ADA

Algorithm

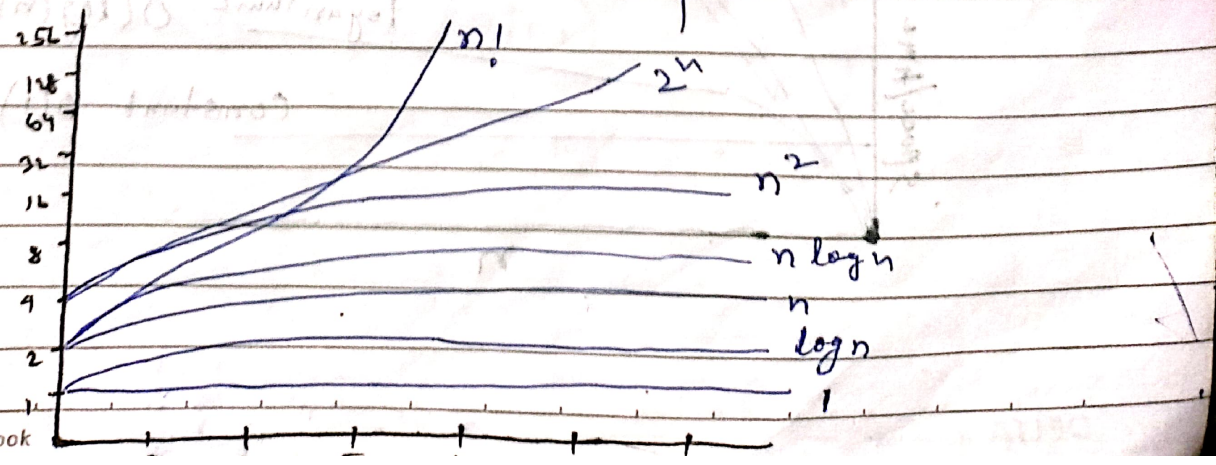
An algorithm is a step by step procedure to solve a particular problem.

In ADA, an Algorithm is evaluated based on its correctness, efficiency and complexity.

Following are the parameters can be considered while analysis of an Algorithm.

1. Time
2. Space
3. Bandwidth
4. Register
5. Battery power

In Asymptotic Analysis, we evaluate the performance of an Algo. in terms of input size. We calculate how does the time or space taken by an Algorithm increase with the input size.

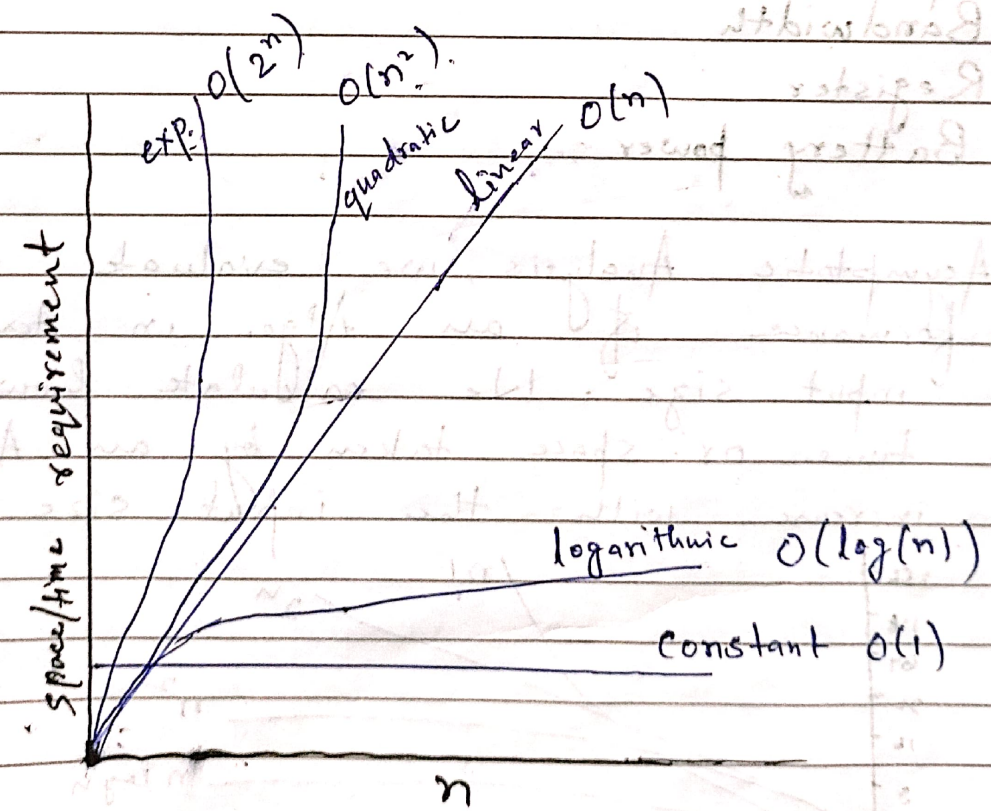


Asymptotic Notation

Asymptotic Notation is used to Analyze the growth of an Algorithm's running time in relation to the input size.

It helps in comparing the efficiency of different algorithms.

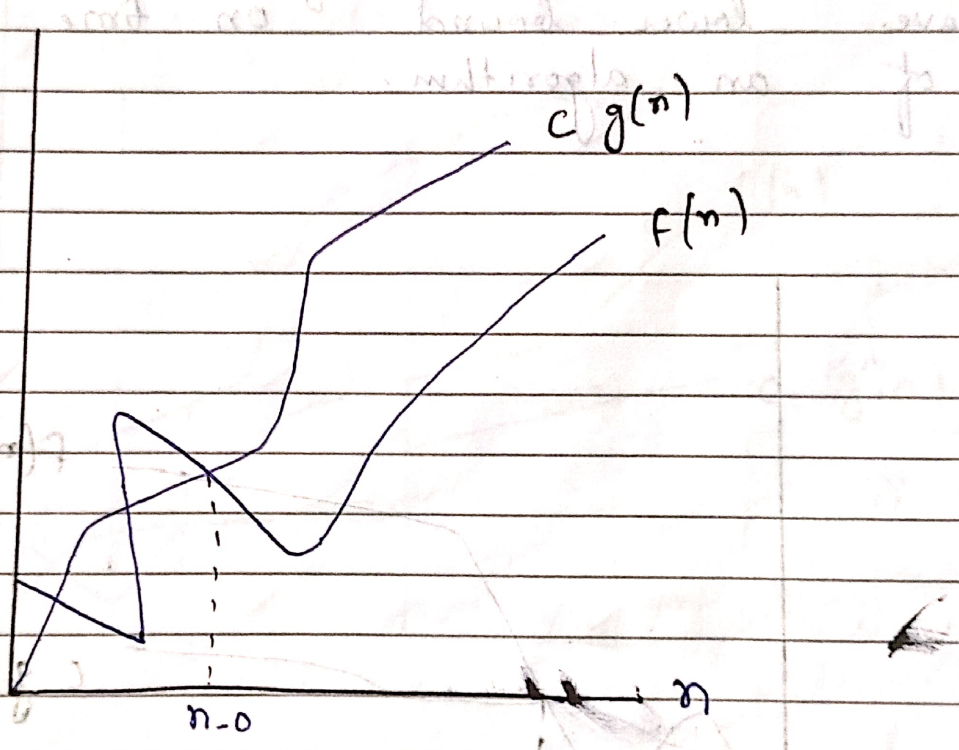
Asymptotic notations are Abstract notation for describing the behaviour of algorithm and determine the rate of growth of a function.



Big-O Notation

The Big O notation defines an upper bound of an algorithm, it bound a function only from above.

It Represents the worst-case time complexity of an Algorithm and defines the maximum amount of time an algorithm take.



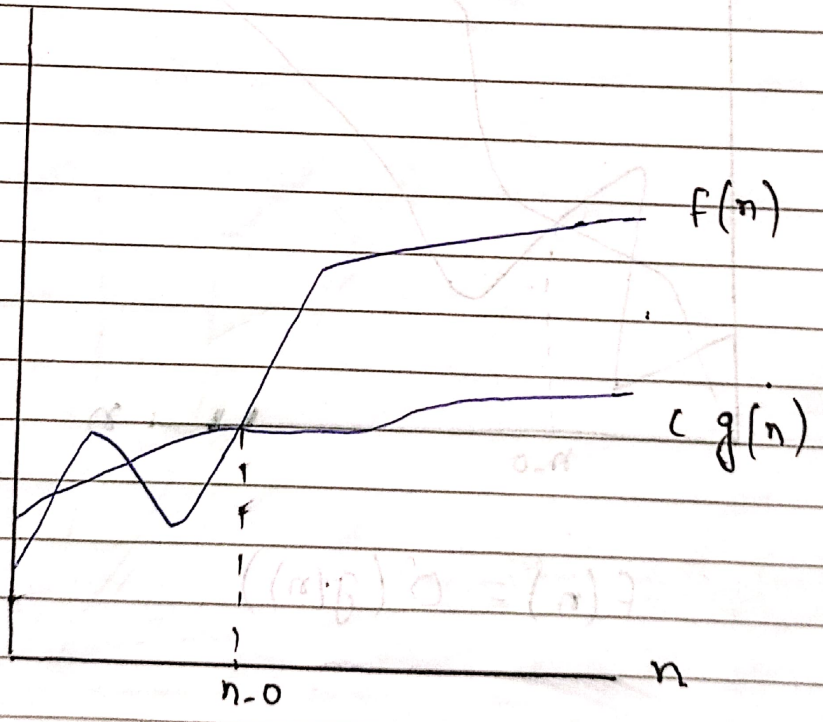
$$f(n) = O(g(n))$$

Ω (omega) Notation

Ω notation provides an asymtotic lower bound of a function.

It Represents the best case time complexity and defines the minimum amount of time an Algorithm will take.

Ω notation can be useful when we have lower bound on time complexity of an algorithm.

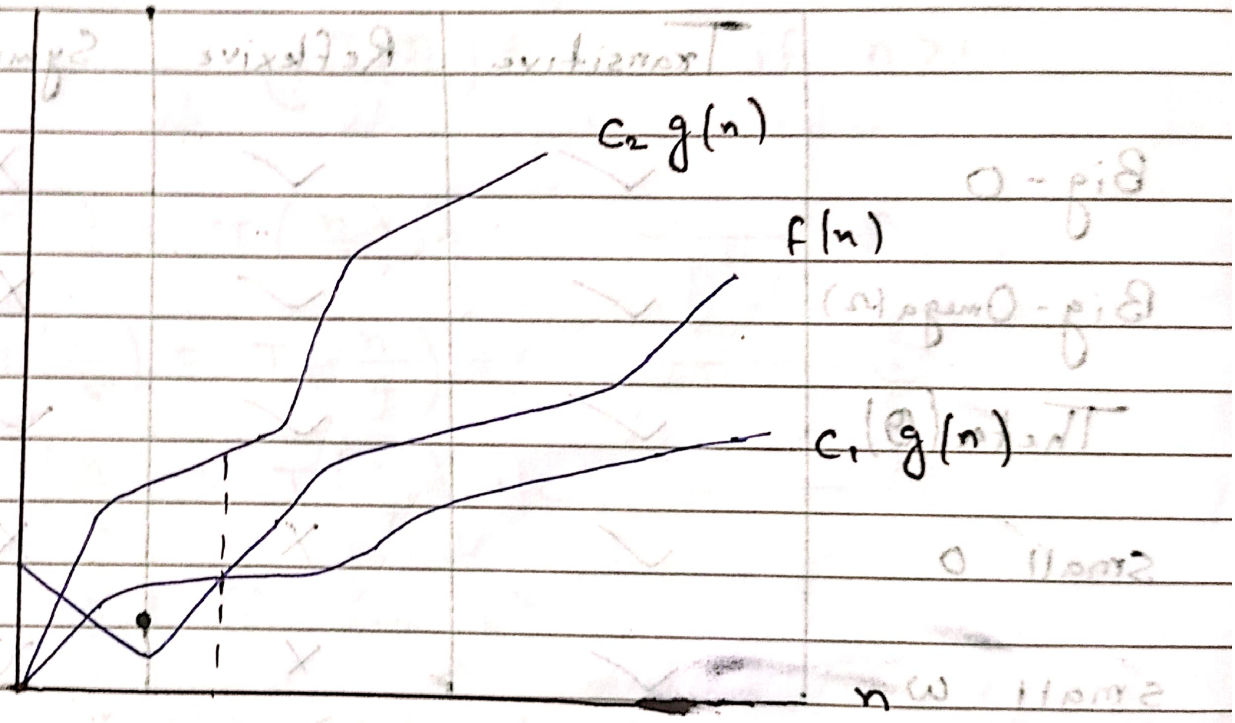


$f(n) = \text{Omega}(g(n))$

Theta Notation

The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour.

It Represents the average-case time complexity and provides both upper and lower bounds.



$f(n) = \text{theta}(g(n))$

O — $f(n) \leq c \cdot g(n)$

Ω — $f(n) \geq c \cdot g(n)$

Θ — $c \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

small o — $f(n) < c \cdot g(n)$

small ω — $f(n) > c \cdot g(n)$

	Transitive	Reflexive	Symmetric
Big-O	✓	✓	✗
Big-Omega (Ω)	✓	✓	✗
Theta (Θ)	✓	✓	✓
small o	✓	✗	✗
small ω	✓	✗	✗

Ascending Order of Complexity

$O(1) < O(\log \log n) < O(\log n) < O(n^{1/2}) < O(n)$

$< O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n^n)$

$< O(2^{2^n})$

Recurrences

1. Substitution Method
2. Master Theorem / Method
3. Recursion Tree Method

Substitution Method

Que.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + c & \text{if } n > 1 \end{cases}$$

$$T(n) = T\left(\frac{n}{2}\right) + c \quad \text{——— (1)}$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + c \quad \text{——— (2)}$$

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + c \quad \text{——— (3)}$$

put (2) in (1)

$$T(n) = T\left(\frac{n}{4}\right) + c + c$$

$$T(n) = T\left(\frac{n}{4}\right) + 2c \quad \text{——— (4)}$$

put (3) into (4)

$$T(n) = T\left(\frac{n}{8}\right) + c + 2c$$

$$T(n) = T\left(\frac{n}{8}\right) + 3c$$

After 'K' steps

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

$$T\left(\frac{n}{2^K}\right) = T(1)$$

$$\frac{n}{2^K} = 1$$

$$n = 2^K$$

Taking log both side

$$\log n = \log 2^K$$

$$\log n = K \log 2$$

$\because \log 2 = \text{constant}$

$$\boxed{K = \log n}$$

Substitute $K = \log n$

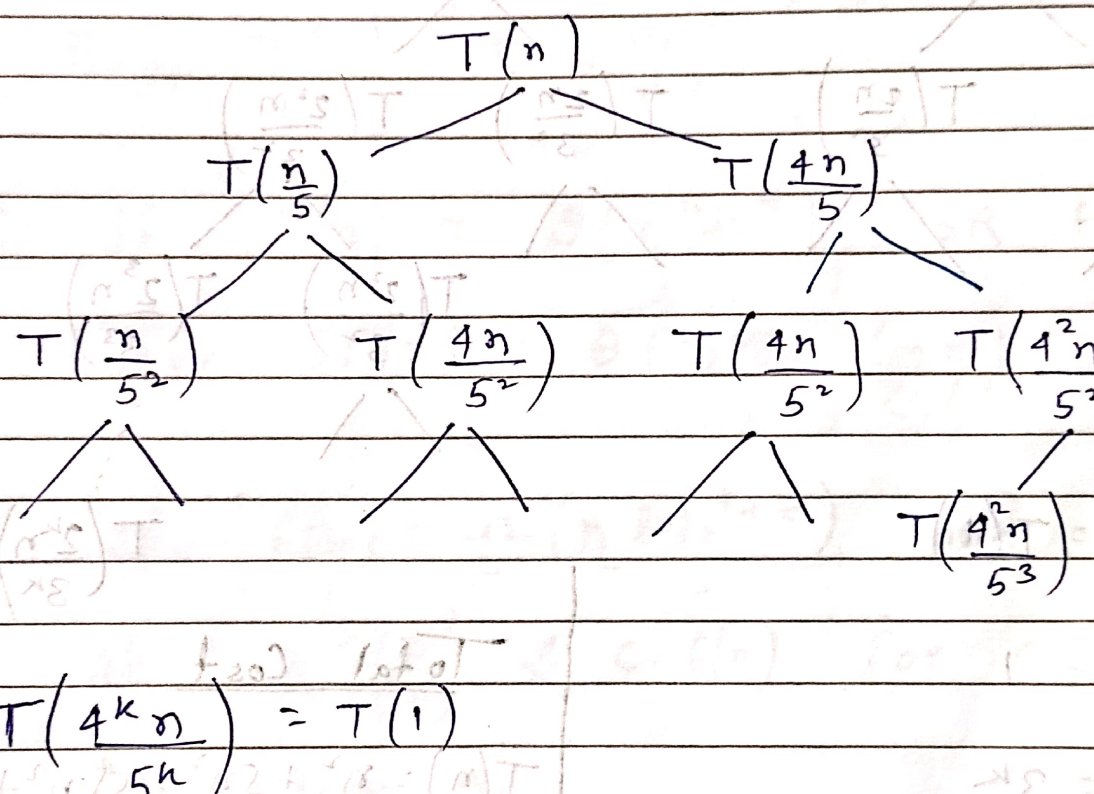
$$T(n) = T(1) + (\log n)\, c$$

$$T(n) = 1 + \log n \cdot c \leq \text{const.}\ g(n)$$

$$\boxed{T(n) = O(\log n)}$$

Recursion Tree Method

Que $T(n) = \begin{cases} T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$



Cost

n

+

n

+

n

.

.

.

L

$T\left(\frac{4^k n}{5^k}\right) = T(1)$

$\frac{4^k n}{5^k} = 1$

$4^k n = 5^k$

Taking log both side

$\log(4^k n) = \log(5^k)$

$k \log 4 + k \log n = k \log 5$

$k \log 5 - k \log 4 = k \log n$

$k \log \frac{5}{4} = \log n$

$\boxed{k = \log n}$ $\because \log \frac{5}{4} = 0$

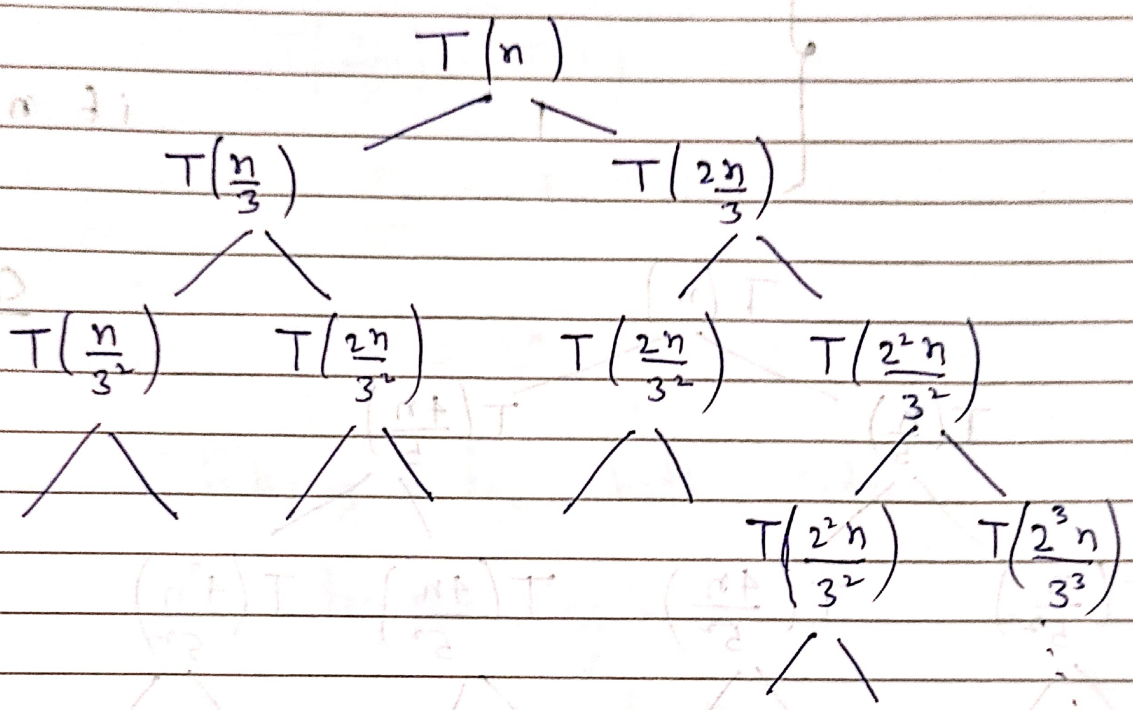
Total cost :-

$= n \cdot \log n$

$= O(n \log n)$

Delta

Que $T(n) = \begin{cases} T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n^2 & \text{if } n > 1 \\ 1 & \text{if } n \le 1 \end{cases}$



Cost

n^2

$\frac{5n^2}{9}$

$\frac{25n^2}{81}$

$\frac{5kn^2}{9k}$

$T\left(\frac{2^k n}{3^k}\right) = T(1)$

$\frac{2^k n}{3^k} = 1$

$2^k n = 3^k$

$\log 2^k + \log n = k \log 3$

$k \log 3 - k \log 2 = \log n$

$k \log \frac{3}{2} = \log n$

$\boxed{k = \log n}$ $\because \log \frac{3}{2} = \text{const.}$

Total Cost

$T(n) = n^2 + \frac{5n^2}{9} + \left(\frac{5}{9}\right)^2 n^2 + \cdots$

$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}$

$n^2 \sum_{k=0}^{\log n} 1 \cdot \left(\frac{5}{9}\right)^k = n^2 \cdot \frac{1}{1-\frac{5}{9}}$

$T(n) = n^2 \cdot \frac{9}{4}$

$T(n) = O(n^2)$

Master Method:

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

where $a \geq 1$, $b > 1$

① If $f(n) = O(n^{\log_b a - \epsilon})$, $\epsilon > 0$ OR $f(n) < n^{\log_b a}$

Then $T(n) = \Theta(n^{\log_b a})$

② If $f(n) = \Theta(n^{\log_b a})$ OR $f(n) = n^{\log_b a}$

Then $T(n) = \Theta(n^{\log_b a} \log n)$

③ If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and OR $f(n) > n^{\log_b a}$

If $a f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for $c < 1$ and very large n

Then $T(n) = \Theta(f(n))$

Que $T(n) = 9T\left(\frac{n}{3}\right) + n$

a → a, b → b, n → $f(n)$

$a = 9$, $b = 3$, $f(n) = n$

$f(n) = O(n^{\log_b a - \epsilon})$

$n = O(n^{\log_3 9 - \epsilon})$

$n = O(n^{2-\epsilon})$ $\qquad \epsilon = 1$

$T(n) = \Theta(n^{\log_3 9})$

$T(n) = \Theta(n^2)$

Linear Search

Linear Search is a simple searching Algo. that checks each element in a list one by one until the target element is found.

It is Also known as Sequential Search.

Algorithm

Step 1: Begin

Step 2: Start from the first element of the list.

Step 3: Compare each element with the target element

Step 4: If a match is found, return the index of the element

Step 5: If no match found return -1 or indicates that element is not present.

Step 6: Exit.

DL:
PD: Delta

Time Complexity

Best Case :- O(1) - When the target element is found at first position.

Worst Case :- O(n) - When the target element is at the last position or not present at all.

Average Case :- O(n) - When the element is somewhere in the middle or not present.

Binary Search

Binary Search is an efficient searching algorithm used to find an element in a sorted array.

Algorithm

Step 1: Begin

Step 2: Find the middle element of the Array.

Step 3: If the middle element is equal to the target, return its index.

Step 4: If the middle element is greater than the target, search the left half.

Step 5: If the middle element is smaller than the target, search the right half.

Step 6: Repeat the process until the element is found or the search space becomes empty.

Step 7: Exit

$$mid = \left[\frac{l+h}{2}\right]$$

Stoping Criterea :- $h < l$

Time Complexity

Best Case :- $O(1)$ - If the middle element is the target.

Worst/Average Case :- $O(\log n)$ - As we divide the search space in half

Delta

Matrix Multiplication using Divide & Conquer

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2\times 2} \qquad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_{2\times 2}$$

$$C = A \times B = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

where,

$C_{11} = a_{11}b_{11} + a_{12}b_{21}$
$C_{12} = a_{11}b_{12} + a_{12}b_{22}$
$C_{21} = a_{21}b_{11} + a_{22}b_{21}$
$C_{22} = a_{21}b_{12} + a_{22}b_{22}$

$P \times \underbrace{Q \quad Q}_{} \times R$

$= P \times R$

$$C[i,j] = \sum_{k=1}^{n} a[i,k] * b[k,j]$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & A_{11} & & A_{12} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ & A_{21} & & A_{22} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ & B_{11} & & B_{12} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ & B_{21} & & B_{22} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$
$C_{12} = A_{11}B_{12} + A_{12}B_{22}$
$C_{21} = A_{21}B_{11} + A_{22}B_{21}$
$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2$$

$$= O(n^3)$$

Strassen's Matrix Multiplication

$$P = (A_{11} + A_{22})(B_{21} + B_{12})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{21} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} - A_{12})B_{22}$$

$$U = (A_{21} - A_{22})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Delta

String Matching

Text [1 . . . n]

Pattern [1 . . . m]

$m \leq n$

Naive String Matching

It is the simplest method for pattern matching.

Performs checking at all position in the text, whether an occurence of the pattern starts there or not.

After each attempt the Algorithm shifts the pattern by exactly one position to the right.

Text a d a b c a a b

Pattern a b c a

$i = 2$

$j = 1$

Delta

Algorithm

```
n = T.length
m = P.length
for s=0 to n-m
    if P[1...m] == T[s+1...s+m]
        print "Pattern found with shift" s
```

Example :-

Text acaabc

Pattern aab

pattern found with shift 2

Time Complexity -

Best Case - O(n)

Worst Case - O(m·n)

Average Case - O(m·n)

DL:
PD: Delta

KMP: Knuth - Morris Pratt Algo.

This Algorithm works on proper prefix and proper suffix.

π-table of String

String 1.

	1	2	3	4
	a	b	a	b
	0	0	1	2

String 2

	1	2	3	4	5	6	7
	a	b	c	d	a	b	c
	0	0	0	0	1	2	3

Example on KMP Algorithm

Text : a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Pattern : a b a b d

π-table

J	1	2	3	4	5
0	a	b	a	b	d
	0	0	1	2	0

* It is better than Naive String Matching.

Delta

Algorithm

(i) Take two variables i and j

i = string (T(1)) , j = P[0]

(ii) Compare T(i) with P(j+1)

(i) if Match is found (Move both i and j to right)

(ii) if Mismatch (Move j to the location as per π table index)

(iii) if j = 0 (Move i to the right)

Pros

→ The Algo. ensures that the characters of the text are never compared more than once.

→ No backtracking needed.

Cons

→ The preprocessing step requires additional time and memory.

→ The Algorithm is more complex to understand and implement.

Rabin Karp String Matching

The Rabin-karp algorithm uses hashing to find any set of pattern occurences. Instead of checking all occur characters of pattern at every position, it checks a hash value.

Text: a a a a b a

Pattern: a b a

97 + 98 + 97

= 292

ASCII Value

a = 97

b = 98

But In Some Case there will be False positive.

Text: c c a c c a a e d b a

Pattern: d b a

4 + 2 + 1

= 7

3 + 3 + 1

= 7 false positive

Hash func

a = 1

b = 2

c = 3

d = 4

Method to Remove False Positive

aba - $1 \times 2^2 + 2 \times 2^1 + 1 \times 2^0$

= 9

aab - $1 \times 2^2 + 1 \times 2^1 + 2 \times 2^0$

= 8

Delta

Pros

→ Faster than the naïve approach on average

→ Very efficient for multiple pattern searches at once.

Cons

→ Requires a good hash function to avoid frequent spurious hits.

→ The worst time complexity can be as bad as the Naive algorithm if many hash collision occur.

String Matching with Finite Automata

Finite Automata = $\{Q, \Sigma, q_0, \delta, F\}$

Q = finite non-empty set of state
Σ = input Alphabet {a,b,c...z}, {1,2,...9}
q_0 = Initial state
δ = transition function ($\delta : Q \times \Sigma \to Q$)
F = Final state.

Algorithm

```
Finite Automata [T, δ, m]
n ← length[T]

q ← 0
    for i → 1 to n
        do q → δ(q, T[i])
            if q == m

                then print "Pattern occur
                with shift" i-m.
```

Example:-

Text: a b a b a b a c a b a

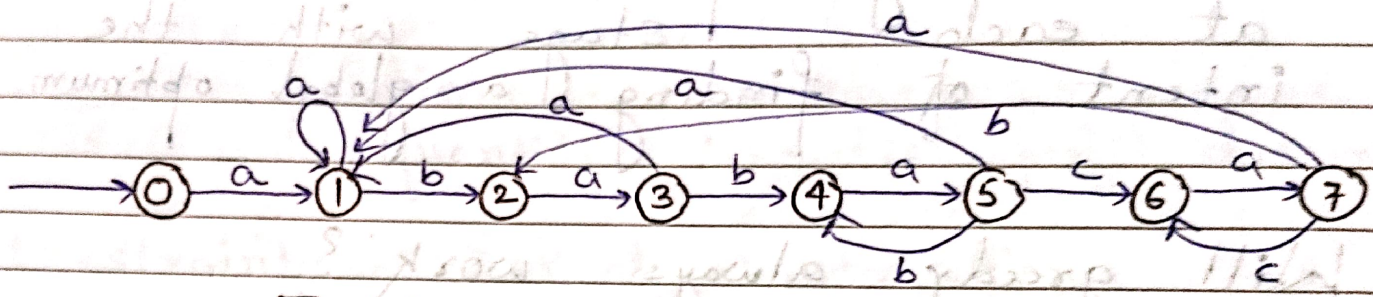
Pattern: a b a b a c a

$\Sigma = \{a, b, c\}$

$Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$

$q_0 = \{0\}$

$F = \{7\}$



States	Inputs a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

1 2 3 4 5 6 7 8 9 10 11

T = a b a b a b a c a b a

0 1 2 3 4 5 4 5 6 7

pattern occured with shift i-m

= 9 - 7

= 2

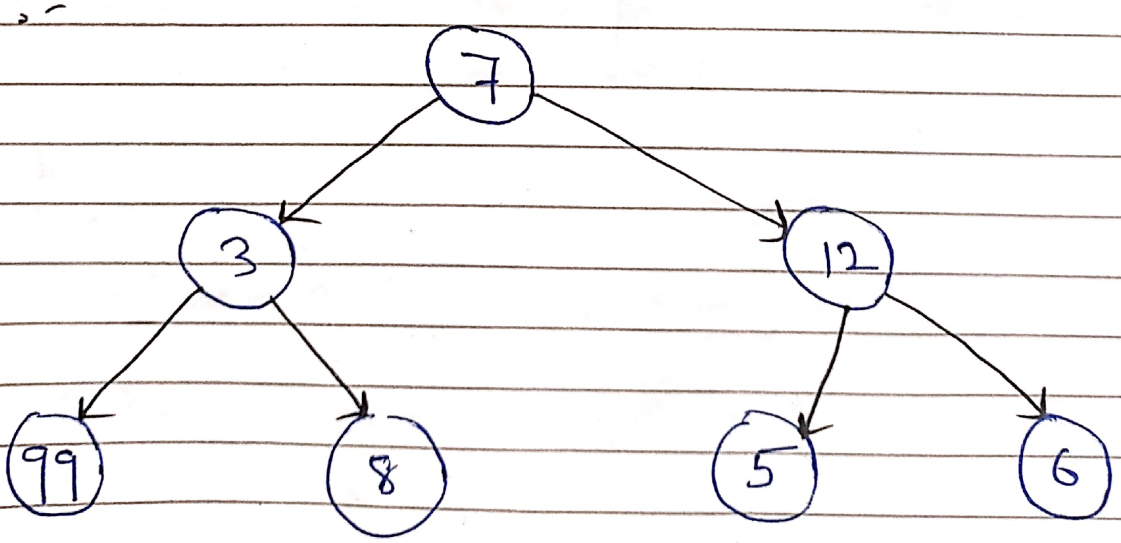
Greedy Algorithm

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.

Will greedy always work?

In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Eg:-



7 → 12 → 6

Optimization Problem

An optimization problem is a type of problem that seeks to find the best solution from all feasible solutions.

Objective :- Minimize or maximize some quantity (like cost, profit, distance).

Constraints :- Set of restrictions or conditions that the solution must satisfy.

Feasible Solution :- A solution that meets all constraints.

Optimal Solution :- A feasible solution that yields the best value of the objective function.

Knap Sack Problem

The knapsack or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

- It can be studied in two versions
  - (i) Fractional Knap Sack
  - (ii) 0/1 Knap Sack - Dynamic programming

Fractional Knap Sack

Que Consider the weights and values of items listed below. Note that there is only one unit of each item.

Max Weight = 20

Object	O_1	O_2	O_3
profit	25	24	15
Weight	18	15	10

Greedy by profit :-

Object	O_1	O_2	O_3
Profit	25	24	15
Weight	18	15	10
Solution	1	2/15	0

$25 \times 1 + 24 \times \frac{2}{15} + 15 \times 0$

$= 25 + 3.2 + 0$

$= 28.2$

Greedy by Weight :-

Object	O_1	O_2	O_3
Profit	25	24	15
Weight	18	15	10
Solution	0		

$25 \times 0 + 24 \times \frac{10}{15} + 15 \times 1$

$0 + 16 + 15$

$= 31$

Greedy by Profit/Weight :-

Object	O_1	O_2	O_3
Profit	25	24	15
Weight	18	15	10
Profit/Weight	1.38	1.6	1.5
Solution	0	1	5/10

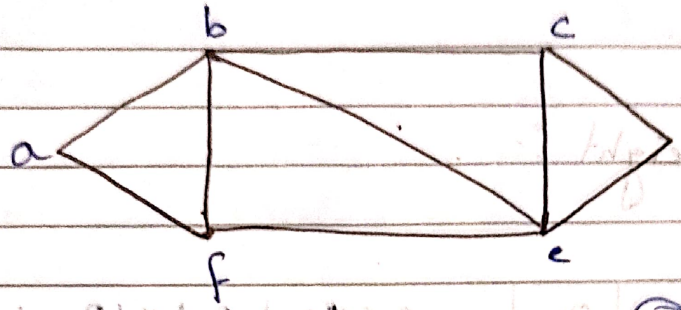
$25 \times 0 + 24 \times 1 + 15 \times \frac{5}{10}$

$24 + 7.5$

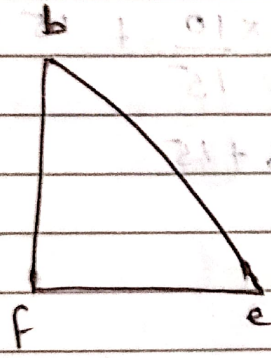
$= 31.5$

Spanning Tree

A tree T is said to be spanning tree of a connected graph G, if T is a subgraph of G and T contains all vertices of G.

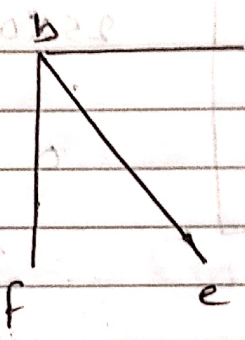


(i)



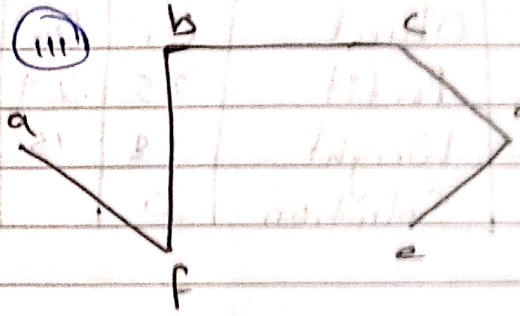
No
(Cycle)

(ii)

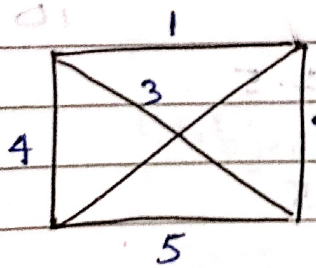


No

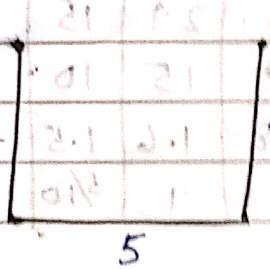
(iii)



Yes

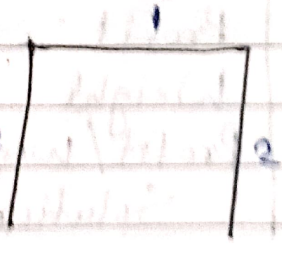


Undirected Graph



Spanning Tree

Cost = 4 + 5 + 2
= 11



Minimum Spanning Tree

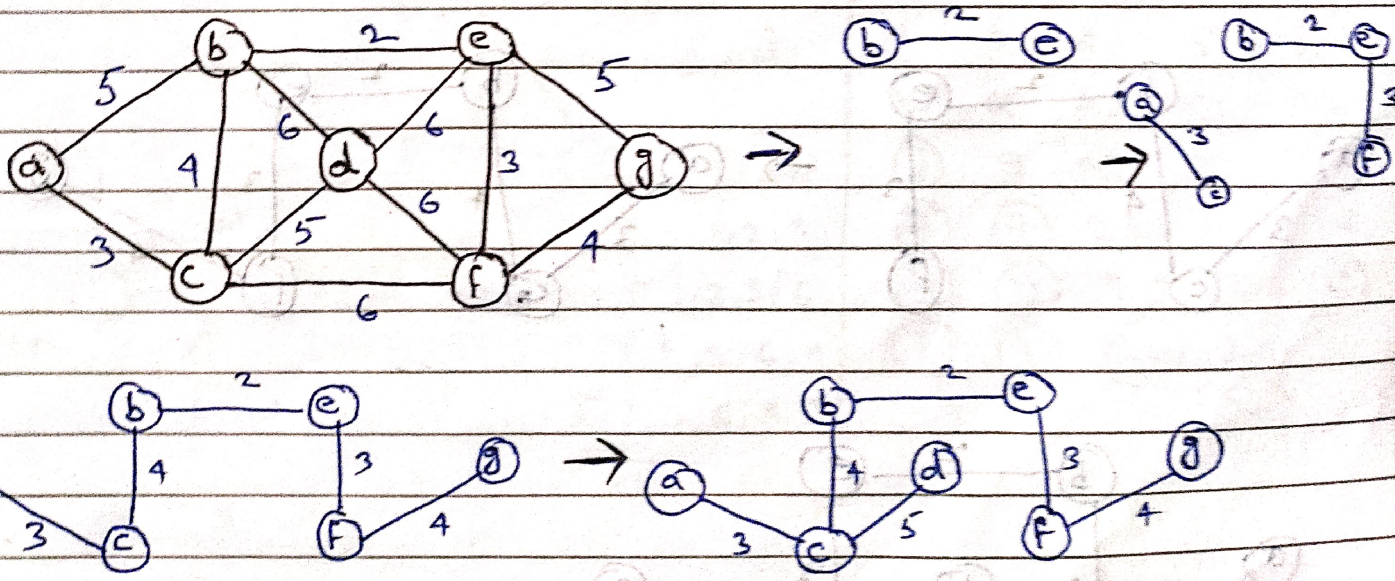
Cost = 4 + 1 + 2
= 7

Minimum Spanning Tree :-

MST is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Kruskal Algorithm

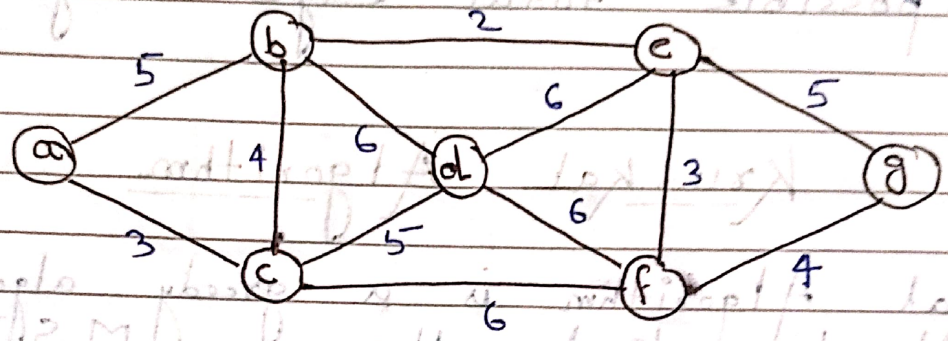
Kruskal Algorithm is a greedy algorithm used to find the MST of a connected undirected, and weighted graph. The goal is to connect all the vertices with the minimum total edge weight, without any cycle.

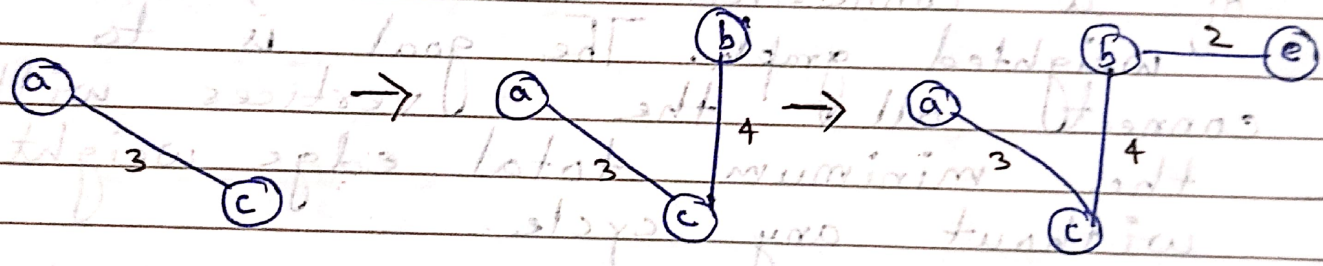


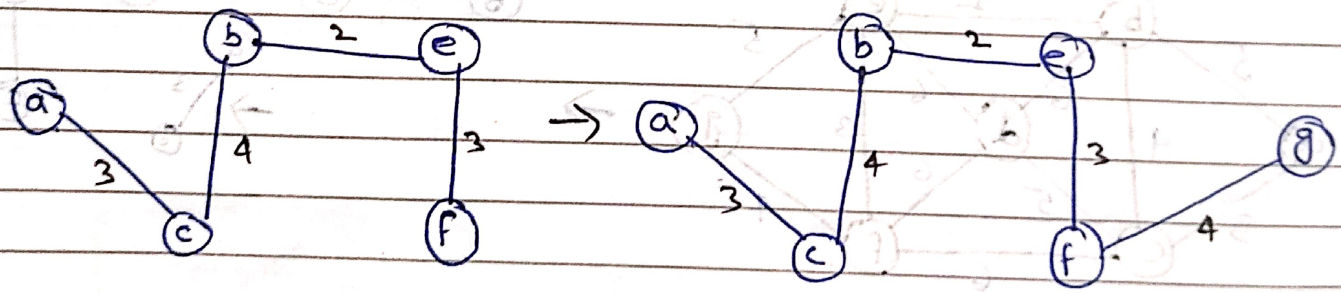
Prim's Algorithm

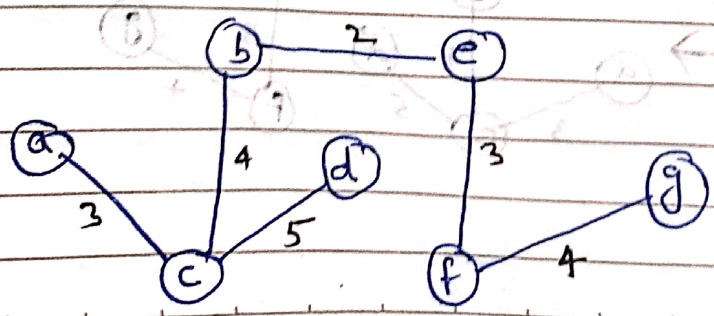
Prim's Algorithm grows the MST from a starting node, always adding the cheapest edge that connects a vertex in the MST to a vertex outside the MST.

Eg:-







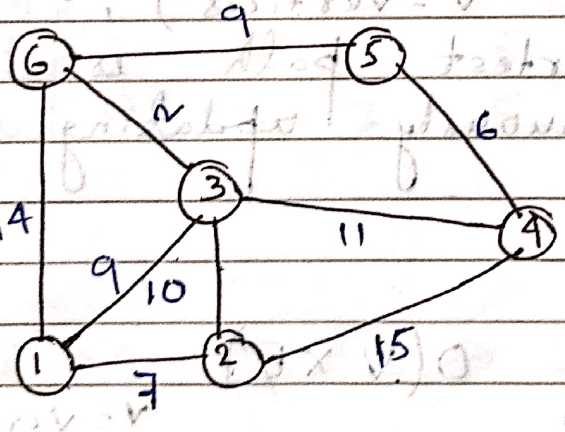


Single Source Shortest Path

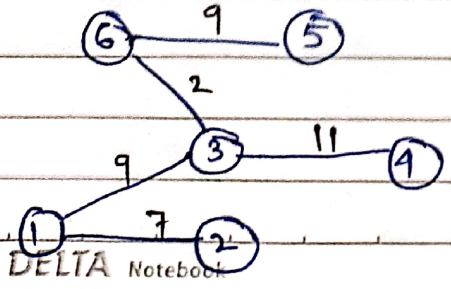
In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

Dijksta's Algorithm

This Algorithm is used for finding the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights.



Source	Destination				
1	2	3	4	5	6
	∞	∞	∞	∞	∞
1, 2	(7)	9	∞	∞	14
1, 2, 3	(7)	(9)	22	∞	14
1, 2, 3, 6	(7)	(9)	20	∞	(11)
1, 2, 3, 6, 4	(7)	(9)	(20)	20	11
1, 2, 3, 6, 4, 5				(20)	



Drawbacks of Dijkstra's Algorithm

Does not Work with Negative Weights

Reason: Dijkstra assumes that once a node's shortest distance is found, it cannot be improved.

Solution: Use Bellman-Ford Algorithm if the graph has negative edge weights.

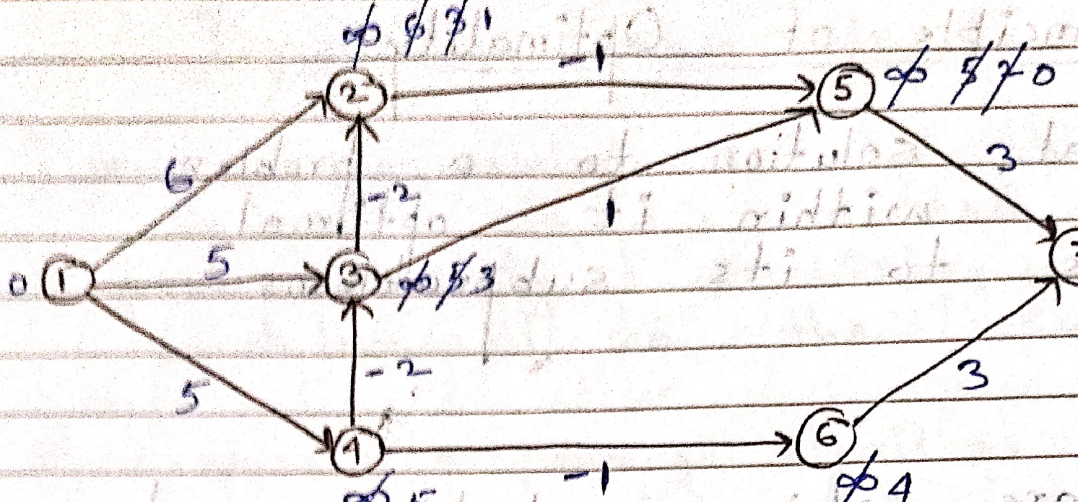
Bellman-Ford Algorithm

Bellman-Ford relaxes all the edges V-1 times (where V = vertices), ensuring the shortest path is found by continuously updating distances.

Time Complexity :- $O(V \times E)$

V = vertices

E = edges



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

	1	2	3	4	5	6	7
0	∞	∞	∞	∞	∞	∞	∞
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	5
4	0	1	3	5	0	4	3
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

1 → 4 → 3 → 2 → 5 → 7 = 3

Delta

Principle of Optimality

An optimal solution to a problem contains, within it optimal solutions to its subproblems.

Means :-

If you are solving a problem and you break it into smaller parts then :

- Solving each subproblems optimally
- Guarantees that the overall solution will also be optimal.

Eg:- Suppose the shortest path from A to D is :

$A \rightarrow B \rightarrow C \rightarrow D$

Then :-

- The path from A to B must be optimal.
- The path from B to C must be optimal.
- The path from C to D must also be optimal.

Dynamic Programming

Dynamic programming is like the divide and conquer method, solve problems by combining the solutions to the subproblems.

A dynamic programming algorithms solves every subproblems just one and then saves its answer in a table there by avoiding the work of recomputing the answer every time the subproblem is encountered.

There are four steps of dynamic programming

- Characterize the solution of an optimal solution
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution in a bottom up fashion
- Construct an optimal solution from computed information

0/1 Knap Sack Problem

Problem Def^{n} :- More formally there are n number of objects $O_1, O_2, O_3 \cdots O_n$ each has a weight associated with its W_i, and a profit associated with it P_i, we can take x_i either 0 or 1.

Weight Condition $\sum_{i=1}^{n} W_i \cdot X_i <= M$

Profit $\sum_{i=1}^{n} P_i \cdot X_i$

$$KS(n,w) = \begin{cases} 0 & n=0 \text{ or } w=0 \\ KS(n-1,w) & wt[n] > w \\ \max \begin{cases} KS(n-1,w) - wt[n] + P[n] \\ KS(n-1,w) \end{cases} \end{cases}$$

Eg:-

Object	O_1	O_2	O_3	O_4
Profit	1	2	5	6
Weight	2	3	4	5

$n = 4$, $W = 8$

	P	Wt	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7	W=8
n=0	0	0	0	0	0	0	0	0	0	0	0
n=1	1	2	0	0	1	1	1	1	1	1	1
n=2	2	3	0	0	1	2	2	3	3	3	3
n=3	5	4	0	0	1	2	5	5	6	7	7
n=4	6	5	0	0	1	2	5	6	6	7	8

All pair Shortest path problem

It involves finding the shortest paths between every pair of vertices in a weighted graph.

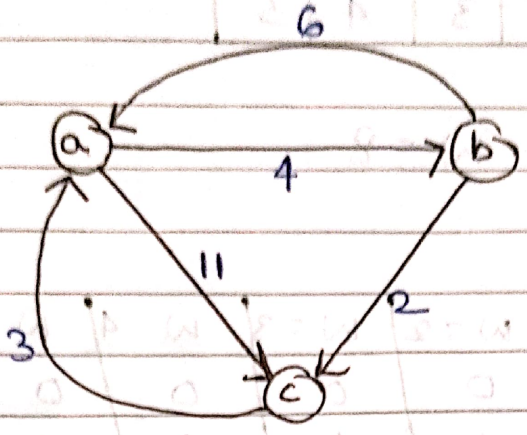
Algorithm to Solve APSP

Floyd-Warshall Algorithm

It uses a dynamic programming approach to incrementally improve the solution by considering all possible paths

Time Complexity : $O(V^3)$

Floyd Warshall problem



$$D^0 = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array} \qquad \pi^0 = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} a & a & a \\ b & b & b \\ c & - & c \end{bmatrix} \end{array}$$

$$D^a = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array} \qquad \pi^a = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} a & a & a \\ b & b & b \\ c & a & c \end{bmatrix} \end{array}$$

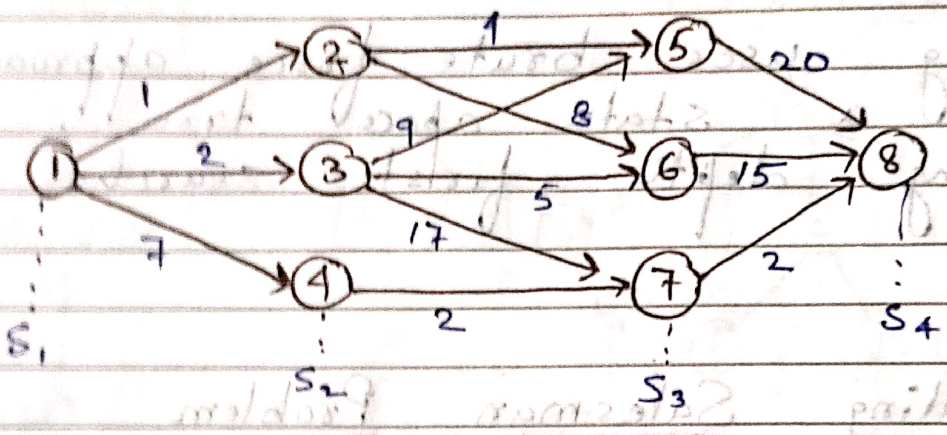
$$D^b = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array} \qquad \pi^b = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} a & a & a \\ b & b & b \\ c & a & c \end{bmatrix} \end{array}$$

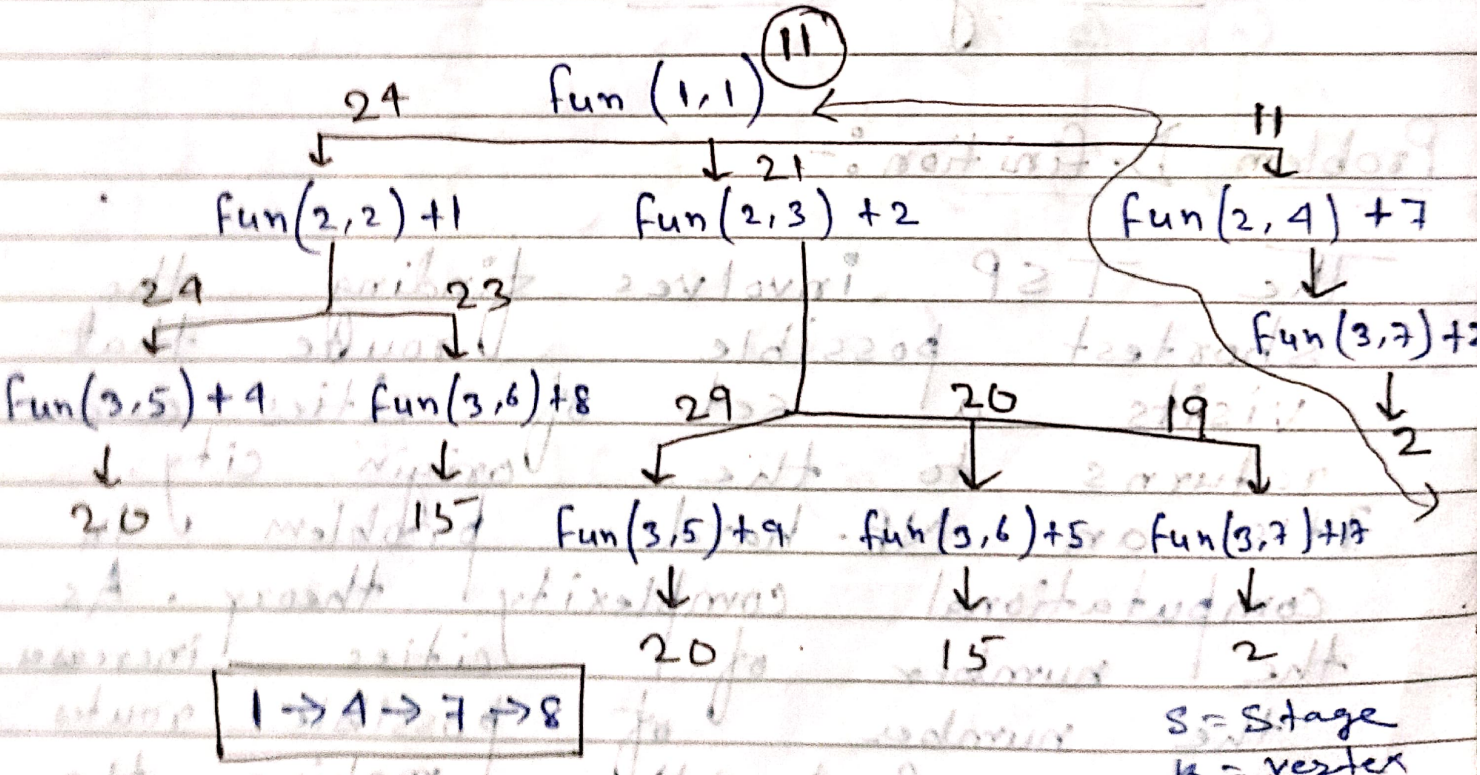
$$D^c = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array} \qquad \pi^c = \begin{array}{c} \\ a \\ b \\ c \end{array}\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{bmatrix} a & a & b \\ c & b & b \\ c & a & c \end{bmatrix} \end{array}$$

Time Complexity :- $O(n^3)$ (n = vertices)

Multi-Stage Graph Problem

Multistage Graph problem is a type of shortest path problem where the graph is divided into stages, and you must find the shortest path from the source (start of stage 1) to the destination (end of last stage).





S_i = Stage
k = vertex

$$fun(S_i, V_j) = \min \begin{cases} fun(S_{i+1}, k) + c(V_j, k) \\ c(V_j, D) \quad \text{if } S_i = S_{fi-1} \end{cases}$$

Backtracking

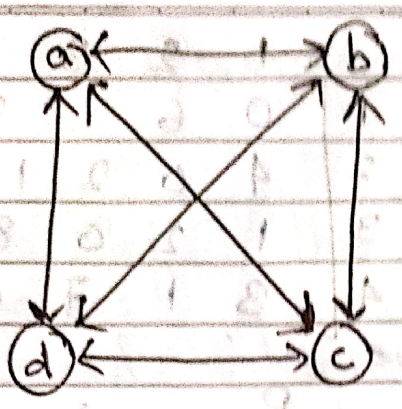
- Backtracking is a programming method used to solve problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints.
- Backtracking uses brute force approach generating a state space tree, following depth first search.

Travelling Salesman Problem

Problem Definition :-

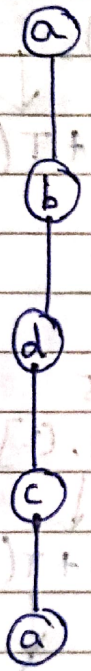
The TSP involves finding the shortest possible route that visits a set of cities and returns to the origin city. It's an NP-hard problem in computational complexity theory. As the number of cities increases, the number of possible routes increases factorially, making the problem computationally intensive.

Eg:-

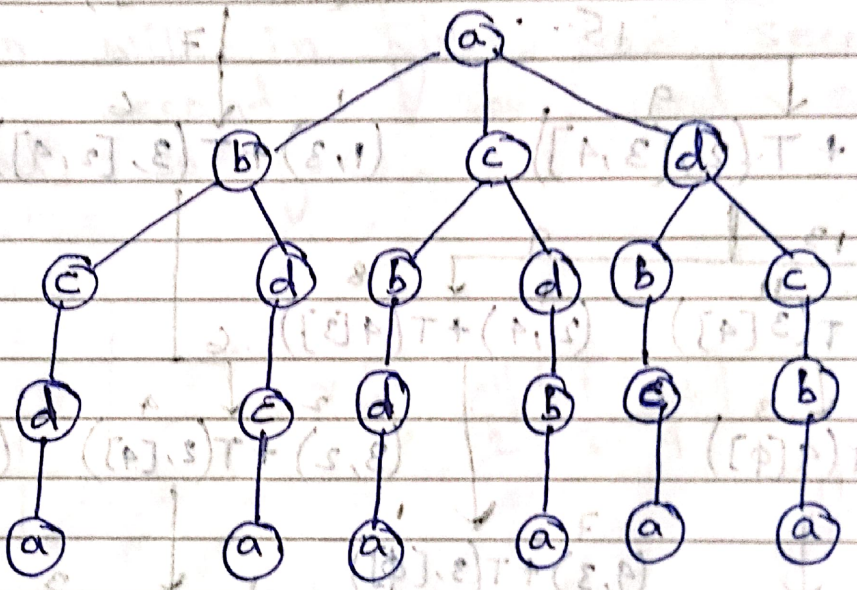


	a	b	c	d
a	0	10	15	20
b	5	0	25	10
c	15	30	0	5
d	15	10	20	0

Greedy



Backtracking / Brute Force

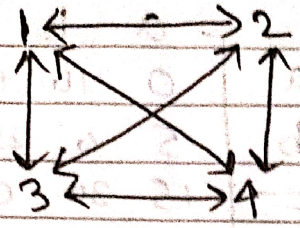


$O(n!)$

TC:- $O(n^n)$

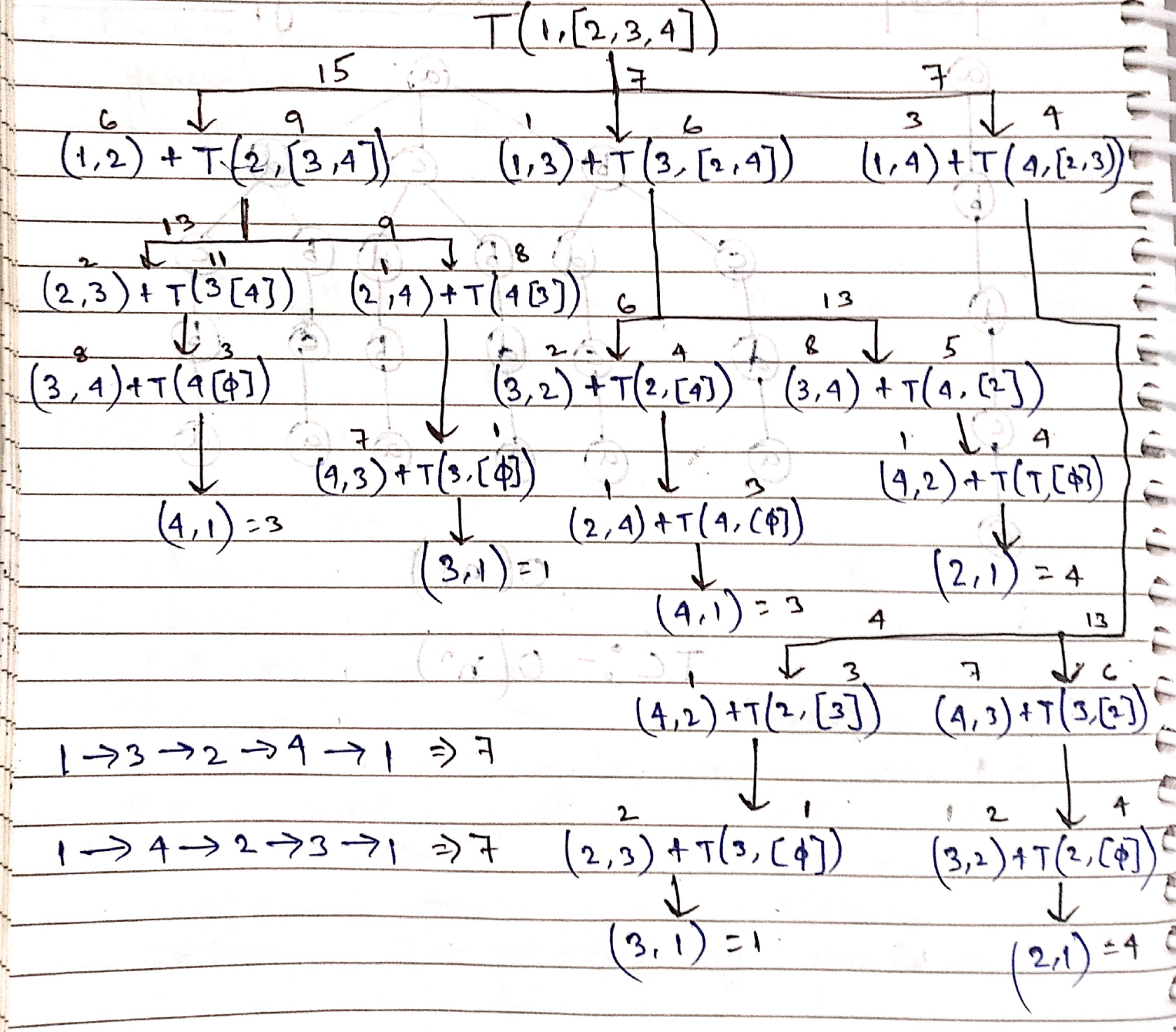
Delta

Ex.:-



	1	2	3	4
1	0	6	1	3
2	4	0	2	1
3	1	2	0	8
4	3	1	7	0

Using Dynamic Programming



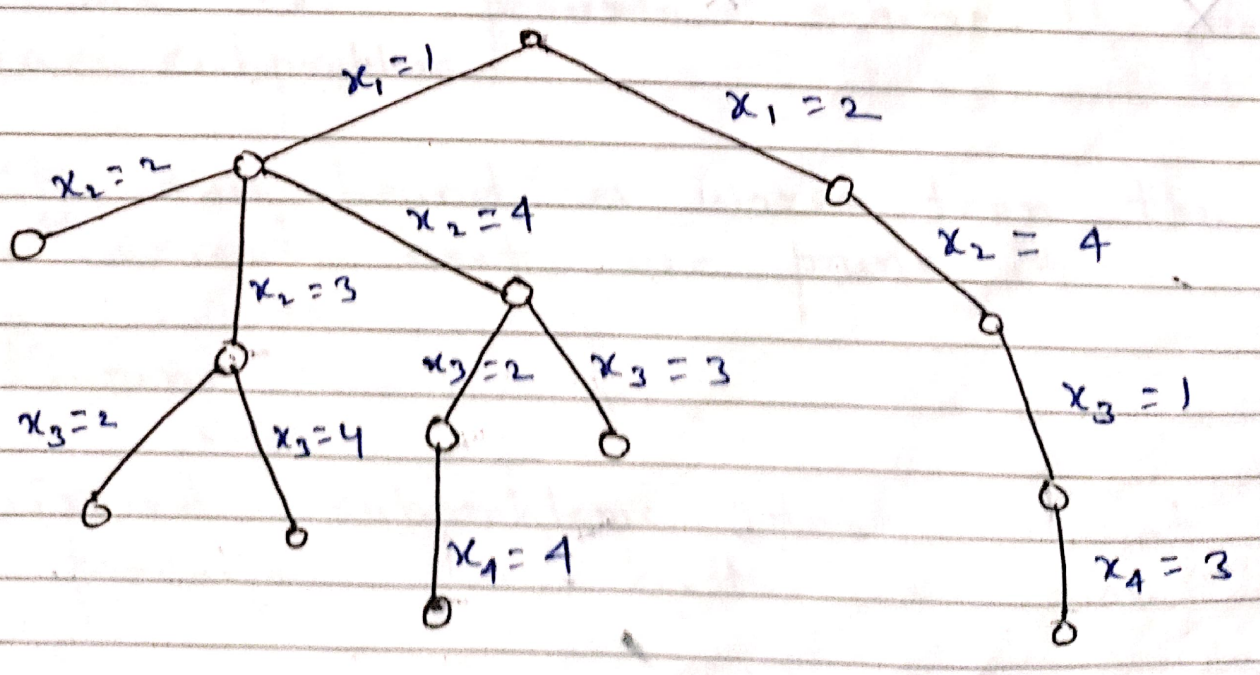
N-Queens problem

↳ Backtracking.

- The Aim of this problem is to place N chess queens on an nxn chessboard without any queen threatening another. This means no two queens can be in the same row, column, or diagonal.
- First queen will in first row, second queen in second row and so on and so for.

	1	2	3	4
1		Q_1		
2				Q_2
3	Q_3			
4			Q_4	

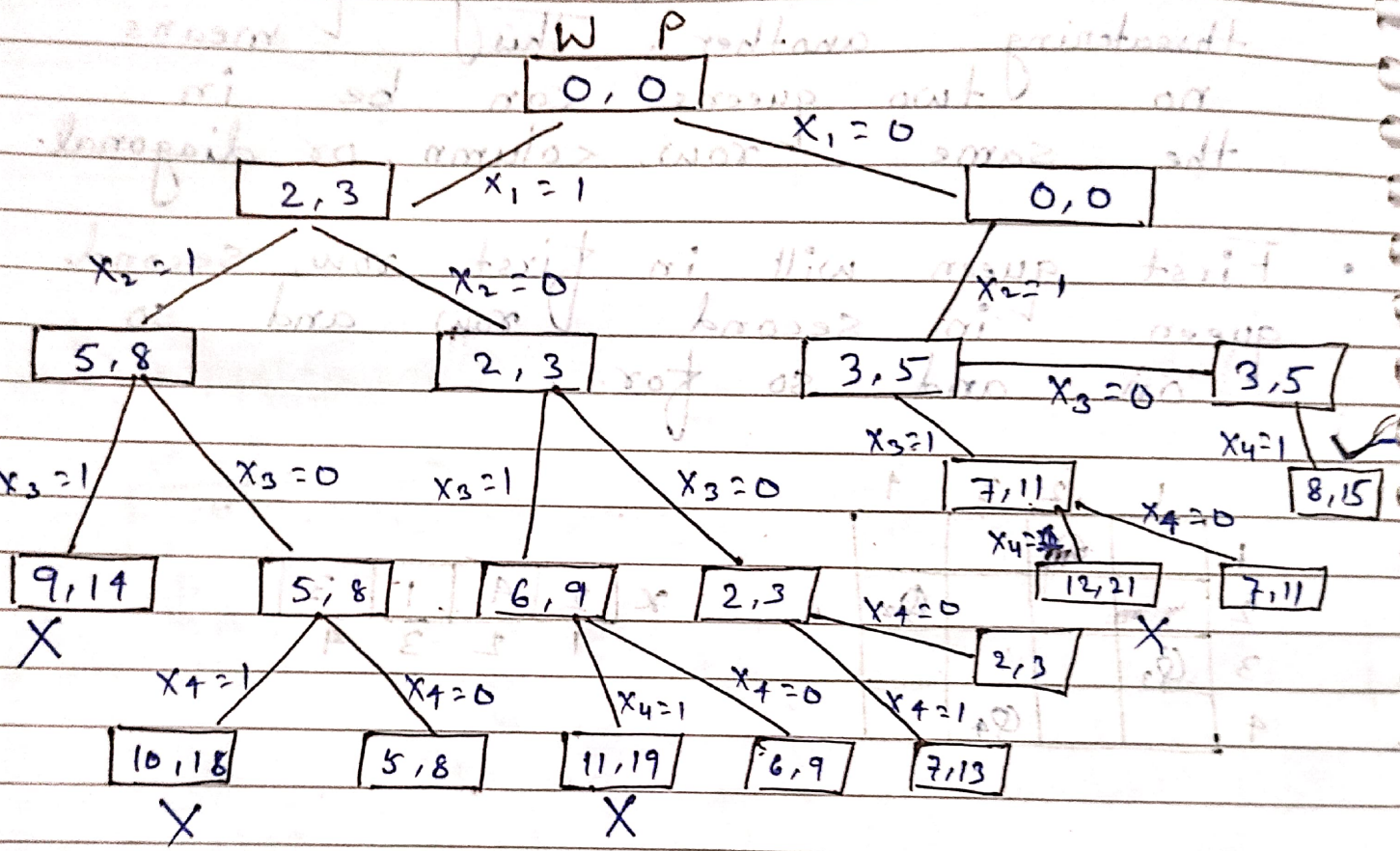
x	2	4	1	3
	1	2	3	4



0/1 Knapsack problem - using Backtracking

Object	1	2	3	4
Profit	3	5	6	10
Weight	2	3	4	5

M = 8



Branch and Bound

Branch and Bound systematically explores all possible solutions by dividing (branching) the problem into subproblems and eliminating (bounding) subproblems that cannot yield better solutions than the current best.

Method :-

- **Branching :-**

Divide the main problem into smaller subproblems (like tree nodes).

- **Bounding :-**

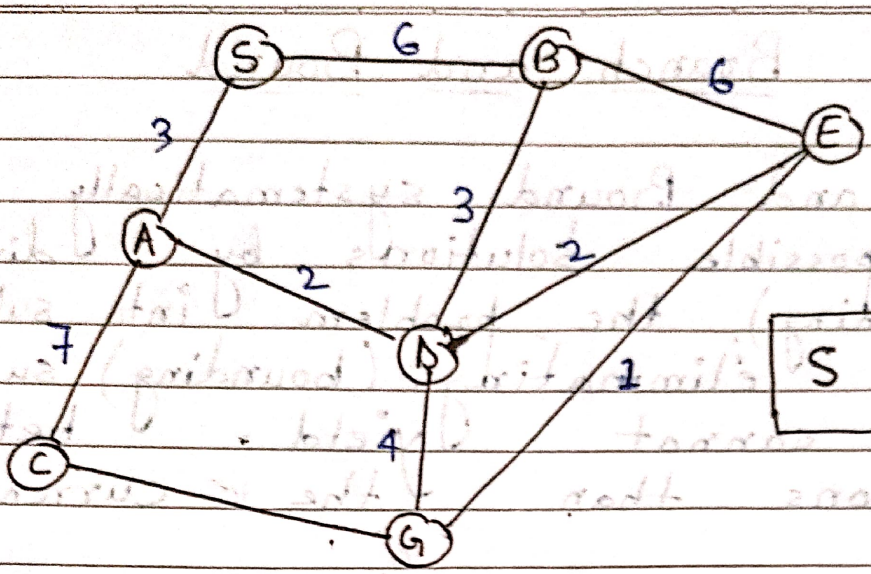
→ Calculate a bound (lower or upper) on the best possible solution from a subproblem.

→ If this bound is worse than the current best, we prune it.

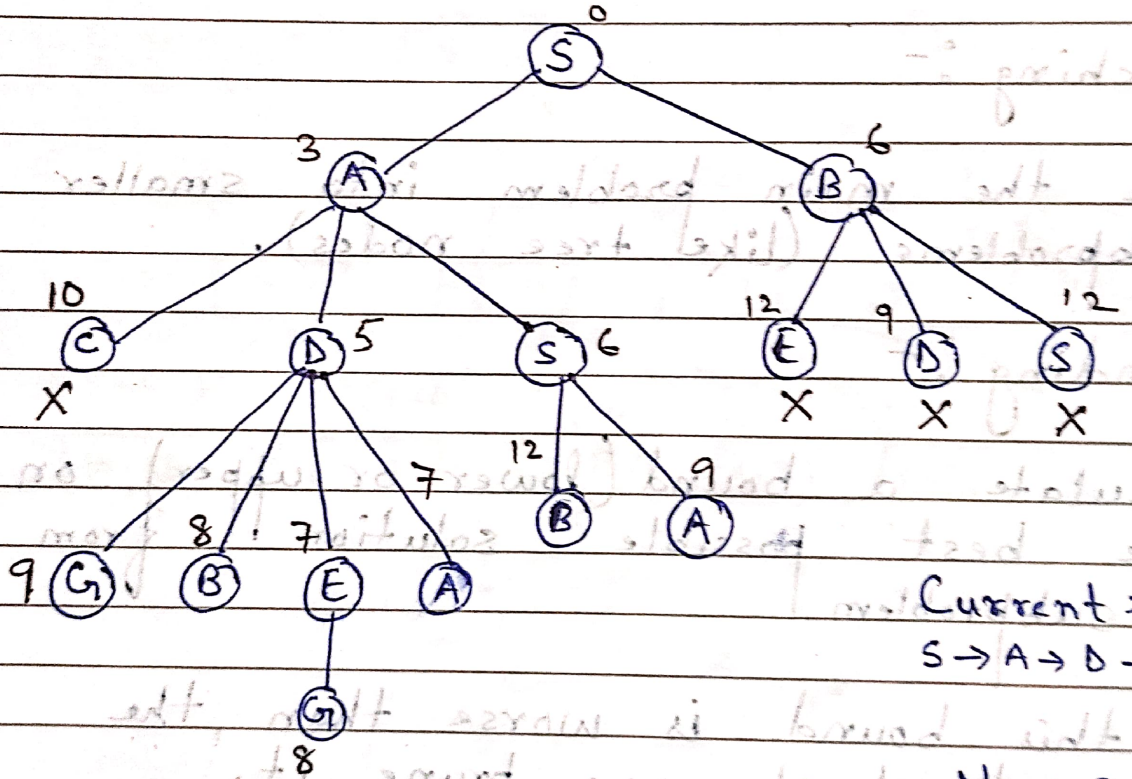
- **Pruning :-**

Discard subproblems that cannot produce a better solution.

E.g :-



S to G



Current = 9 X

S → A → D → G

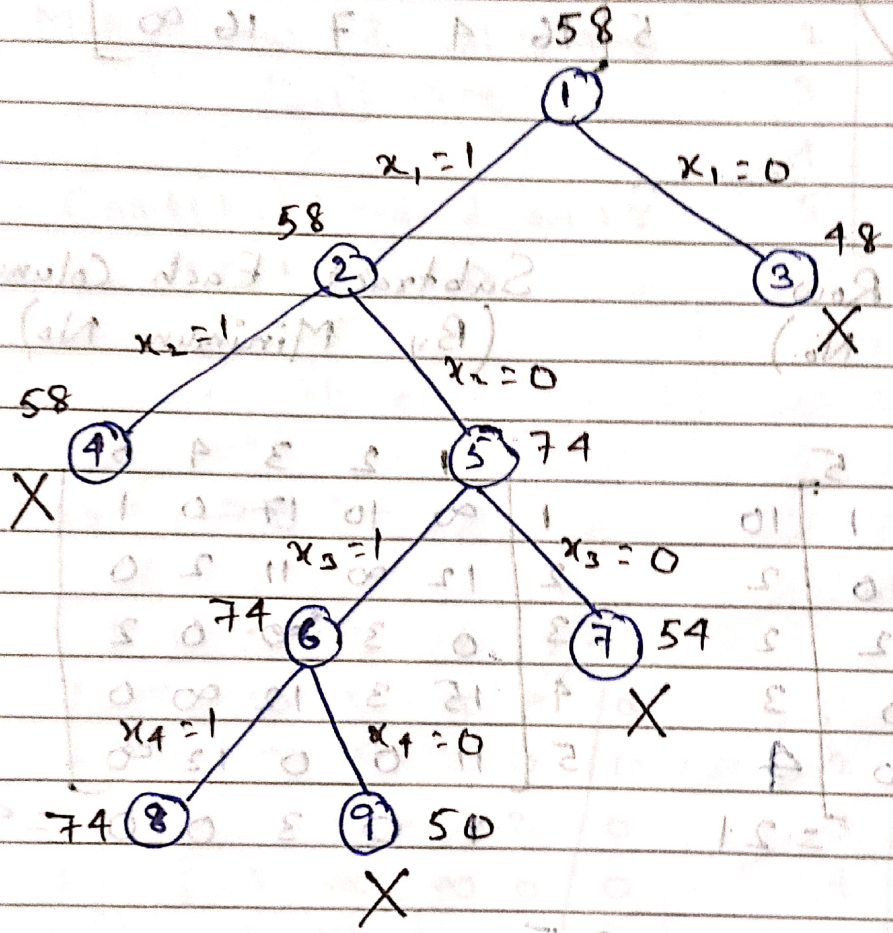
Now = 8

S → A → D → E → G

0/1 Knapsack problem using Branch & Bound

Object	1	2	3	4
Profit	30	28	20	24
Weight	5	7	4	2

M = 12



Current

2	28
1	30
	58

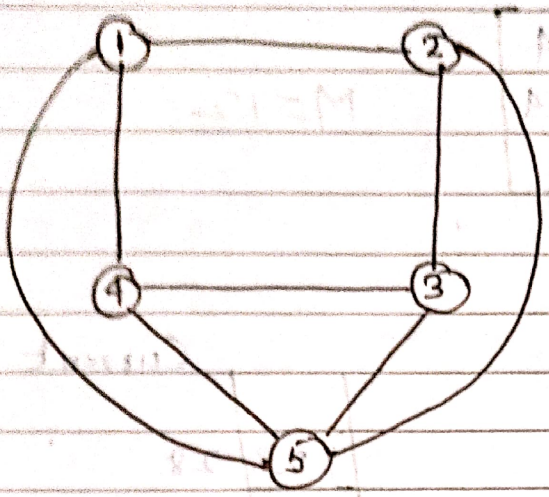
X

4	24
3	20
1	30
	74

Object = 1 + 3 + 4

$1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 8$

Traveling Sales person - Branch & Bound



	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

Subtract Each Row (By Minimum No.)

	1	2	3	4	5	
1	∞	10	20	0	1	10
2	13	∞	14	2	0	2
3	1	3	∞	0	2	2
4	16	3	15	∞	0	3
5	12	0	3	12	∞	4

= 21

Subtract Each Column (By Minimum No)

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

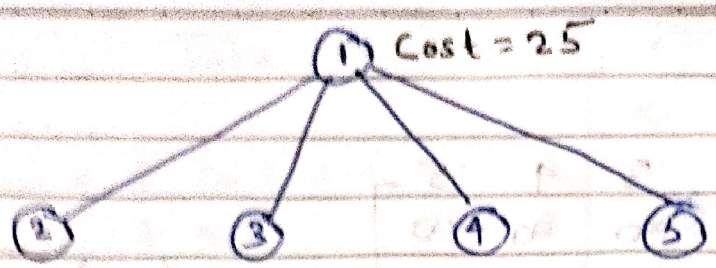
1 0 3 0 0 = 4

21 + 4 = 25

Reduced Matrix

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

Reduced Cost = 25



For 1→2 :-

Make Row 1 & Col 2 → ∞

(2,1) → ∞

Cost(1,2) + reduced cost + $\hat{r}$

10 + 25 + 0

35

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

$\hat{r}$ = is there any new reduction (where All are ∞ or 0)

For 1→3 :-

Subtract Col-1 by 11

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	∞
2	12	∞	∞	2	0	0
3	∞	3	∞	0	2	0
4	15	3	∞	∞	0	0
5	11	0	∞	12	∞	0
	11	0	∞	0	0	

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	∞
2	1	∞	∞	2	0	0
3	∞	3	∞	0	2	0
4	4	3	∞	∞	0	0
5	0	0	∞	12	∞	0
	0	0	∞	0	0	

Cost(1,3) + r + $\hat{r}$

17 + 25 + 11 = 53

For 1 → 4 :-

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

$Cost(1,4) + r + \hat{r}$

$0 + 25 + 0$

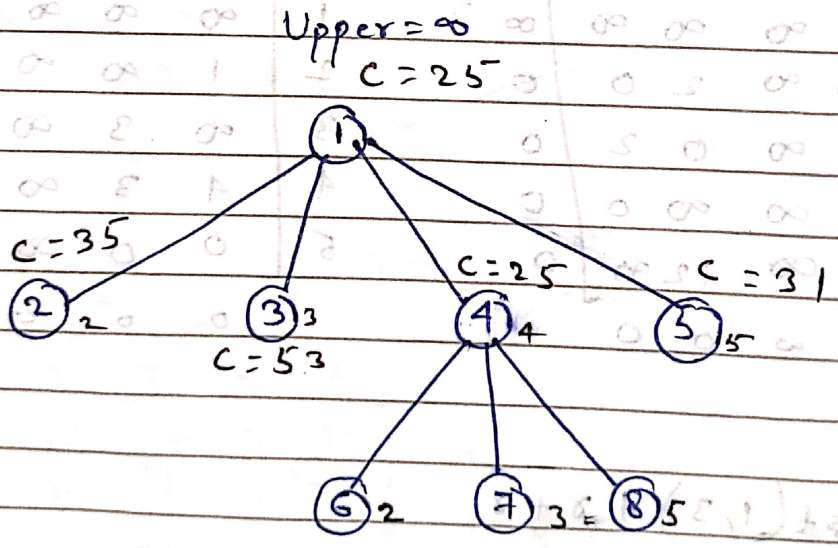
$= 25$

For 1 → 5 :-

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

$Cost(1,5) + r + \hat{r}$

$C = 31$



For 4 → 2 :- Take Matrix 1 → 1 :-

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

Make Row 4 & Col 2 → ∞

(2,1) → ∞

Cost(4,2) + Cost(4) + $\hat{r}$

3 + 25 + 0 = 28

For 4 → 3 :-

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

Cost(4,3) + Cost(4) + $\hat{r}$

= 50

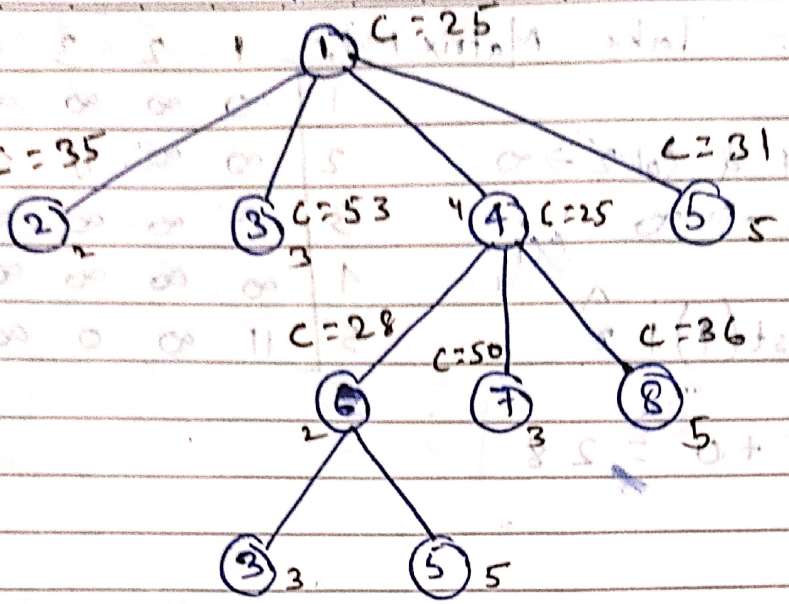
For 4 → 5 :-

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

Cost(4,5) + Cost(4) + $\hat{r}$

= 36

Upper = ∞



For 2 → 3 :- Take Matrix - 4 → 2.

(3,1) → ∞

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

$Cost(2,3) + Cost(2) + \hat{r}$

$= 52$

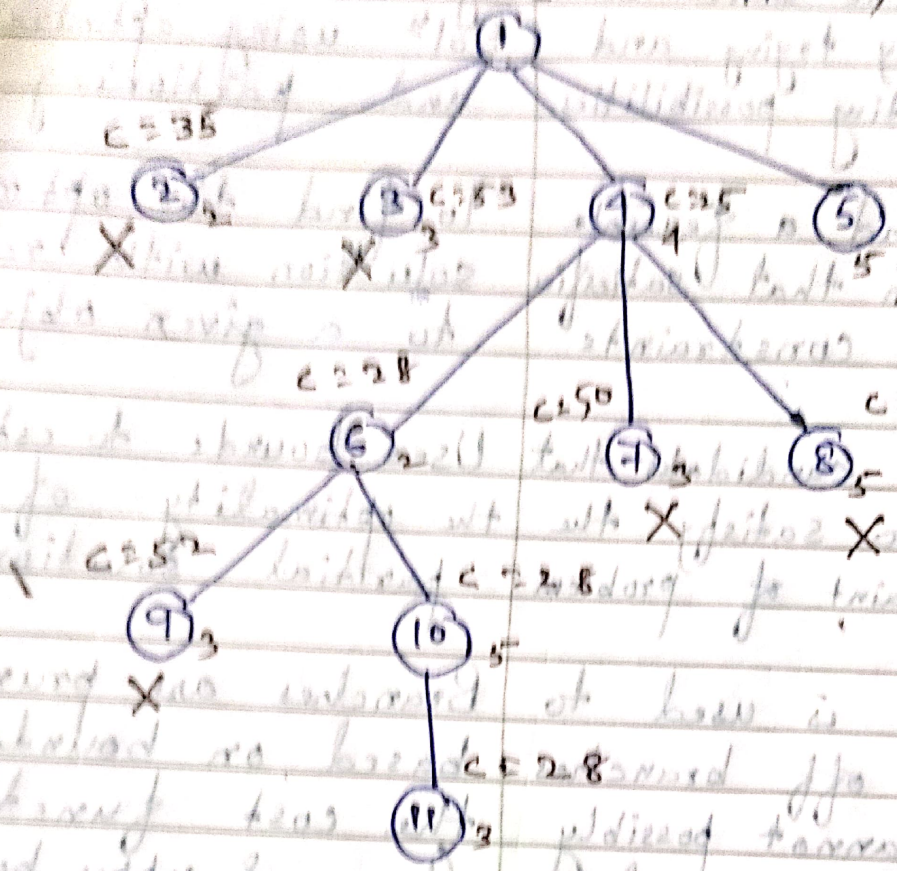
For 2 → 5 :-

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

$Cost(2,5) + Cost(2) + \hat{r}$

$= 28$

Similarly for 5 → 3 Cost = 28



1 → 4 → 2 → 5 → 3 → 1.

DL:
PG:

Aspect	Backtracking	Branch & Bound
Approach	Systematic search for Soln by trying and eliminating possibilities	Systematic search for Soln using optimization and partioning
Goal	To find a feasible solution that satisfies all constraints	To find the optimal solution with respect to a given objective
Technique	Removes candidates that fail to satisfy the constraint of problem	Uses bounds to estimate the optimality of partial solution
State Space Tree	Pruning is used to chop off branches that cannot possibly lead to a solution	Branches are pruned based on bounds of the cost function (lower & upper bound)
Application	Suitable for decision problem like puzzle	Used for problems like integer programming.