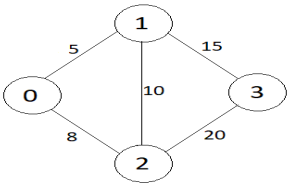
P_i	=	10	10	12	18
W_i	=	2	4	6	9
X_i	=	1	1	1	3/9

The optimal solution is,

$\sum p_i x_i$ = 10*1 + 10*1 + 12*1 + 18*(3/9) = 38

Maximum profit is 38 and the solution vector is (X_1 , X_2 , X_3 , X_4) = (1, 1, 1, 3/9).

b) **Construct Minimum cost spanning tree using Prim's algorithm.** CO2 7M



(0, 1) be an edge of minimum cost in E;

Min-cost = 5;

t[1, 1] = 0; t[1, 2] = 1;

Initial near[j] table

0	1	2	3
~~1~~	~~0~~	0	1

near[0] = near[1] = X;

for i :=2 to n-1

When i = 2

possibilities for j = 2, 3 among these vertex 2 has minimum cost[2, near[2]]

t[2, 1] = 2; t[2, 2] = near[2] = 0;

min-cost = 5 + cost[2, near[2]] = 5 + cost[2, 0] = 5 + 8 = 13;

near[2] = X;

Update near table.

0	1	2	3
~~1~~	~~0~~	~~0~~	1

Partially constructed Spanning tree t

0	1
2	0

When i = 3

possibilities for j = 3 vertex 3 has minimum cost[3, near[1]]

t[3, 1] = 3; t[3, 2] = near[3] = 1;

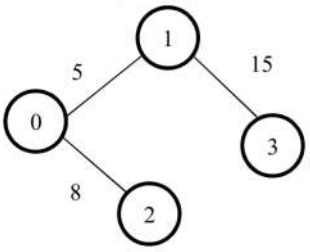
min-cost = 13 + cost[3, near[3]] = 13 + cost[3, 1] = 13 + 15 = 28;

near[3] = X;

Update near table.

0	1	2	3
~~1~~	~~0~~	~~0~~	~~1~~

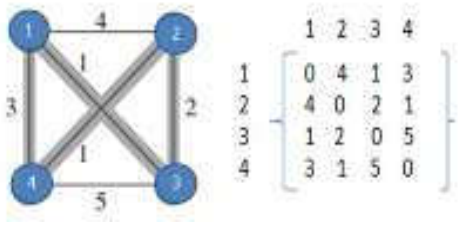
Minimum cost of the Spanning tree t is 28 and the Minimum cost the Spanning tree t is



0	1
2	0
3	1

Unit –III

6. a) **Solve the following travelling sales person problem using Dynamic Programming.** CO3 14M



	1	2	3	4
1	0	4	1	3
2	4	0	2	1
3	1	2	0	5
4	3	1	5	0

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,

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \quad \text{----(1)}$$

Generalizing equation (1) , we obtain (for i $\notin$ S)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad \text{----(2)}$$

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.

Initially, $g(i, \emptyset) = c_{i1}$ for $1 \le i \le n$

$g(1, \emptyset) = c_{11} = 0$

$g(2, \emptyset) = c_{21} = 4$

$g(3, \emptyset) = c_{31} = 1$

$g(4, \emptyset) = c_{41} = 3$

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

$g(2, \{3\}) = c_{23} + g(3, \emptyset) = 2 + 1 = 3$

$g(2, \{4\}) = c_{24} + g(4, \emptyset) = 1 + 3 = 4$

$g(3, \{2\}) = c_{32} + g(2, \emptyset) = 2 + 4 = 6$

$g(3, \{4\}) = c_{34} + g(4, \emptyset) = 5 + 3 = 8$

$g(4, \{2\}) = c_{42} + g(2, \emptyset) = 1 + 4 = 5$

$g(4, \{3\}) = c_{43} + g(3, \emptyset) = 5 + 1 = 6$

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

$$g(2, \{3, 4\}) = \min_{j \in 3,4} \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\}$$

$$= \min_{j \in 3,4} \{2 + 8, 1 + 6\}$$

$$= \min_{j \in 3,4} \{10, 7\}$$

$= 7$ (when $j = 4$)

$$g(3, \{2, 4\}) = \min_{j \in 2,4} \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\}$$

$$= \min_{j \in 2,4} \{2 + 4, 5 + 5\}$$

$$= \min_{j \in 2,4} \{6, 10\}$$

$= 6$ (when $j = 2$)

$$g(4, \{2, 3\}) = \min_{j \in 2,3} \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$$

$$= \min_{j \in 2,3} \{1 + 3, 5 + 6\}$$

$$= \min_{j \in 2,3} \{4, 11\}$$

$= 4$ (when $j = 2$)

Finally, from equation (1) we obtain

$$g(1, \{2, 3, 4\}) = \min_{j \in 2,3,4} \{c_{12} + g(2, \{3,4\}),\ c_{13} + g(3, \{2,4\}),\ c_{14} + g(4, \{2,3\})\}$$

$$= \min_{j \in 2,3,4} \{4 + 7,\ 1 + 6,\ 3 + 4\}$$

$$= \min_{j \in 2,3,4} \{11,\ 7,\ 7\}$$

$$= 7 \qquad (\text{when } j = 3 \text{ and } 4)$$

An optimal tour of the graph has length **7.**

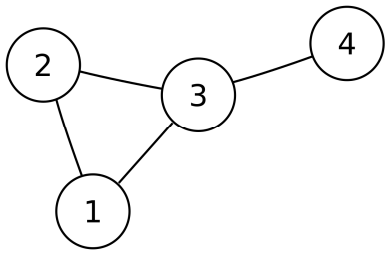
A tour of this length can be constructed if we retain with each g(i, S) the value of j that minimizes the right hand side of the equation (2). Let J(i, S) be this value.

When J(1, { 2, 3, 4 }) = 3. Thus the tour starts from 1 and goes to 3. The remaining tour may be obtained from g(3, { 2, 4 }). So J(3, { 2, 4 }) = 2. Thus the next edge is (3, 2). The remaining tour is for g(2, { 4 }). So J(2, { 4 }) = 4. The optimal tour is **1→3→2→4→1**.

When J(1, { 2, 3, 4 }) = 4. Thus the tour starts from 1 and goes to 4. The remaining tour may be obtained from g(4, { 2, 3 }). So J(4, { 2, 3 }) = 2. Thus the next edge is (4, 2). The remaining tour is for g(2, { 3 }). So J(2, { 3 }) = 3. The optimal tour is **1→4→2→3→1**.

(OR)

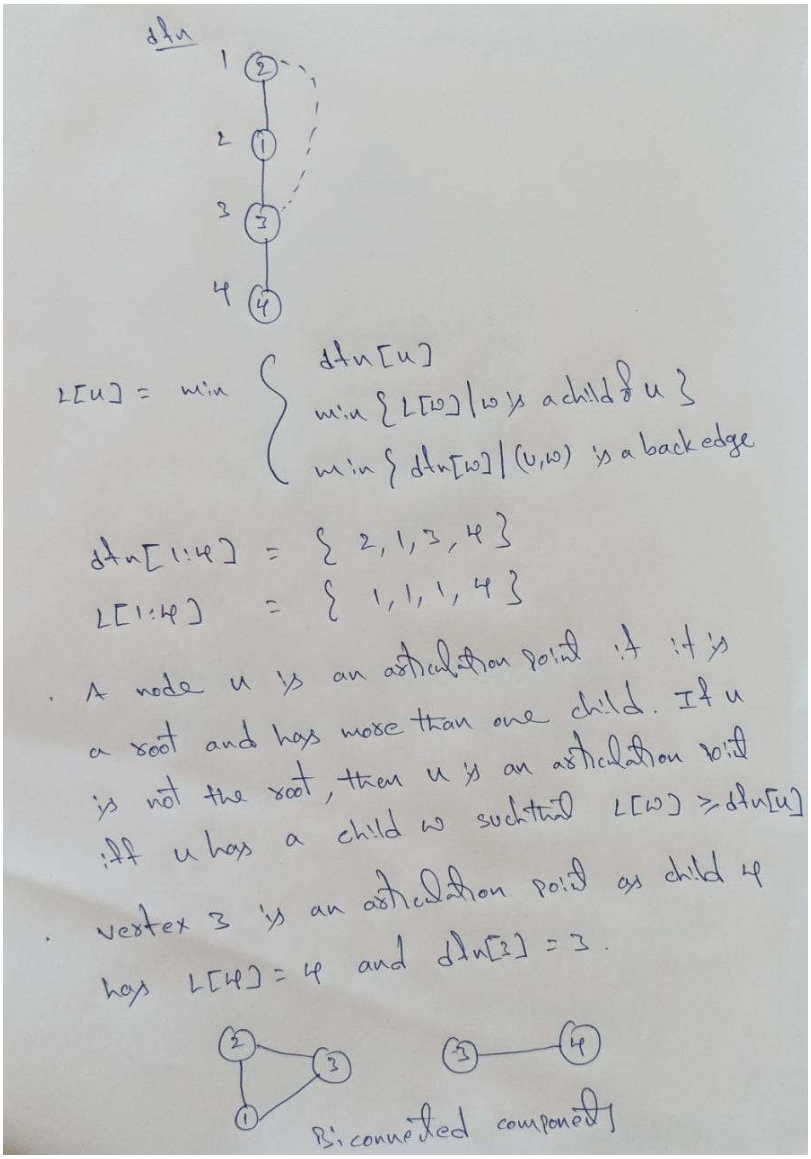
7. a) **What is meant by connected component? What is Bi- connected graph? Find Bi-connected components for the given graph.** CO3 7M



If G is a connected undirected graph, then all vertices of G will get visited on the first call to BFS (Algorithm 6.5). If G is not connected, then at least two calls to BFS will be needed. Hence, BFS can be used to determine whether G is connected. Furthermore, all newly visited vertices on a call to BFS from BFT represent the vertices in a connected component of G. Hence the connected components of a graph can be obtained using BFT. For this, BFS can be modified so that all newly visited vertices are put onto a list. Then the subgraph formed by the vertices on this list make up a connected component. Hence, if adjacency lists are used, a breadth first traversal will obtain the connected components in $\Theta(n + e)$ time.

In this section, by "graph" we always mean an undirected graph. A vertex v in a connected graph G is an *articulation point* if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.

A graph G is *biconnected* if and only if it contains no articulation points.



b) **What are the graph traversal techniques? Explain BFS with an example.** CO3 7M

Graph traversal techniques

- Breadth first traversal
- Depth first traversal

Breadth first search

• In Breadth first search we start at vertex v and mark it as having been reached. The vertex v at this time is 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. There are new unexplored vertices. Vertex v has now been explored.

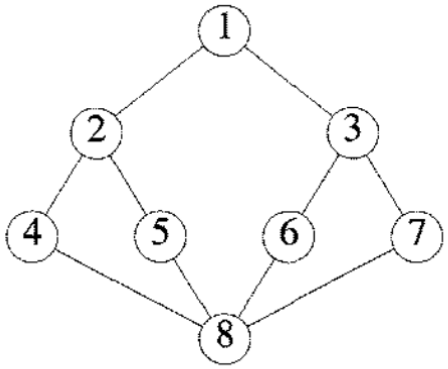
• The newly visited vertices have not been explored and are put onto the end of the list of unexplored vertices. The first vertex on this list is the next to be explored.

• Exploration continues until no unexplored vertex is left. The list of unexplored vertices acts as a queue and can be represented using any of the standard queue representations.

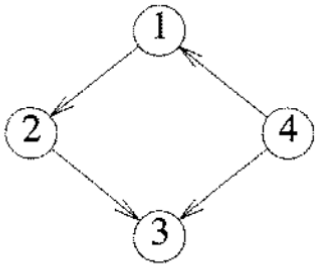
```
Algorithm BFS( v )
// A breadth first search of 'G' is carried out. Beginning at vertex v; For any node
// i, visit.  if 'i' has already been visited. The graph 'v' and array visited [ ] are
// global; visited [ ]  initialized to zero.
        {
                        u := v;          // q is a queue of unexplored
                        visited[ v ] := 1;
                        repeat
                        {
                                for all vertices w adjacent from u do
                                {
                                        if ( visited[ w ] = 0 ) then
                                        {
                                                Add w to q;
                                                visited[ w ] := 1;
                                        }
                                }
                                if q is empty then return;
                                delete u from q;
                        } until (false)
        }
```

- If BFS is used on a connected undirected graph G, then all vertices in G get visited and the graph is traversed. However, if G is not connected, then at least one vertex of G is not visited. A complete traversal of the graph can be made by repeatedly calling BFS each time with a new unvisited starting vertex. The resulting traversal algorithm is known as breath first traversal.

```
Algorithm BFT( G, n )
        {
                        for i := 1 to n do
                                visited[ i ] := 0;
                        for i := 1 to n do
                                if ( visited[ i ] = 0 ) then BFS( i )
        }
```



(a) Undirected graph G

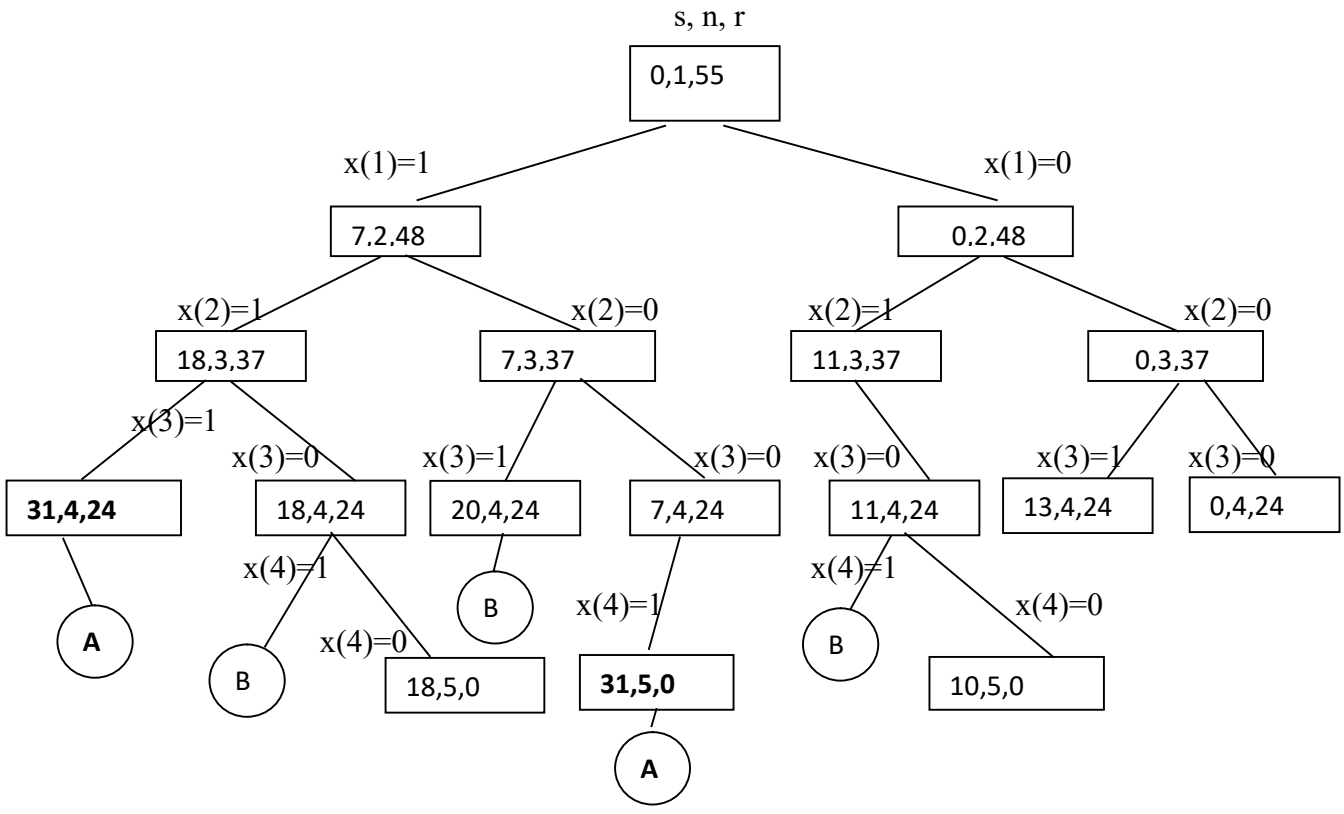


(b) Directed graph

Example 6.1 Let us try out the algorithm on the undirected graph of Figure 6.4(a). If the graph is represented by its adjacency lists as in Figure 6.4(c), then the vertices get visited in the order 1, 2, 3, 4, 5, 6, 7, 8. A breadth first search of the directed graph of Figure 6.4(b) starting at vertex 1 results in only the vertices 1, 2, and 3 being visited. Vertex 4 cannot be reached from 1. □

Unit –IV

8. a) **Let m = 31 and weights W(7, 11, 13, 24) draw a portion of state space tree using an algorithm of sum of subsets in backtracking approach?** CO4 7M



1st solution is (1, 1, 1, 0)

2nd solution is (1, 0, 0, 1)

b) **Explain the backtracking solution to the 4-queens problem and draw a portion of the tree that is generated during backtracking?** CO4 7M

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[l : n, l : 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

$$i - j = k - l \qquad \text{or} \qquad i + j = k + l$$

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 |.**

All solutions to n-queens problem can therefore be represented as n-tuples (x_1, …., x_n), where xi is the column on which queen i is placed.

Explicit constraints S_i = { 1, 2, 3, 4, ……, n }, $1 \le i \le n$

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

Example: 4-queens.

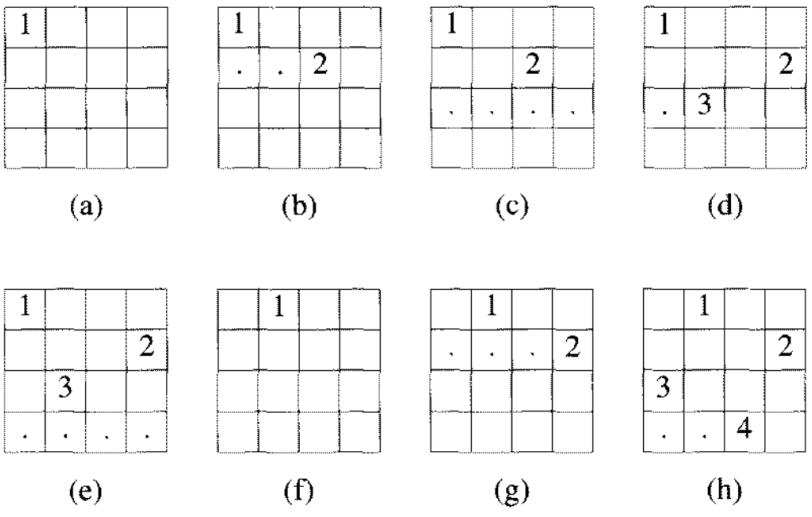


Figure 7.5 Example of a backtrack solution to the 4-queens problem

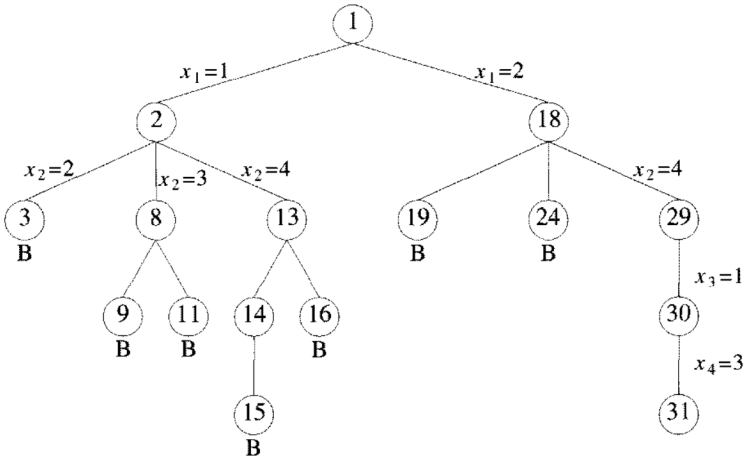


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

Two possible solutions are

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

Solution 2 : (3, 1, 4, 2)

(OR)

9. a) **Draw a portion of the state space tree generated by LCBB for the following Knapsack problem? Where n = 4, m = 15, (P1, P2, P3, P4) = (10, 10,12,18), (W1, W2, W3, W4, W5) = (2,4,6,9) Clearly show the solutions obtained?** CO4 14M

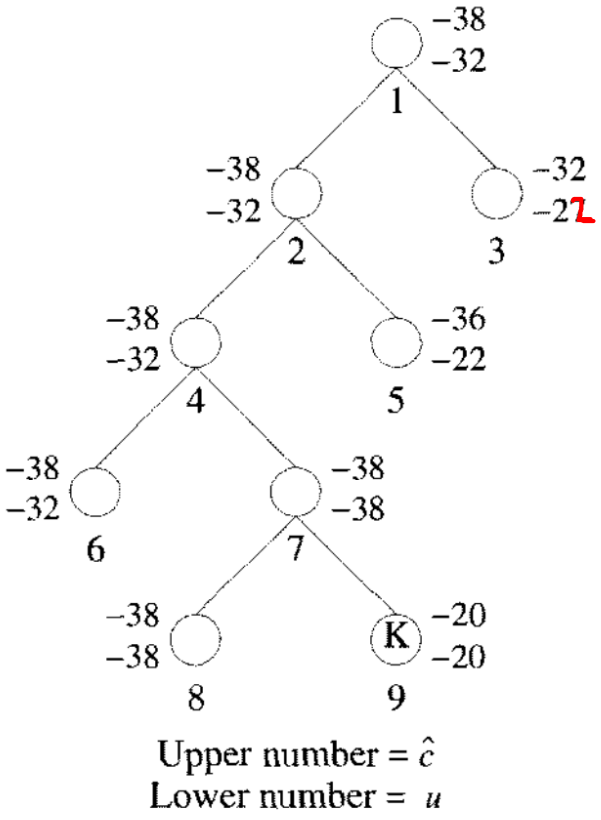


Figure 8.8 LC branch-and-bound tree for Example 8.2

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value UBound$(0, 0, 0, 15)$. UBound scans through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned. In Function UBound, c and b start with a value of zero. For $i = 1, 2$, and 3, c gets incremented by $2, 4$, and 6, respectively. The variable b also gets decremented by $10, 10$, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, LCBB sets $ans = 0$ and $upper = -32$ (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node 4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and $upper$ is updated to -38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then $upper$ is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > upper$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq upper$ and the search terminates with node 8 the answer node. At this time, the value -38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i = upper$. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, tag. The sequence of tag bits from the answer node to the root give the x_i values. Thus, we have $tag(2) = tag(4) = tag(6) = tag(8) = 1$ and $tag(3) = tag(5) = tag(7) = tag(9) = 0$. The tag sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1, x_3 = 0, x_2 = 1$, and $x_1 = 1$. □

Maximum profit is **38** and the solution vector is (X_1, X_2, X_3, X_4) = **(1, 1, 0, 1).**

HOD, CSE