Space Complexity:: Input: 'n'

Space complexity: O(n)

Huffman Coding: (An application of optimal merge pattern)

↳ {data encoding + data compression technique.}

↓ representing an element with some code.

eg: 4 characters/elements

Straight forward / normal encoding: 2 bits

- 00 - a
- 01 - b
- 10 - c
- 11 - d

eg: 100 characters

→ 7 bits/character

$\lceil \log_2 N \rceil$ bits required.

* Data compression is never possible with uniform encoding.
* Compression is possible with non uniform encoding.
* Codes are assigned based on frequency [non-uniform encodings]

↓

Huffman Coding

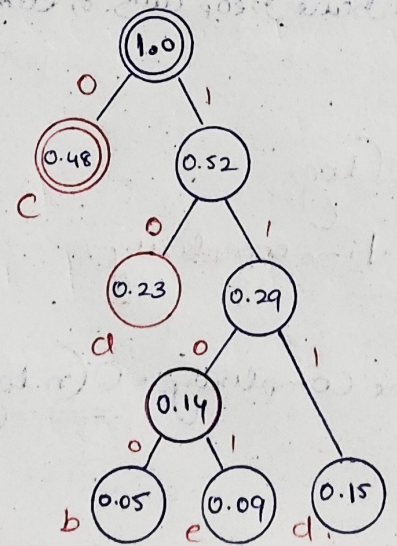
$L = \Sigma(a, b, c, d, e)$ → uniform encoding: 3 bits/character.

→ non-uniform.

* we are given frequency

= <0.23, 0.05, 0.48, 0.15, 0.09>

(↳ we need to apply optimal merge pattern algorithm)



a → 10 (2 bit)
b → 1100 (4 bit)
c → 0 (1 bit)
d → 111 (3 bit)
e → 1101 (4 bit)

eg: Text : <ccaccccabdc>

(i) uniform encoding : 10x8 = 80bits

(ii) Huffman coding : <0010000101100111O> : 17bit

: Given binary string : 1001000101000111....

get text format : acaccaaccd....

Average no. of bits : $\sum_{i=1}^{n} d_i * q_i$:

(Huffman Coding)

$= 2 \times 0.23 + 4 \times 0.05 + 1 * 0.48 + 3 \times 0.15 + 4 \times 0.09$

$= 1.95$ bits/characters

Drawback of Huffman Coding: It is very weak with regard to error, one bit error can corrupt the whole text/data.

Q1). The characters 'a' to 'h' have the set of frequencies based on the first 8 fibonacci numbers as follows:

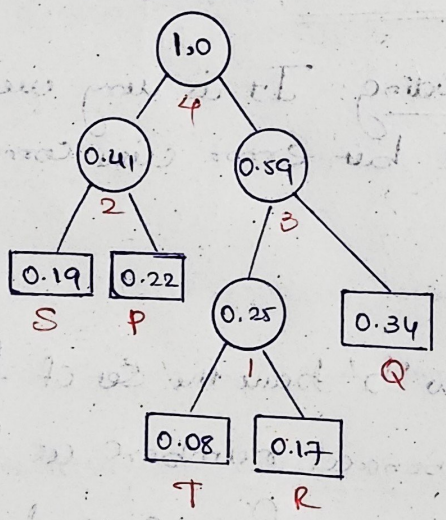
{a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21}

A huffman coding is used to represent the characters. What is the sequence of characters corresponding to the following code? 1101111001111010 = ?

Q2) A message is made up entirely of characters from the set $X = \{P, Q, R, S, T\}$. The table of probabilities of each of the characters is shown below.

Character	Probability
P	0.22
Q	0.34
R	0.17
S	0.19
T	0.08
Total	1.00

If a message of 100 characters over X is encoded using Huffman coding, then the expected length of the encoded message in bits is 22.5



P → 01 (2)
Q → 11 (2)
R → 101 (3)
S → 00 (2)
T → 100 (3)

$$\sum_{i=1}^{n} d_i * q_i = (2\times 0.22 + 2\times 0.34 + 3\times 0.17 + 2\times 0.19 + 3\times 0.08)$$

$$\Rightarrow 2.25$$

Q3) Consider the string abbccddeee. Each letter in the string must be assigned a binary code satisfying the following properties:

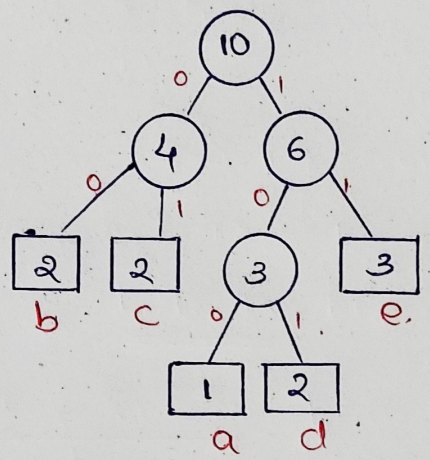
(1) For any two letters, the code assigned to one letter must not be a prefix of the code assigned to other letter.

(2) For any two letters of same frequency, the letter which occurs earlier in the dictionary order is assigned a code whose length is atmost the length of the code assigned to the other letter.

Among the set of binary codes assignments which satisfy the above two properties, what is the min length of the encoded string?

→ Frequencies:

a = 1
b = 2
c = 2
d = 2
e = 3.



a → 100 (3)
b → 00 (2)
c → 01 (2)
d → 101 (3)
e → 11 (2)

Total: (3 + 2*2 + 2*2 + 3*2 + 2*3)

⇒ 23

Problem (4) : Minimum Cost Spanning Tree

(Graph Based Problem)

Graph (non-linear D.S)

→ G = (V, E)
- V → set of vertices
- E → set of edges

|V| = n (vertices)
|E| = e (edges)

- Undirected / directed
- weighted / unweighted

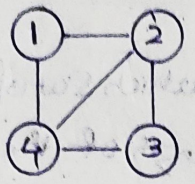
Representation of Graphs:

→ 1> Adjacency Matrix (Array-sequential)
→ 2> Adjacency List (list-based).

10/8/2023

$G = (V, E)$ $|V| = n$, $|E| = e$.

Representation of Graphs → Adjacency matrix
→ Adjacency list.



(i) Matrix [n×n]

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

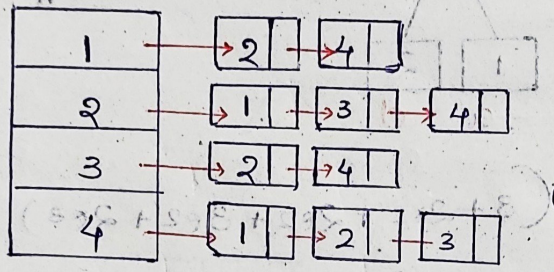
n×n
$O(n^2)$

$A[i,j] = 1$, $\langle i,j \rangle \in E$

$A[i,j] = 0$, $\langle i,j \rangle \notin E$

(ii) Adjacency List

Array of linked list.

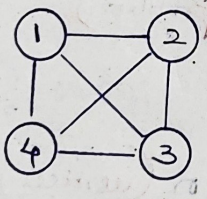


(a) Undirected graph
$(n+2e)$ – Size of adjacency list.

(b) directed graph
$(n+e)$

Time Complexity = $O(n+e)$

"$e \propto n$" (in worst case)



Complete Graph.

Undirected: $e = \frac{(n-1)n}{2}$ → $O(n^2)$

Directed: $e = n(n-1)$ → $O(n^2)$

$e = O(n^2)$

$\log e \sim O(\log n)$

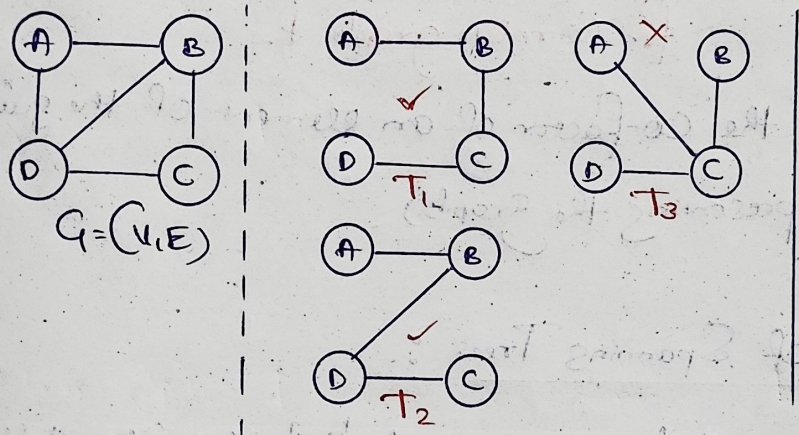
Graphs

Dense Graphs (almost complete) — $e \sim O(n^2)$ (Matrix representation is better)

Sparse Graphs (not complete) — Adjacent list $\Rightarrow O(n+e)$

$O(n+e) = O(n+n^2)$

Definition of Spanning Tree: A subgraph $T(V, E)$ of the given Graph $G(V, E)$, where E' is subset of E $(E' \subset E)$ is a spanning tree, if T is a Tree (acyclic).



→ A tree with n vertices will have (n-1) edges.

Q1) Given a graph having n-vertices and e-edges, then the no. of edges that must be removed from the ~~knights~~ Graph to get a spanning tree is ____

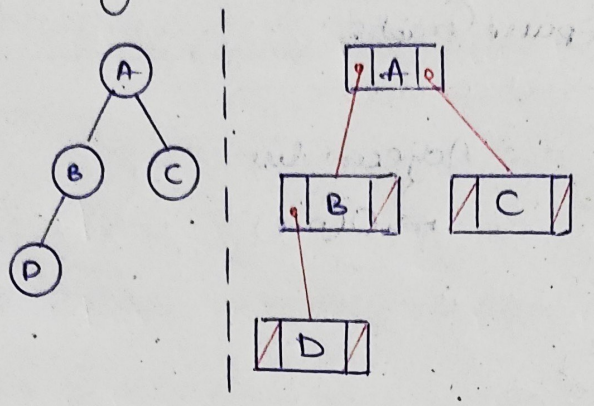
→ Total edges in Graph : e

→ Spanning tree with 'n' vertices will have : (n-1) edges (used)

Removed edges : e - (n-1)

: e-n+1

→ A tree with n-vertices will have (n+1) no fields.
(Binary). (null fields)



Total links = 2n

Edges = (n-1)

Unused = 2n - (n-1)

(Null) fields : (n+1)

→ For a given graph with n-vertices and 'e' edges the maximum number of spanning tree (solution space) will be $n^{(n-2)}$

(for complete graph k_n)

→ Determining the co-factor of an element of the given matrix (representing the graph).

Applications of Spanning Trees :

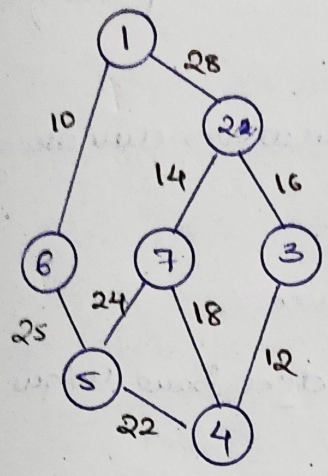
(i) Multicasting and broadcasting in both wired and wireless (ST) connections.

(ii) Construct/implement circuits (electric/electrical) in (MCST) communication networks.

Algorithms for Constructing Min Cost Spanning Tree

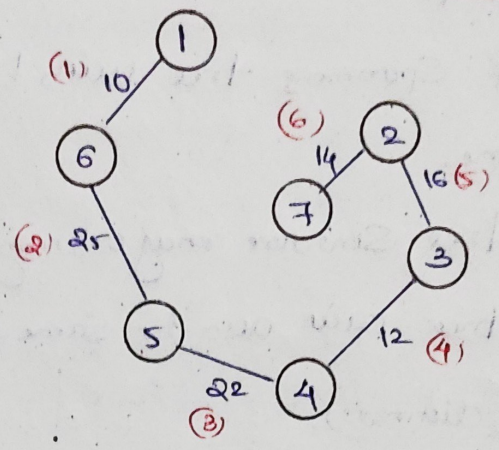
1. Prims - Jarnik algorithm.
2. Kruskal algorithm.
3. Dijkstra's algorithm.

$G = (V, E)$



Prims Algorithm

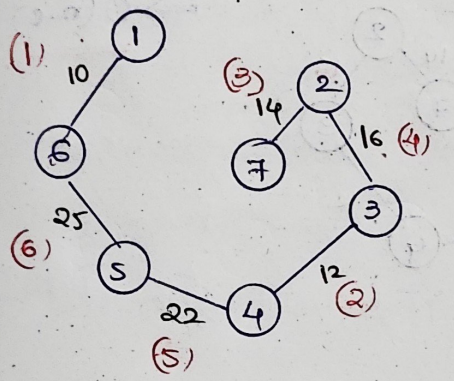
$T.C = O(n^2)$



Kruskal's Algorithm.

Min-Heap

$\langle 10, 12, 14, 16, 18, 22, 24, 25, 28 \rangle$

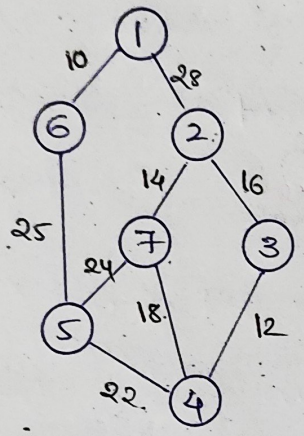


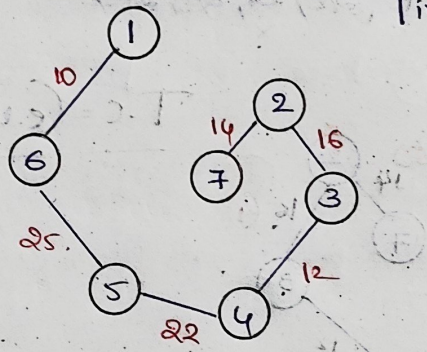
$T.C = (e \log e)$ — e: no. of edges

Prim's Algorithm	Kruskal's Algorithm
* Time Complexity is $O(n^2)$	Time Complexity is $O(e \log e)$ where e is no of edges.
* Prim's method always maintains tree structure property at each step.	Kruskal's method does not always maintain tree structure property at each step.
* Prims algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest (disconnected components) at any instance as well as it can work on disconnected components

- Prims method always maintains tree structure property at each step.
- Cost of Spanning tree with both applications will always be same.
- The tree structure may or may not be same.
- The tree will also be same if all edges have unique cost (distinct).

Dijkstra's Algorithm:

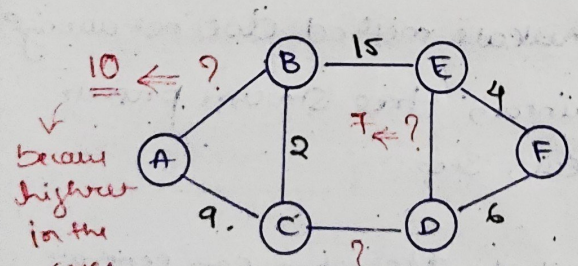




Time Complexity
$\Rightarrow O(n \cdot e)$

Q1). Consider the following graph whose minimum Cost Spanning tree marked with ~~top~~ edge values has a weight of 36.

* Minimum possible sum of all edges of the graph G is ____. (Assume that all edges have distinct cost).



because highest in the cycle

during construction of Spanning tree, the edge edge with max value will be discarded

(9, 2 ?) ⇒ max value among the cycle that is minimum
= 10

(4, 6, ?) ⇒ 7

(2, 15, 7 ?) ⇒ 16.

add edge have during con

∴ 36 + 7 + 16 + 10 = 69

* If all edges aren't distinct, cost = 66

Algorithm Prim (E, cost, n, t) (Tree Structure, edges will be constructed)

```
{ Let (k,l) be an edge of min. cost in E.        (i.e.
  mincost = cost[k,l];                                  -1
  t[1,1] = k , t[1,2] = l                               -1
  for (i=1 to n) do
  { if (cost[i,l] < cost[i,k]) then near[i] = l;
    else near[i] = k; }                                 } n.
  near[k] = near[l] = 0;
  for (i=2 to n-1): do // adding remaining (n-2) edges  (n-2) edges.
  { // find (n-2) additional edges
    // let j be an index such that near[j] ≠ 0 and cost[j, near[j]]
       is minimum.
    t[i,1] = j;   t[i,2] = near[j];
    mincost = mincost + cost[j, near[j]];
    near[j] = 0;
    for (k=1 to n) do // update near[]
    { if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k,j]))
        { then mark
          near[k] = j; }
    }
  }
  return mincost;
}
```

Time Complexity

$T(n) = e + c + n + n^2 = O(n^2)$

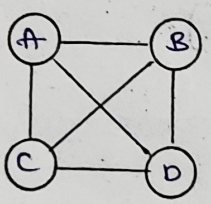
High level Implementation of Kruskal's Algorithm

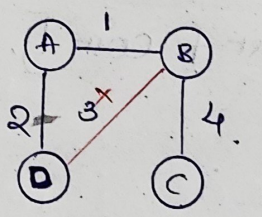
```
t = 0
while ((t has less than n-1 edges) and (E ≠ 0)) do
{
    Choose an edge (v,w) from E of lowest cost;
    Delete (v,w) from E.
    if (v,w) does not create a cycle then add (v,w) to t;
    else discard (v,w);
}
```

Time Complexity := O(e log e)

* Real algorithm will be discussed after 'Set' data structure introduction.

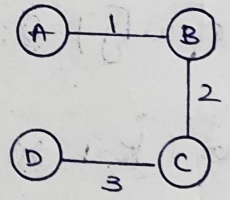
Q2) Let G be a complete undirected graph with 4 vertices and edge weights are {1, 2, 3, 4, 5, 6}. The maximum possible weight that a minimum Spanning tree can have?





1+2+4 = 6

Normal Situation

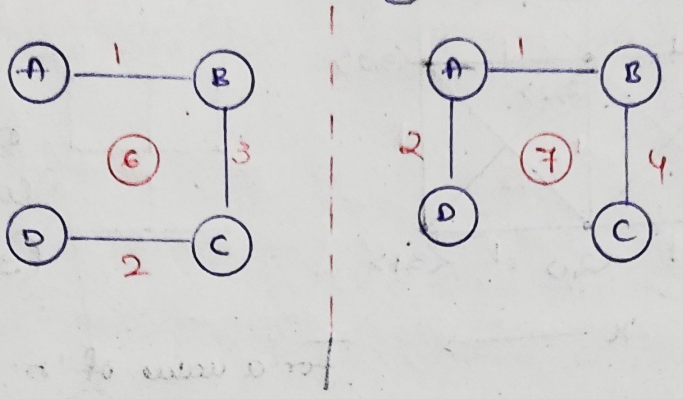


1+2+3 = 6

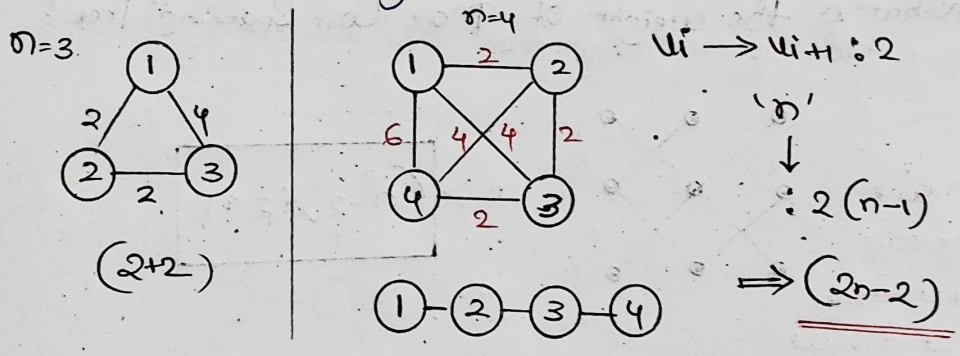
∴ The max possible weight that a minimum Spanning tree can have is 7

12/8/2023

Q1> Let G be a complete undirected graph with 4 vertices and edge weights are {1, 2, 3, 4, 5, 6}. The maximum possible weight that a minimum weight spanning tree can have is 7



Q2> Consider a complete weighted graph with n vertices numbered v_1 to v_n. Two vertices v_i and v_j having edge between them has a cost value of $2|i-j|$. The weight of minimum cost spanning tree of such a graph is _____.

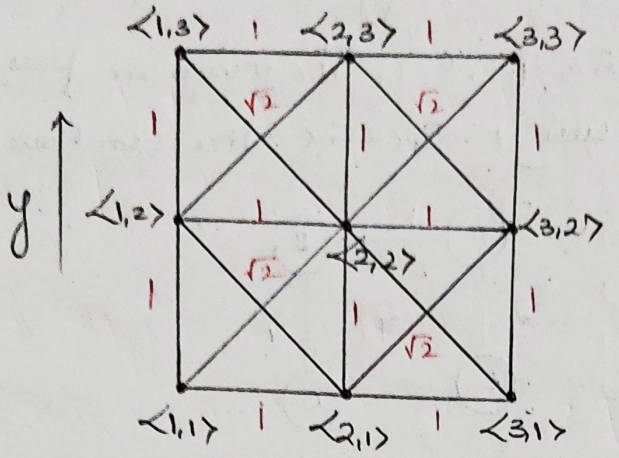


Q3> Let G be a connected undirected graph of 100 (99) vertices and 300 edges. The weight of minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes _____.

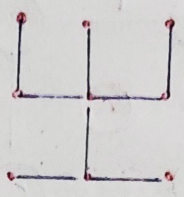
$\Rightarrow 500 + 99 \times 5 \Rightarrow 995$

Q4> Consider a graph whose vertices are points in a plane with integer co-ordinates (x, y) where $1 \le x \le n$, $1 \le y \le n$, $n > 2$ is an integer. 2 vertices $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$ are adjacent if $|x_1 - x_2| \le 1$ and $|y_1 - y_2| \le 1$. The cost of such an edge is given by the distance between them. Compute the weight of min cost spanning tree of such graph for given value of n.

→ n=3:



* distance formula $= \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$



n=3
Cost = 8
$(3^2-1)+1$

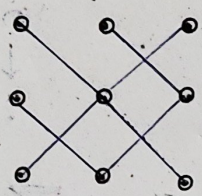
For a value of 'n'.

No. of vertices of 'G' = (n^2)

No. of edge of tree = (n^2-1).1

* Addition question to same Problem

→ What is the weight of Max Cost Spanning Tree?

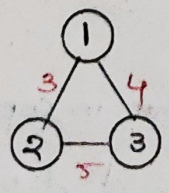


adding one edge will make it a cycle.

$\Rightarrow \boxed{(n^2-2)\sqrt{2}+1}$

Q5) Consider a graph with 'n' vertices. n>2 the vertices are numbered 1 to n. Two vertices V_i and V_j are adjacent if $0 < |i-j| \leq 2$. The weight of such an edge is i+j. The weight of minimum cost Spanning tree for such a graph for value n is ____

n=3.

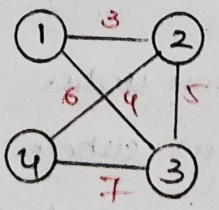


: (3+4)

↓

(3+2×2)

n=4.



: (3+4+6)

↓

(3+2×2+2×3)

Generalized formula:

'n'

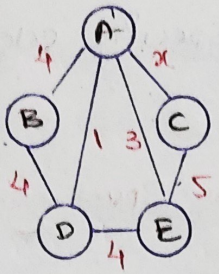
: $3+2(2+3+4\ldots n-1)$

: $3+2\left[\frac{n(n-1)}{2}-1\right]$

: $3+[n^2-n-2]$

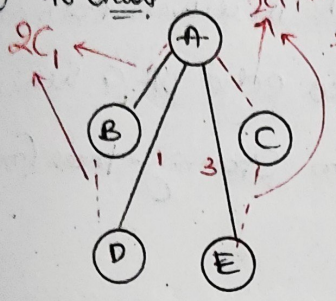
$\Rightarrow \boxed{n^2-n+1}$

Q6) Consider the following undirected Graph G.



Choose a value for x that will maximize the number of minimum weight Spanning tree (MWSTs) of G. The number of MWSTs of G for the value of x is ___

anyone of '4' to choose



$2C_1$, if $x=5$

↓ if $x>5$ then Number = 2.

↓ if $x<5$ then also Number = 2

↓ if $x=5$ then in right side - 2 options

$= 2C_1 + 2C_1$

(left side, right side)

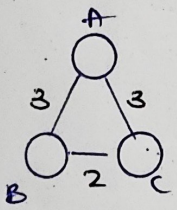
Total no. of MCST

$\Rightarrow 2_4 + 2C_1$

$\Rightarrow 4$

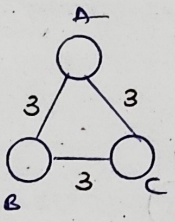
No of Minimum Cost Spanning Tree Possible

(nC_r)

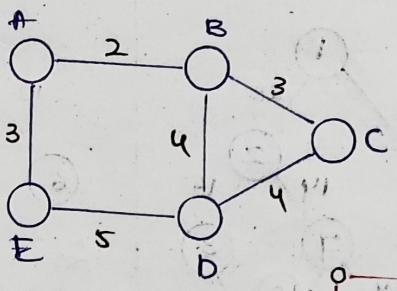


No. of MCST

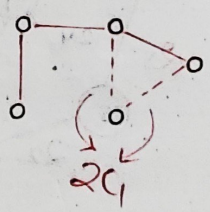
$= 2C_1$



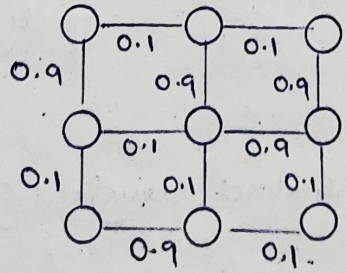
$3C_2 = 3$



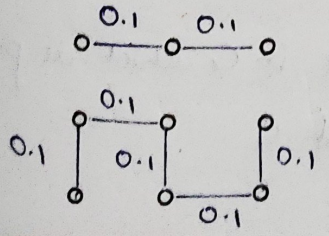
$2C_1 = 2$



Q7) Consider the following undirected graph with edge weights as shown.



The ~~minimum~~ number of minimum weight Spanning trees of the Graph is $3C_1 = 3$



$\Rightarrow 3C_1$

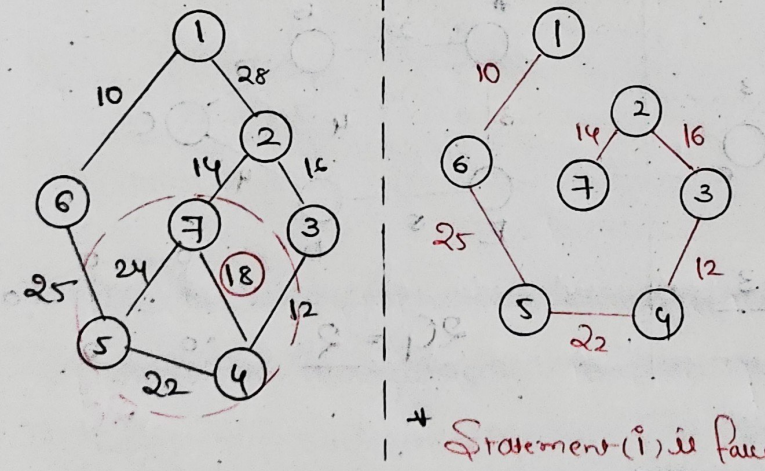
Q8> Let w be the minimum weight among all edge weights in an undirected connected graph. Let 'e' be a specific edge of weight 'w'. Which of the following is false?

→ 'e' is present in every ~~spanning~~ minimum Spanning tree.

Q9> G=(V, E) is an undirected simple graph in which each edge has a distinct weight, and e is a particular edge of G. Which of the following statements about minimum spanning tree (MST) of G is/are true?

(i) If e is the lightest edge of some cycle in G, then every MST of G includes e.

(ii) If e is the heaviest edge of some cycle in G, then every MST of G excludes e.



→ Statement (i) is false and Statement (ii) is always true.

<Dijkstra's algorithm property>

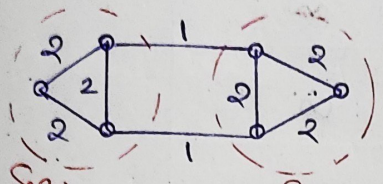
Q10> Let G be a connected undirected weighted graph. Consider the following two statements

S1: There exists a minimum weight edge in G which is present in every minimum spanning-tree of G.

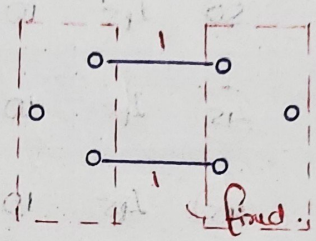
S2: If every edge in G has a distinct weight, then G has a unique minimum Spanning tree.

→ S_1 is false and S_2 is true.

Q11) Compute number of minimum Spanning tree:



H/W



$[2C_1 \times 3C_2 \times 2C_1]$

$\Rightarrow 12$

Problem (5): Single Source Shortest Paths (SSSP)

(1) Single pair Shortest path (Trivial) $s \longrightarrow d$

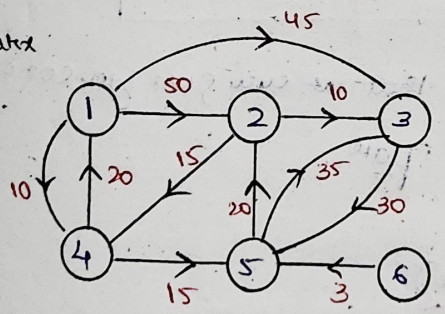
(2) Single source shortest paths
- → Dijkstra Algorithm. (Greedy) (may not work with -ve edges)
- → Bellman Ford (D.P) ↳ works with -ve edges

$s \Leftrightarrow$ (n-1) dist.

(3) All-pairs Shortest Path
- <Floyd-warshall's Algorithm> (D.P)

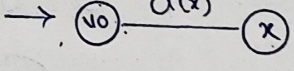
Dijkstra's Greedy Algorithm.

Source vertex v_0.

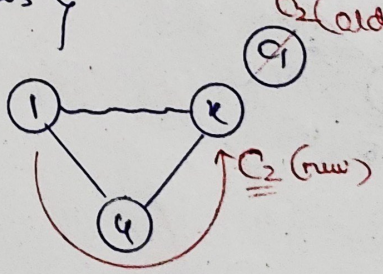


C	1	2	3	...	6
1	0	50	45	...	
2	:				
3	:	C<i,j> cost of edge.			
:	:				
6	:				

$d(x)$ = distance from source (v_0) to vertex 'x' known so far.

→ 

{Relaxation Process}



* if($C_2 < C_1$)
  update (d-value of x).

a) Matrix Method : $V_0 = 1$.

D-values.

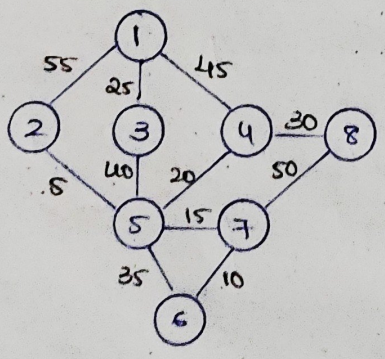
Vertex Selected	1.	2.	3.	4.	5.	6
Step 1.	—	50	45	(10)	∞	∞
Step 2 {1, 4}	—	50	45	10	25	∞
Step 3 {1,4,5}	—	(45)	45	10	25	∞
Step 4 {1,2,4,5}	—	(45)	45	10	25	∞
Step 5 {1,2,3,4,5}	—	(45)	45	10	25	∞

$V_0 = 6$.

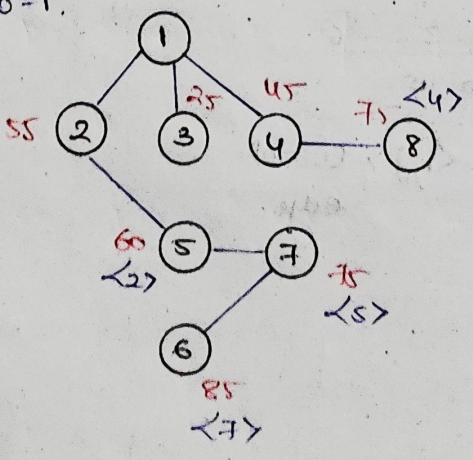
Vertex selected.	1	2	3	4	5	6
(1) {6}	∞	∞	∞	∞	3	—
(2) {6,5}	∞	23	38	∞	3	—
(3) {6,2,5}	∞	23	38	38	3	—
(4) {6,2,3,5}	∞	23	33	38	3	—
(5) {6,2,3,5,1}	58	23	33	38	3	—

* The drawback of matrix method is that it will give you cost of path but won't give you the exact path.

Dijkstra's Algorithm (Spanning Tree)



$V_0 = 1$.

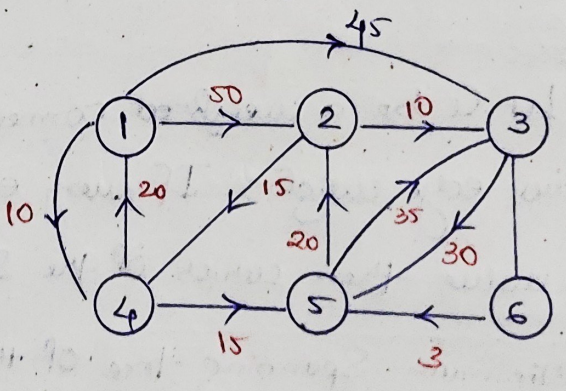


Algorithm ShortestPath(v, cost, dist, n) 14/8/2022

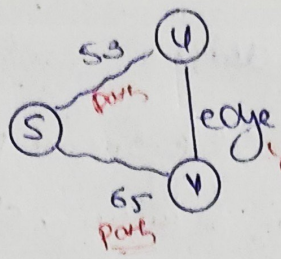
```
{ // dist[j], 1<=j<=n is set to lengths of the shortest paths from vertex v
  // to vertex j in a digraph G with n vertices. dist[v] is set to zero.
  // G is represented by its adjacency matrix cost[1:n, 1:n]

  for(i=1 to n) do ->O(n)
  { S[i] = false, dist[i] = cost[v,i] }  // initialize S.

  S[v] = true; dist[v] = 0.0. // put v in S

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

Time Complexity = $n + n\log n + n\log n$

$(n+e)\log n$.



Q) Consider a unified undirected graph with positive edge weights and let uv be an edge in the graph. It is known that the shortest path from the source vertex s to v has weight 53, and the shortest path from s to v has weight 65. Which of following st. is true?

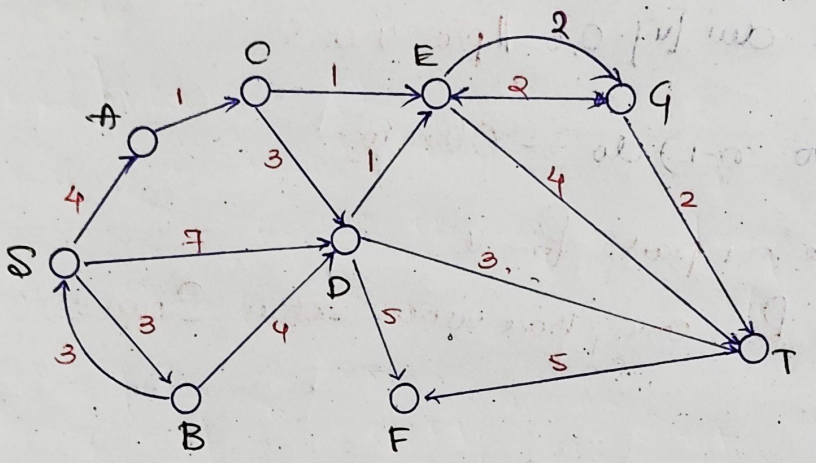


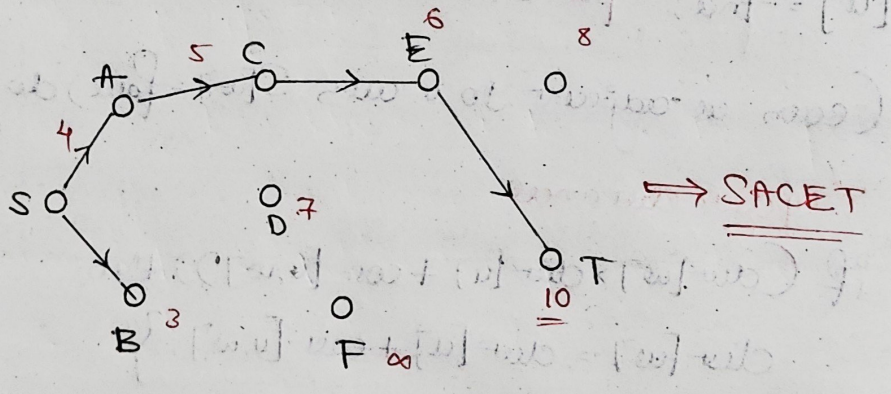
wt[u,v] = ?

can be 11, wt[u,v] = ≥ 12

→ weight (u,v) ≥ 12

Q2> Apply Dijkstra Algorithm over the given Graph, which pattern is reported from 'S to T'?





⇒ SACET

Q3> Let G be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the statements is/are true?

1. Minimum Spanning tree of the graph does not change. — True
2. Shortest path between any pair of vertices do not change. — False

Q4) Let $G = (V, E)$ be any connected undirected edge-weighted graph. The weights of the edges in E are positive and distinct. Consider the following statements:

(i) Minimum Spanning tree of G is always unique. → True

(ii) Shortest path between any two vertices of G is always unique. → False

Which of the statements is true?

→ (i) is true and (ii) is false.

Q5) Consider the following.

Algorithm.	Design Paradigms
(P) Kruskal	(i) Divide and Conquer
(Q) Quick Sort	(ii) Greedy
(R) Floyd-Warshall	(iii) Dynamic Programming

14/2/2023

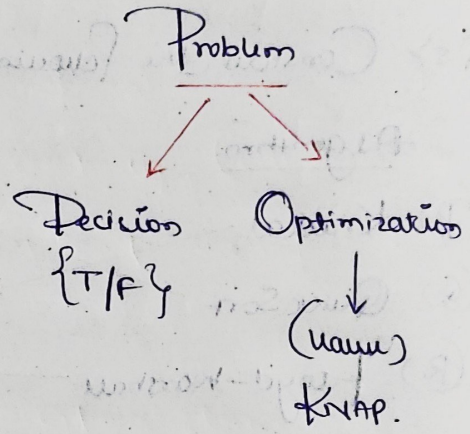
Dynamic Programming.

- design strategy proposed by Richard Bellman

Dynamic Programming (DP): is an algorithm design strategy (method) paradigm, used for solving problem, whose solutions are viewed as a result of making a sequence of decisions.

Terminology

* problem definations
* Constraints — implicit (local)
  — explicit (boundary)
* Solution Space (∞, 2^n, $n!$, ...)
* Objective function — min, max } criteria
* Optimal Solution: unique solution.



* One way of making these decisions is to make them one at a time in a step-wise (sequential) step-by-step manner and never make an erroneous decision. This is true of all problems solvable by Greedy method.

* For many other problems it is not possible to make step-wise decisions based on local information available at every step, in such a manner that the sequence of decisions made is optimal.

Problem I : Coin Change.

Given a set of coin values, construct a sum of money using as fewer as possible. we can use each coin value any number of time

Coins = $<C_1, C_2, C_3 \ldots>$

Target Money = N.

eg:1. N=12, coin = {1, 2, 5}

Greedy method: 5+5+2 → 3 coins → Optimal Solution

eg:2 N=6, coins = {1, 3, 4}

Greedy method = 4+1+1 = 6 → not optimal (failed).

3+3 = 6 } 2 coins → Optimal

eg:3

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

Number	Weight (Kgs)	Value (Rupees)
1	10	60
2	7	28
3	4	20
4	2	40

0/1 Knapsack

Using Greedy

× $\frac{P_1}{w_1} = \frac{60}{10} = 6$

× $\frac{P_2}{w_2} = \frac{28}{7} = 4$

✓ $\frac{P_3}{w_3} = \frac{20}{4} = 5$

✓ $\frac{P_4}{w_4} = \frac{24}{2} = 12$

<2+4>

$x_4 = 1$

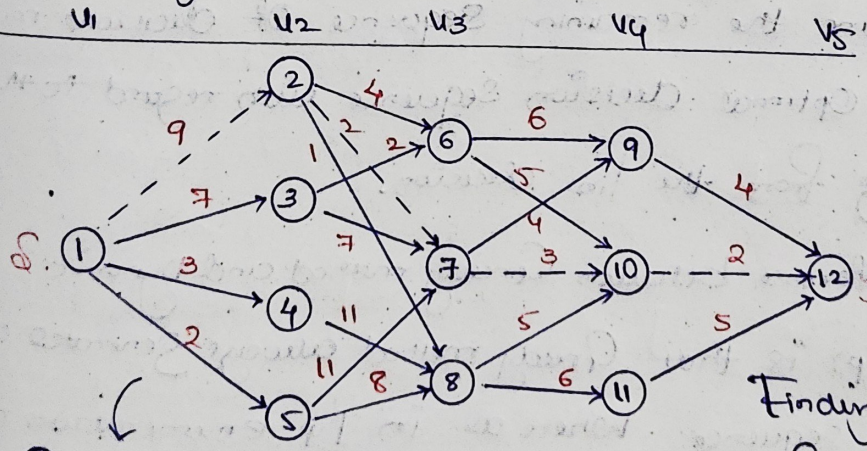
$x_2 = 1$

44

↓

not Optimal

Multi Stage Graphs



Optimal : 1 → 2 → 7 → 10 → 12.

⇒ 16

Finding shortest path from 's' to 't'

Greedy method = 1 → 5 → 8 → 10 → 12

= 17 (not Optimal)

Using Dijkstra Algorithm: (SSSP)

s → t : <1-2-7-10-12> : 16

16/8/2023

- One way of solving problems in which it is not possible to make sequence of decisions in step-wise manner leading to optimal solution is to enumerate all decision sequences (Brute Force) and then pick up the best solution (optimal).

  But the drawback with Brute force / Enumeration is excessive time/space requirements.

- Dynamic Programming (DP) based on enumeration often tried to reduce the amount of enumeration by curtailing (cut down) those decision sequence from where there is no possibility of getting optimal solution.

  In Dynamic programming these set of optimal decisions are made by applying Principle of optimality.

  global optimality / Optimal substructure property.

Principle of optimality: States that whatever the initial state and decision are the remaining sequence of decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

- Essential difference between Greedy method and Dynamic Programming (DP) is that Greedy method always generates only one decision sequence. Where as in DP enumeration many decision sequences can be generated.

- Another important feature of D.P is that optimal solutions of the subproblems are retained (cached/stored in a table) to avoid recomputing their values. Invariably this leads to saving of time

Dp Implementation → Memoization (Top-down approach) (Recursive)
→ Tabulation (Bottom-up approach) (Iterative)

Elements of DP:

(i) Splitting of Original problem into Subproblem: Be able to split original problem into subproblems in a recursive manner (so that the subproblem can be further divided into sub-problem). this process of splitting should continue till the sub problems become small.

(ii) Subproblems Optimality (Optimal Substructure): An optimal solution to the problem must result from optimal solutions to the subproblem with combine operation.

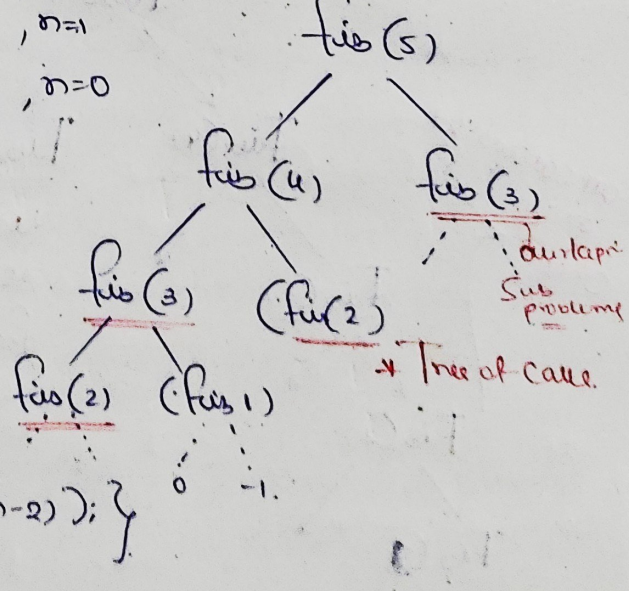
(iii) Overlapping Subproblem: Many sub problems themselves contain common sub-problems. Therefore, it is desirable to solve the small sub-problem and store/cache their results, so that they can be used in other sub-problems.

eg: Fibonacci Numbers: 0, 1, 1, 2, 3, 5, ...

$Fib(n) = Fib(n-1) + Fib(n-2)$, $n>1$
$Fib(n) = 1$, $n=1$
$Fib(n) = 0$, $n=0$

→ Normal Recursive Algorithm

```
{ Algo Fib (int n)
  if (n<=1) return n;
  else return (Fib(n-1) + Fib(n-2)); }
```



$T(n) = c, \quad n \le 1$
$= T(n-1) + T(n-2) + c, \quad n \ge 2$

$T(n) = T(n-1) + T(n-2) + a$ — (1)

$\because T(n) < T(n-1) + T(n-1) + a$

$T(n) < 2 \cdot T(n-1) + a$ — (2)

$\Rightarrow T(n) < 2^n$

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

Top-down – Memorization Implementation

memo[0.....n] – Global array – to store value.

```
Algo memiFb(n)
{ if (mn[n] is undefined)        // if result not in memo array
  { if (n<=1) result = n;        // calculate
    else
      result = memiFb(n-1) + memiFb(n-2);
    memo[n] = result; }
  return memo[n];                // finally return result from memo array
}
```

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

n=5

↳ memfib(5) = 5

→ memfib(4) = 3

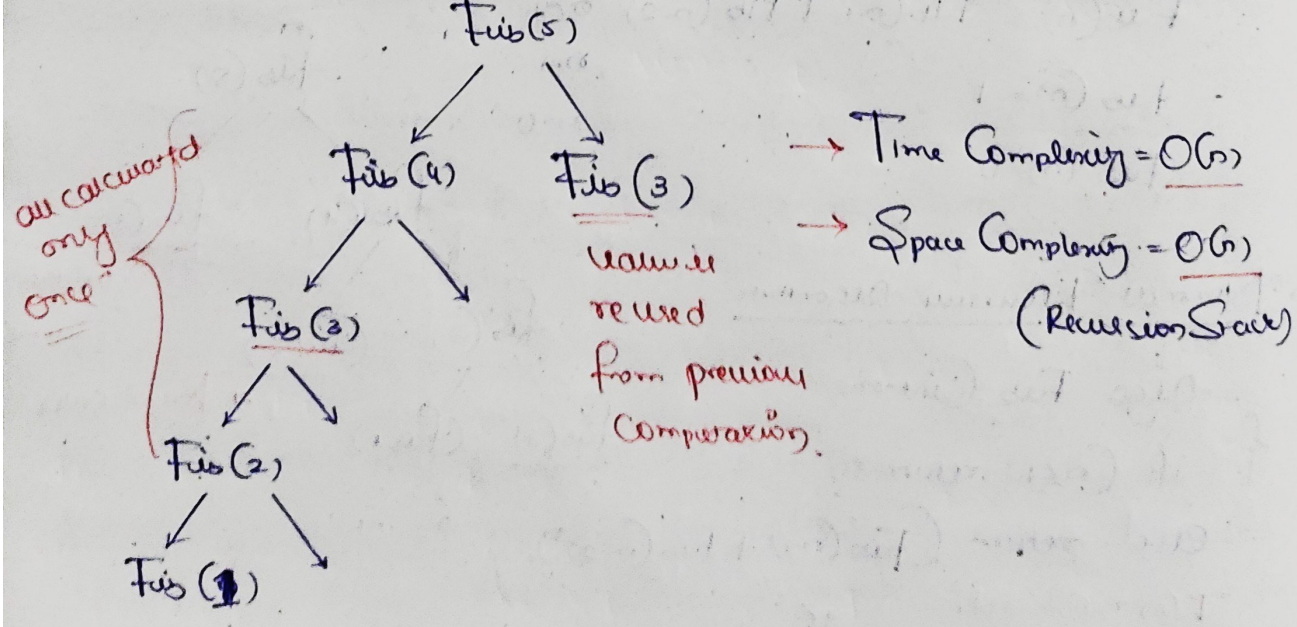
→ memfib(3) = 2+1 = 3

→ memfib(2) = 1

→ memfib(1) = 1

→ memfib(0) = 0

first time calculation



→ Time Complexity = O(n)

→ Space Complexity = O(n) (Recursion Stack)

Bottom-up Approach

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

```
Algo memFib(n)
{ m[0]=0;  { initializing
  m[1]=1;  { the first 2 values.
  for (i=2 to n) do
  { m[i] = m[i-1] + m[i-2]; }
  return (m[n]); }
```

Time Complexity = O(n)

Space Complexity = O(n) (space of array).

When to use Tabulation and when Memoization:

* If the original problem requires all sub problems to be solved, tabulation usually out performs memoization.
* Tabulation has no overhead of recursion and can use pre allotted array.
* If only some of the sub problems need to be solved for the original problem, then memoization technique is preferable because the sub problems are solved lazily.

Dynamic Programming V/s Greedy Method V/s Divide and Conquer

Greedy method: Building up the solution to the problem is done in a stepwise manner (incrementally) by applying local options only.

Divide and Conquer: Breaking up a problem into seperate problems (independent), then solve each sub problems to get the solution of original problem.

Dynamic Programming: Breaking up of a problem into series of overlapping sub-problems and building up solution of larger and larger sub problems.

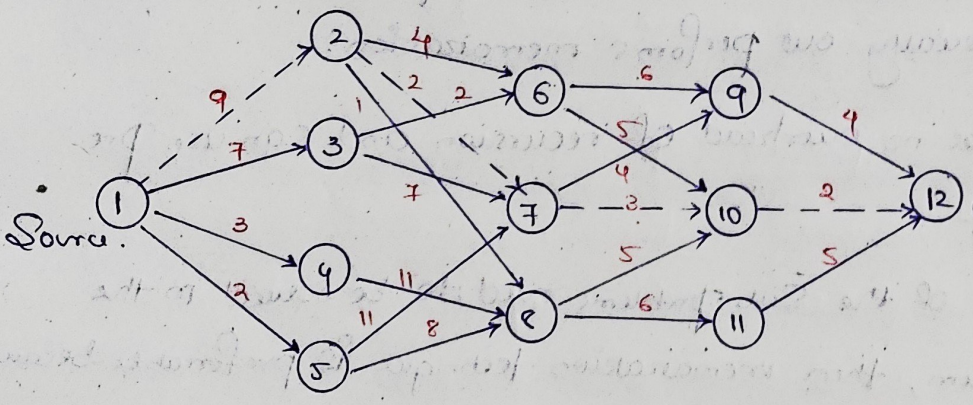
* Unlike Divide and Conquer, D.P typically involves solving all subproblems, rather than a small portion of subproblem.

* D.P tends to solve each subproblem only once since the results of the subproblems are stored, which are used later again when required. This is going to reduce the computation drastically.

eg: Fibonacci numbers → Brute Force = $O(2^n)$

→ Dynamic Programming : $O(n)$

Multi-Stage Graph.



c	1 2 3 ... 12
1	
2	
3	c(i,j) = edge cost.
⋮	
12	

Let cost(i,j) represents cost of the path from vertex 'j' present in Stage 'i' to reach destination vertex 't'.

$$Cost(1,1) = \min_{\substack{k \in V_2 \\ \langle 1,k \rangle \in E}} \{ c(1,k) + cost(2,k) \}$$

Cost(i, j) → Stage #. ; ↓ Vertex in Stage 'i'

$$Cost(1,1) = \begin{cases} \text{original path} \\ cost(1,k) + cost(2,k) \\ \langle 1 - k, \ldots\ldots\ldots\ldots\ldots\ldots - t \rangle \end{cases}$$

$k \in V_2$, $\langle 1,k \rangle \in E$

Sub path: cost(2,k)

Homework ↙

Generalize formula: Cost(i, j)

17/8/2023

$$Cost(i,j) = \min_{k \in V_{i+1}, \langle j,k \rangle \in E} \{ c(j,k) + Cost(i+1,k) \} \quad -1.$$

$$Cost(i-1,j) = c(j,f) \quad -2.$$

$D(i,j)$ = 'k' that minimizes the eqn 1.

$$Cost(1,1) = \min \{ c(1,2) + Cost(2,2), \; c(1,3) + Cost(2,3), \; c(1,4) + Cost(1,4), \; c(1,5) + Cost(2,5) \}$$

$Cost(4,9) = c(9,f) = 4$
$Cost(4,10) = c(10,f) = 2$
$Cost(4,11) = c(11,f) = 5$

} 4th Stage value of Cost

$$Cost(3,6) = \min \{ c(6,9) + Cost(4,9), \; c(6,10) + Cost(4,10) \} = 7$$

$D(3,6) = 5.$

$Cost(3,7) = 5, \quad D(3,7) = 10$

$Cost(3,8) = 7 \quad D(3,8) = 10$

$Cost(2,2) = \min \{ 4+7, 2+5, 1+7 \} = 7 \quad | \quad D(2,2) = 7$

$Cost(2,3) = \min \{ 2+7, 7+5 \} = 9. \quad | \quad D(2,3) = 6$

$Cost(2,4) = 18 \quad | \quad D(2,4) = 8$

$Cost(2,5) = 15. \quad | \quad D(2,5) = 8.$

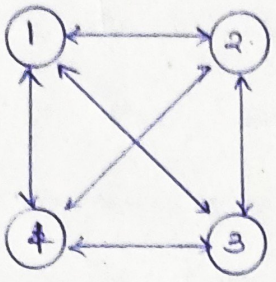
Final cost $\Rightarrow$ 16.

Path Construction = $\langle 1 \to 2 \to 7 \to 10 \to 12 \rangle$

(2.) Travelling Salesperson Problem.

v_0 = home city.

$G = (V, E)$



The tour of TSP should start from home city v_0 and visit remaining $(n-1)$ cities exactly once and come back to home city (v_0). Start that the cost of tour is minimum.

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

→ Greedy Method = (1-2-3-4-1) : 39.
($v_0 = 1$)

Optimal Solution

(1-2-4-3-1) : 35

* Solution Space for brute force approach = $(n-1)!$

→ Let $g(i, s)$ represent cost of the tour of travelling salesperson from vertex 'i' and visiting all vertices in the set 's' exactly once and terminating the tour at v_0.

$v_0 = 1$

$$g(1, \{2,3,4\}) = \min_{\substack{k \in S \\ \langle 1,k \rangle \in E}} \{ c(1,k) + g(k, S-\{k\}) \}$$

$c(1,k)$

$\langle 1-(k \ldots\ldots v_0) \rangle$

$k \in S$

$\langle 1,k \rangle \in E.$

$$g(i,s) = \min_{\substack{k \in S \\ \langle i,k \rangle \in E}} \{ c(i,k) + g(k, S-\{k\}) \} \quad \text{(1)}$$

$$g(i, \phi) = c(i, v_0)$$

$i - v_0$

$J(i,s)$ = value of 'k' that minimizes the eqn (1)

$v_0 = 1$

$$g(1,\{2,3,4\}) = \min\{\underset{k=2}{c(1,2)+g(2,\{3,4\})},\ \underset{k=3}{c(1,3)+g(3,\{2,4\})},\ \underset{k=4}{c(1,4)+g(4,\{2,3\})}\}$$

$$= \min\{10+25,\ 15+25,\ 20+23\} \Rightarrow \underline{35}$$

$|S| = 0$

$g(2,\phi) = c(2,1) = 5$

$g(3,\phi) = c(3,1) = 6$

$g(4,\phi) = c(4,1) = 8.$

$|S| = 1$

$g(2,\{3\}) = c(2,3) + g(3,\phi) = 9+6 = 15.$

$g(2,\{4\}) = 10+18 = 18$

$g(3,\{2\}) = 13+5 = 18$

$g(3,\{4\}) = 12+8 = 20$

$g(4,\{2\}) = 8+5 = 13$

$g(4,\{3\}) = 9+6 = 15.$

T.SP is the one of the problem in literature where there is no polynomial time algorithm in literature

$|S| = 2$

$g(2,\{3,4\}) = 25$

$J(2,\{3,4\}) = 4.$

$g(3,\{2,4\}) = 25$

$J(3,\{2,4\}) = 4$

$g(4,\{2,3\}) = 23$

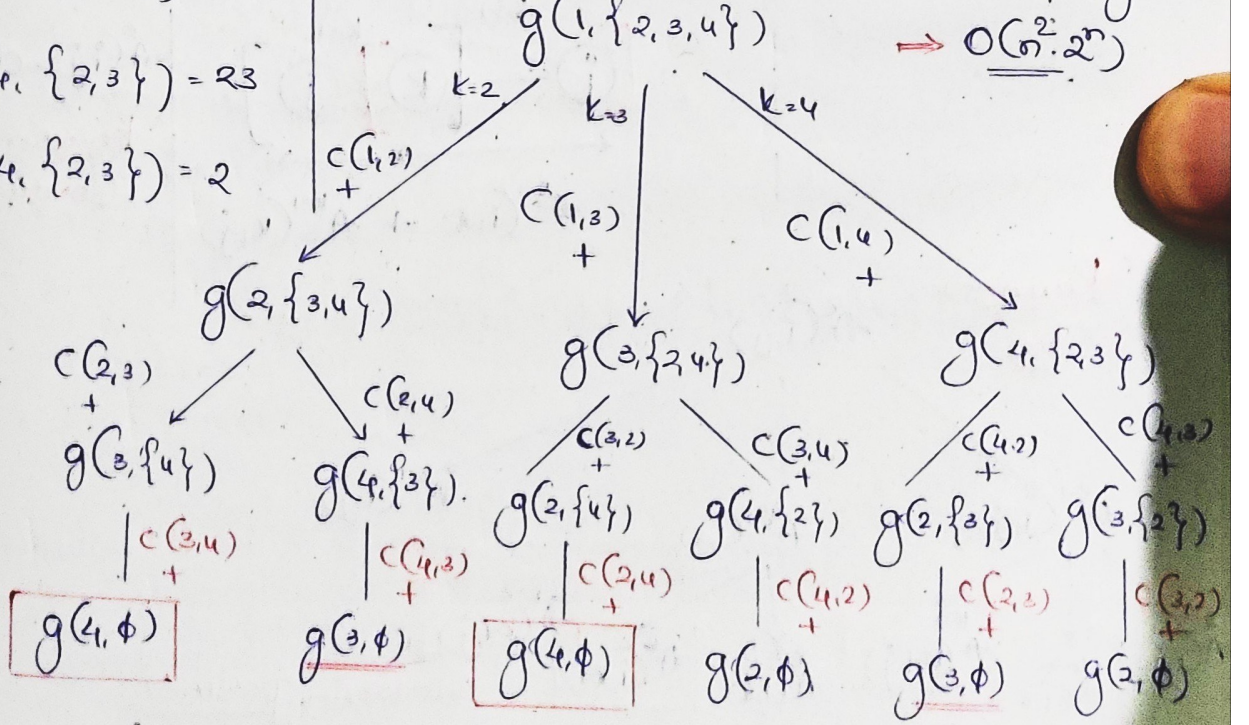
$J(4,\{2,3\}) = 2$

Tour Construction

$v_0 = 1.$

$\langle 1-2-4-3-1 \rangle = 35.$

* Held-Karp Algorithm $\Rightarrow O(n^2 \cdot 2^n)$



18/8/2023

(3.) All-pairs Shortest Paths

→ This problem is solvable by greedy method also.

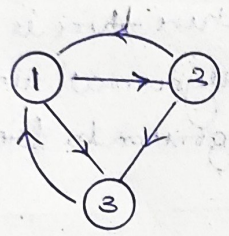
↳ Single source shortest paths

↓

Apply it n times, changing source vertex each time!

→ Solvable by dynamic programming

"Floyd-Warshall's algorithm"



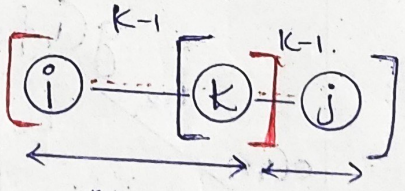
Cost	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Let $A^k(i,j)$ represents cost of the path from vertex 'i' (src) to vertex 'j' (dest), with 'k' being the highest intermediate vertex along the path.

$A^k(i,j)$: i ... k ... j

$$A^k(i,j) = \min\left\{ A^{k-1}(i,k) + A^{k-1}(k,j),\ A^{k-1}(i,j) \right\}$$

$k = n, 1.$

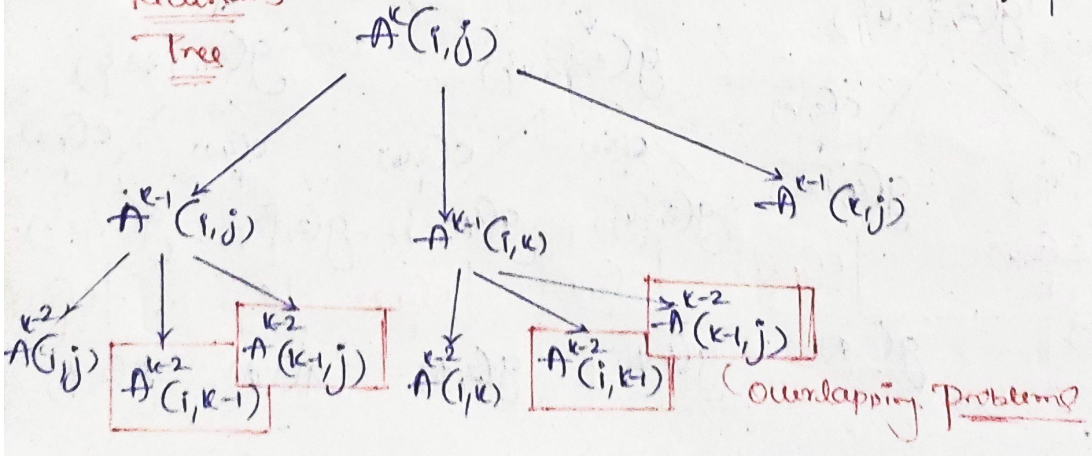


$A^{k-1}(i,k) + A^{k-1}(k,j)$

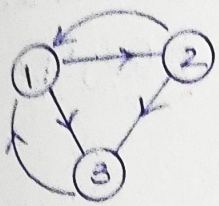
$A^0(i,j) = C(i,j)$

Boundary Condition

Recursion Tree



(overlapping problems)



$A^k(i,j) = \min\{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\}$

$A^3 \to A^2 \to A^1 \to A^0$

C	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

formula/eqn

$A^1(1,2) = \min\{A^0(1,2), A^0(1,1) + A^0(1,2)\}$

$\to 4$

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Shortest path from a vertex to another vertex

Final Answer

Algorithm Floyd-Warshall (G, n, e, c)

```
{
    integer c[1...n, 1...n];
    integer A[1...n, 1...n];
1.  for i ← 1 to n
        for j ← 1 to n
            A[i,j] = c[i,j];
2.  for k ← 1 to n .— intermediate vertex
        for i ← 1 to n — Src
            for j ← 1 to n: — dest
                A[i,j] = min{A[i,j], A[i,k] + A[k,j]}
}
```

Time = $O(n^3)$

Space = $O(n^2)$

* Transitive closure of a matrix (representing a Graph)

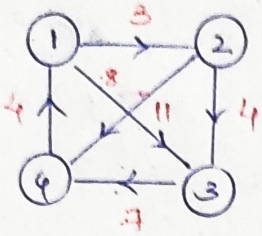
$i \to j$, $j \to k$ } $i \to k$.

path

→ Floyd-Warshall Algorithm can be used to obtain transitive closure of a matrix (representing a graph) in $O(n^3)$ T.C

HW

Solve using Floyd Warshall Algorithm.



4. 0/1 Knapsack (Binary Knapsack)

→ KNAP Cap : m

→ no. of objects : n.

(Oi)

wi pi xi

→ Max $\sum_{i=1}^{n} p_i x_i$

→ Subject to cost $\sum_{i=1}^{n} w_i x_i \leq m$.

$x_i = 0/1$.

Solution Space $= O(2^n)$

$\langle x_1, x_2, x_3 \ldots x_n \rangle$

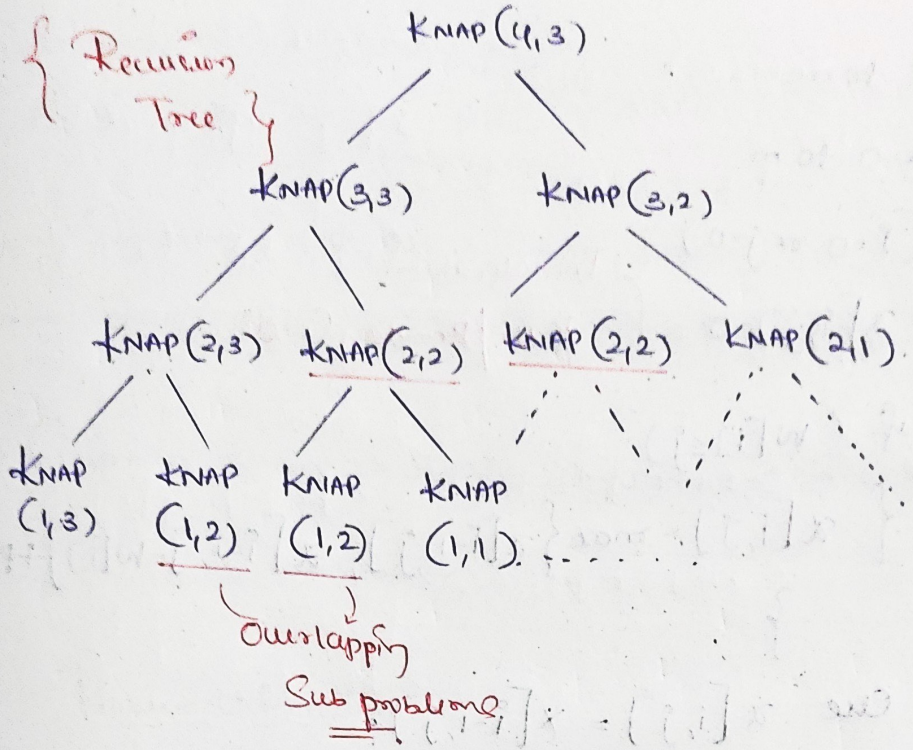
Let 0/1 KNAP(n,m) represents profit using n objects and KNAP of Capacity 'm'.

0/1 KNAP(n,m) = 0/1 KNAP(n-1, m), $w_n > m$.

$\Rightarrow$ Max { 0/1 KNAP(n-1, m) [$x_n = 0$], 0/1 KNAP(n-1, m-w_n) + p_n [$x_n = 1$] }.

Boundary Conditions { 0/1 KNAP(n,m) = 0, n=0, or m=0

profit must be zero — no objects — zero or less Capacity in knapsack



Data: $n=4$, $m=8$ $w[2,3,4,5]$, Profits $[1,2,5,6]$

let $x[0\ldots n, 0\ldots m]$ be an array of entries $x[n,m]$ = profit.

$\S$ – (bottom-up-tabulation).

X			0	1	2	3	4	5	6	7	8	
P_i	W_i	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	0	1	1	1	1	1	1	1	
2	3	2	0	0	1	2	2	3	3	3	3	
5	4	3	0	0	1	2	5	5	6	7	7	
6	5	4	0	0	1	2	5	6	6	7	(8)	

Max profit

Max profit = $x[n][m] = 8$

Objects included = $\langle x_1 x_2 x_3 x_4 \rangle$

0 1 0 1

$x[i,j] = \max\{x[0,j], x[0,j-1] + P_i\}$

→ Algorithm 0IKNAP (m, n, w, P)

{ $P[1\ldots n]$, $w[1\ldots n]$ // Global variables

integer $x[0\ldots n, 0\ldots m]$.

```
for i ← 0 to n
    for j ← 0 to m.
        if (i=0 or j=0).        } Initialization
            x[i,j]=0            } - Base / Boundary Condition
    else { if (w[i] ≤ j)
            { x[i,j] = max{ x[i-1,j], x[i-1, j-w[i]]+Pi }
            }
        else x[i,j] = x[i-1,j];
        }
    }
```

Time = $O(n*m)$

Space = $O(n*m)$

⑤ Longer Common Subsequence - (String Manipulation).

String : a group of one/more characters

Substring

- A group of one or more characters taken in contigous order from given String

Subsequence

A group of one or more characters taken from the given string that may not be contigous. but however their relative order is maintained.

Q1. Given a string of length n-characters, then no. of substrings possible are $n(n+1)/2 \Rightarrow O(n^2)$

Subsequences of <A, B, C>

{A, B, C, AB, AC, BC, ABC} ← valid subsequences

CA ? CB ← not valid

* Every Substring is a also a Subsequence.
* Every Subsequence mayor may not be substring

Q2) Given a String of length n characters, the no. of subsequences possible are $O(2^n)$

Problem definations

19/8/2023

Given two strings x and y of lengths 'n' and 'm' characters, a subsequence that is common to both x and y is known as common subsequence.

eg: x = ABCD. y = BDC LCS = BC, max length = 2.

⇒ The problem of LCS is to determine a subsequence of longest length that is common to both x and y.

Applications of LCS

1. Geonomics: Genetic engineering for DNA Strange samplings.
2. Version Change: in software engineering. (determining similarity).
3. Data Gathering Systems (eg Google).
4. Plagarism Checking System.

DP based Solution:

let i and j be the indices into the strings 'x' and 'y' as shown.

$$X = \langle x_1^0, x_2^1 \ldots x_n^i \rangle \quad Y = \langle y_1^0, y_2^1 \ldots y_n^j \rangle$$

→ Let L(i,j) denote the length of common subsequence of the strings x and y as defined above.

Case I. $x = \langle GTTCCTAATA \rangle$ (0 1 2 3 4 5 6 7 8 9)

$y = \langle CGATAATTGAGA \rangle$ (0 1 2 3 4 5 6 7 8 9 10 11)

$L(9,11)$

$x[i] = y[i]$ if last char matches

$\Rightarrow L(i,j) = 1 + L(i-1, j-1)$

Case II.

$x = \langle GTTCCTAATA \rangle$ (0 1 2 3 4 5 6 7 8 9)

$y = \langle CGATAATTGAG \rangle$ (0 1 2 3 4 5 6 7 8 9 10)

$L(9,10)$

$x[i] \neq y[i]$

(if last char doesn't match)

$L[i,j] = \max\{L(i-1,j), L(i,j-1)\}$

Recursive formula:

$$L(i,j) = 1 + L(i-1,j-1), \text{ if } x[i] = y[j]$$

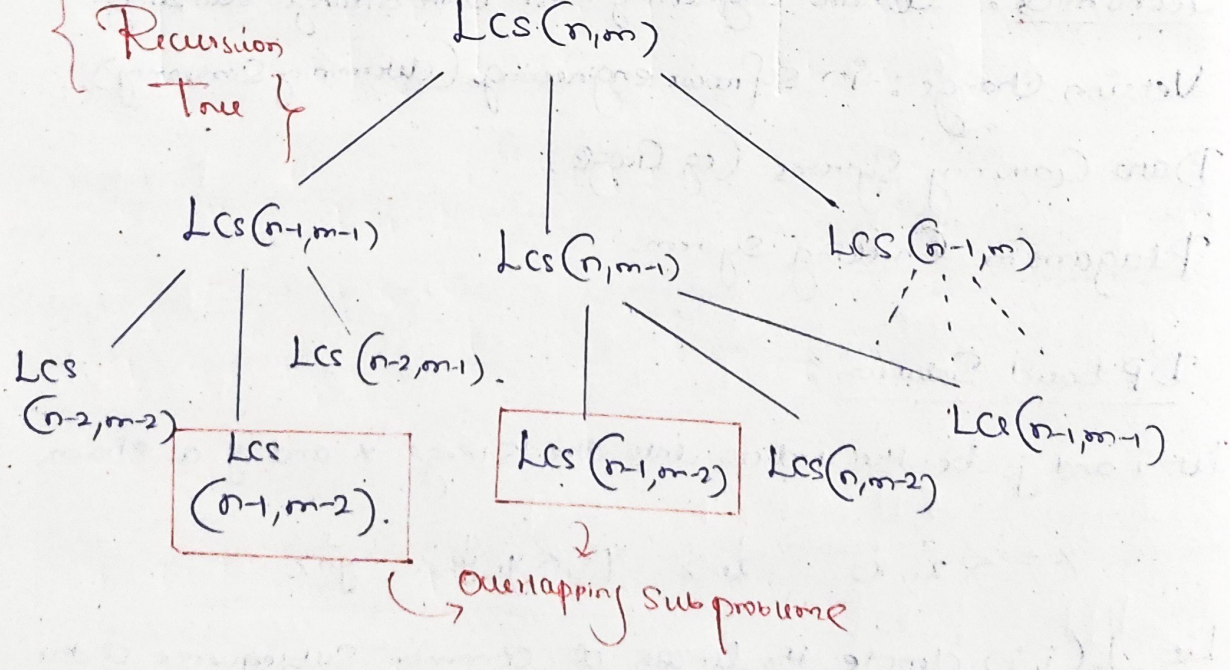
$$L(i,j) = \max\{L(i-1,j), L(i,j-1)\} \text{ if } x[i] != y[j]$$

Base Case:

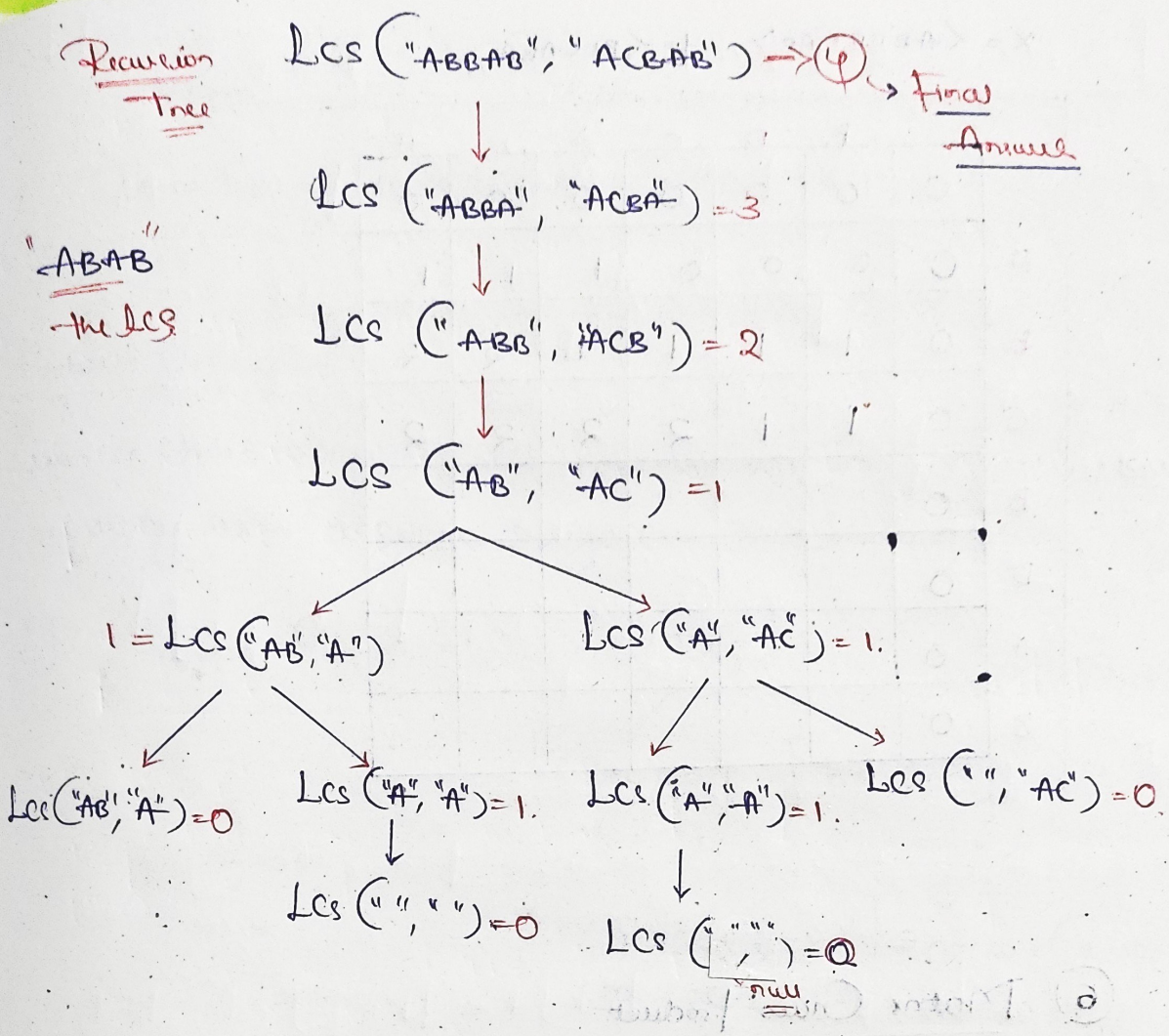
$$L(-1,j) = 0, \text{ if } i=-1 \text{ and } j\text{-anyno.}$$

$$L(i,-1) = 0, \text{ if } j=-1 \text{ and } i\text{-anyno.}$$

Recursion Tree



Recursion Tree



Algorithm LCS Based on Bottom-up Approach Tabulation Method

```
Algorithm LCS (x, y)
integer, x[0....n], y[0...m];
{ integer L[0...n, 0...m];
1) for i ← 0 to n-1
       L[i-1] = 0
2) for j ← 0 to m-1
       L[-1, j] = 0
3) for i ← 0 to n-1.
       for j ← 0 to n-1
           if (x[i] == y[j]) L[i,j] = 1 + L[i-1, j-1];
           else L[i,j] = max { L[i-1, j], L[i, j-1] };
```

Boundary Conditions

Time = O(n x m)
Space = O(n x m)

$X = \langle ABCBDAB \rangle$, $Y = \langle BDCABA \rangle$

		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0						
D	0						
A	0						
B	0						

(6) Matrix Chain Product

Given 2 square matrices A and B of order $n \times n$.

The no. of scalar multiplications needed to multiply them

$A \times B = n^3$.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{n \times n} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_{n \times n} \quad C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}_{n \times n}$$

$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$

$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$

$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$

$c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$

$2 \times 2 \times 2 = 8$.

(11) Non-Square Matrix multiplications.

$A_{2\times3} \times B_{3\times4} = C_{2\times4}$. Total multiplications $= 2 \times 4 \times 3 = 24$.

Should be same

$A_{n \times m} * B_{m \times k} \Rightarrow C_{n \times k}$.

Total Scalar multiplications $= n \times m \times k$.

Problem Definition

Given a chain of compatible non-square matrices, it is required to multiply them together to get one final matrix.

eg: $\langle B_{2\times10}, C_{10\times50}, D_{50\times20} \rangle \Rightarrow BCD$.

$[(B\times C)\times D]$ approach 1. $[B\times(C\times D)]$ approach 2.

⇒ No. of matrix multiplications

approach 1: $(2\times10\times50)+(2\times50\times20)$

$1000+2000=3000$

approach 2: $2\times10\times20+(10\times50\times20)$

$400+10,000$

$=10,400$

Difference. (Just by changing the parenthesis or multiplying approach)

Given a chain of matrices $\langle A_1, A_2 \ldots A_n \rangle$ where matrix A_i is of dimension $P_{i-1} \times P_i$. The problem of a MCP is to fully parenthesize the chain, such that the total no of scalar multiplications is minimum.

eg: $\langle A_1 A_2 A_3 A_4 \rangle$

- $A_1 (A_2 A_3 A_4)$
  - $A_2 (A_3 A_4)$ — I
  - $(A_2 A_3) A_4$ — II
- $(A_1 A_2)(A_3 A_4)$ — III
- $(A_1 A_2 A_3) A_4$
  - $(A_1)(A_2 A_3)$ — IV
  - $(A_1 A_2) A_3$ — V

20/8/2023

Derivation of D.P based recurrence of M.C.P

→ Let the resultant matrix $A_{i..j}$ be the product $\langle A_i \dots A_j \rangle$

* Any optimal parenthesization, must split the chain about the matrix A_k and A_{k+1} such that total no. of matrix multiplications is minimum.

$$\langle (A_i, A_{i+1} \dots A_k)(A_{k+1} \dots A_j) \rangle \quad [i \le k \le j-1]$$

k-splitting point.

→ Let $m[i,j]$ denote the number denote the number of scalar multiplications to get the resultant matrix $A_i \dots A_j$.

$$m[i,j] = \min \{ m[i,k] + m[k+1,j] + P_{i-1} * P_k * P_j \}$$
$$i \le k \le j.$$

eg: $A_{1\dots6}$ (2×6) $= \langle (A_1 A_2 A_3)(A_4 A_5 A_6) \rangle$ — I, II

$A_1 \to P_0 \times P_1 = 2 \times 3$
$A_2 \to P_1 \times P_2 = 3 \times 5$
$A_3 \to P_2 \times P_3 = 5 \times 8$
$A_4 \to P_3 \times P_4 = 8 \times 4$
$A_5 \to P_4 \times P_5 = 4 \times 7$
$A_6 \to P_5 \times P_6 = 7 \times 6$

$(A_{1\dots3})$ (I) $\leftrightarrow$ $(A_{2\dots6})$ (II)

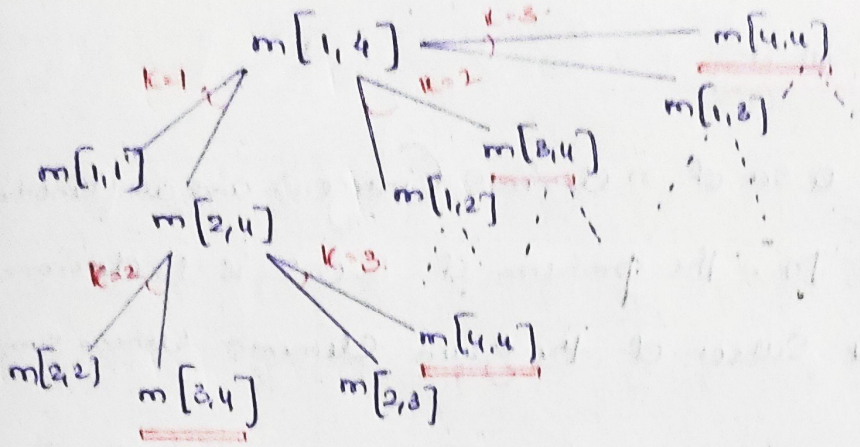
$A_{1\dots3}$: 2×8, $P_0 \times P_3$ — $A_{2\dots6}$: 8×6, $P_3 \times P_6$

$\Rightarrow m[1,3] + m[4,6] + P_0 P_3 P_6$

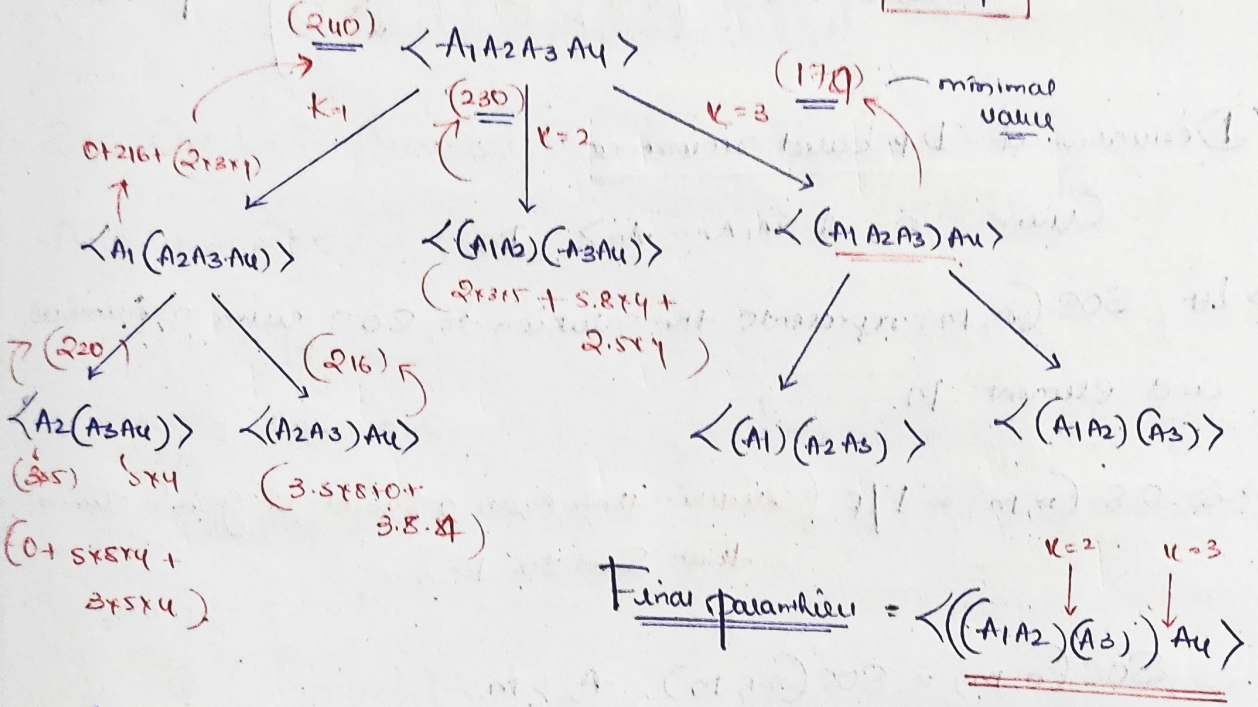
$$m[i,j] = \min \{ m[i,k] + m[k+1,j] + P_{i-1} * P_k * P_j \}$$
$$i \le k \le j.$$

$m[i,j] = 0$

$S[i,j] = k$ (point of split)



eg : $\langle A_1 A_2 A_3 A_4 \rangle = \langle 2\times3, 3\times5, 5\times6, 6\times4 \rangle \Longrightarrow$ **174**



Final parenthesis = $\langle ((A_1 A_2)(A_3))A_4 \rangle$ (k=2, k=3)

Algorithm Matrix-Chain-Product(p)

```
n ← length[p] - 1        // n - no of matrices in chain
for i ← 1 to n
    do m[i,i] ← 0
for l ← 2 to n           // l is the length of chain
  for i ← 1 to n-l+1
      j ← i+l-1
      m[i,j] = ∞
      for k ← i to j-1
          q ← m[i,k] + m[k+1,j] + P_{i-1} P_k P_j
              if (q < m[i,j]) then m[i,j] ← q
                                   s[i,j] ← k
return m and s
```

Time Complexity : $O(n^3)$

Space Complexity : $O(n^2)$

(7) Sum of Subsets

Definition: Given a set of n elements (integers) and also another element (number) 'm'. The problem of S.O.S is to determine if there exists a subset of the given elements whose sum equals to 'm'.

eg: $n=5$.
$\langle 10, 20, 30, 40, 50 \rangle$, $m = 50$
(1, 2, 3, 4, 5)
Subsets are = $\{1, 4\}, \{2, 3\}, \{5\}$ } It is a decision Problem.

Derivation of D.P based recurrence.

Given: n, ~~A~~ $\langle A_1 A_2 \cdots A_n \rangle$, m.

→ Let $SOS(n, m)$ represents the solution to SOS with n, number and element m.

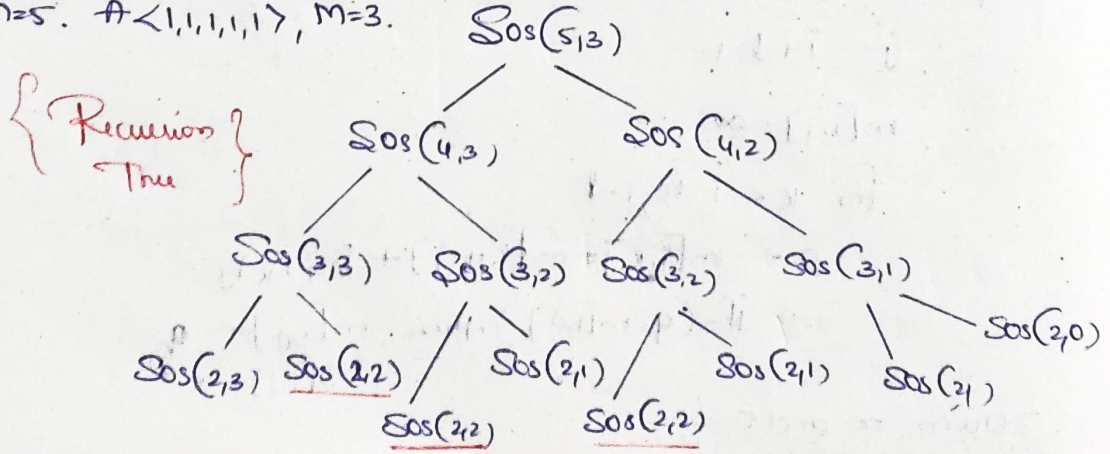
→ $S.O.S(n, m) = T/F$ (whether there exists a subset of given 'n' elements that sum to m).

$SOS(n, m) = SOS(n-1, m)$, $A_n > m$.
(last element, if not included because greater than m)

$= [SOS(n-1, m)$ or $SOS(n-1, m-A_n)]$, $A_n \leq m$.

= False, $n=0, m>0$
= True, $n \geq 0, m=0$ } Base conditions

eg: $n=5$. $A\langle 1,1,1,1,1 \rangle$, $M=3$.



{ Recursion Tree }

eg $n=5$, $m=6$, $A=\langle 2,8,4,11,9\rangle$

$x[i,j] = T/F$

i \ j	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
2 1	T	F	T	F	F	F	F
8 2	T	F	T	F	F	F	F
4 3	T	F	T	F	T	F	T
11 4	T	F	T		T		
9 5	T	F	T		T		

Weather there exist a subset from the given 'i' elements whose sum is 'j'.

→ SOS can be implemented using bottom-up approach.

Algorithm SOS (n, m, A)

{

A[1...n]

Integer x[0...n, 0...m]

for i ← 0 to n

for j ← 0 to m

if (i ≥ 0 and j = 0)

x[i, j] = T

else

if (i = 0 and j > 0)

x[i, j] = F;

Base Case.

else if (A[i] > j) x[i, j] = x[i-1, j]

else x[i, j] = x[i-1, j] ∨ x[i-1, j - A[i]]

}

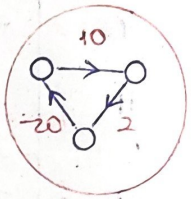
Time : O(n × m)

Space : O(n × m).

(8) Bellman Ford Algorithm (Single Source Shortest Path)

(i) Dijkstra's Algorithm (greedy method) always works, provided the graph has +ve weight edges.

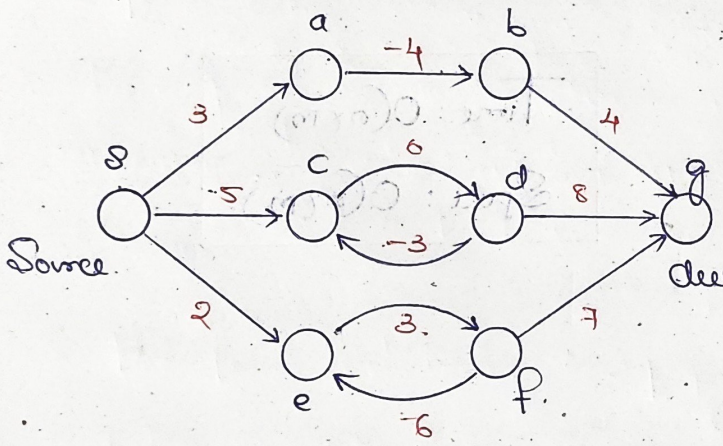
(ii) If the Graph has -ve weight edges but no -ve weight cycle then Dijkstra's Algorithm may/may not work.



-8

(iii) If the graph have -ve weight edges and no -ve weight cycle then Bellman Ford Algorithm always works correctly.

(iv) If the graph has -ve weight cycle reachable from source then No-algorithm works.



1. $S \to a = 3$
2. $S \to b = -1$
3. $S \to g = 3$. (?)

4. $S \to C = 5$

$S \to C \to d \to c = 8$

$= 11$

$= 14$

5. $S \to e = 2$

$= -1$

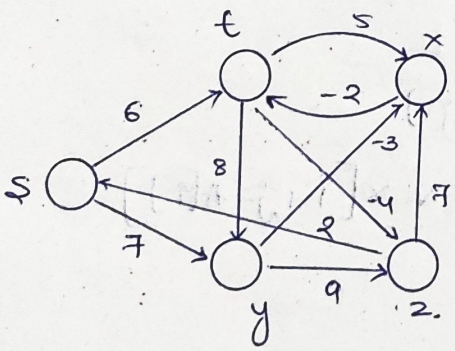
$= -4$

6. $S \to e \to f = 5$

$= 2$

$= -1$.

Homework:



v.	s	t	x	y	z
{s}	-	6	4	7	2
		x			x

By dijkstra Algorithm

$S \to z$

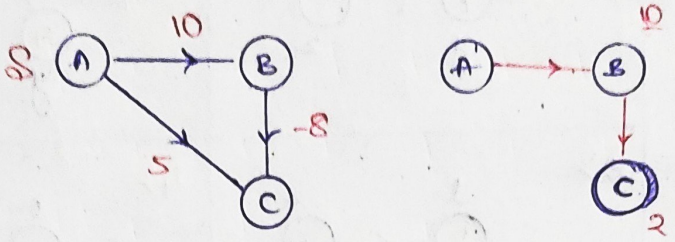
$S \to y \to x \to t$ } Optimal Cost.

$7 - 3 - 2 = 2$. }

↳ Dijkstra's Algorithm failed to find Shortest Paths for few vertices from Source.

→ In dijkstra's algorithm (Greedy), the relaxation process is carried out w.r.t vertices. The d-value once selected cannot be relaxed further.

→ In Bellman Ford's algorithm, the relaxation is carried out w.r.t edges in multiple iterations.



* In a graph having n-vertices and any unrestricted path between any two pair of vertices cannot have more than n-1 edges without any cycle.

Algorithm Bellman-Ford(G, w, s)

{ Initialise-Single-Source(G,s)

for i ← 1 to (|V[G]| - 1)

do for each edge (u,v) ∈ E[G]

do Relax(u, v, w)

for each edge (u,v) ∈ E[G]

do if d[v] > d[u] + w(u,v) } to detect -ve weight cycle.

then return false;

return True;

Time : n+n.e+e

⇒ O(n.e)

Initialise-Single Source (G,s)

{ for each vertex v ∈ V[G]

do d[v] ← ∞

π[v] ← NIL

d[s] ← 0

}

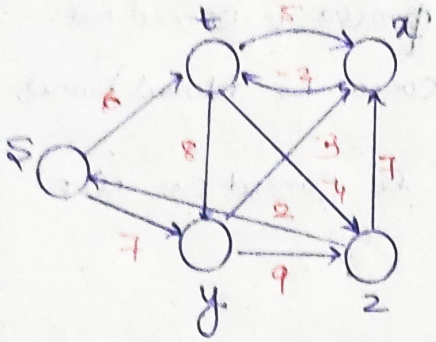
Relax(u,w)

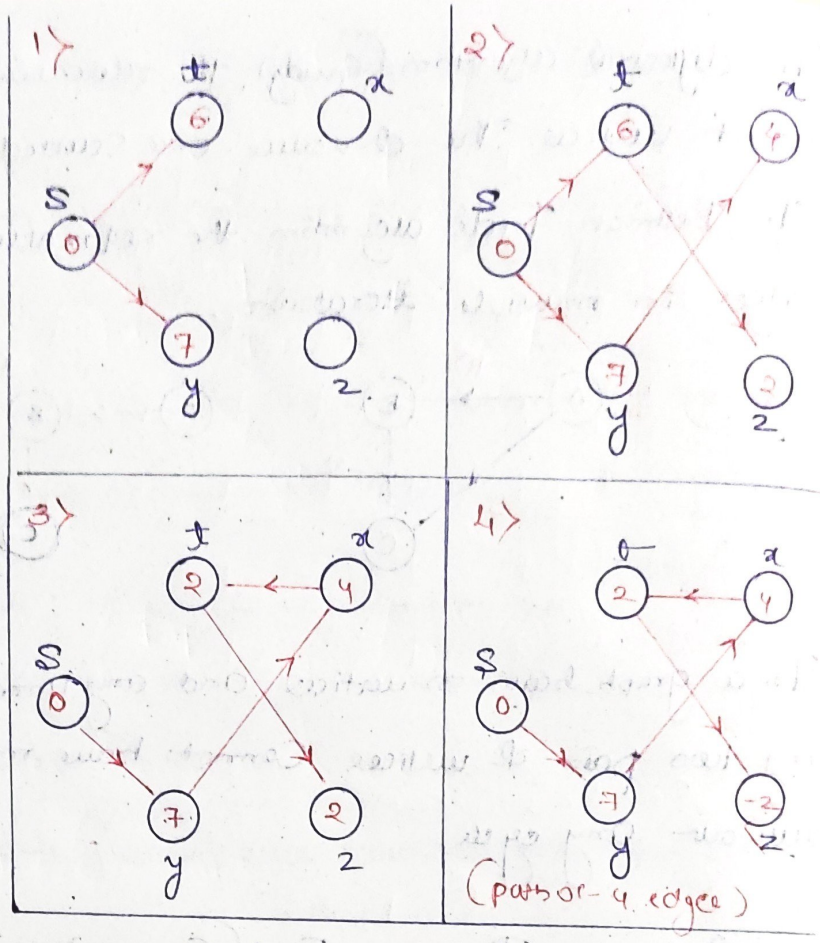
{ if d[v] > d[u] + w(u,v)

then d[v] ← d[u] + w(u,v)

π[v] ← u.

}





→ If graph has no -ve weight cycle, reachable from source, then shortest paths are always determinable after completing $(n-1)$ th relaxation. And additional relaxations will not change the d-value.

→ The time-complexity of B.F algorithm for a complete Graph having 'n' vertices is $O(n^3)$. { Complete Graph $e = O(n^2)$ }

(Bellman Ford)

Graph Techniques

21/8/2023

Traversal: visiting all the nodes of the tree/graph in a specified order and processing the info only once.

Graph traversals:

- Depth First Search
  - Undirected
    - Connected
    - disconnected
  - Directed
    - Connected
    - disconnected
- Breadth First Search (BFS)
  - FIFO BFS
  - LIFO BFS

(I) DFS in undirected Graphs

(a) Connected Graphs:

Terminologies:

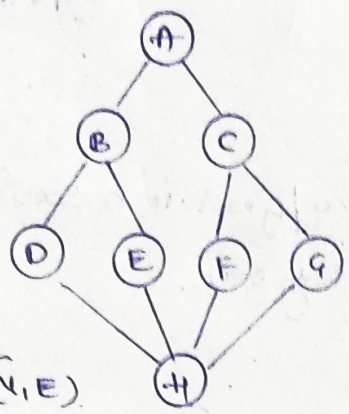
(i) Status of a Node:

- E-node: Exploring node (node which is currently being explored).
- Live-node: Node which is not fully explored (live nodes are stored in some D.S).
- Dead node: Node which is fully been explored.

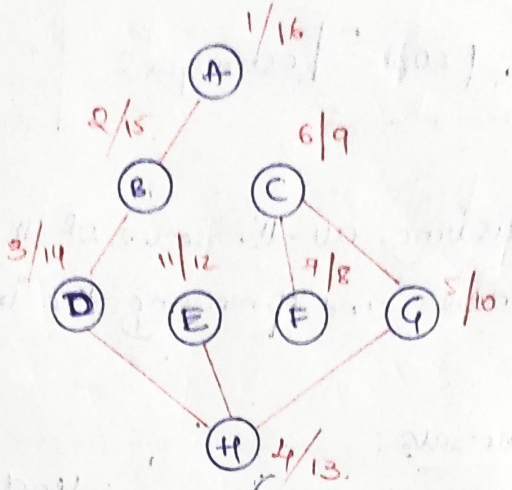
(ii) Timing-values associated with nodes, during traversal.

Discovery-time: d(x): The time at which the node is visited for the first time.

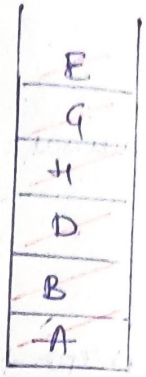
Finishing-time: f(x): The time at which the node becomes dead node.



G = (V, E)



(Start / end time)



(Until Stack Empty)

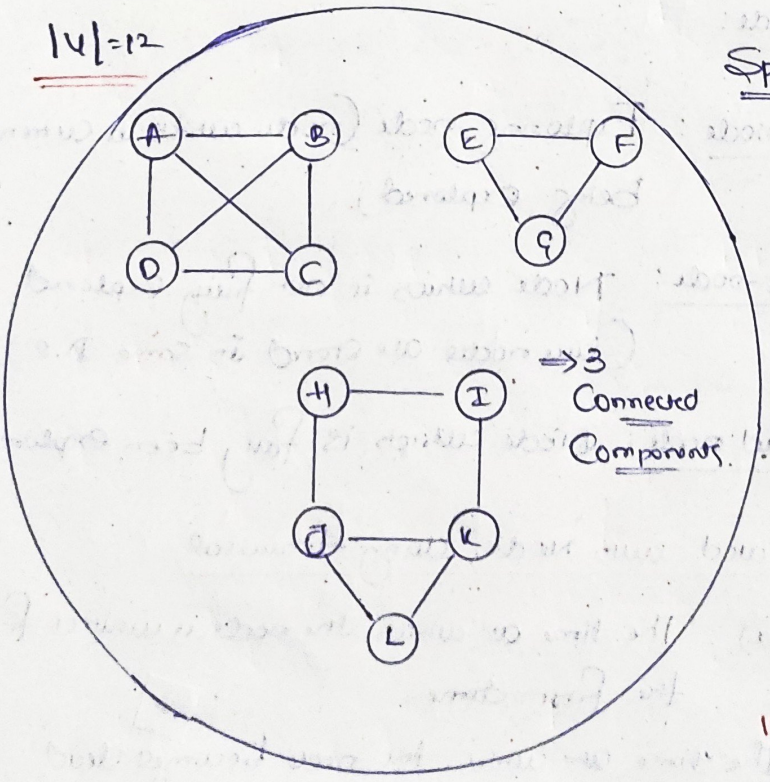
{DFS Spanning Tree}

Which of the traversals below are valid?

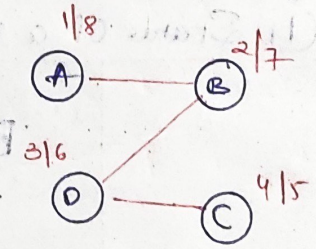
1. A, B, E, H, D, F, C, G - valid
2. A, C, F, E, H, G, B, D. - invalid.
3. H, D, B, E, A, C, F, G - valid
4. H, F, C, B, A, D, E, G - invalid
5. G, C, F, H, E, B, D, A. - valid.

(b) Disconnected / Disjoint Graphs (DFT)

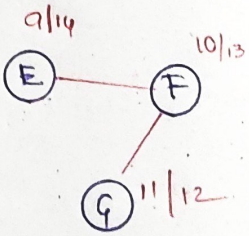
DFS Spanning Tree (forest)



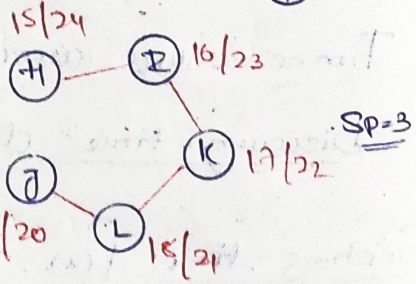
Sp=1



Sp=2



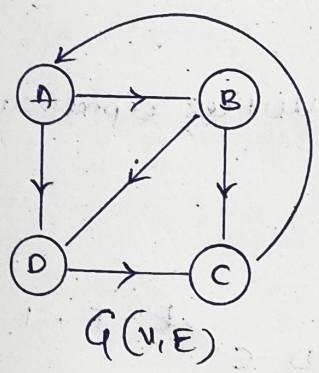
Sp=3



Q) Consider an undirected Graph with 4 vertices <P, Q, R, S>
DFS is carried out generating the following d/f times.

	P	Q	R	S	
1.	<3,25>	<5,18>	<8,15>	<10,12>	– Connected
2.	<12,25>	<5,10>	<6,8>	<15,20>	– disconnected
3.	<8,10>	<18,22>	<3,15>	<6,12>	– disconnected <R S P> <Q>
4.	<18,22>	<12,15>	<8,10>	<25,30>	– disconnected 4 components

II) DFS in directed Graphs



G(V,E)

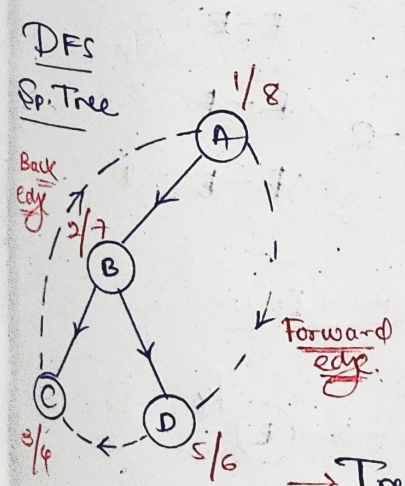
DFS when Carried out on a directed graph leads to following type of edges:

1) Tree Edge: are part of DFS Spanning tree or forest.

2) Forward edge: leads from a node to its non child descendent in the Spanning tree.

3) Back edge: leads from a node to its ancestor

4) Cross edge: leads to an edge node which is neither ascendig nor descending.

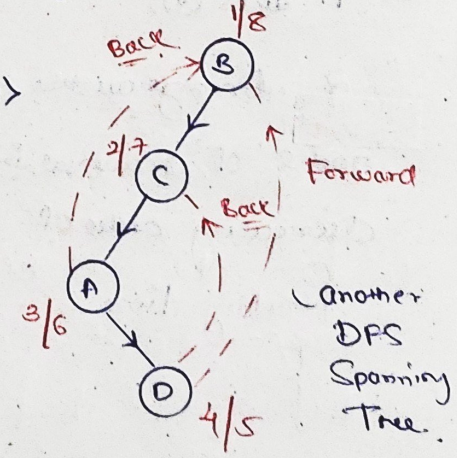


→ Tree edges: <AB> <BC> <BD>

→ Forward edge: <AD>

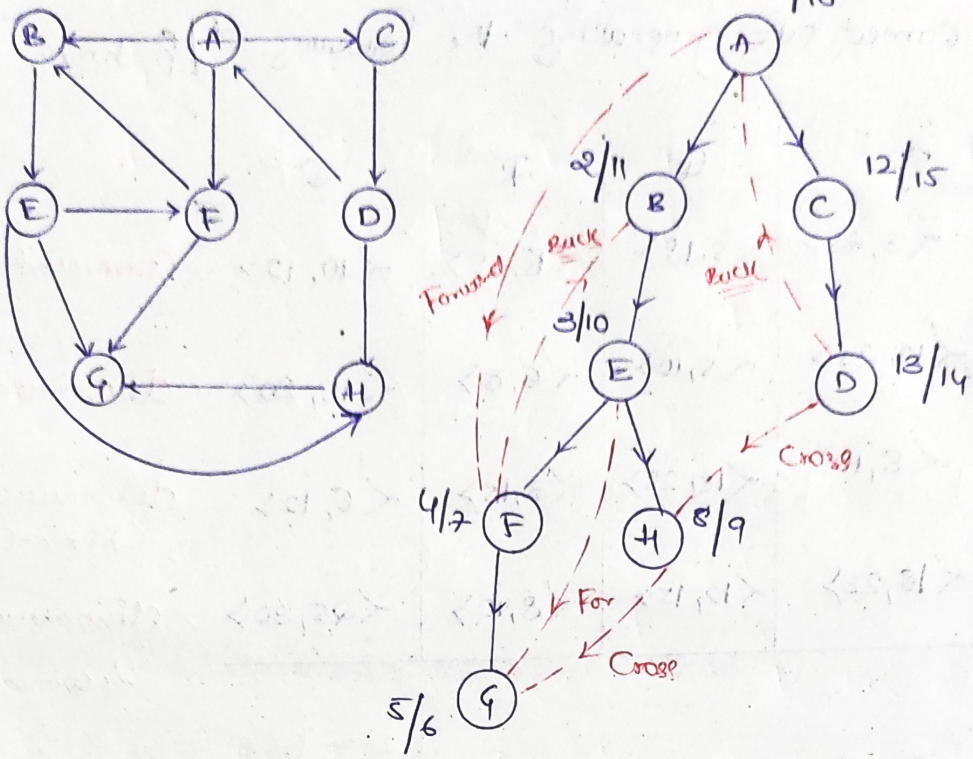
→ Back edge: <CA>

→ Cross edge: <DC>



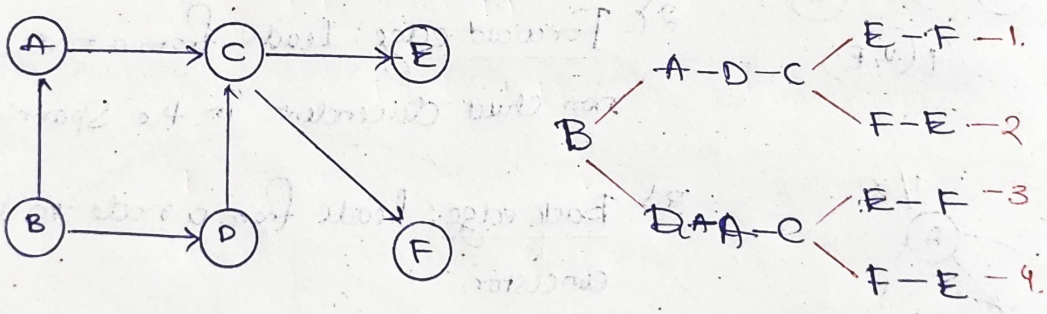
Another DFS Spanning Tree.

22/8/2023



DFS in Directed Acyclic Graph

Topological Sort: Linear order of the vertices representation the activities maintaining precedence.



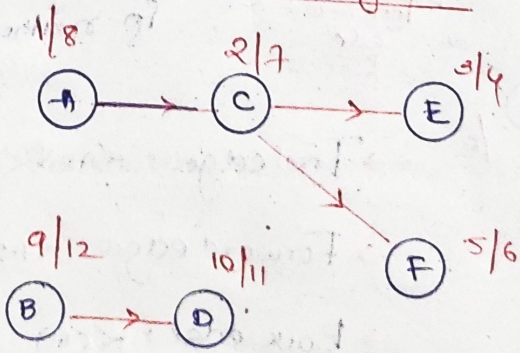
Algorithm Topo (G)

{

1. DFS (V);
2. Arrange all the node of traversal in decreasing order of finishing time.

}

DFS Spanning Forest



⟹ B-D-A-C-F-E.

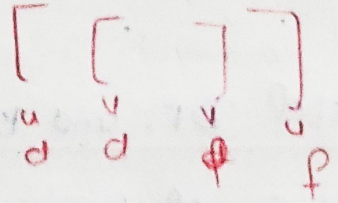
Parenthesization Theorem.

In any Depth-First-Search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v having an edge between them exactly one of the following three conditions holds :.

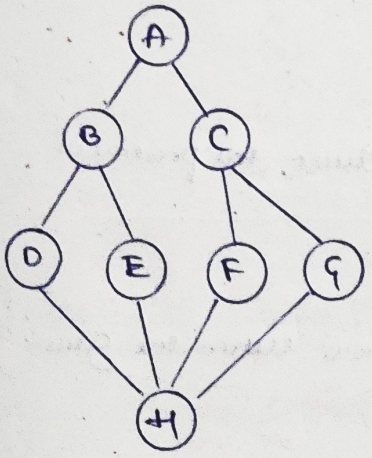
I.) The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u or v is a descendant of the other in the depth first forest. → Cross edge

II.) The interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$ and u is a descendant of v in a depth-first tree, or → Back edge

III.) The interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$ and u is descendant of a u in a depth-first tree. → Forward Tree.

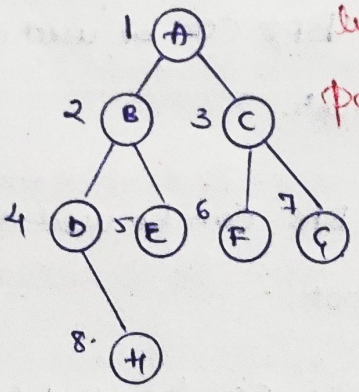


Breadth First Search (BFS)



$G = (V, E)$

FIFO BFS : ABCDEFGH

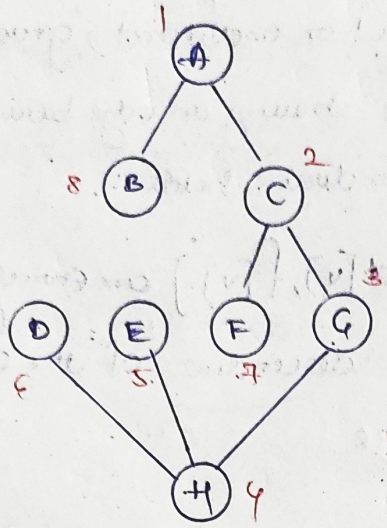


Vertex	B	C	D	E	F	G	H
Parent	A	A	B	B	C	C	D

* Single Source Shortest Paths.

{ BFS-Spanning Tree }

LIFO BFS : ACGHEDFB.



Line	B	E	F	G	H	D	E	F
Parent	A	A	C	G	G	H	H	H

Q1). Consider an undirected graph (unweighted). If BFS of G is done from a node 'r' let $d(r,u)$ and $d(r,v)$ be the lengths of the shortest paths from r to u. If 'u' is visited before 'v', during the traversal then which is true?

→ $d(r,u) \leq d(r,v)$

Applications of DFS and BFS

1. Time complexity of DFS and BFS depends upon representation of Graphs:
   (i) Adjacency matrix: $O(n^2)$
   (ii) Adjacency list: $O(n+e)$

2. Both DFS and BFS can be used to detect the presence of cycle in the Graph.

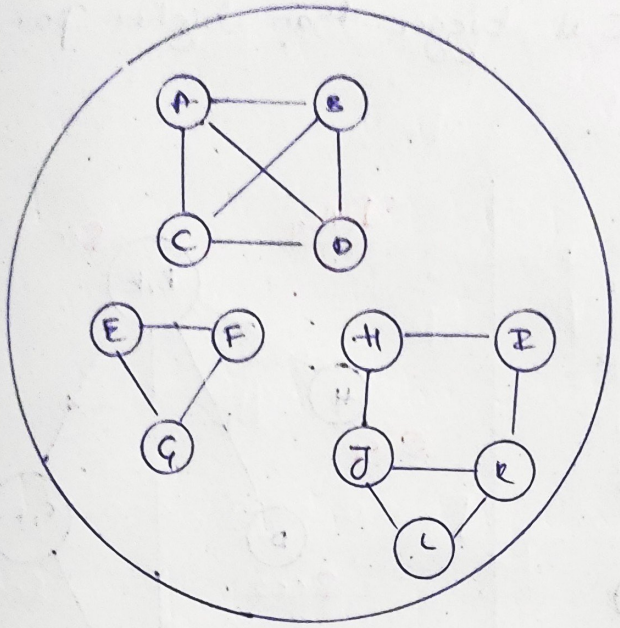
3. Both DFS and BFS can be used to know when the given graph is connected or not.

4. Both DFS and BFS can be used to know weather the two vertices u and v are connected or not.

5. DFS is used to determine connected, strongly connected, biconnected components and articulation points.

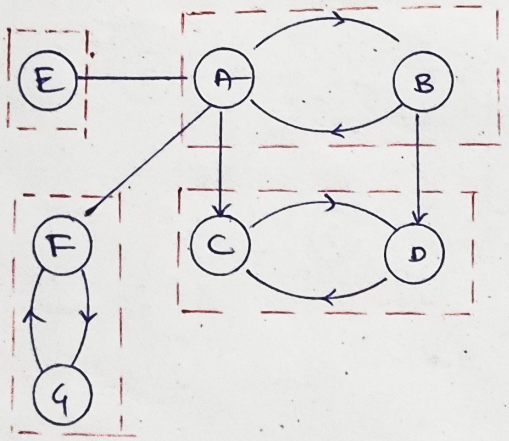
Connected Components : {Undirected Graphs}

↳ Maximal Subgraph that is connected



Strongly Connected Components {Directed Graphs}

Two nodes 'u' and 'v' of a directed graph are connected, if there is a path from 'u' to 'v' and a path from 'v' to 'u'. This relation partitions the vertex of 'V' into disjoint sets known as Strongly Connected components. (Maximal)



⇒ 4 Strongly Connected Components

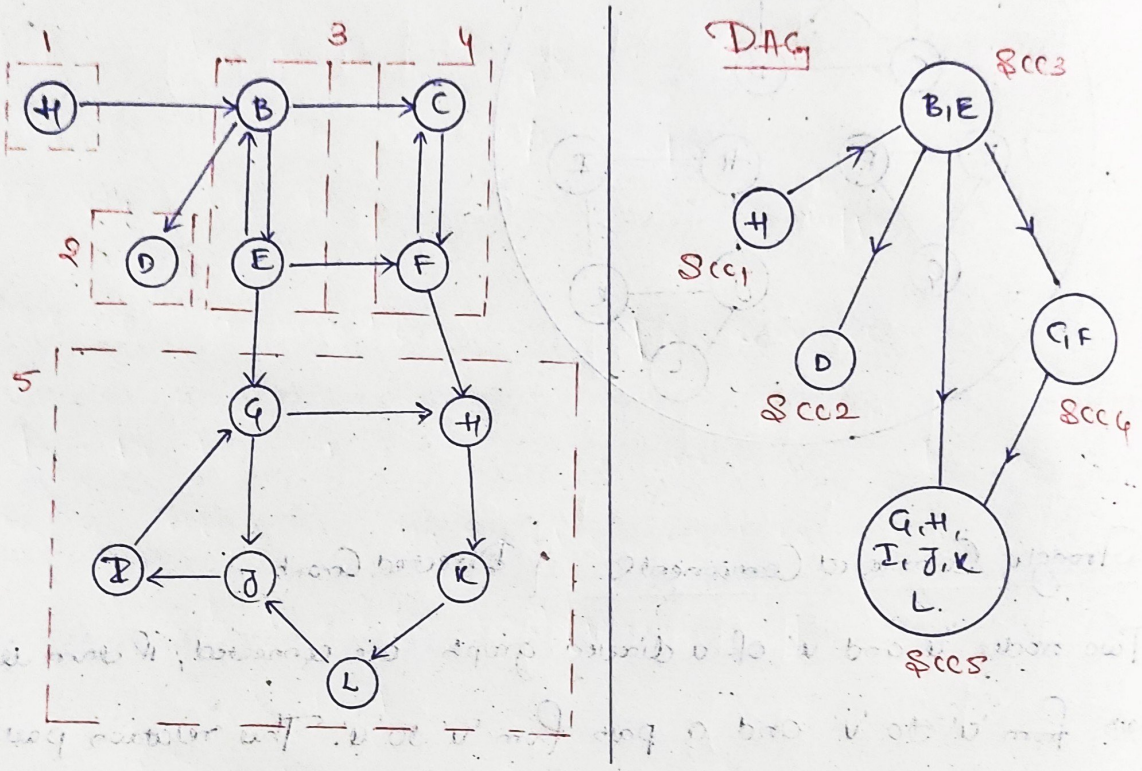
{A,B} {C,D} {F,G} {E}

Properties of Strongly Connected Components

1.) Every directed graph is a D.A.G of strongly connected components

2.) Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$ and $u', v' \in C'$, Suppose that there is a path $u \rightsquigarrow u'$ in G, then there cannot be a path $v' \rightsquigarrow v$ in G.

3.> If 'c' and 'C' are strongly connected components of, and there is an edge from a node in C to a node in C', then the highest-post number in C is bigger than highest post number in C'.



Q2> DFS is performed on a directed acyclic graph. D(u) is discovery time and f(u) is finishing time. Which is true for all edges (u,v) in the graph?

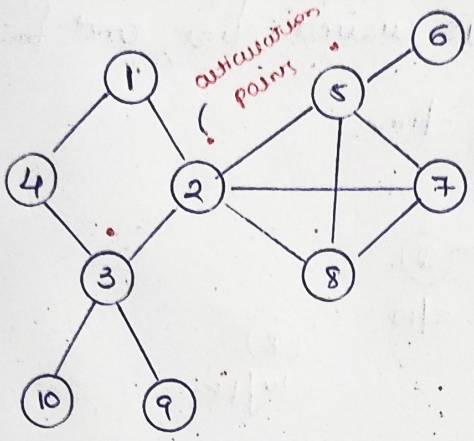
→ f[u] > f[v]

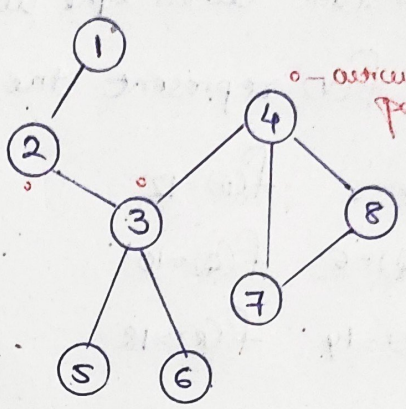
Articulation Points (Cut vertex) and Biconnected Components

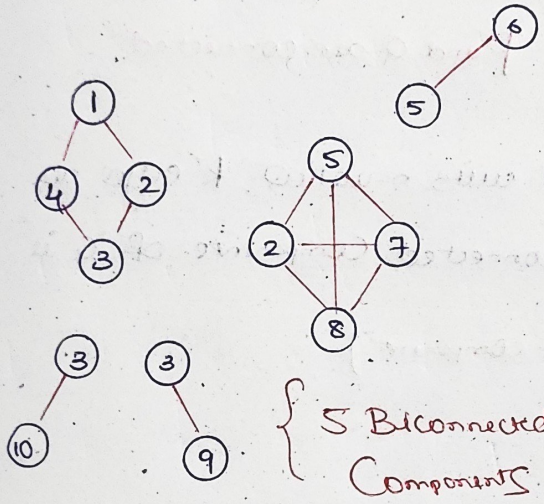
* Articulation point: it is that vertex in the graph, the removal of which along with all its edges, partitions the graph into 2 or more non-empty components

* A graph is said to be "bi-connected" if it does not contain any articulation point.

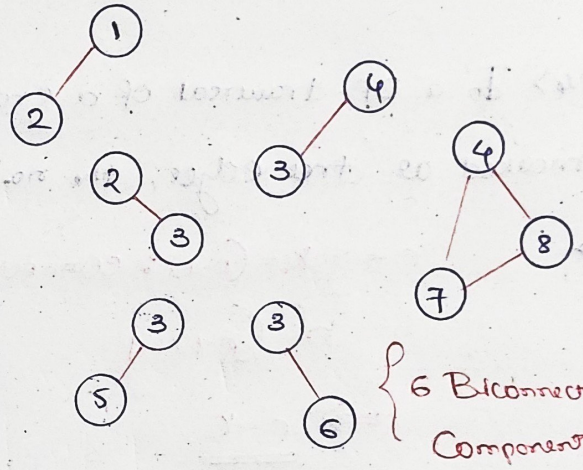
* If the graph is not bi-connected than its maximal subgraph which is biconnected is "bi-connected component".



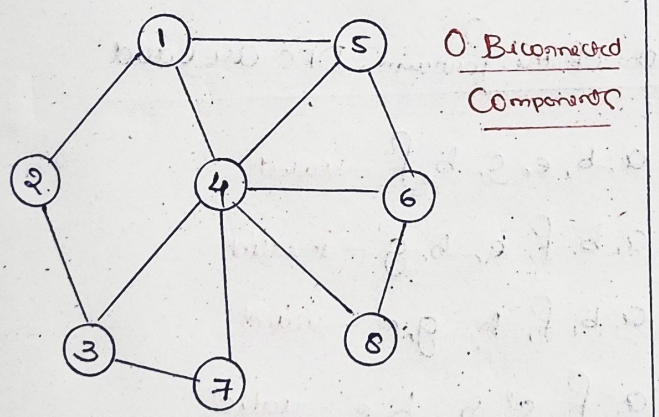




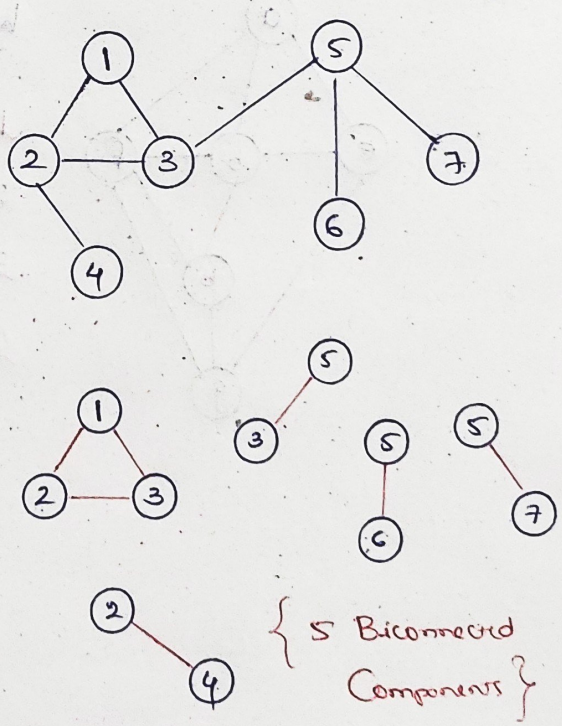
{ 5 Biconnected Components }



{ 6 Biconnected Components }



0 Biconnected Components



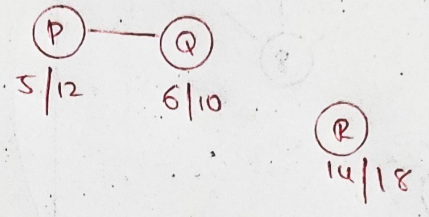
{ 5 Biconnected Components }

Q3> Consider the depth first search of an undirected graph with 3-vertices, P, Q and R. Let discovery time d(v) represent the time instant when the vertex 'v' is visited first, and finish time f(v) represent the finishing time.

$d(P)=5 \quad f(P)=12$

$d(Q)=6 \quad f(Q)=10$

$d(R)=14 \quad f(R)=18$



→ There are 2-connected components, P and Q are connected.

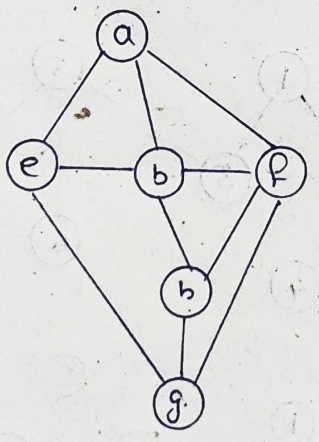
Q4> In a DF-traversal of a graph 'G' with n-vertices, 'k' edges are marked as tree edges, the no. of connected components of 'G' is

→ $n \longrightarrow k=(n-1)$: connected [1-component]

$n-(n-1)=1$

$\Rightarrow n-k$

Q5>



Which of the following DFS are valid

1. a, b, e, g, h, f – valid
2. a, b, f, e, h, g – invalid
3. a, b, f, h, g, e – valid
4. a, f, g, h, b, e. – valid

Sorting Techniques

22/8/2022

Classification of Sorting Techniques

(i) Internal and external sort

(ii) Comparision and non-comparision based
(Radix Sort)

(iii) Recursive and Iterative.

(iv) Inplace and not in-place
(Space required is generally O(1) or O(log n) atmost, for recursion stack)
(merge sort - O(n) (stack/space required)

(v) Stable v/s Unstable. — {quicksort, heapsort}
(relative order of same-elements are maintained)

Let A[1...n] be an array of n-elements

Let i, j be two indices (i ≠ j)

if (A[i] == A[j]) and if A[i] precedes A[j] in the pre-sorted array then the sorting method is stable if A[i] precedes A[j] in sorted list also.

	1	2	3	4	5	6
Arr	10	8	6	8	2	3

Sorted Array

2	3	6	8	8	10

(Stable)

Parameters that influence the time complexity of comparision based sort

Time-Complexity (comparison based sort) = f (no. of comparisons, no of swaps)

(Traversion of an array.)

Merge Sort
→ O(n log n)
Swaps — Comparisons

as a size of input (n)
f(n², n log n) Time = O(n²)

Inversion of an Array

→ Let $A[1 \ldots n]$ be an array, 'i' and 'j' be indices $(i \neq j)$ if $(i<j$ and $A[i]>A[j])$ then the pair $\langle i,j \rangle$ is known as Inversion of the array.

	1	2	3	4	5	6
Arr:	8	9	4	5	1	2

Swappings:

$\langle 1,3 \rangle \langle 1,4 \rangle \langle 1,5 \rangle \langle 1,6 \rangle = 4$

$\langle 2,3 \rangle \langle 2,4 \rangle \langle 2,5 \rangle \langle 2,6 \rangle = 4$

$\langle 3,5 \rangle \langle 3,6 \rangle = 2$

$\langle 4,5 \rangle \langle 4,6 \rangle = 2$

12

Q1) Given an array of size 'n' elements.

(a) What will be lower-bound (minimum) no of inversions

→ '0'

(b) What is the upper bound (maximum) no. of inversions.

→ for n elements: $\frac{n(n-1)}{2} = O(n^2)$

Algorithm BUBBLE_SORT (A, n)

```
{ for pass=n to 1.        → for "optimization"
    int flag;
  { for i←1 to (pass-1)
      flag = 0;
    { if (A[i] > A[i+1])   → inversion
        { swap (A[i], A[i+1]);
          flag = 1; }
    }
    if (flag == 0) break;
    (in first loop)
}
```

Time Complexity

	Comparisons	Swaps
Increasing	$n(n-1)/2$	0
Decreasing	$n(n+1)/2$	$\frac{n(n-1)}{2}$

Time Complexity = $O(n^2)$

Optimised Bubble Sort (flag)

(i) Best Case : $O(n)$ - elements are in decreasing order.

(ii) Worst Case : $O(n^2)$ - decreasing order

→ Let $T(n)$ represents the time complexity of (worst case)

Bubble Sort : $\boxed{T(n) = n + T(n-1)} \Rightarrow O(n^2)$

Best Case : $\boxed{T(n) = n-1+c} \Rightarrow O(n)$

Space Complexity = $O(1)$. } Inplace ✓ Stable ✓

Selection Sort.

1. Find the smallest element in the list.
2. Swap it with the value in the current its position.
3. Repeat this process for all elements until the array is sorted.

Time Complexity

Comparisions: $\frac{n(n-1)}{2}$

Swaps: 'n' (zero or more)

Overall = $O(n^2)$

Algorithm SELECTIONSORT (A, n)

```
{ for int i ← 1 to n-1 do
  { min ← i;
    for j ← i+1 to n
    { if (A[j] < A[min]) then
        { min ← j }
    }
    swap (A[min], A[i]);
  }
}
```

Space Complexity: $O(1)$ } Inplace ✓ Stable ✓

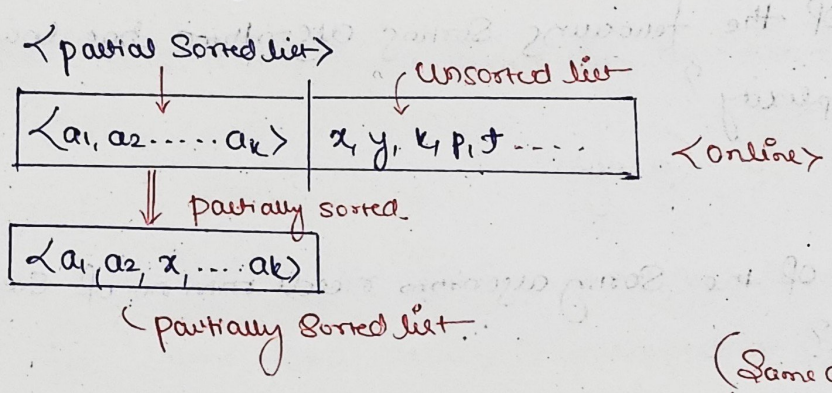
* Selection sort is the only sorting technique in comparision based sorting that requires least no. of swaps ($O(n)$) in the worst case..

23/8/2023

Sorting Techniques {Continued}

Insertion Sort

* Effective for small data sets. ($O \leftrightarrow n^2$)
* If the input list is pre-sorted, then it takes time of $O(n+d)$
  d = no. of inversions.
* It is on-line.
* Each pass of insertion sort removes an element from the input data set, inserts it into the correct position in the already sorted list.
* It does not require all the elements to be sorted to be available at the time of sorting, unlike other sorting techniques.



Algorithm INSERTION_SORT (A, n)

```
{  for j ← 2 to n
   { key ← A[j];
     i ← j-1
   while (i>0 and A[i]>key)
   { A[i+1] ← A[i];
     i ← i-1; }
   A[i+1] ← key;
} }
```

Time Complexity (Same as Optimized Bubble Sort, further no need of flag)

1. Best Case : $O(n)$ – Increasing order of elements
   0-swaps, 0-Comparisons.
2. Worst Case : $n^2 + n^2 = O(n^2)$ – decreasing order

Radix Sort (non-comparision based sorting).

Base : <r=10>. ← each number is stored in buckets of numbers 0-9 ← for base 10

	1	2	3	4	5	6	7	8	9	10
Arr.	723	64	99	83	124	4	7	65	333	545
Pass 1:	723	83	333	64	124 ~~333~~	4	65	545	7	99
Pass 2:	4	7	723	124	333	545	64	65	83	99
Pass 3:	4	7	64	65	83	99	124	333	545	723

Time Complexity : $O(d*(n+b)) \sim O(d \times n)$

b = base

d = no of bits / digits of largest number

Q1> Which of the following sorting algorithms has lowest worst case complexity?

→ Merge sort - $(n \log n)$

Q2> Which of the sorting algorithms needs min no. of swaps and is in-place?

→ Selection Sort.

Q3> What would be the worst case complexity of insertion sort if the inputs are restricted to permutations of 1 to n with atmost 'n' inversions?

→ Time : $O(n+d)$ d = no of inversions

$d = n$

$O(n+n) = O(n)$

Q4> Let 'S' be the sorted array of 'n' integers and T(n) denote the time taken for most efficient algorithm to determine if there are 2 elements in the array with sum <1000. → Since it is sorted array, we can just compare

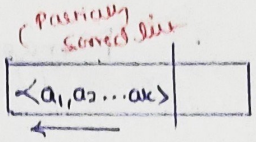
Find 2 or last 2 elements whose sum = 1000; ∴ T(n) is O(1)

Q5> The traditional insertion sort to sort an array can be of n elements uses linear search to identify the position where an element is to be inserted into already sorted part of the array, if instead binary search is used to identify the position of newly inserted element then worst case complexity will be order of O(n²)

W.C = $n^2 + n^2 = O(n^2)$

W.C = $n \log n \cdot n^2$

$= O(n^3)$



If worst case no. of comparisions asked the answer will be O(n logn)

Q6> The worst case running times of insertion sort, merge sort, and Quicksort respectively are:

→ $O(n^2)$, $O(n \log n)$, and $O(n^2)$

Q7> You have n lists, each consisting of integers sorted in ascending order merging these lists into a single sorted list will take time

→

Q8> If we use Radix sort to sort n integers in the range $(n^{k/2}, n^k)$ for some $k>0$ which is independent of n, the time taken would be O(n log n)

O(dkn) $\log n^k$

$d = k \log n$

∴ O(n x log n)

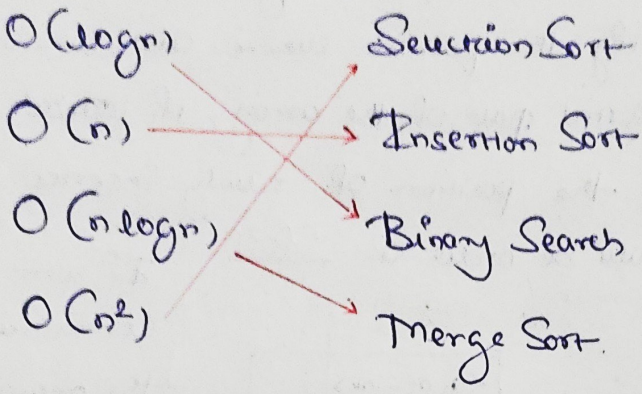
Q9> Following algorithm (s) can be used to sort n integers in the range $[1 \ldots n^3]$ in O(n)

→ Radix Sort

Q10> For merging two sorted lists of sizes m and n into a sorted list of size m+n, we require comparisions of

→ O(m+n)

Q11) Match the following



Q12) Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order which of the following is True?

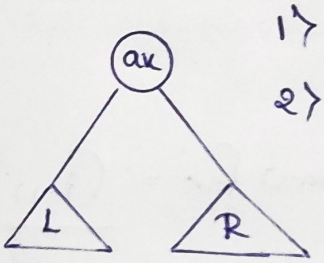
I. Quicksort runs in $\Theta(n^2)$ time
II. Bubble sort runs in $\Theta(n^2)$ time
III. Merge sort runs in $\Theta(n)$ time
IV. Insertion sort runs in $\Theta(n)$ time.

{ I and IV are true }

Heap Algorithms

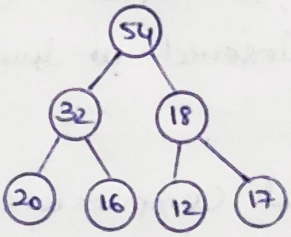
24/8/2023

Definition: A "heap" is a complete binary tree with the property that the value at each node is atleast as large as the values at its children (if they exist).

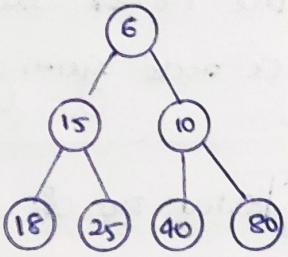


1) $ak > |L, R|$ — Max Heap

2) $ak < |L, R|$ — Min-Heap.



Max Heap



Min Heap

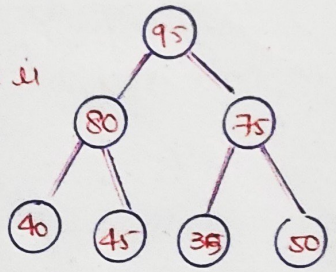
3) $|L| < ak |R|$

Binary Search Tree.
— (need not be complete)

Heap Construction:

(I.) **Insertion Method:** — Insert one element at a time starting from an empty tree.

eg: <40, 80, 35, 95, 45, 50, 75>



each node is added & compared, swapped if it doesn't follow the property.

Best Case: decreasing order: $O(n)$

Worst Case: increasing order: $O(n \log n)$

```
Procedure INSERT (A, n)
integer i, j, n;
j ← n; i ← ⌊n/2⌋ item ← A(n)
while (i > 0 and A(i) < item) do
    A(j) ← A(i) // move parent down.
    j ← i; i ← ⌊i/2⌋
repeat
A(j) ← item; // a place for A(n) is found
End INSERT
```

```
for i ← 2 to n do
    Call INSERT (A, i)
repeat;
```

Time Complexity:

→ Max no of nodes at level 'i' = 2^{i-1}.
of a binary tree.

→ The no. of level comparisions (movements) for a node getting inserted at level 'i' = $(i-1)$

→ Total no. of level comparisions for (max) nodes at level 'i' = $(i-1) \cdot 2^{i-1}$

→ Time = $T(n)$ = No of comparisions / movements for all nodes at all levels $(1 \ldots k)$ = $\sum_{i=1}^{k} (i-1) \cdot 2^{i-1}$

$$\sum_{i=1}^{k} (i-1)\, 2^{i-1} = \frac{1}{2}\left[\sum_{i=1}^{k} i \cdot 2^i - \sum_{i=1}^{k} 2^i \right]$$

$$\Rightarrow \frac{1}{2}\left[(k-1)2^{k+1} + 2 - \left(2^{k+1} - 2\right) \right]$$

$$\Rightarrow \frac{1}{2}\left[k \cdot 2^{k+1} - 2^{k+1} + 2 - 2^{k+1} + 2 \right]$$

$$\Rightarrow k \cdot 2^k - 2 \cdot 2^k + 2$$

$$\Rightarrow n \log n - 2n + 2$$

Let $n = 2^k$

$k = \log n$.

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

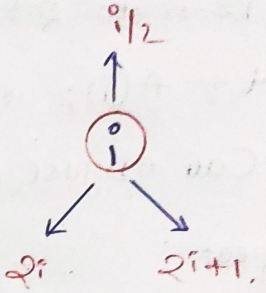
II Insert Operation:

Given a heap with n-elements, the time complexity to insert an element into it is $O(\log n)$

III Build Heap / Heapify

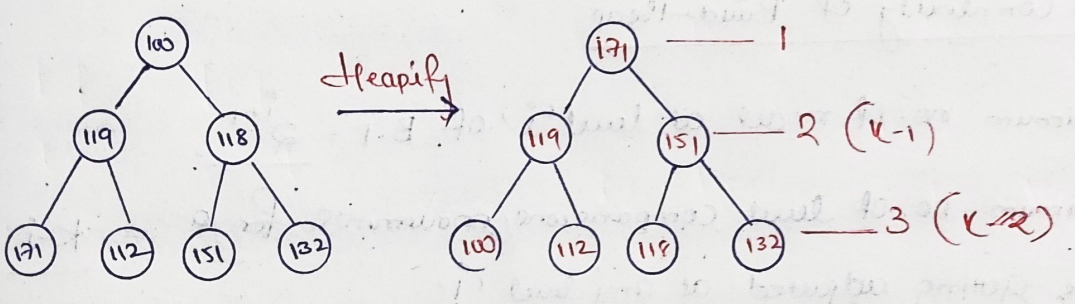
→ Tree already exists

→ Here by here we can adjust the nodes.



A:

1	2	3	4	5	6	7
100	119	118	171	112	151	132



Procedure ADJUST (A, i, n)

Integer i, j, n;

j ← 2*i, item ← A(i)

while (j ≤ n) do

if (j ≤ n and A(j) < A(j+1)) then

j ← j+1 // j points to larger child

endif

if (item ≥ A(j)) then

exit // a position for item is found

else

A(⌊j/2⌋) ← A(j)

j ← 2*j // move the larger child up a level.

endif

repeat

A(⌊j/2⌋) ← item

end ADJUST

```
Procedure HEAPSORT (A,n)
// A(1:n) Contains Elements to be Sorted
Call HEAPIFY (A,n)
for i ← n to 2 by -1 do
    t ← A(i); A(i) ← A(1); A(1) ← t
    Call ADJUST (A, 1, i-1)
repeat
end HEAPSORT
```

Time Complexity of Build-Heap

→ Maximum no. of nodes at level 'i' of B.T = 2^{i-1}

→ Maximum no. of level comparisions/movements for a node getting adjusted at any level 'i' = $k-i$

→ For all nodes (max) no. of comparisions = $(k-i).2^{i-1}$

→ Total time = Sum of all level comparisions for all nodes at all ~~levels~~ nodes at all levels $(1 \ldots k) = T(n) = \sum_{i=1}^{k} (k-i).2^{i-1}$

$$T(n) = \sum_{i=1}^{k} (k-i)\, 2^{i-1}$$

$$= \frac{1}{2}\left[\sum_{i=1}^{k} k.2^{i} - \sum_{i=1}^{k} i.2^{i}\right] = \frac{1}{2}\left[k(2^{k+1}-2) - \left((k-1)2^{k+1}+2\right)\right]$$

$$= \frac{1}{2}\left[k.2^{k+1} - 2k - k.2^{k+1} + 2^{k+1} - 2\right]$$

$$= 2^{k} - k - 1$$

$n = 2^k$

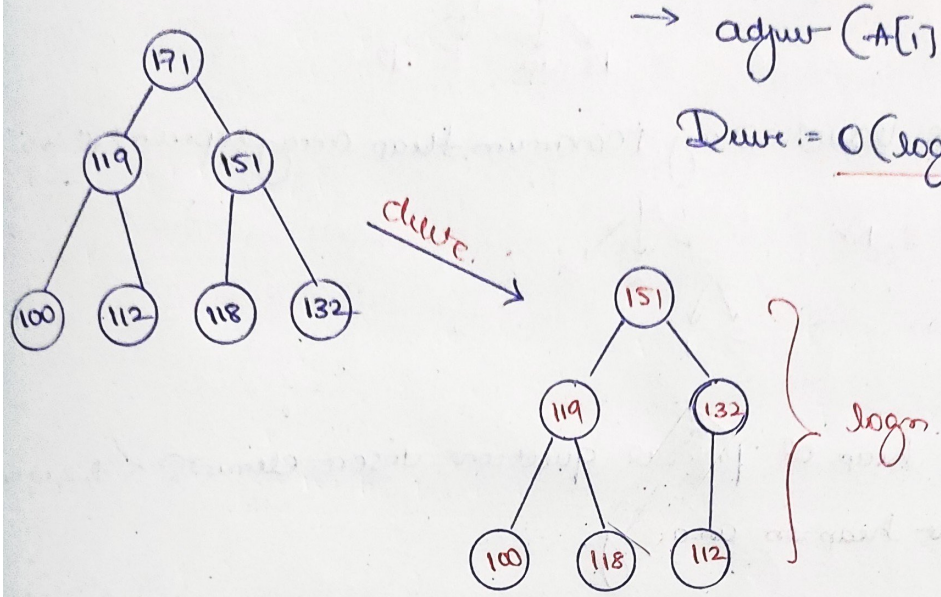
$$\Rightarrow T(n) = \boxed{n - \log n - 1} = O(n)$$

Delete - operation in a Heap : deleting the root.

→ Swap Root (A[1], A[n]) } - 1

→ adjust (A[1]) } - logn

Time = O(logn)



Heap Sort.

Procedure HEAPSORT (A, n)

// A(1...n) contains elements to be sorted.

Call HEAPIFY (A, n)

for i ← n to 2 by -1 do

t ← A(i); A(i) ← A(1); A(1) ← t

Call Adjust (A, 1, i-1);

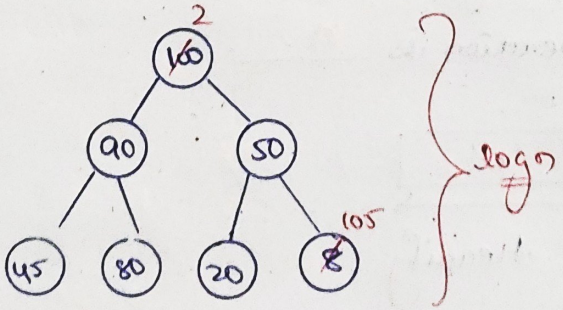
repeat.

End HEAPSORT

Time Complexity

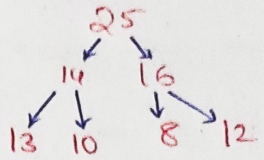
= O(n.logn)

→ Given a heap with n-elements, the time complexity to perform the operation of inc/dec key is O(logn)



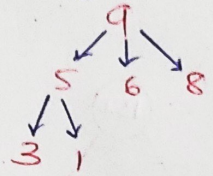
Q1> Which array representation is valid binary max-heap?

→ <25, 14, 16, 13, 10, 8, 12>



Q2> Which one is valid - 3-ary maximum heap array representation

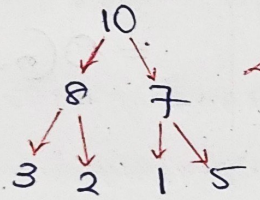
→ <9, 5, 6, 8, 3, 1>



Q3> To the valid heap of previous question insert elements <7, 2, 10, 14> Indicate the resultant heap in an array.

Q4> Level order traversal of a binary max heap generated <10, 8, 5, 3, 2>. To this heap Insert <1, 7>. What is the resultant level order traversal.

→

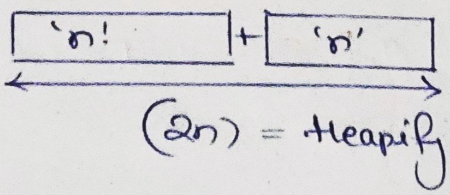


<10, 8, 7, 3, 2, 1, 5>

Q5> In a Binary Max-Heap with n elements, Smallest element can be found in the time of O(n)

→ Convert max heap to min heap ⇒ O(n)

Q6> Given binary heap with 'n' elements and it is required to insert n more elements not necessarily one after another into this heap. Total time for this operation is n.



Q7> Given Binary Heap in array with the smallest at the root, the 7th smallest element can be found in the time complexity of. _____

→ "Min-Heap"

Traditional approach

- 7 delete operations.

O(log n)

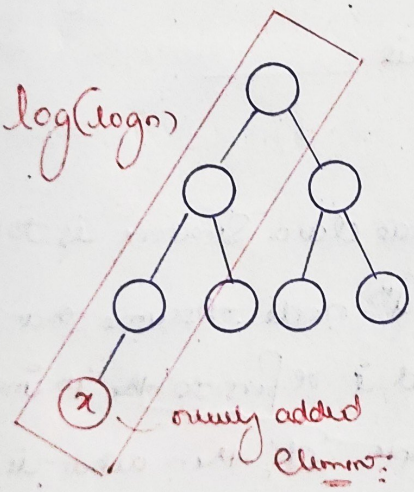
Non traditional approach

$(2^7 - 1)$ — because 7th smallest cannot lie beyond 7th level.

= O(1)

∴ our search will be limited (constant)

Q8> Consider a binary heap in an array with n elements. It is desired to insert an element into the heap. If a binary search is performed along the path from the newly inserted element to the root then no. of comparisions made is order of log log n



⟹ log(log n)

Q9> the approximate no. of elements that can be sorted in O(log n) time using heap sort is $O\left(\frac{\log}{\log \log n}\right)$

with n elements → n log n TC

* Should be less than log n.

with n 'log n' elements

Time = [log n × log log n]

* no. of elements with be < 0

No of elements = 1

Time = O(1)

Proof → P.T.O

Element → n → $n^x \log n$

$\log \rightarrow \frac{\log}{\log\log n} \Longrightarrow \left[\frac{\log n}{\log\log n} \times \log\left(\frac{\log n}{\log\log n}\right)\right]$

$$\frac{\log n}{\log\log n}\left(\log n \log n - \log\log\log n\right)$$

$$\log - \left(\frac{\log n}{\log\log n} \times \log\log\log n\right)$$

$= O(\log n)$ ∴ Hence proved!

HW

Q10> Given $[\log n]$ sorted lists each having $\lfloor n/\log n \rfloor$ elements. The time complexity to merge the given list into single sorted list, using heap data structure is _____.

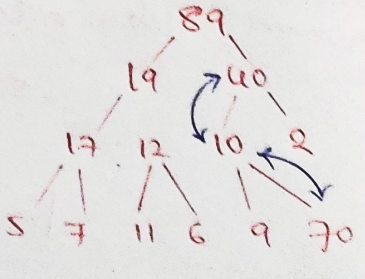
Q11> An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i^{th}-node. Assume that the heap is implemented in an array and i refers to the i^{th} index of the array. If the heap tree has depth 'd', then what is the time complexity to re-fix the heap efficiency after the removal of the element?

→ O(d) but not O(1)

Q12> The minimum number of interchanges needed to convert the array into max-heap is

<89, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70>

→ 2



Q13> An array of integers of size n can be converted into heap by adjusting the heaps rooted at each internal node of the complete binary tree starting at the node $\lfloor (n-1)/2 \rfloor$ and doing this adjustment upto the root node (root node is at index 0) in the order $\lfloor (n-1)/2 \rfloor$, $\lfloor (n-3)/2 \rfloor$ .... 0. The time required to construct a heap in this manner is ___O(n)___ => Build Heap / Heapify — O(n).

Q14> An array X of n distinct integers is interpreted as a complete binary tree. The index of the first element of the array is 0. If only the root node does not satisfy the heap property, the algorithm to convert the complete binary tree into a heap has the best asymptotic time complexity of. ___O(log n)___

Q15> Consider a complete binary tree where the left and right subtrees of the root are max-heaps. The lower bound for the no. of operations to convert the ~~heap~~ tree to a heap is. ___Ω(log n)___

Q16> Consider a max-heap represented by the array:

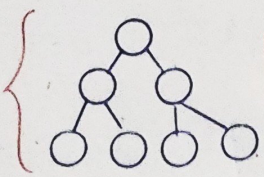
< 40, 30, 20, 10, 15, 16, 17, 8, 4>

1 2 3 4 5 6 7 8 9.

Assume that a value is inserted into heap, after this new heap is:

→ <40, 35, 20, 10, 30, 16, 17, 8, 4, 15>

Q17> The minimum possible min-heaps containing each value from {1, 2, 3, 4, 5, 6, 7} exactly once is ___80___



$[{}^{6}C_3 \times 2! \times 2!] = 80$

$T(n) = {}^{(n-1)}C_k \times T(k) \times T(n-k-1)$

k = no. of nodes in left subtree

$T(7) = {}^{6}C_3 \cdot T(3) \cdot T(3)$

Q18) Consider the following Statements:

I. The smallest element in a max heap is always at leaf node.

II. The second largest element in a max-heap will always a child of root node.

III. A max heap can be constructed from binary search tree in $O(n)$ time.

IV. A binary search tree can be constructed from a max-heap in $O(n)$ time.

Which of the above Statements are true?

→ I, II and III are true.

Q19) Let H be a binary min heap consisting of n-elements implemented as an array. What is the worst case time complexity of an optimal algorithm to find the maximum element in H?

→ (c) $O(n)$

25/8/2023

Recursion tree method for solving D&C recurrences

(i) $T(n) = a \cdot T(n/b) + f(n)$ — Back Substitution / Master method / Recursion tree.

(ii) $T(n) = T(\alpha n) + T((1-\alpha)n) + f(n)$ — Recursion tree

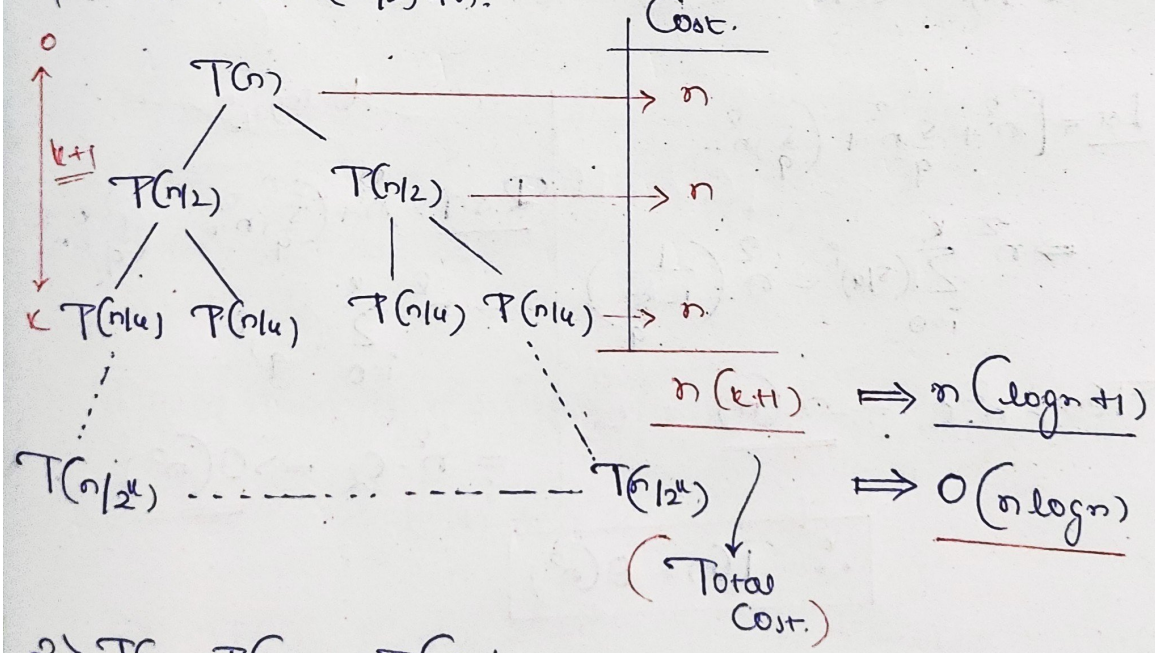
(iii) $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + f(n)$ — Recursion tree

Note:

Back Substitution — value / order.
Master method — order
Recursion tree — order.

} What we get using each method of solving recurrence.

1.) $T(n) = 2 \cdot T(n/2) + n$.

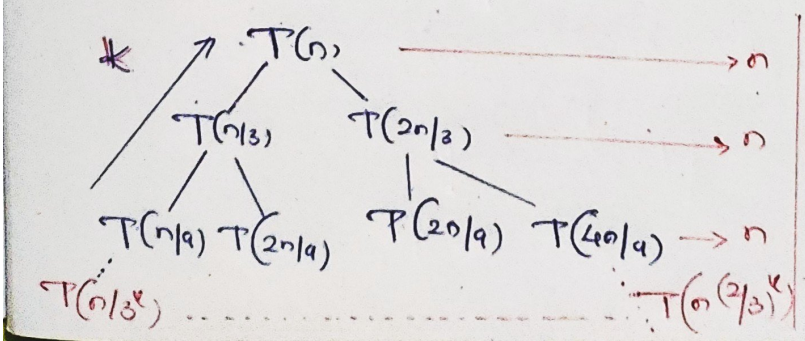


$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2 n$

2.) $T(n) = T(n/3) + T(2n/3) + n$.



$T(n) \geq n \times k \longleftrightarrow T(n(2/3)^k)$

$\geq n \cdot \log_3 n$

$\Omega(n \log_3 n)$

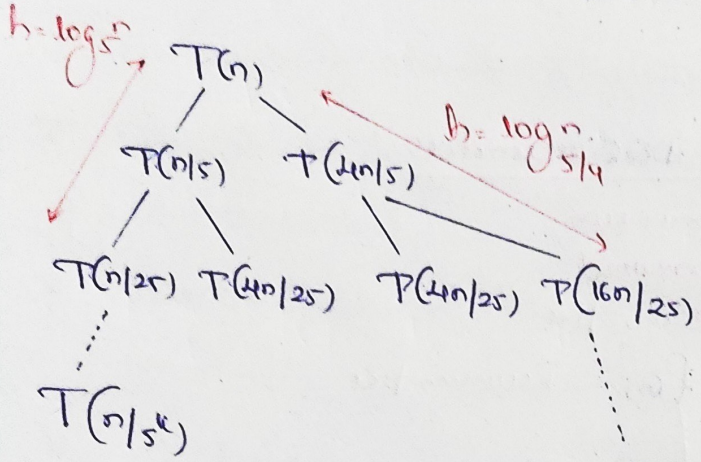
$T(n) \leq n \times k$

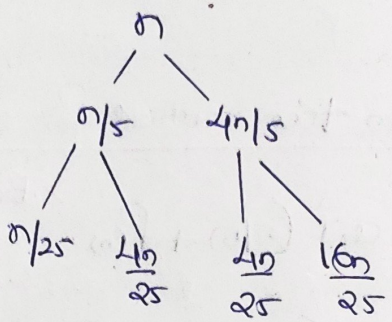
$\leq n \times \log_{3/2} n$

$O(n \log_3 n)$

$\Rightarrow \Theta(n \log n)$

3) $T(n) = T(n/5) + T(4n/5) + n$





$T(n) \geq n \log_5 n$

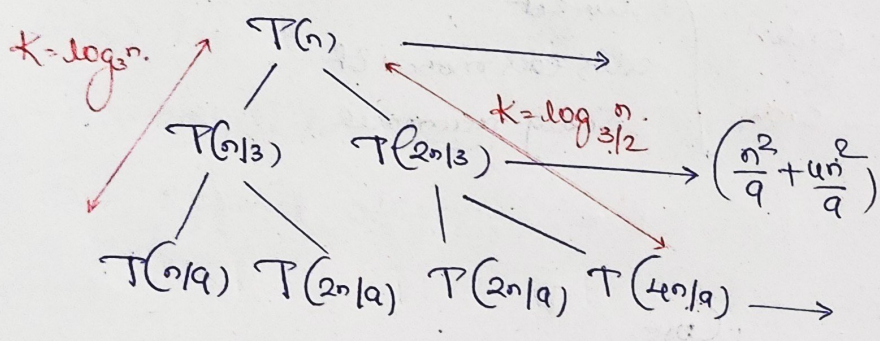
$\Omega(n \log n)$

$T(n) \leq n \log_{5/4} n$

$O(n \log n)$

$\Rightarrow \Theta(n \log n)$

4) $T(n) = T(n/3) + T(2n/3) + n^2$



$S_n = \frac{a(1-r^n)}{1-r}$

$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad x<1$

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

$\Rightarrow n^2 \sum_{i=0}^{k} (5/9)^i \sim n^2 \left(\frac{1}{1-\frac{5}{9}}\right)$

$T(n) \geq n^2 \cdot c = \Omega(n^2)$

$k = \log_{3/2} n$

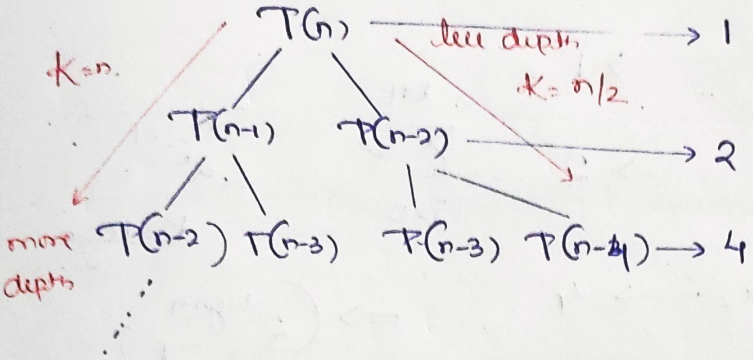
$T.S.T = n^2 + \left(\frac{5}{9}\right) n^2 + \cdots \left(\frac{5}{9}\right)^k \cdot n^2$

$= n^2 \sum_{i=0}^{k} \left(\frac{5}{9}\right)^i$

$= n^2 \cdot c_2 \Rightarrow O(n^2)$

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

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



$$S_n = \frac{a(r^n - 1)}{r-1}$$

$$= \frac{1(2^{n+1} - 1)}{2-1}$$

$$= \underline{2^{n+1}}$$

L.S.T $= \sum_{i=0}^{n} 2^i = (2^{n+1} - 1)$

$T(n) \leq 2^{n+1} - 1$

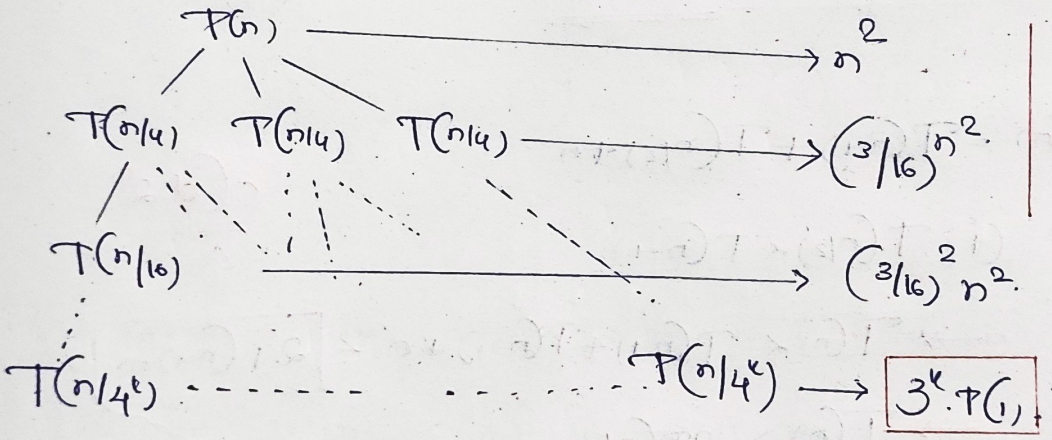
$= \underline{O(2^n)}$

R.S.T $\Rightarrow T(n) \geq \sum_{i=0}^{n/2} 2^i = \frac{1(2^{k+1} - 1)}{2-1}$

$= 2^{n/2+1} - 1.$

$T(n) = \underline{\Omega(2^{n/2})}$

6) $T(n) = 3T(n/4) + n^2.$



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

$k = \log_4 n.$

$$T(n) = \left[n^2 + \left(\frac{3}{16}\right)n^2 + \left(\frac{3}{16}\right)^2 n^2 + \cdots + \left(\frac{3}{16}\right)^{k-1} n^2\right] + 3^k \cdot T(1)$$

$$= n^2 \sum_{i=0}^{k-1} \left(\frac{3}{16}\right)^i + 3^{\log_4 n}.$$

$T(n) = c \cdot n^2 + n^{\log_4 3}.$

$= O(n^2), \Omega(n^2)$

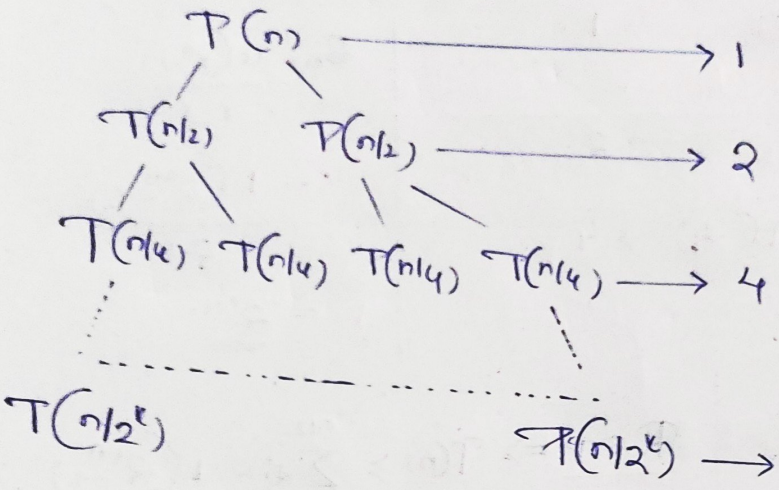
$\therefore \boxed{T(n) = \Theta(n^2)}$

Homework

1) $T(n) = T(n/2) + T(n/3) + T(n/4) + n.$

2) $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + T\left(\frac{2n}{5}\right) + 7n & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$

$T(n) = 2T(n/2) + 1$



$\sum_{i=0}^{k+1} 2^i \qquad \frac{2}{2^k} = 1$

$k = \log n$

$\Rightarrow (2^{\log n + 1} - 1)$

$\Rightarrow O(n)$

7) $T(n) = 2T(n/2 + 17) + n$. → Induction

$= T(n) = 2T(n/2) + n$

↳ $O(n \log n)$

8) $T(n) = T(n-1) + T(n/2) + n$

(i) $T(n/2) < T(n-1)$

$\rightarrow T(n) < T(n-1) + T(n-1) + n < 2.T(n-1) + n$

$O(d \cdot 2^n)$

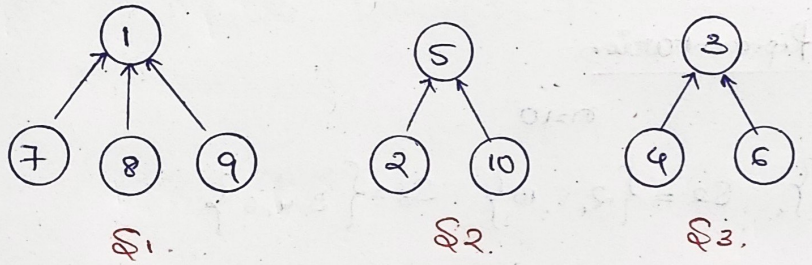
$\rightarrow T(n) > 2T(n/2) + n$

$\Rightarrow n \log n$

Sets

It is assumed that the elements of the sets are the numbers 1, 2, 3 ... n, these numbers might in place, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (that is if S_i, and S_j, $i \neq j$, are two sets, then there is no element that in both S_i and S_j).

For example, when n=10, the elements can be partitioned into three disjoint sets, $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, $S_3 = \{3, 4, 6\}$. Figure our one possible representation for these:

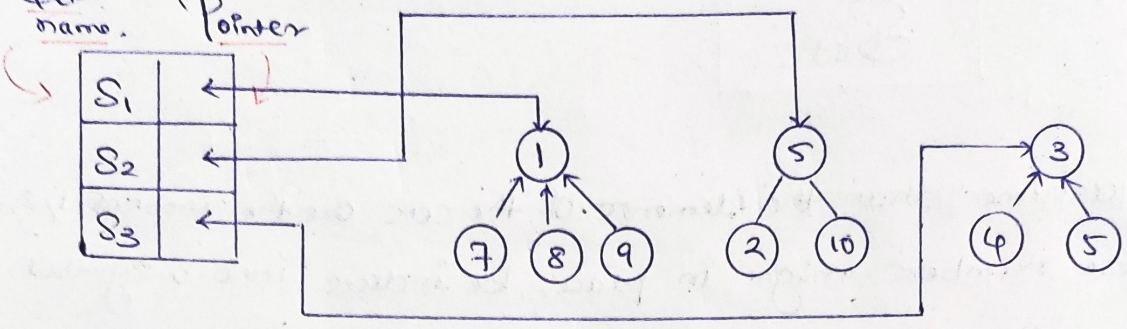


S_1. S_2. S_3.

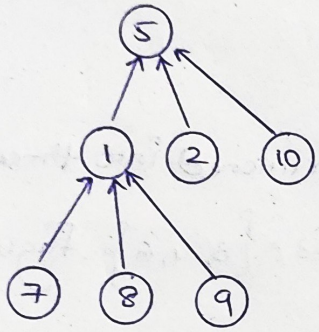
The operations we wish to perform on these sets are:

1. Disjoint Set Union: If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j. Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all the sets are disjoint, we can assume that the following the union of S_i and S_j, the sets S_i and S_j do not exist independently, that is why they are replaced by $S_i \cup S_j$ in the collection of sets.

2. Find (i): Given the element i, find the set containing i. Thus 4 is in set S_3, and 9 is in set S_1.



Data Representation for S_1, S_2 and S_3.



Possible representation of $S_1 \cup S_2$.

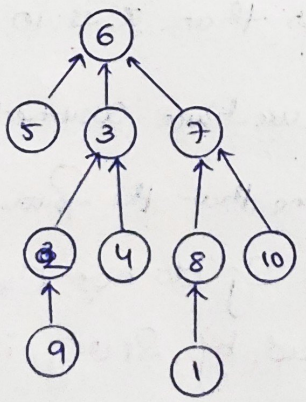
Array based Representation

n=10

$S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$ $S_3 = \{3, 4, 6\}$

i	1	2	3	4	5	6	7	8	9	10
Parent:	-1	5	-1	3	1	3	1	1	1	5

$S_6 = \{1 \ldots . 10\}$



i	1	2	3	4	5	6	7	8	9	10
P.	8	3	6	3	6	-1	6	7	2	7

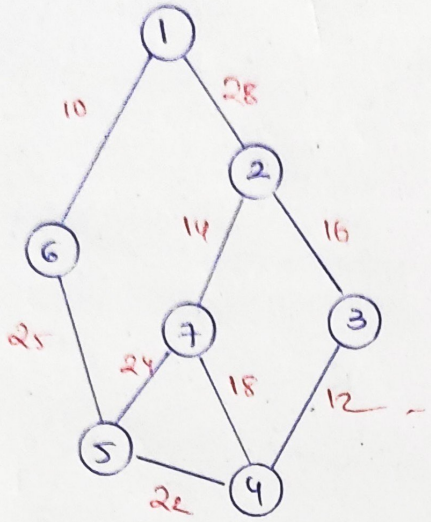
```
Algorithm Union(i, j)
{   p[i] = j; }

Algorithm Find(i)
{   while(p[i] > 0) do
        i = p[i];
    return i;
    }

Algorithm KRUSKAL(E, cost, n, t)
{ // Construct a heap out of the edge costs using heapify.
  for i = 1 to n: do parent[i] = -1;
  i = 0; mincost = 0.0;
  while((i < n-1) and (heap not empty)) do
  { // delete a minimum cost edge (u, v) from the heap and reheapify
    using Adjust.
    j = Find(u), k = Find(v);
    if(j ≠ k) then
    {   i = i+1;
        t[i,1] = u; t[i,2] = v;
        mincost = mincost + cost[u, v];
        Union(j, k); }
    }
  if(i ≠ n-1) then Print("no spanning tree");
  else return mincost;
  }
```

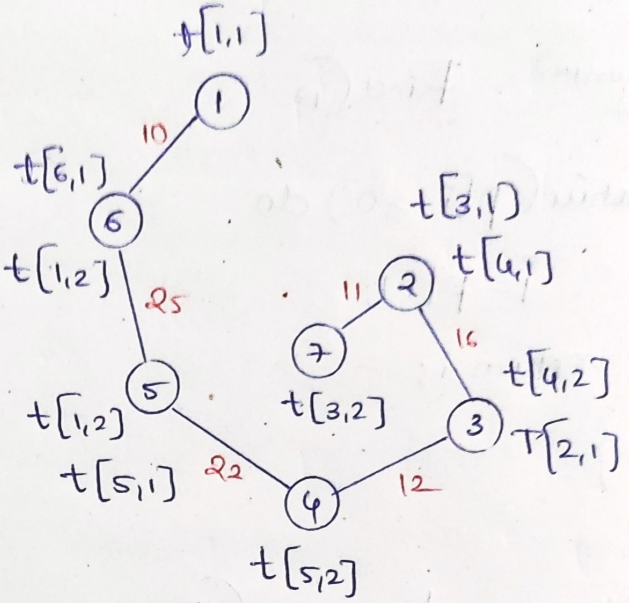
Time Complexity:

$= c + e + n + e \log e$.

$\Rightarrow O(e \log e)$

Kruskal's Algorithm Implementation

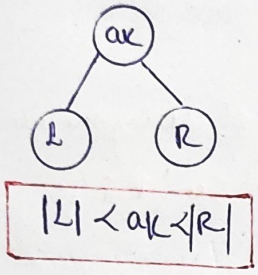


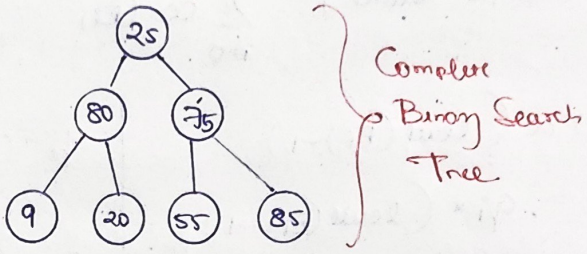


26/8/2023

Optimal cost binary Search tree <OBST>/D.P.

Definition B.ST: It is a binary tree with the property that, the value at each node is greater than values of left children and less than the value of right children.





→ **Keyword Table**

$\langle a_1, a_2, a_3 \rangle = \langle$ do, if, while $\rangle$

$\langle p_1, p_2, p_3 \rangle, \langle q_1, q_2, q_3 \rangle$

n-identifiers

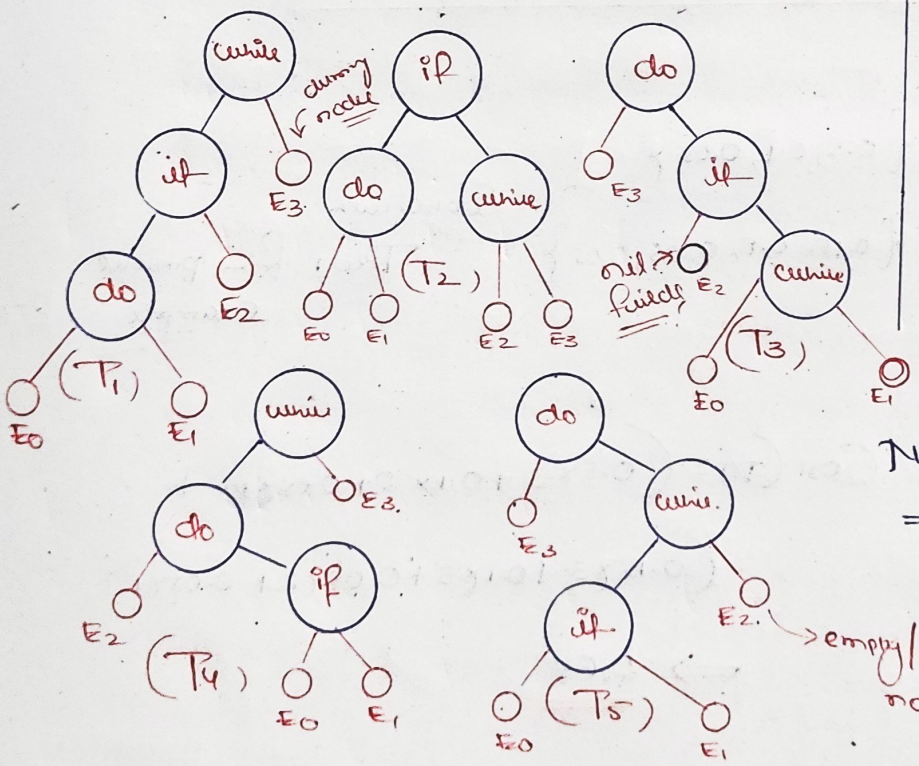
no. of binary trees

$\langle$ Catalan no $\rangle$

$\frac{1}{(n+1)} \cdot {}^{2n}C_n$

$n = 3$

$\Rightarrow 5$



No. of unsuccessful Searches = no of NIL fields = $n+1$

Procedure to determine cost of B.S.T

1.) Cost (T) = cost (Successful Search) + cost (unsuccessful search)

2.) Cost (Successful Search) = $\sum_{i=1}^{n}$ cost (ai)

3) $Cost(a_i) \propto level(a_i)$

$Cost(a_i) = p_i * level(a_i)$

(probability of using a_i in program

$\Rightarrow$ $Cost(\text{Successful Searches}) = \sum_{i=1}^{n} p_i * level(a_i)$

4) $Cost(\text{Unsuccessful Searches}) = \sum_{i=0}^{n} cost(E_i)$

5) $Cost(E_i) \propto (level(E_i) - 1)$

$= q_i * (level(E_i) - 1)$;

q_i = Prob of using the identifiers in the set E_i

6) Total cost of (unsuccessful searches) $= \sum_{i=0}^{n} q_i * (level(E_i) - 1)$

$$Cost(T) = \sum_{i=1}^{n} p_i * level(a_i) + \sum_{i=0}^{n} q_i * (level(E_i) - 1)$$

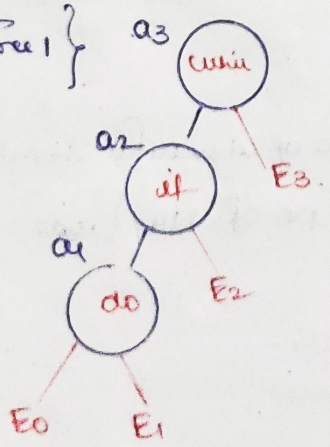
eg: $\langle p_1 \cdots p_3 \rangle = \{0.5, 0.1, 0.05\}$

$\langle q_0 \cdots q_3 \rangle = \{0.15, 0.1, 0.05, 0.05\}$

Solutions

Tree 1 from previous example.

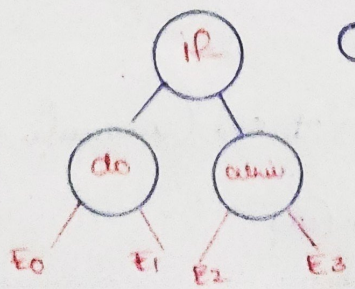
{Tree 1}



$Cost(T) = (0.5 \times 3 + 0.1 \times 2 + 0.05 \times 1) + (0.15 \times 3 + 0.1 \times 3 + 0.05 \times 2 + 0.05 \times 1)$

$\Rightarrow$ 2.65

{Tree 2}



$Cost(T) = ($

Cost Comparisions:

Tree 1 = 2.65

Tree 2 = 1.9

Tree 3 = 1.05 → Tree with minimum cost

Tree 4 = 2.15

Tree 5. = 1.6

Note: The cost of the B.S.T. is dependent on both height (level) and also probabilities.

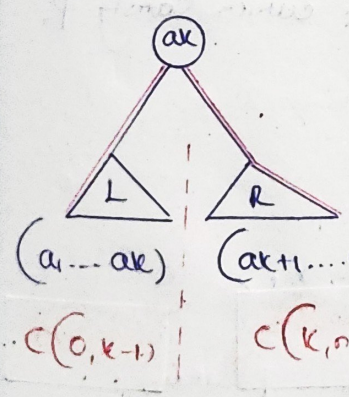
Construction of Optimal Binary Search Tree.

$$I/p \begin{cases} \langle a_1, a_2 \dots a_n \rangle \\ \langle p_1, p_2 \dots p_n \rangle \\ \langle q_0, q_1 \dots q_n \rangle \end{cases}$$

$$\text{Cost}(T) = P_k + \text{Cost}(L) + \text{Cost}(R)$$

Let $c(0,n)$ be the cost of the ~~tree~~ $T_{1 \dots n}$.

$$c(0,n) = \min_{1 \le k \le n} \{ P_k + c(0, k-1) + c(k, n) \} + w(0,n)$$



$w(0,n)$ = weight that is added to LST and RST to balance it with respect to root.

$$c(i,j) = \min_{i < k \le j} \{ P_k + c(i, k-1) + c(k, j) \} + w(i,j) \quad \rightarrow (1)$$

$$c(i,i) = 0$$

$R(i,j) = k$ that minimizes the eqn (1)

Time Complexity: $O(n^3)$

Space Complexity: $O(n^2)$

Backtracking.

27/8/2022

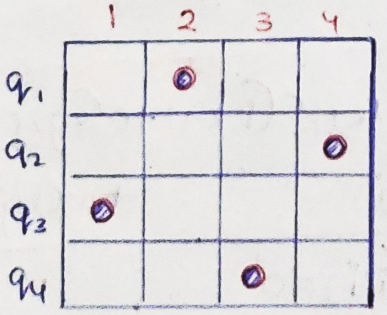
It represents one of the most general techniques. Many problems which deal with searching for a set of combinations/solutions or which ask for an Optimal Solution satisfying the constraints can be solved using backtracking formulation.

The name backtrack was first coined by D.H. Lehmer in the 1950s. Early workers also studied the procedure where R.J Walker who gave an algorithmic account of it in 1960 and Golomb and Baumert also presented a very general description of the backtracking coupled with a variety of applications.

In order to apply the backtrack method, the desired solution must be expressed as an n-tuple (x_1, x_2) where x_i are chosen from one finite set S_i. Often the problem to be solved calls for finding one vector which maximises (or satisfies) a criterion function $P(x_1 \dots x_n)$. Sometime it seeks all such vector which satisfy P.

1) n Queens : $\langle q_1 \dots q_n \rangle$ – n x n square board

n=4



$x[1 \dots n] = \langle x_1, x_2, x_3, x_4 \rangle$

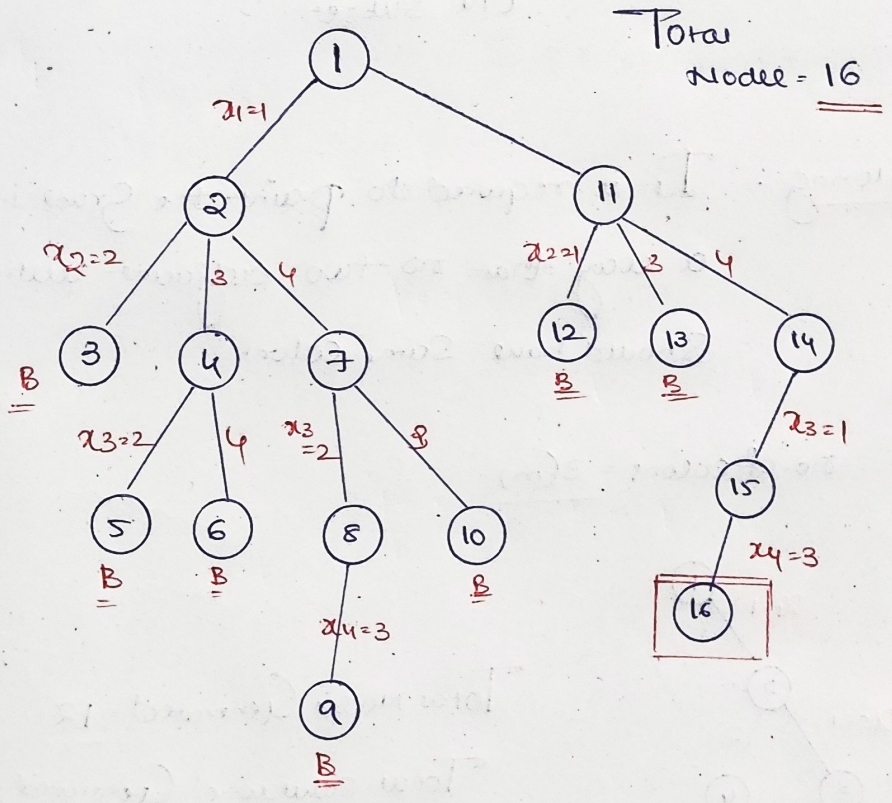
$1 \le x_i \le n$.

x_i = position of q_i i.e placed in row 'i'

→ Backtracking uses D.F.S to search for feasible solutions in the solution space and it also applies bounding functions to kill or bound those nodes that are not feasible;

> Backtracking = D.F.S + Bounding functions

Portion of State-Space Tree Generated due to backtracking Process for 4 Queens



An optimal Solution

⟨2, 4, 1, 3⟩

2) Variable tuple Size formation of State Space Tree.

Sum of Subsets : Given n elements : m. ↳ target

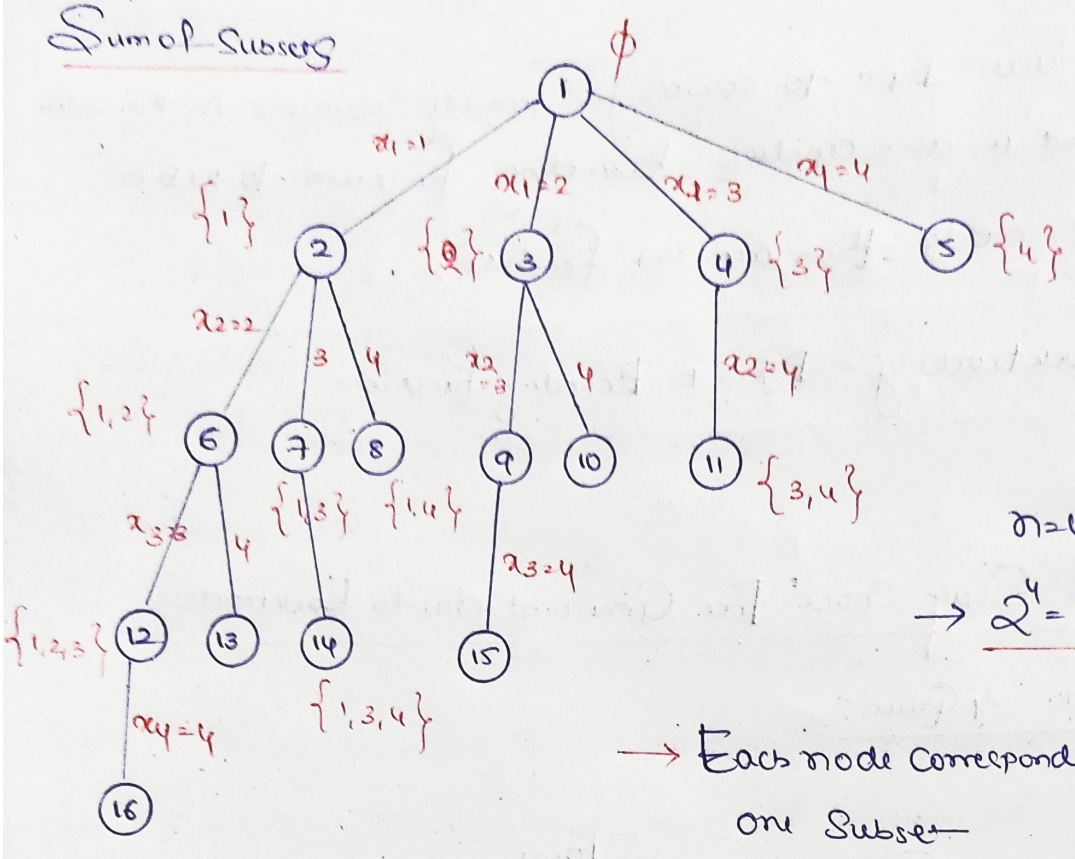
A : ⟨1 ... n⟩

n = 5, m = 50

A : ⟨10, 20, 30, 40, 50⟩ — $\{2^n\}$ - all possible Subsets

{20, 30}, {10, 40}, {50}

Sum of Subsets

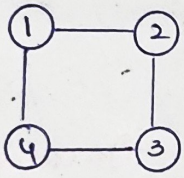


n=4

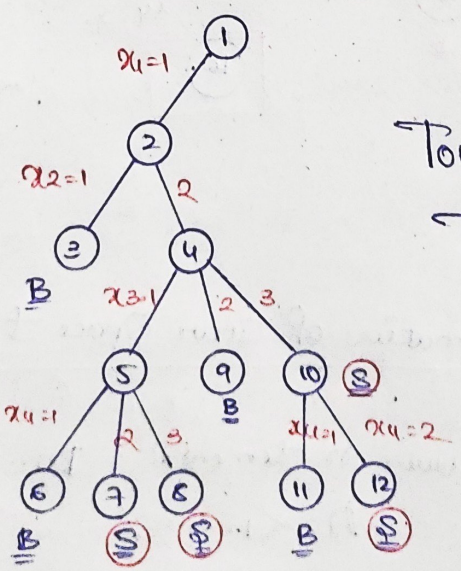
→ $2^4 = 16$

→ Each node corresponds to one Subset.

3). Graph Coloring: It is required to paint the Graph in such a way that no two adjacent vertices should have same color.



no. of colors = 3(m)



Total Node Generated = 12

Total solutions Generated = 4

Popular problems solved by Backtracking

1. N-Queens
2. Sum of Subsets
3. Graph coloring
4. Hamiltonian Cycle
5. 0/1 Knapsack.

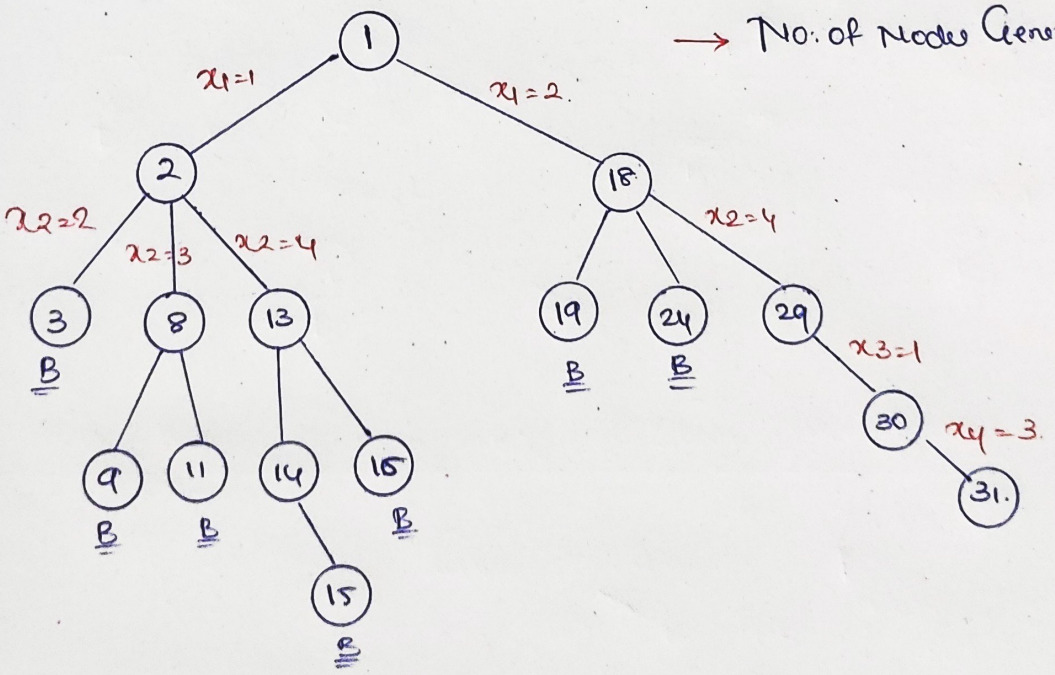
Branch and Bound

Backtracking = D.F.S + Bounding function → uses Stack

Branch and Bound = B.F.S + Bounding function

→ Queue : → LIFO – BB
→ FIFO – BB

4 Queens Problem using FIFO BB



→ No. of Nodes Generated = 31

Popular Branch & Bound Problems :

1. n-queens
2. 15 puzzle Problem.
3. J.S.D
4. T.S.P
5. 0/1 Knapsack.