DAA THEORY ASSIGNMENT - II

1Q) Design and analysis algorithm for the following ?

A) Quick sort

B) Merge sort.

SRILEKHA DEVINEEDI
19131A05M7
CSE-4

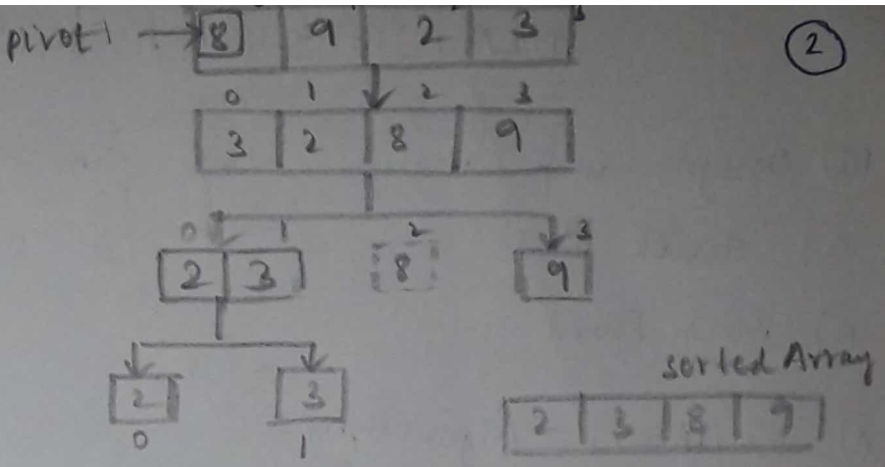
A) Quicksort Algorithm:

Input: Unsorted list

Output: Sorted list.

```
Partition (a,s,e)
{
   left = s+1
   right = e
   pivot = s
  while left<= right
 {
  while ( left<=right and a[left]<pivot)
  {  left ++
  }
  while ( left <= right and a[right]>pivot)
  {  right ++
  }
   if (right> left)
   {  break the loop
   }
   else
   {  swap a[left] and a[right]
   }
 }
 swap a[s] and a[right]
 return right
}
```

```
quicksort (a,s,e)
{ if s<e
  {
    p=partition(a,s,e)
    quicksort (a,s,p-1)
    quicksort (a,p+1,e)
  }
}
```



Time complexity:-

Best case: The best case occurs when the partition picks the middle element as pivot. Following is recurrence for best case

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

($\frac{n}{2}$ → Division of array; $\Theta(n)$ → No. of comparisons; $\frac{n}{2}$ → elements in the sub array)

The solution of above recurrence is $\Theta(n \log n)$. [Masters theorem case 2]

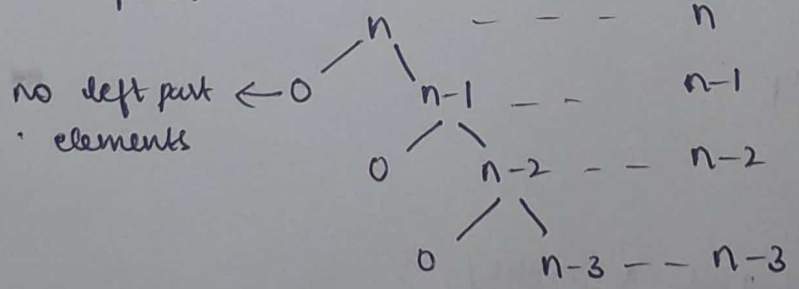
Average case: Average case occurs when we divide the array into unequal sub parts. In Quick sort both ~~Best~~ and Average case complexity is same

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

$\left[\log_b^a = K \quad p > -1 \quad O(n^K \log^{p+1} n)\right]$ $\quad a=2, b=2, K=1, p=0$

Worst case: In quicksort worst case will occur if the given array is sorted.

eg: 10 20 30 40 50
pivot



$$1+2+3+\ldots\ldots \Rightarrow \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

$$\Rightarrow O(n^2) //$$

B) MERGE SORT

Merge Sort Algorithm

Input: unsorted list

Output: sorted list

```
merge (left, right, a)
{
  i=0  j=0  k=0
  while(i<length(left) and j< length(right))
  {
    if left[i] < right[j]
    {
      a[k] = left[i]
        k++
        i++
    }
    else
    {
      a[k]= right[j]
        k++
        j++
    }
  }
  while i< length(left)
  {
      a[k] = left[i]
        k++
        i++
  }
  while j< length(right)
  {
      a[k] = right[j]
        j++
        k++
  }
}
  mergesort(a)
{
  if length(a)>1
  { mid = length(a)/2
    mergesort(
    left = a[first element to mid]
```

```
    right = a[ mid to last element ]
    mergesort (left)
    merge sort (right)
    merge (left, right, a)
  }
}
```

Time complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- 2: Dividing array into 2 parts
- $\frac{n}{2}$: Each part contains $n/2$ elements
- n: minimum n comparisons

Best Case :- If we divide the given array into exactly two parts

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$a=2, b=2, f(n) = O(n^k \log^p n) \Rightarrow k=1, p=0$.

$\log_2^2 = 1 = k$, $p > -1$ so,

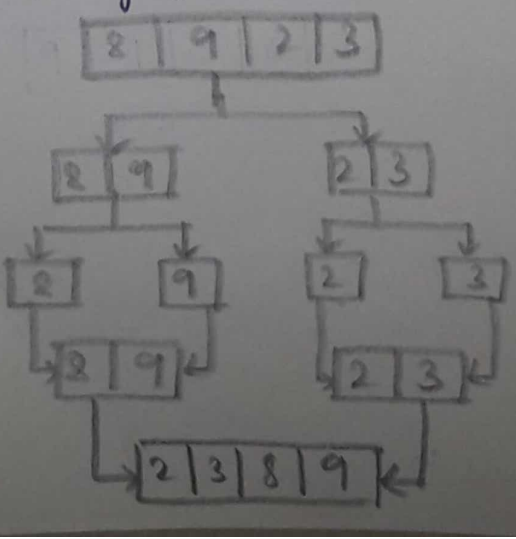
$O(n^k \log^{p+1} n)$

$\Rightarrow O(n \log n)$

Time complexity of merge sort in all the three cases is the same

Average case $\Rightarrow O(n \log n)$

Worst case $\Rightarrow O(n \log n)$.



2Q) Explain 0/1 Knapsack and Traveling salesman Problem. Design and analysis algorithm for both using the following techniques (5)

A) Dynamic Programming

B) BackTracking

C) Branch and Bound.

0/1 Knapsack :- Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum value (profit) in the knapsack. In other words given two integer arrays val[0--n-1] and wt[0--n-1] which represents values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the max value subset of val[] (profits) such that the sum of the weights of this subset is smaller than or equal to W. We cannot break the weights, either pick the complete item or don't pick it

A) Dynamic Programming :-

weights = {3, 4, 6, 5} W = 8

profits = {2, 3, 1, 4}

		i ↓ / w →	0	1	2	3	4	5	6	7	8
Pi	Wi	0	0	0	0	0	0	0	0	0	0
2	3	1	0	0	0	2	2	2	2	2	2
3	4	2	0	0	0	2	3	3	3	5	5
4	5	3	0	0	0	2	3	4	4	5	6
1	6	4	0	0	0	2	3	4	4	5	6

formula:

$V[i, w] = \max\{ V[i-1, w], V[i-1, w-w[i]] + P[i] \}$.

the max profit = 6.

6 is generated by including 3rd object in the bag.

remaining profit = 6 - 4 = 2.

we got 2 by including the 1st object so,

remaining profit = 2-2 = 0.

{1, 2, 3, 4}
1 0 1 0

Bag contains 1st & 3rd objects.

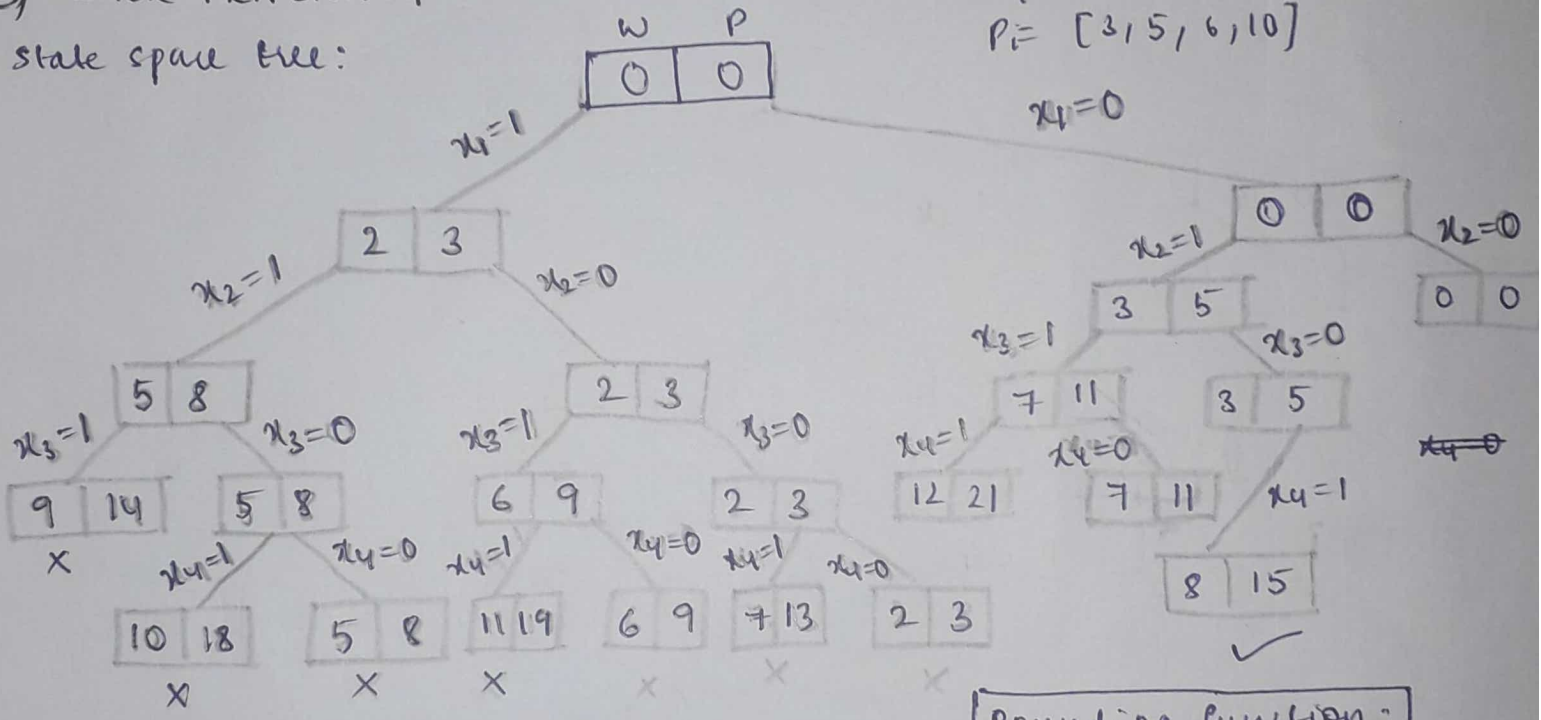
Time complexity = $O(nW)$

where, n = no. of items, W = capacity of the bag

(B) BACK TRACKING:

x_1 x_2 x_3 x_4
$w_i = [2, 3, 4, 5]$ $W = 8$
$P_i = [3, 5, 6, 10]$

State space tree:



Bounding function:

$$\sum_{i=1}^{n} w_i x_i \leq M.$$

{x_1, x_2, x_3, x_4}
0 1 0 1

2nd & 4th objects are included in the bag.

Time complexity:

Time complexity = $O(2^n)$

Here 2^n solutions would be generated.

c) BRANCH AND BOUND.

$x_i \in \{0,1\}$ $x_i = 0$ Did not consider

$x_i = 1$ Consider

To Maximise $\sum_{i=1}^{n} b_i x_i$

constraint $\sum_{i=1}^{n} w_i x_i \leq M$.

x_1	x_2	x_3	x_4	---	x_n
0/1	0/1	0/1	0/1		
2	2	2	2	--	2^n selections

$b = \{45, 30, 45, 10\}$

$w = \{3, 5, 9, 5\}$

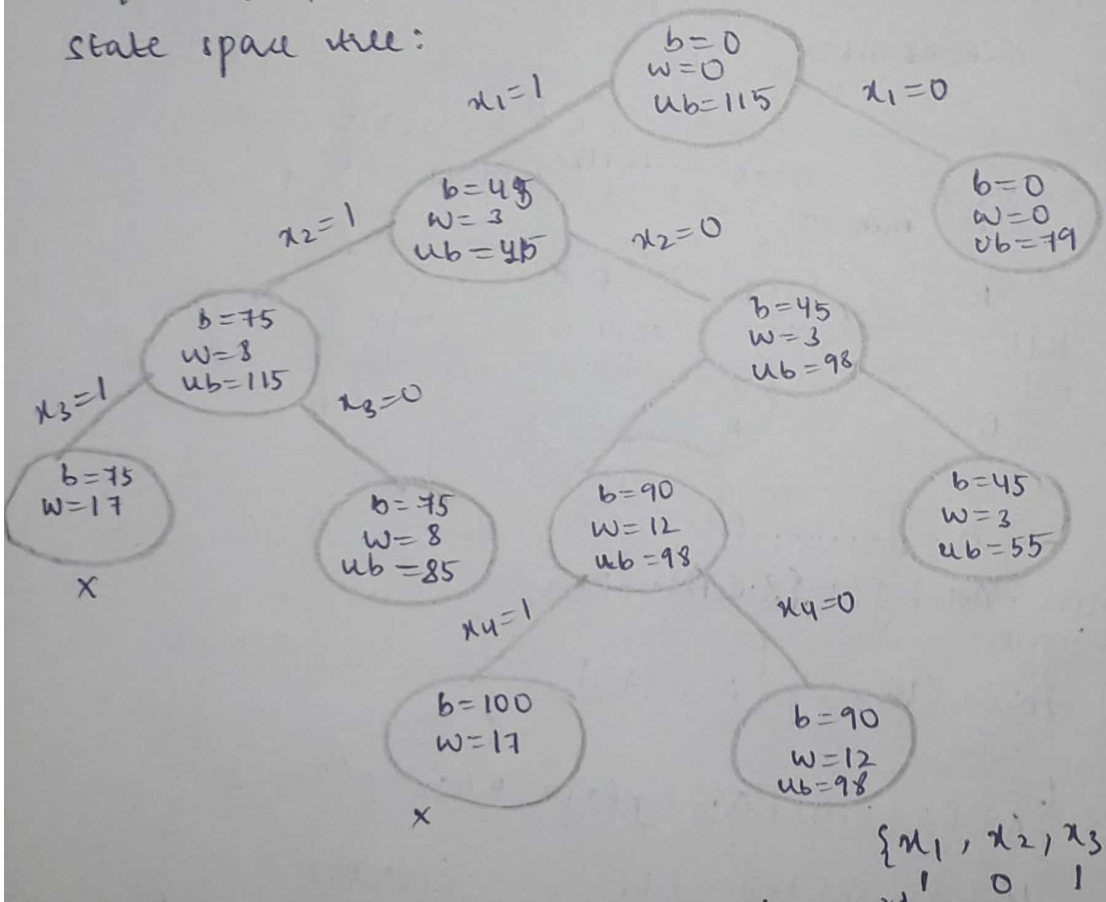
$b/w = \{15, 6, 5, 2\}$

$W = 16$.

upper bound $= 45 + 30 + \frac{8}{9} \times 45$

$= 115$ //

State space tree:



$\{x_1, x_2, x_3, x_4\}$

$\{1, 0, 1, 0\}$

Bag contains 1st & 3rd item.

Time complexity:- The time complexity of 0/1 Knapsack is $O(2^n)$

Here 2^n solutions would be generated.

0/1 Knapsack Dynamic Programming

Algorithm:

```
Knapsack (object, profit, weights, capacity)
for i in range (len (object) + 1)
    for w in range (capacity + 1)
        if i == 0 or w == 0                    // 0th row & 0th column
            v[i][w] = 0
        else if weight[i] <= w                 // weight[i] <= capacity
            v[i][w] = max { v[i-1][w], v[i-1][w - weight[i-1]] +
                                        profit[i-1])
        else
            v[i][w] = v[i-1][w]                // fill with previous values
```

0/1 Knapsack Back Tracking:

Algorithm:

Step 1: Consider the root node with 0 profit and 0 weight.

Repeat the following steps until you find a solution

Step 2: Include the next item in the bag by adding the values of its profit and weight to the previous one.

Step 3: If the item is not included keep the previous profits and weights as it is.

Step 4: If the weight calculated is equal to given weight then return true

Step 5: If the calculated weight > ~~actual~~ given weight stop and backtrack

Step 6: If you reached the last item and didn't find a feasible solution, backtrack and explore the remaining by including and not including every item.

Repeat the above steps to find all possible solutions.

0/1 Knapsack Branch & Bound:

Algorithm

START

Step 1: Sort all items in decreasing order of ratio of value per unit weight so that an upperbound can be computed using greedy approach.

Step 2: Consider the root node with 0 profit / weight = 0 and the upperbound

Step 3: Repeat the following steps until a solution is found.

3.1. Explore both the cases of including and not including the item from the current node.

3.2. If included add the included item weight and profit by the previous one.

3.3. If not included ~~remove the item weight from the upper bound~~ calculate the upperbound without including the item.

3.4 If weight > given weight kill the node and explore the live node which has maximum upper bound.

3.5 If weight <= given weight and producing more upperbound then return true.

STOP.

⇒ TRAVELLING SALESMAN DYNAMIC PROGRAMMING:

Algorithm:

Step 1: If size of S is 2, then S must be $\{1, i\}$

$C(S, i) = dist(1, i)$

Step 2: else if size of S is greater than 2

$C(S, i) = \min \{C(S - \{i\}, j) + dis(j, i)\}$ where j belongs to S, $j != i$ and $j != 1$.

⇒ TRAVELLING SALES MAN BACKTRACKING:

Algorithm:

START

Step 1: Select a vertex as a root node.

Step 2: Explore the next city from the current node and calculate the distance from the root node to the current node for every vertex. Explore it depth wise.

Step 3: If leaf node has been reached, backtrack to the parent node and explore the remaining possibilities. Repeat step 2 for every vertex.

Step 4: Consider the least distance containing leaf node and that is the minimum path.

END

⇒ TRAVELLING SALESMAN BRANCH AND BOUND:

Algorithm:

START

Step 1: Let A be the reduced cost matrix for the node 'R'. Let S be the child of R such that the edge (R,S) corresponds to including edge (i,j) in the tour.

Step 2: If 's' is not a leaf node, then the reduced cost matrix for node 's' can be obtained as follows.

i) change all the entries in row i & column j to '∞'.

ii) set A(j,1) to '∞'

iii) Apply row reduction and column reduction except for rows and columns containing '∞'.

iv) Total cost of a node 's' can be calculated as

$$\hat{c}(s) = \hat{c}(R) + A(i,j) + r.$$

where r is the total amount subtracted in step 3.

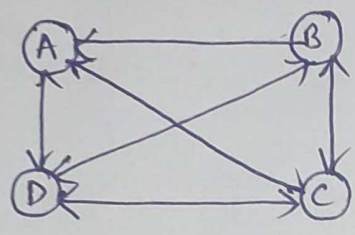
Step 3: Select the minimum cost node and traverse until the leaf node

EXIT

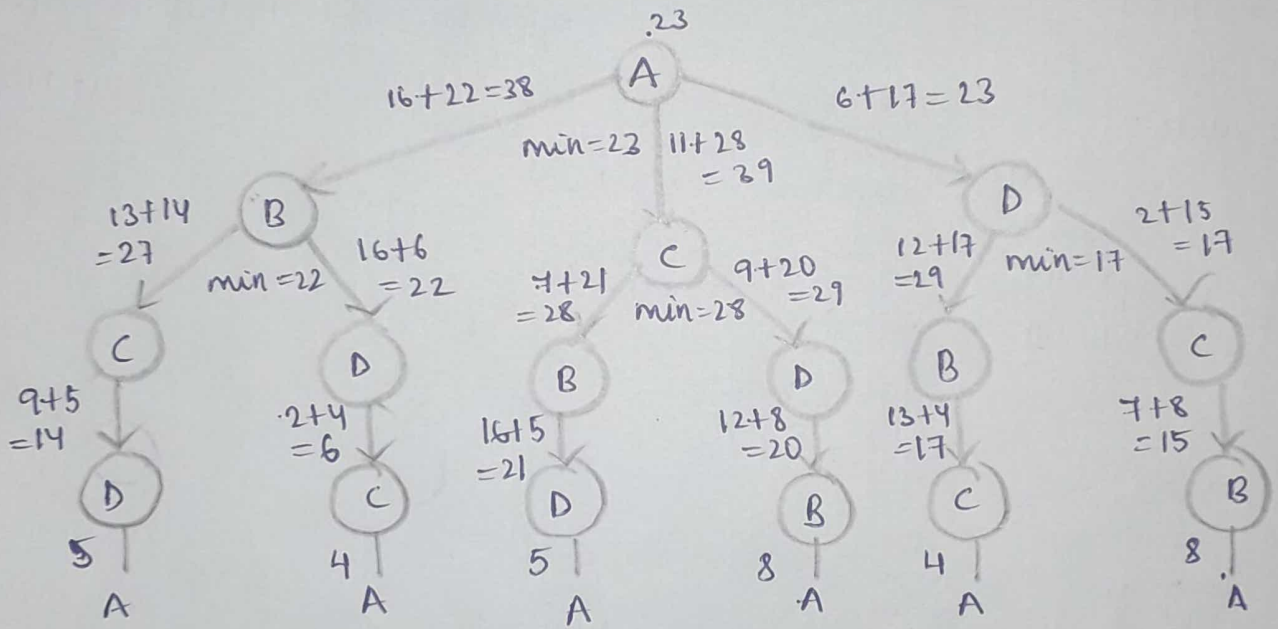
⇒ TRAVELLING SALESMAN PROBLEM

The travelling sales man problem is the challenge of finding the shortest yet most efficient route for a person to take a list of specific destinations. Select any vertex as the starting vertex and it should go through all the vertices and return to the first vertex. It should form a cycle.

A) DYNAMIC PROGRAMMING:



	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0



Eg:- $g(A,\{B,C,D\}) = \min_{K\in\{B,C,D\}} \{W_{ik} + g(K,\{B,C,D\}-\{K\})\}$

Formula: $g(i,S) = \min_{K\in S} \{W_{ik} + g(K, S-\{K\})\}$.

$g(A,\{B,C,D\}) = \min\{A_{12}+g(3,4), A_{13}+g(2,4), A_{14}+g(2,3)\}$

$16+9,\ 11+16,\ 6+13$

$g(A,\{B,C,D\}) = \min\{W_{AB}+g(B,\{C,D\}), W_{AC}+g(C,\{B,D\}), W_{AD}+g(D,\{B,C\})\}$

$g(B,\{C,D\}) = \min\{W_{BC}+g(C,\{D\}), W_{BD}+g(D,\{C\})\}$

$g(C,\{D\}) = \min\{W_{CD}+g(D,\phi)\}$.

$9 + 05\ [D-A]$

$= 14$

$g(D, \{C\}) = w_{DC} + g\{C, \{\phi\}\}$

$= 2+4 = 6$ //

~~$= 6+4$~~

~~$= 10$~~ //

$g(B, \{C, D\}) = \min\{$~~$4+10$~~, $13+4, 16+6\}$

$= 22$ //

$g(C, \{B, D\}) = \min\{w_{CB} + g\{B, \{D\}\}, w_{CD} + g(D, \{B\}\}$

$g(B, \{D\}) = \min\{w_{BD} + g\{D, \{\phi\}\}\}$

$16 + 5$

$= 21$ //

$g(D, \{B\}) = \min\{w_{DB} + g(B, \{\phi\})\}$

$= 12$~~8~~ ~~+8~~

$= 20$ //

$g(D, \{B, C\}) = \min\{w_{DB} + g(B, \{C\}), w_{DC} + g\{C, \{B\}\}\}$

$g(B, \{C\}) = \min\{w_{BC} + g(C, \{\phi\})\}$

$13 + 4 = 17$ //

$g(C, \{B\}) = \min\{w_{CB} + g(B, \{\phi\})\}$

$7 + 8 = 15$ //

$g(C, \{B, D\}) = \min\{7+21, 9+20\}$

$= 28$ //

$g(D, \{B, C\}) = \min\{13+17, 2+15\}$

$= 17$ //

$g(A, \{B, C, D\}) = \min\{16+22, 11+28, 6+17\}$

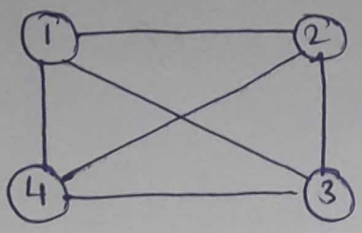
$\min\{38, 39, 23\}$

$= 23$ //.

Time complexity :-

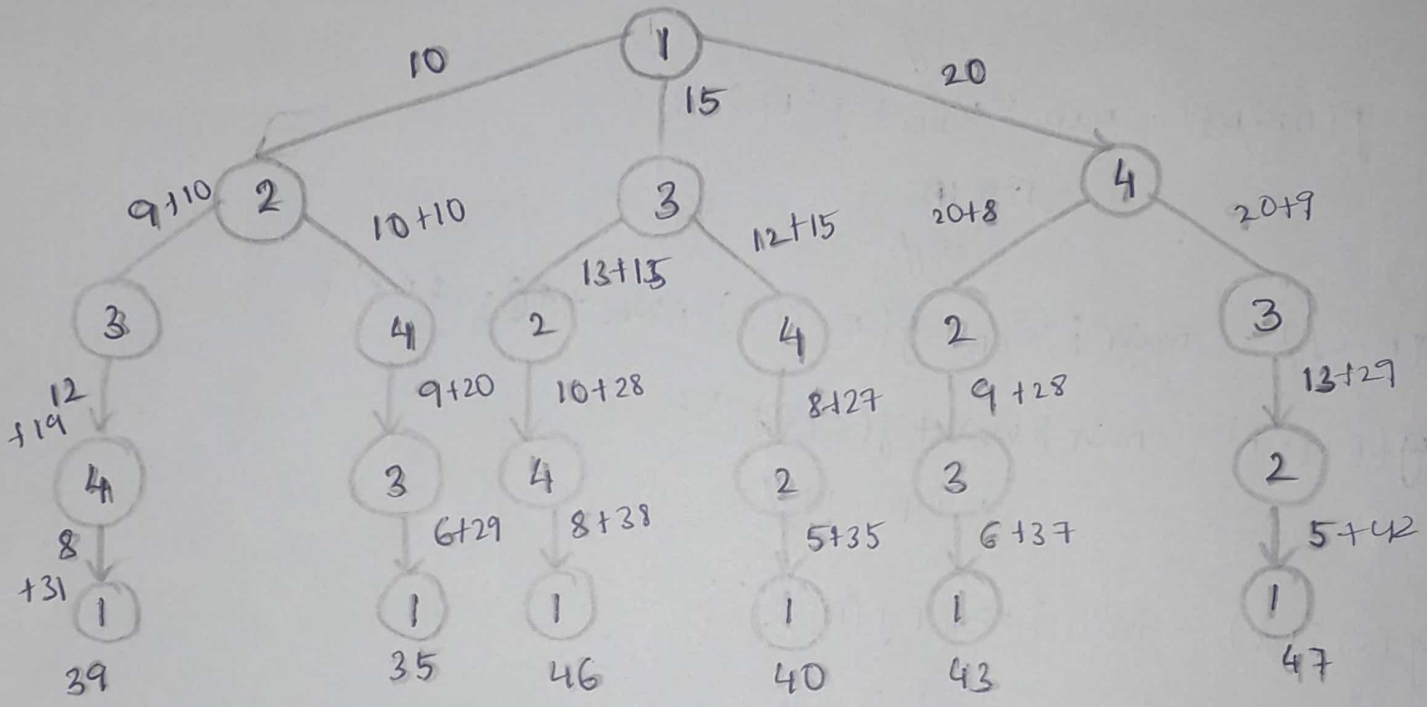
The time complexity of travelling salesman algorithm is $O(n^2 \times 2^n)$ where n is the no. of nodes. There are almost $n \times 2^n$ subsets. Each subset requires $O(n)$ time complexity.

B) BACK TRACKING:



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

state space tree:



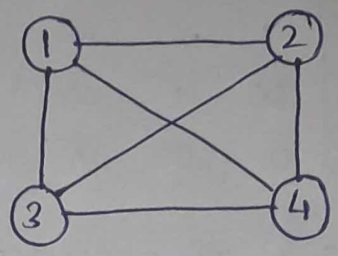
Here 35 is the minimum tour and it is given by the path 1-2-4-3-1.

Time complexity:

The time complexity of travelling saleman algorithm is $O(n^2 \times 2^n)$ where n is the no. of nodes. There are almost $n \times 2^n$ subsets. Each subset requires $O(n)$ time complexity.

c) BRANCH AND BOUND:

	1	2	3	4
1	∞	10	5	3
2	8	∞	9	7
3	1	6	∞	9
4	2	3	8	∞



Row reduction

				Min
	∞	10	5	3	3
	3	∞	9	4	7
	1	6	∞	9	1
	2	3	8	∞	2

13

Row Reduction ⟹

	∞	7	2	0
	1	∞	2	0
	0	5	∞	8
	0	1	6	∞
min		1	2	⟹ 3

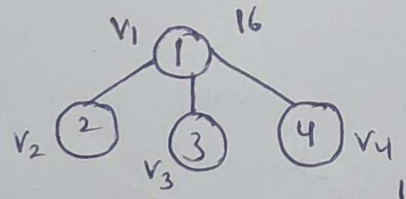
Column Reduction ⟹

	1	2	3	4
1	∞	6	0	0
2	1	∞	0	0
3	0	4	∞	8
4	0	0	4	∞

Cost of reduction =
Cost of row reduction + Cost of column reduction

Cost of reduction = 13 + 3 ⟹ 16

The minimum cost tour will be atleast or maybe 16.



v - Vertice
1, 2 - Node count.

Node 2
1 − 2

[1st row & 2nd column ∞] 1-7∞ & 2-1∞.

	1	2	3	4	Min
1	∞	∞	∞	∞	∞
2	∞	∞	0	0	0
3	0	∞	∞	8	0
4	0	∞	4	∞	0
Min:	0	0	0	0	

Cost of reduction = 0.

Cost (1−2) = edge (1,2) + Cost (vertex 1) + Cost (reduction)

6 + 16 + 0

Cost (1−2) = 22

1-3

Node 3

	1	2	3	4	Min
1	∞	∞	∞	∞	0
2	1	∞	∞	0	0
3	∞	4	∞	8	4
4	0	0	∞	∞	0

Row Reduction ⟹

	∞	∞	∞	∞
	1	∞	∞	0
	∞	0	∞	4
	0	0	∞	∞
Min	0	0	∞	0

Cost of reduction = 4

cost (1-3) = edge (1,3) + cost (vertex 1) + cost (reduction)

= 0 + 16 + 4

cost (1-3) = 20.

1-4

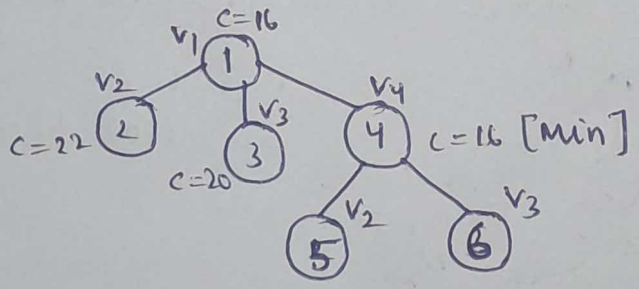
Node 4

	1	2	3	4	Min
1	∞	∞	∞	∞	∞
2	1	∞	0	∞	0
3	0	4	∞	∞	0
4	∞	0	4	∞	0
Min	0	0	0	∞	

cost (1-4) = edge (1,4) + cost (vertex 1) + cost (reduction)

0 + 16 + 0

cost (1-4) = 16.



4-2

Node 5

	1	2	3	4	Min
1	∞	∞	∞	∞	∞
2	∞	∞	0	∞	0
3	0	∞	∞	∞	0
4	∞	∞	∞	∞	∞
Min	0	∞	0	∞	

1-4-2

2-1-∞

2-4-∞

1 & 4 row -∞

2 column -∞

reduction = 0.

cost (4-2) = edge (4,2) + cost (vertex 4) + cost (reduction)

0 + 16 + 0

cost (4-2) = 16.

4-3

Node 6

1-4-3

	1	2	3	4	Min
1	∞	∞	∞	∞	∞
2	1	∞	∞	∞	1
3	∞	4	∞	∞	4
4	∞	∞	∞	∞	∞
					5

Row Reduction ⟹

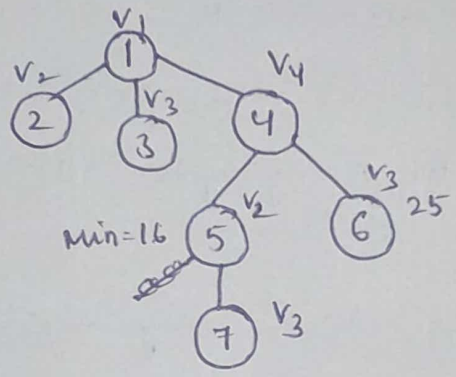
	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	∞	∞
3	∞	0	∞	∞
4	∞	∞	∞	∞

cost of Reduction = 5

cost (4-3) = Edge (4,3) + cost (vertex 4) + cost (reduction)

4 + 16 + 5

cost (4-3) = 25



2-3

Node 7

1-4-2-3

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞
4	∞	∞	∞	∞

1, 4, 2 Row - ∞

+ column - ∞

3-1, 3-4 - ∞ ; 3-2 - ∞

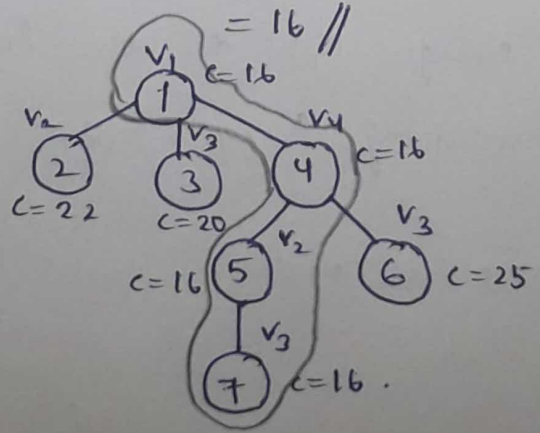
cost of reduction = 0

cost (2-3) = Edge (2,3) + cost (vertex 5) + cost (Reduction)

cost (2-3) = 0 + 16 + 0

= 16 //

state space tree:



Route = 1-4-2-3-1

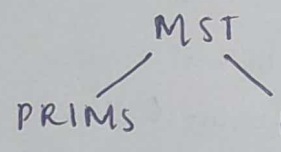
Time complexity: Time complexity of travelling salesman problem is $O(n^2)$ because algorithm has two iterations embedded one inside the other for calculation of reduced cost matrix.

3Q) Explain minimum cost spanning tree and single pair shortest path. Give an algorithm for both using greedy method.

A) Minimum cost spanning tree (MST):

A spanning tree is a subset of an undirected graph that has all the vertices connected by minimum no. of edges. If all the vertices connected in the graph, then there exists atleast one spanning tree. In a graph there may exist more than one spanning tree.

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.



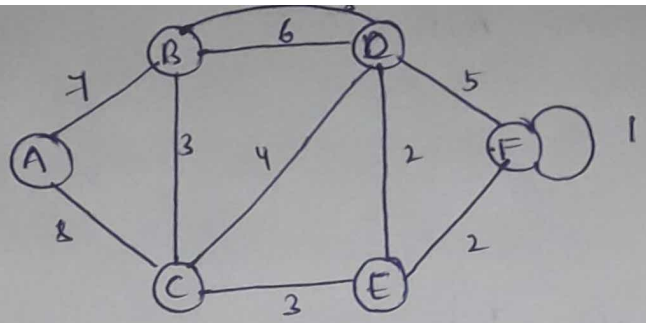
Prim's Algorithm:

Prim's algorithm is a greedy algorithm to find the minimum spanning tree.

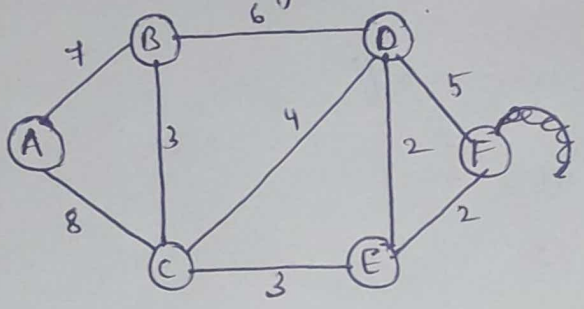
Algorithm:

1. Choose a starting vertex and put the remaining vertex in the fringe list
2. Not each vertex in the fringe list do the following.
   A. Select a edge connecting fringe vertex with minimum weight.
   B. Add the edge and vertex in MST.
   [END OF LOOP]
3. EXIT.

Eg:

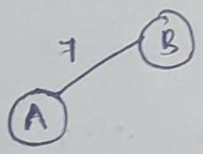


Remove all parallel edges and loops. Always remove the edge which has more weight.

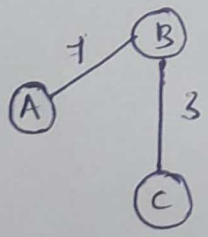


⇒ Choose an Arbitary node as ~~minimum~~ root node.

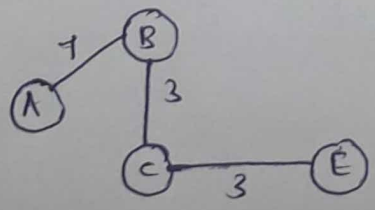
⇒ Choose an edge of A with minimum weight



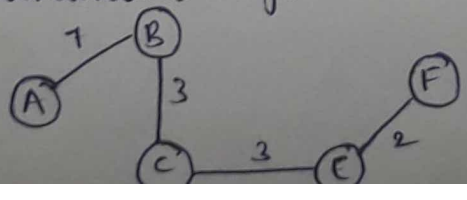
⇒ Now we've two vertices A & B. check all the outgoing edges from A & B and pick the edge which is minimum.



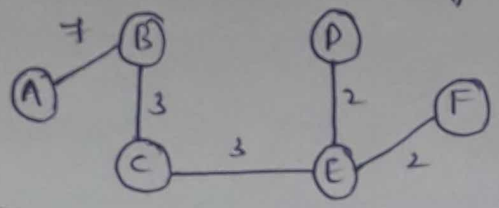
⇒ Pick the minimum edge from A, B, C:



⇒ Pick the minimum edge from A, B, C, E



Pick the minimum edge from A, B, C, E, F.



MST Edges = No. of vertices in Graph − 1

MST vertices = Original graph vertices

Edges = 6 − 1

Edges = 5 // vertices = 6 //

So we can stop here.

Time Complexity:

> If adjacency list is used to represent the graph, then using BFS, all the vertices can be traversed in O(V+E) time. We traverse all the vertices of graph using BFS and use a min heap for sorting the vertices not yet included in the MST. To get the minimum weight edge, we use min heap as a priority queue. Min heap operations like extracting minimum element and decreasing key value takes O(log V) time. Min heap operations like extracting minimum element and decreasing key value takes O(log V) time.

So overall time complexity:

$$= O(E+V) \times O(\log V)$$

$$= O(E \log V + V \log V)$$

$$= O(E \log V).$$

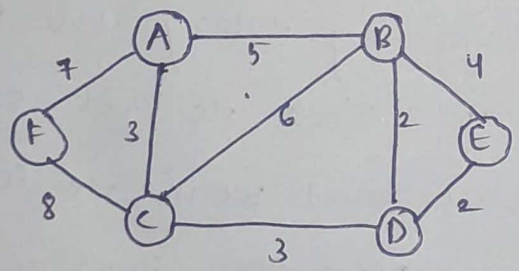
KRUSKALS ALGORITHM:

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of this algorithm is used to find the subset of edges by using which we can traverse every vertex of the graph. It follows greedy approach.

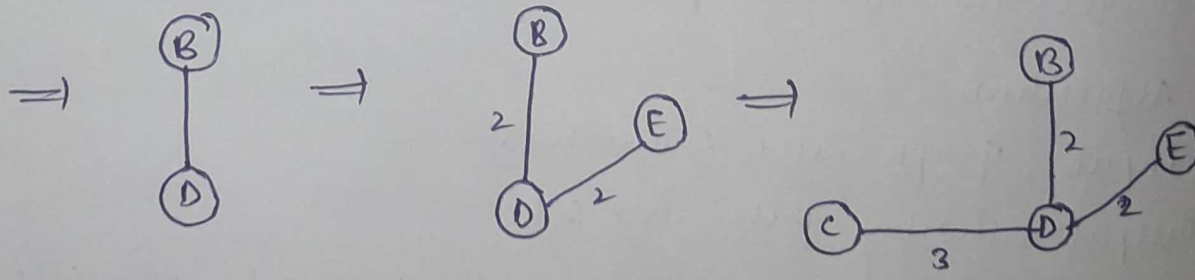
Algorithm:

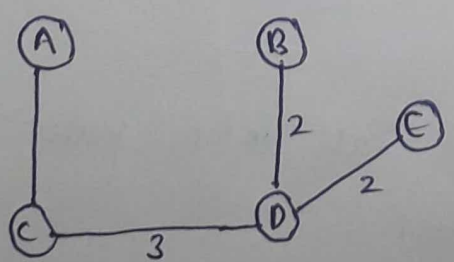
1. Sort all the edges in non-decreasing order of their weight
2. Pick the edge with smaller weight.
   A. check whether it is forming a cycle or not if not, add it to the MST.
3. Repeat step 2 until there are (v-1) edges in the spanning tree.
4. EXIT.

Eg:-

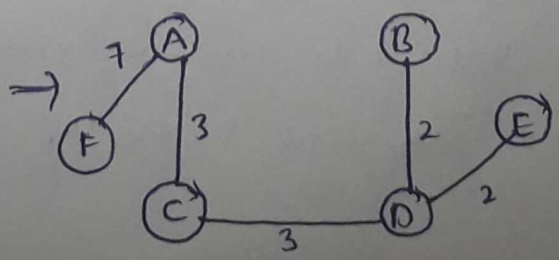


Pick the minimum edge weight and add it to the MST one by one.





Next $min^{BE}_{AB, BC}$ forms a cycle so add AF



Time complexity:

The edges are maintained as min heap. The next edge can be obtained in O(logE) time if graph has E edges. Reconstruction of heap takes O(E) time. So time complexity is

Time complexity = O(ElogE).

⟹ SINGLE PAIR SHORTEST PATH

We've to find the shortest path from the given node.

Dijsktra's Algorithm:

If a weighted graph is given then we've to find a shortest path from starting vertex to all other vertices. As we've to find the shortest path it is a minimization problem. And minimiza-tion problem is an optimization problem so, optimization problems can be solved using greedy method. Greedy method says that a problem should be solved in stages by taking one step at a time and considering one input at a time to get optimal solution. There are predefined procedures and we've to follow the procedures to get the optimal solution. Dijsktras gives a solution to get the shortest path.

Algorithm

Input: Graph and source node

Output: shortest path tree set.

1. Initially all the nodes are marked unvisited and call it unvisited set.
2. Set distance to source vertex 0 and all other vertex to infinity And set source vertex as current vertex.
3. Relax all the adjacent vertex of current vertex who are not visited.

Relaxation:

If u, v are adjacent

if (distance [u] + cost (u,v) < distance [v])

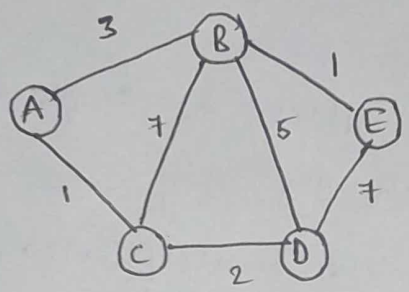
distance [v] = distance [u] + cost (u,v).

4. Update current vertex.

Choose the shortest vertex from current vertex, from unvisited set whose distance is minimum from current vertex.

5- Repeat 3 & 4 until unvisited set is empty.

Eg:-



source vertex = A

Next Minimum	A	B	C	D	E
A	0	3	1	∞	∞
C	0	3	1	3	∞
D	0	3	1	3	10
B	0	3	1	3	4
E	0	3	1	3	4

shortest path = A C D B E

Time complexity: It is finding shortest path to all the vertices If there are n vertices, while finding shortest path it is relaxing, we cannot predict how many it can relax it depends on the graph, so it can relax almost n vertices so, O(n x n) (n vertices, n vertices are relaxed)

Time complexity = $O(n^2)$ or $O(v^2)$ [worst case]

we will get the worst case if the given graph is complete. Optimising it by using a min-priority queue reduces to $O(V + E \log V)$.

Time complexity $= O(V + E \log V)$

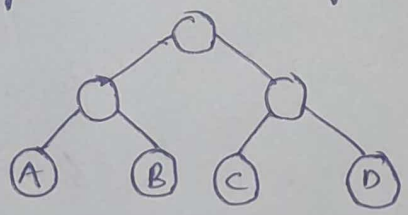
- V: cost to make unvisited vertex.
- E: cost to relaxation
- $\log V$: cost to choose next vertex.

4Q) Explain Matrix chain multiplication and All pair shortest path. Give an algorithm for both using dynamic programming method.

Matrix Chain Multiplication: Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not to perform the multiplications, but merely to decide in which order to perform the multiplications.

Eg:- If we have matrices A, B, C, D then we have

$\frac{\lfloor 2n}{\lfloor n \lfloor n+1}$ ways to multiply them $n = 3$



$\frac{\lfloor 6}{\lfloor 3 \lfloor 4} = 5$ ways //

n = no. of nodes

i) (A B)(C D) ii) (A(B C))D iii) A(B(C D)) iv) A((B C) D) v) ((A B)C)D

Tabularization:

	1	2	3	4
1	0	5785	1530	2856
2		0	1335	1845
3			0	9078
4				0

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

Splitting table.

A · B · C · D

13×5 5×89 89×3 3×34

i) ~~M[1,2] =~~

Multiplying 2 matrices

$M[1,2] = 13 \times 5 \times 89 = 5785$

$M[2,3] = 5 \times 89 \times 3 = 1335$

$M[3,4] = 89 \times 3 \times 34 = 9078$

o-splitting

Multiplying 3 matrices [2 possibilities]

$M[1,3] = A \cdot (B \cdot C)$ Or $(AB) \cdot C$

$= M[1,1] + M[2,3] + 13 \times 5 \times 3$ [Or] $M[1,2] + M[3,3] + 13 \times 89 \times 3$

$0 + 1335 + 195 = 1530$ [Or] $5785 + 0 + 3471 = 9256$

min

$M[1,3] = 1530$ //

$M[2,4] = B \cdot (C \cdot D)$ [Or] $(B \cdot C) \cdot D$

$M[2,2] + M[3,4] + 5 \times 89 \times 34$ [OR] $M[2,3] + M[4,4] + 34 \times 3 \times 5$

$0 + 9078 + 15130$ $1335 + 0 + 510$

$= 24280$ $= 1845$

min

$M[2,4] = 1845$

Multiplying 4 Matrices:

$M[1,4] = \min \{ M[1,1] + M[2,4] + 13 \times 5 \times 34,$
$M[1,2] + M[3,4] + 13 \times 89 \times 34,$
$M[1,3] + M[4,4] + 34 \times 3 \times 13 \}$

$\Rightarrow \min \{ 0 + 1845 + 2210,$
$5785 + 9078 + 39338,$
$1530 + 0 + 1326 \}$

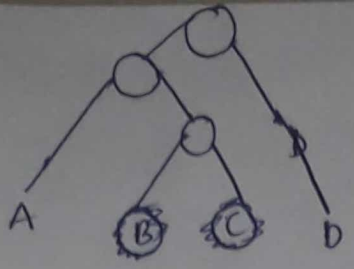
$\min \{ 4055, 54201, 2856 \}$

$M[1,4] = 2856$. No. of multiplications required to multip

~~Time comp~~ From splitting table $M[1,4] = 3 \Rightarrow (A\,B\,C)\,D$ & $M[1,3] =$ $(A\,(B\,C))\,D$ //

Formula:

$M[i,j] = \min \{ M[i,K] + m[K+1,j] + d_{i-1} * d_K * d_j \}$



Time complexity:-

We're preparing half table = $\frac{n(n-1)}{2} = n^2$

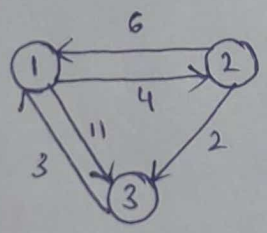
Each element we are getting by calculating the minimum so, it takes atmost n. So

Time complexity = $O(n^3)$

⇒ ALL PAIR SHORTEST PATH:

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

Eg:-



Adj Matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

vertex 1 as intermediate node [1st row & column same]

$$A' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

$A'[2-3] = \min[A^0[2,3], A^0[2,1] + A^0[1,3]]$

$\min[2, 6+11]$

$= 2.$

$A'[3-2] = \min[A^0[3,2] + A^0[3,1] + A^0[1,2]]$

$\min[\infty, 3+4]$

$= 7$

Vertex 2 as intermediatory node

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

$A^2[1-3] = \min[A^1[1,3], A^1[1,2]+A^1[2,3]$

$\min\{11, 6\}$

$A^2[1-3] = 6$

$A^2[3-1] = \min[A^1[3,1], A^1[3,2]+A^1[2,1]]$

$\min[3, 7+6]$

$A^2[3-1] = 3$

Vertex 3 as intermediatory node

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

$A^3[1-2] = \min[A^2[1,2], A^2[1,3]+A^2[3,2]]$

$= \min[4, 6+7]$

$A^3[1-2] = 4$ //

$A^3[2-1] = \min[A^2[2,1], A^2[2,3]+A^2[3,1]]$

$= \min[6, 2+3]$

$A^3[2-1] = 5$ //

Shortest path b/w all the pair of vertices.

Formula:

$$A^K[i,j] = \min\{A^{K-1}[i,j], A^{K-1}[i,K]+A^{K-1}[K,j]\}$$

Time complexity:

The Algorithm consists of 3 nested for loops so time complexity is,

Time complexity $= O(n^3)$

Algorithm:

```
for (k=1; k<=n; k++)
{  for (i=1; i<=n; i++)
   {  for (j=1; j<=n; j++)
      A[i,j] = min(A[i,j], A[i,k]+A[k,j]).
   }
}
```

4. MATRIX CHAIN MULTIPLICATION - DYNAMIC PROGRAMMING

ALGORITHM

Input : Dimensions of the matrices

Output : Order of matrix multiplication .

```
Matrix Chain Multiplication (d,n)
    for l in range (2,n)
        for i in range (1,n-l+1)
            j = i+l-1
            m[i][j] = ∞
            for k in range (i,j)
            r= m[i][k]+m[k+1][j]+d[i-1]*d[k]*d[j]
            if r< m[i][j]
                s[i][j] =k
                m[i][j]= q .
    print ( m[1][n-1] , minimum cost)
    print (order =, inorder (s, 1-n, n))

inorder (s, i, j, n)
    if (s[i][j] != 0)
        print ("(", end=" ")
        inorder (s, i, s[i][j], n)
    if s[i][j]==0
        if (i==n-1 and j==n-1)
            print ("M", n-1)
        else
            print (M, s[i][j+1])
    if s[i][j] != 0
        inorder (s, s[i][j]+1, j, n)
        print (" )")
```

5a) Write an Algorithm for the following using Back Tracking. (24)

A) Graph Coloring B) Hamiltonian cycle C) N-queen Problem

D) Sum of subset Problem.

A) Graph coloring:

A graph and some colours will be given. We've to colour the vertices of the graph such that NO 2 vertices adjacent to each other have the same colour. So, Given a graph we need to know the minimum no. of colors to color a graph.

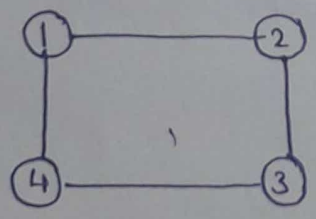
Algorithm:

Step 1: Arrange the vertices of the graph in some order.

Step 2: Choose the first vertex and color it with the first color.

Step 3: Choose the next vertex and colour it with the ~~first colour~~ lowest numbered colour that has not been coloured on any vertices adjacent to it. If all the adjacent vertices are coloured with this colour, assign a new colour to it. Repeat this step untill all the vertices are coloured.

Eg:-



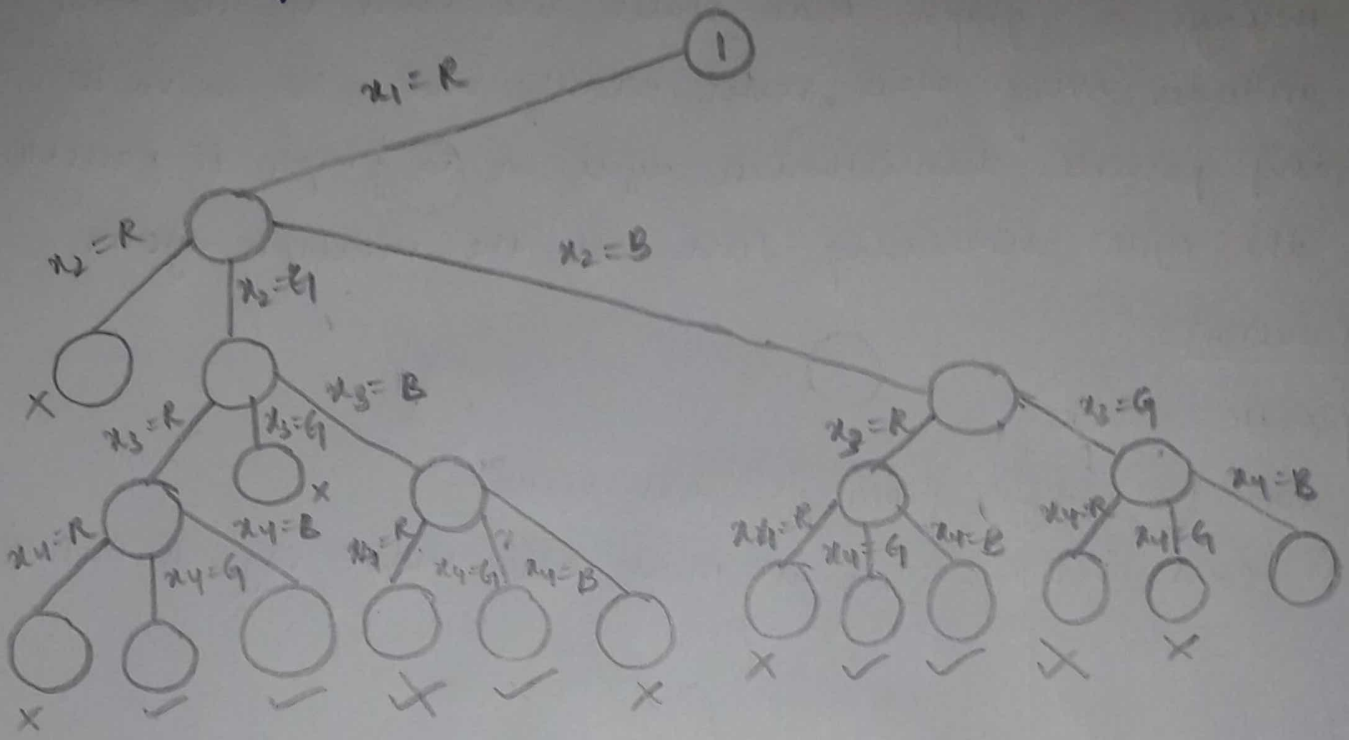
m = 3

{R, G, B}

Bounding function: NO 2 Adjacent can have same colour.

d(u, v) ≠ ∞ then they are not adjacent.

Backtracking:



So like that we can find many solutions:
nine solutions from the above backtracking

i) RGRG ii) RGRB iii) RGBG iv) RBRG v) RBRB.

Time complexity:

Time complexity of Graph colouring problem is $O(n \cdot m^n)$ as we have maximum no. of m^n combinations with 'm' colours to be placed in 'n' places. Thus it takes maximum m^n to check each combination.

m colouring Decision Problem:- If a graph is given and some colours are also given, if we want to know whether that graph can be coloured with using the given colours i.e., if we want to know only whether it can be coloured or not is known as m colouring Decision problem.

m colouring optimization problem: If a graph is given and we want to know minimum colours required to colour the graph it is called m-colouring optimization problem.

The smallest no. of colours required to colour a Graph G is called its chromatic number of that graph.

B) HAMILTONIAL CYCLE:

An path through a graph that starts and ends at the same vertex and includes every other vertex exactly once, so we've to check for any possible hamiltonian cycle in a graph if possible then find the cycle similarly find all the multiple cycles.

Bounding function:

1. No duplicate vertex.
2. Always have a edge from previous vertex.
3. If it is last vertex then it must also have edge to initial vertex.

Algorithm:

1. Create an empty path array
2. Add a first vertex to it.
3. Add each vertex except starting vertex.

check

1. It has to be adjacent to previous added vertex.
2. No duplicate vertex.
3. If it is last vertex then must have an edge to the starting vertex.

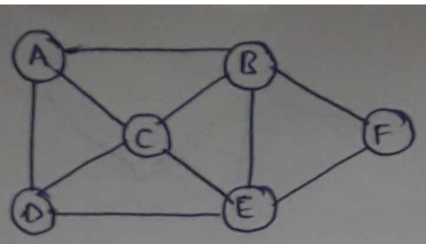
else

Backtrack and explore the remaining possibilities.

Time complexity:

The time complexity of ~~graph colouring~~ Hamiltonian cycle problem is $O(n^n)$ since it tries for n! ways.

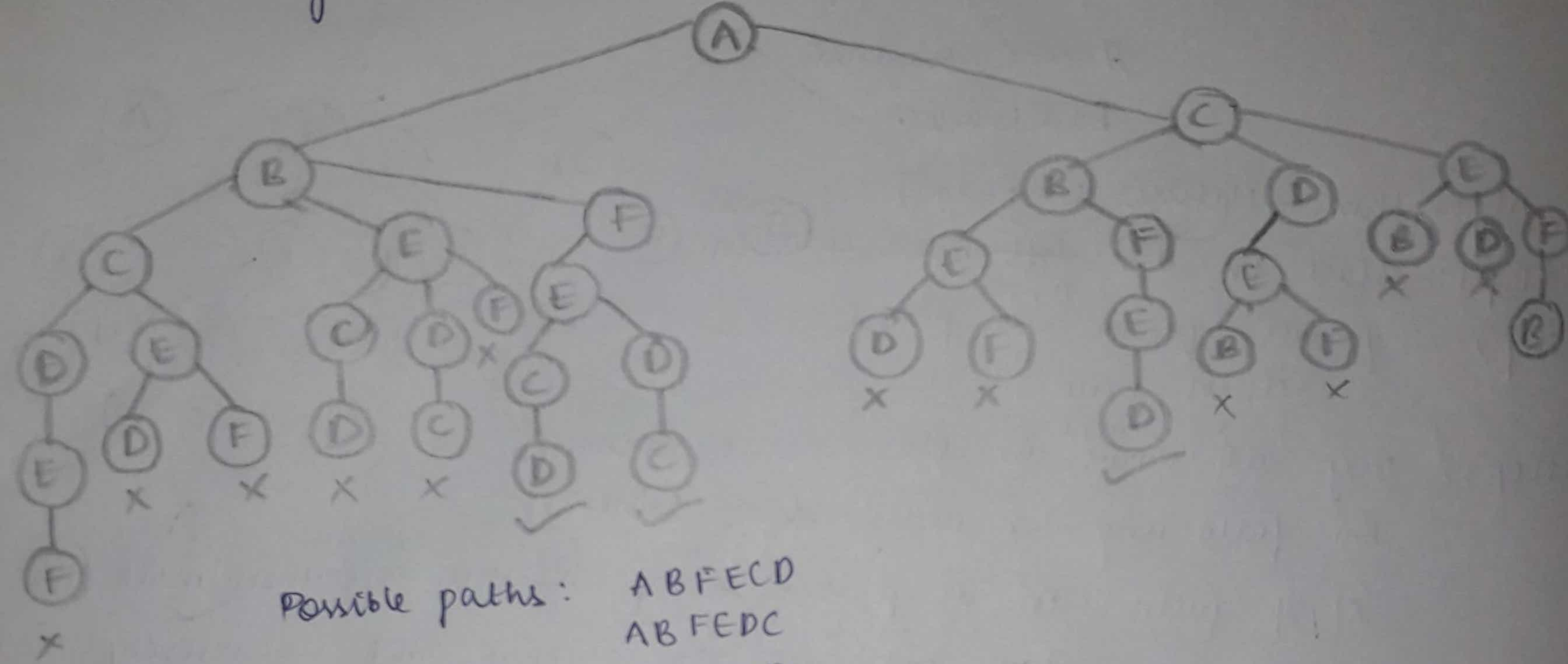
Eg:-



A	~~A~~ B	~~A~~ ~~B~~ C	~~A~~ ~~B~~ ~~C~~ D	~~A~~ ~~B~~ ~~C~~ ~~D~~ E	~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ F

No path from F-A so X

Backtracking:



Possible paths: ABFECD
ABFEDC
ACBFED - - -

c) N-QUEEN PROBLEM:

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on board. The check queens can attack in any direction as horizontal, vertical and diagonal way.

Algorithm:

step1: If all queens are placed at right position then return true

else

step2: for each queen make column choice

see whether if it is safe choice

choose place for next queen.

if not

Backtrack.

Safe choice (definition)

for all queens i

if board[i] == column

backtrack

if |ri - rc| == |ci - cc|

// same diagonal

Backtrack.

Algorithm: (Natural language)

Step 1: Start in the ~~left~~ first row most ~~column~~

Step 2: If all queens are placed.

return true

Step 3: Try all ~~rows~~ columns in the current ~~column~~ row

Do following for every tried ~~row~~ column

a) If queen can be placed safely in this ~~row~~ column then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return true.

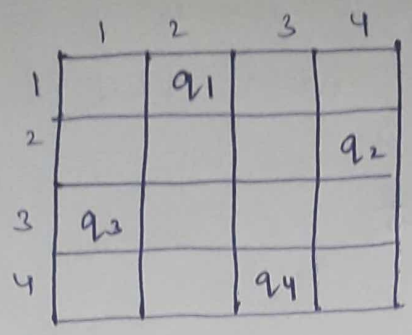
c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

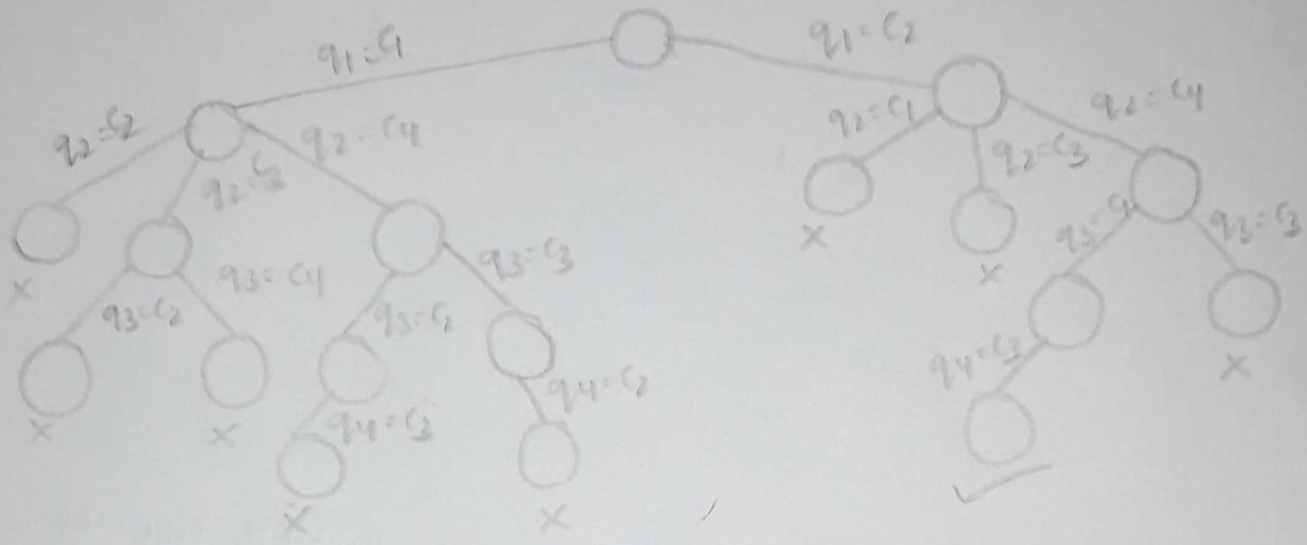
Step 4: If all the ~~rows~~ columns have been tried and nothing worked return false to trigger backtracking.

Bounding function:

Queens should not be placed in the same row (or) same column or in a diagonal.

4-Queen Problem:





columns = 2-4-1-3 [one solution]

Time complexity:

Time complexity of N-Queens problem is $O(n^n)$ since we check every position on mxn board where 'n' is no. of queens.

Time complexity = $O(n^n)$.

D) SUM OF SUBSET PROBLEM:

The subset sum problem is to find a subset's of the given set $S = \{s_1, s_2, s_3 \dots s_n\}$ where the elements of the set s are n positive integers in such a manner that ~~s'∈S and~~ the sum of the elements of subsets is equal to some positive integer X.

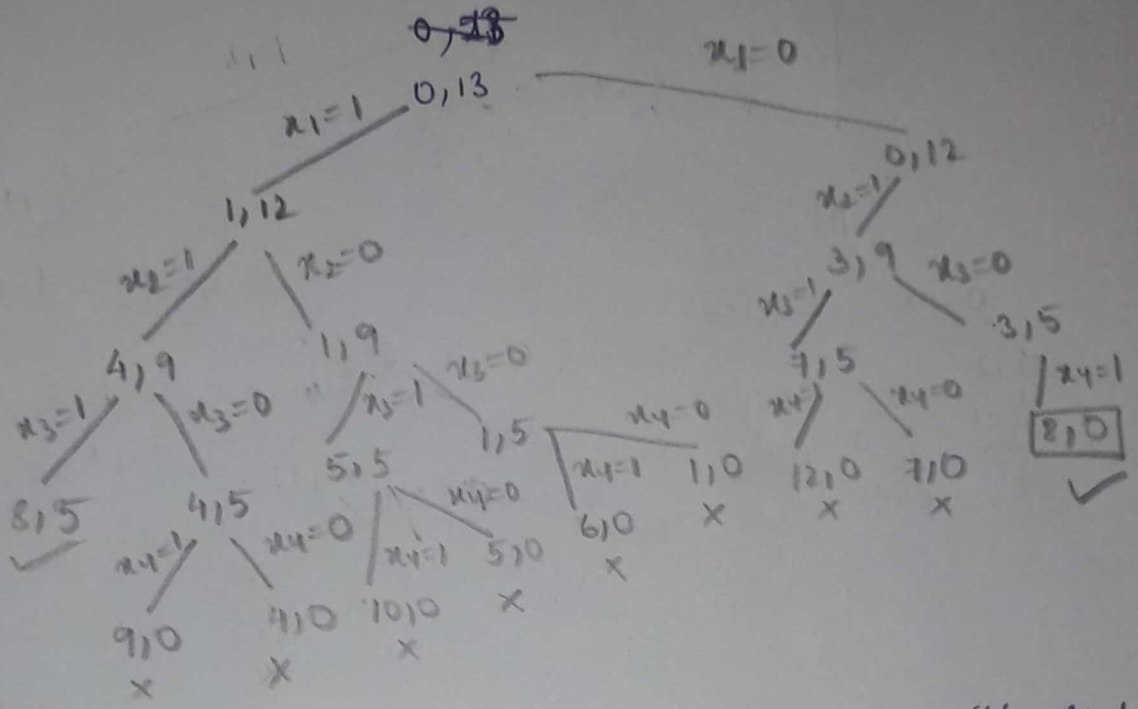
Bounding function:

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq M$$

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > M.$$

Eg:- $A = \{1, 2, 5, 9, 4\}$ $S = \{1, 3, 4, 5\}$

$M = 18$. $M = 8$.



Subset - $\{1, 3, 4\}$, $\{3, 5\}$ are two possible subsets.

Time complexity:

The time complexity of sum of subsets is $O(2^n)$

If there are n weights there will be 2^n paths.

Algorithm:

Let S be a set of elements and m is the expected sum of subsets.

Then,

Step 1: Start with an empty set

Step 2: Add to the subset, the next element from the list.

Step 3: If the subset is having sum m, then stop with that subset as solution.

Step 4: If the subset is not feasible if we've reached the end of the set then backtrack through the subset until we find the most suitable value.

Step 5: If the subset is feasible then repeat step 2.

Step 6: If we've visited all the elements without finding a suitable subset ~~then~~ and if no backtracking is possible then stop without solu-tion

6Q) Explain SAT problem and cook's Theorem.

⟹ SATISFIABILITY PROBLEM:

The satisfiability problem is based on Boolean formula called CNF (Conjunctive Normal Form).

Consider the CNF

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

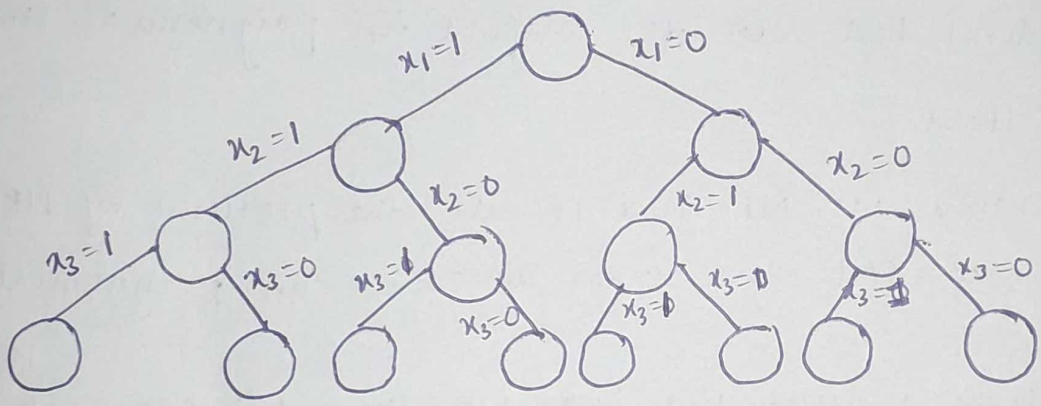
The satisfiability problem is to find out for what values of x_i this formula is true.

using boolean values $x_i = \{x_1, x_2, x_3\}$

x_1	x_2	x_3	
0	0	0	2^3 ways
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

2^n //

Representation using state space tree



> We've to show that the exponential algorithms are similar to this such that if satisfiability solved in polynomial time, all can be solved in polynomial time.

> The boolean satisfiability problem is a problem of determining if there exists an interpretation that satisfies a given boolean formula.

> In other words it asks for whether variables of a given boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to true. If this is the case then the formula is satisfiable.

> Hence the satisfiability problem takes exponential time to solve.

> So satisfiability problem is included in class of NP.

> If one problem in the set can be solved in polynomial time, then any other problem in the same set can be solved in polynomial time

⟹ COOK-LEVIN THEOREM:

The problem of determining whether a boolean expression is satisfiable is NP complete. SAT is NP complete.

> To prove that problem is NP complete, we need to prove that
- The problem is NP.
- The problem is NP-Hard.

> The problem is NP:
- The problem needs to be solved in non-polynomial time. And need to be verified the solution in polynomial time.
- $(p \vee q \vee r)$. It takes 2^n combinations, hence it can be solved in exponential time but can be verified in polynomial time.

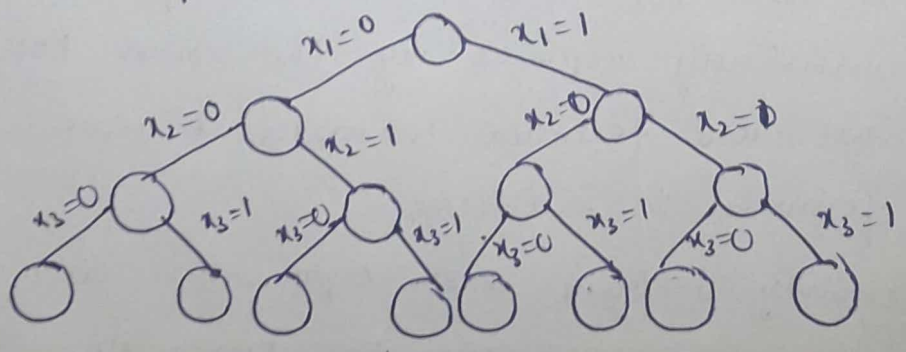
> The problem is NP Hard:
- SAT can be proved as NP-Hard if all the problems of NP can be reduced to SAT and can be solved in polynomial time.

eg:- If satisfiability reduces 0/1 knapsack then 0/1 knapsack also becomes NP Hard.

NP Hard $\propto L_1$ } Transitive $L_1 \propto L_2$ also becomes NP Hard because L_1 is reduced by NP Hard problem and L_2 is reduced by L_1.

> For conversion also we take polynomial time.

Proof:- state space tree for SAT Problem



$x_1 \vee x_2 \vee x_3$

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

2^n possib

Now if you consider 0/1 knapsack problem

$x_1\ x_2\ x_3$

objects = 1, 2, 3

p = 10, 8, 12 $\quad n=3$

w = ~~20, 25, 9~~ 5, 4, 3 $\quad m=8$

x_i = {1/0, 1/0, 1/0}

Each object may or may not be included in the bag so,

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
:	:	:
1	1	1

Here we've to see whether formula is true or not, In this case we've to check whether profit is maximized or not. So, state space tree can solve both the problem. So this is called relation.

> If we solve the state space tree in polynomial time then both the problems can be solved in polynomial time.

Hence Proved that SAT is NP-Hard.

7Q) what is NP-complete and NP-Hard Problems. Given a problem x how will you prove that it is

1) NP complete 2) NP Hard.

NP-complete:

> A problem is said to be NP-complete if and only if

i) x is in NP

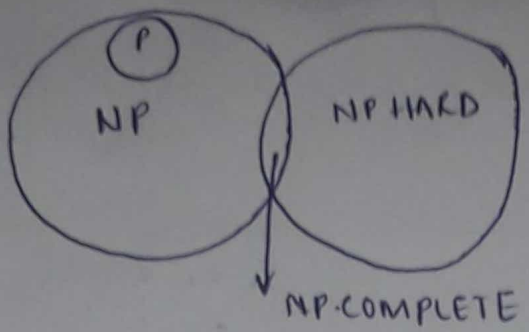
ii) x is in NP-Hard.

If a problem is having non-deterministic algorithm then it becomes NP-complete.

A problem x is NP-complete if

1. It is in NP and
2. For every other problem x' in NP, $x' \propto x$

Cook's theorem: Satisfiability is NP complete. This was the first problem to be shown NP-complete.



Relationship between P, NP, NP complete and NP-Hard classes.

NP Hard: is a class of problems to which ~~are NP-Hard and~~ ~~beta~~ every NP problem reduces.

A problem ns called NP Hard if it has following properties

If it has a polynomial time algorithm that solves this problem then there one for all problems in NP.

> A problem A is NP-Hard if

For every other problem A' in NP, $A' \propto A$.

To prove that a problem X is NP complete and NP Hard

> The problem 'x' should be directly or indirectly related to satisfiability, if this is true then it becomes NP-Hard

> If the problem 'x' has Non-deterministic algorithm then it will be under a class called NP-complete.

Satisfiability is the known NP-complete problem.

$$\text{Satisfiability} \propto \underset{\text{NP-Hard}}{X} \xrightarrow[\text{Algorithm}]{\text{Non deterministic}} \text{NP-complete}.$$

If X is reduced by satisfiability it will be NP-Hard

If there is Non deterministic Algorithm for X then it is NP-complete.