INDEX

NAME: K. Gowtham STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: _____

(11/86) It is clear the optimal profit is 60.

$\langle w_i \rangle : \langle 10, 7, 4, 2 \rangle$

$\langle P_i \rangle : \langle 60, 28, 20, 24 \rangle$

$\langle \frac{P_i}{w_i} \rangle : \langle 6, 4, 5, 12 \rangle$

② ④ ③ ①

Select $O_4 \Rightarrow$ remaining weight $= 11-2 = 9$ $P = 24$

select $O_2 \Rightarrow w_2 > 9$

∴ not possible.

Select $O_3 \Rightarrow$ remaining weight $= 9-4 = 5 \Rightarrow P = 24 + 20 = 44$

Select $O_4 \Rightarrow w_2 > 5$

∴ not possible

$\therefore V_{opt} = 60$ $V_{greedy} = 44$

∴ Ans: 16

Job Sequencing with Deadlines (JSD)

→ n-jobs / programs / Process

→ Each job (J_i) is arrives at time 0 and needs one unit time.

→ Every job is associated with a deadline d_i & profit P_i. The profit is obtained only if the job is completed within its deadline.

Problem stmt:

select a subset of given n-jobs such that the jobs in the subset are completable within their deadlines and generate maximum profit.

→ Here the trivial case is when all the jobs can be scheduled.

→ size of soln space is 2^n.

It means working out this problem using brute force, the time complexity is $O(2^n)$.

> feasible soln for this problem are those subsets having all the jobs which can be scheduled within the deadline (Implicit constraint)

Eg: $n=4$; $(J_1 - J_4)$;

$(P_1 - P_4) = \langle 100, 15, 10, 40 \rangle$

$\langle d_1, d_2, d_3, d_4 \rangle = \langle 2, 1, 2, 1 \rangle$

Let J be soln subset. Initially $J = \emptyset$ ← feasible soln

I. $|J| = 0$

feasible soln

profit, $P = 0$

II: $|J| = 1$

$J = \{J_1\}$

feasible soln ; $P = 100$

$J = \{J_2\}$

feasible soh; $P = 15$

III: $|J| = 2$

$J = \{J_1, J_2\}$

J_2	J_1

feasible soln

profit → 100+15 = 115

Schedule : J_2, J_1

$J_3 = \{J_1, J_3\}$

this is not a feasible soln

Profit = 110

Schedule : J_1, J_3 (or) J_3, J_1

$J = \{J_2, J_4\}$

This is not a feasible soln.

∵ we cannot schedule J_2 & J_4 within their deadlines.

IV. $|J| = 3$

no three jobs can be scheduled

∴ there is not feasible soln with $|J| = 3$.

Note:

The maximum no of jobs that a feasible subset can have = maximum deadline.

(This rule applies only when each job takes exactly 1 unit of time)

Greedy approach for JSD:

$n = 4$; $\langle j_1 - j_4 \rangle$; $\langle P_1 - P_4 \rangle = \langle 100, 15, 10, 40 \rangle$; $\langle d_1 - d_4 \rangle = \langle 2, 1, 2, 1 \rangle$

Let J be the soln subset

→ The job with highest profit will always be part of optimal soln

Step(i): Arrange the jobs in decreasing order of profits.

i.e., j_1, j_4, j_2, j_3

one by one

Schedule the jobs of sorted order such that the job goes into maximum possible slot below its deadline.

∴ Pick j_1:

deadline is 2

slot 2 is free

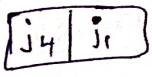
∴

	j_1

Pick 4:

deadline → 1

slot '1' is empty



pick j_2:

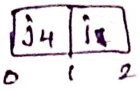
slot 1 is not empty.

pick j_3:

slot 2 is not empty. Now check slot 1.

slot 1 is not empty.

∴ final schedule:



∴ profit = 100 + 40 = 140

Eg: n = 7

$\langle J_1 - J_7 \rangle$;

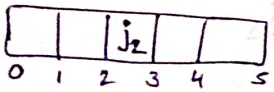
$\langle d_1 - d_7 \rangle = \langle 2, 3, 4, 5, 2, 4, 4 \rangle$

$\langle P_1 - P_7 \rangle = \langle 54, 92, 30, 10, 15, 60, 45 \rangle$

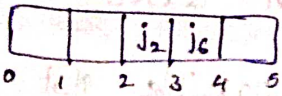
decreasing order of profits

$j_2, j_6, j_1, j_7, j_3, j_5, j_4$

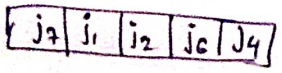
Pick j_2:



Pick j_6:



Pick j_7:



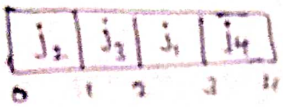
profit = 54 + 92 + 10 + 60 + 45

= 261

Eg: $n=6$; $\langle j_1 - j_6 \rangle$;

$\langle d_1 - d_6 \rangle = \langle 4, 4, 3, 4, 2, 3 \rangle$

$\langle p_1 - p_6 \rangle = \langle 105, 38, 46, 160, 20, 19 \rangle$

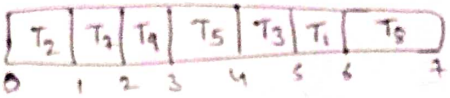


$P = 105 + 38 + 46 + 150$

$= 339$

(H/37)

a)



∴ opt (d)

b) $P = 15 + 20 + 30 + 18 + 25 + 16 + 25$

$= 147$

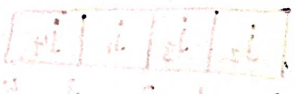
Control Abstraction of Greedy method

```
Procedure GREEDY(A, n)        ↳ i/p size
{
    1. solution ← ϕ
    2. for i ← 1 to n
       {
           x ← SELECT(A);
           if (FEASIBLE(x, solution)) // checks whether including x in soln
                                      is feasible or not.
               ADD(x, solution);
       }
    3. return (solution);
}
```

From this control abstraction,

we say minimum time complexity is $O(n)$

i.e., when, SELECT(), FEASIBLE(), ADD() are $O(1)$



High-level Pseudocode for JSD:

```
Algorithm JSD(d, P, J, n)
// we assume that jobs are arranged in decreasing order of profits
// P1 ≥ P2 ≥ P3 ... ≥ Pn
{
    1. J ← {1};
    2. for i ← 2 to n ——→ O(n)
       {
           if (all job in J ∪ {i} can be completed by their deadlines) then ——→ O(n)
               J ← J ∪ {i};
       }
}
```

Time Complexity of JSD: $O(n^2)$

→ The working of JSD is similar to the worst case behaviour of insertion sort.

Optimal Merge Patterns

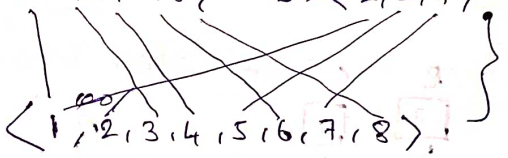
→ Merging of files: Files have records in sorted order.

→ Given n files, it is required to merge them using 2-way merging to get a single sorted file with an objective of minimizing the total no of record movements.

Eg: Consider merging two file A, B with 5, 3 records respectively

~~A = <a1, a2, a3, a4, a5>~~

A = <1, 3, 4, 6, 8> B = <2, 5, 7>



5+3 = 8 record movements

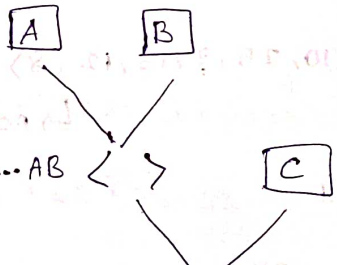
<1, 2, 3, 4, 5, 6, 7, 8>

Eg 2: n=3;

<A, B, C> = <8, 3, 7>

↳ no of records

→ Consider we merge A, B first



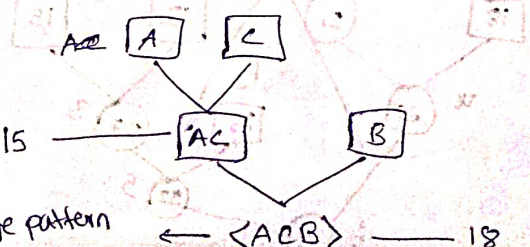
8+3 = 11 record movements ..... AB < >

< > ABC → 11+7 = 18 record movements

↓ merge pattern

∴ 11+18 = 29 records movements.

→ Now consider we merge A, C first



15 — AC — B

merge pattern ← <ACB> — 18

∴ 33 record movements.

→ Here size of soln space = $n!$

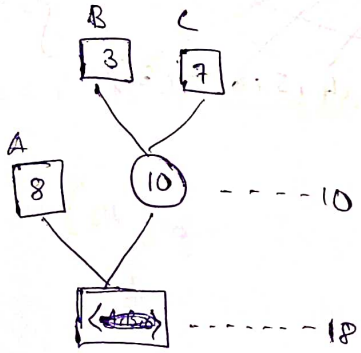
∴ TC with brute force = $O(n^n)$

→ Also here every soln in the soln space is a feasible soln.

Greedy strategy for OMP

At each step (of every (n-1) steps), select the two files with least no of records, merge them and add to the list.

$n=3$; $\langle A,B,C \rangle = \langle 8,3,7 \rangle$



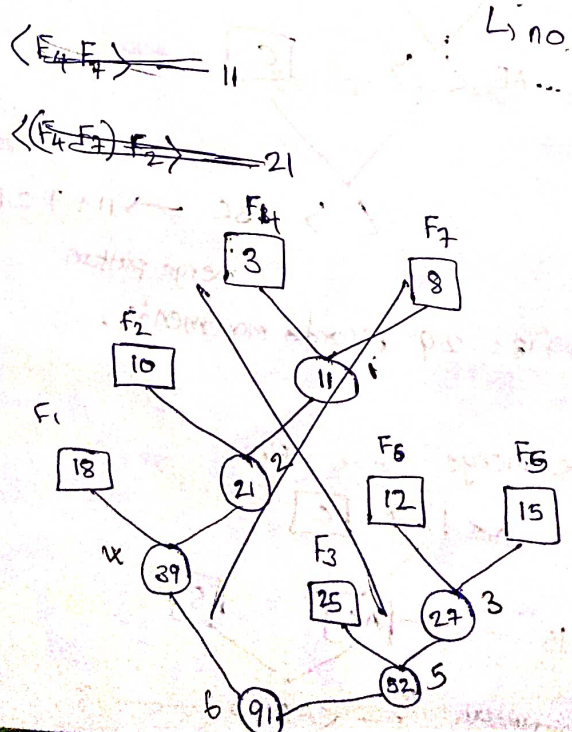
→ 2-way Optimal binary merge tree (reverse of the figure)

∴ Min record movements = $10+18 = 28$

Optimal merge pattern = $\langle B,C,A \rangle$

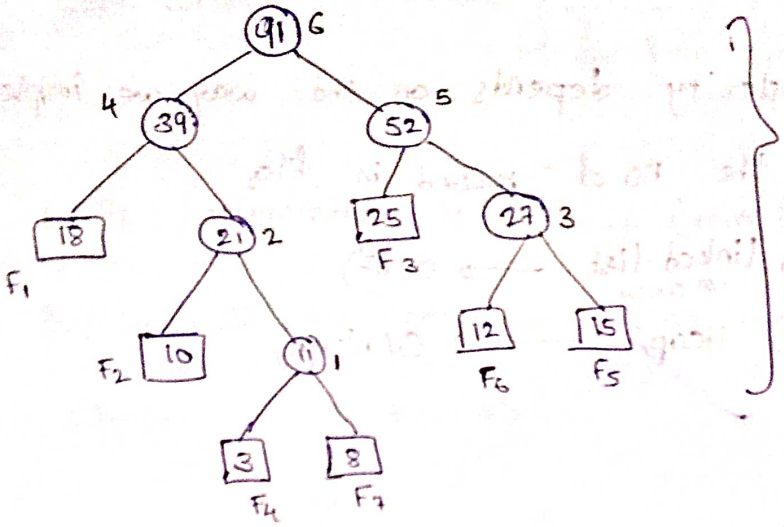
Eg: $n=7$; $\langle F_1 - F_7 \rangle = \langle 18, 10, 25, 3, 15, 12, 8 \rangle$

↳ no. of records per each file



∴ no of record movements = 11 + 21 + 39 + ~~25~~ + 27 + 52 + 91

= 241



optimal binary merge tree.

→ Sum of internal nodes gives min no of record movements.

Note:

Total no of record movements involved during OMP is given by weighted external path length.

i.e., $\sum_{i=1}^{n} q_i * d_i$

d_i → distance from root to F_i

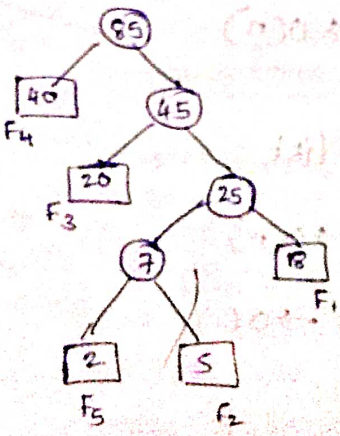
q_i → size of F_i

= (2 × 18) + (3 × 10) + (2 × 25) + (4 × 3) + (3 × 15) + (3 × 12) + (4 × 8)

= 36 + 30 + 50 + 12 + 45 + 36 + 32

= 241

Eg: n = 5; ⟨F1 ... F5⟩ = ⟨8, 5, 20, 40, 2⟩



Min record movement

= 85 + 45 + 25 + 7

= 162

Performance Analysis:

Time Complexity:

→ Time complexity depends on the way we implement or store the no of record in files

(i) linked list (or) array $\longrightarrow O(n^2)$

(ii) Heap $\longrightarrow O(n \log n)$

11/10/20

Algo OMP (L, n)

// L is a list of n single node binary tree as described below

{

1. for i ← 1 to n-1 do
2. call GETNODE(T) // creates a node T
3. LCHILD(T) ← LEAST(L)
4. RCHILD(T) ← LEAST(L)
5. WEIGH(T) ← WEIGHT(LCHILD(T)) + WEIGHT(RCHILD(T))
6. call INSERT(L, T)
7. repeat
8. return (LEAST(L))

}

Here time complexity depends on the way we implement list L.

(i) If L is ordered linear list

LEAST (L) $\longrightarrow O(1)$

INSERT(L, T) $\rightarrow O(n)$

(ii) If L is unordered linear list

LEAST (L) $\longrightarrow O(n)$

INSERT (L, T) $\rightarrow O(1)$

Thus if the algorithm is implemented with a list

∴ time complexity ⟶ $O(n^2)$

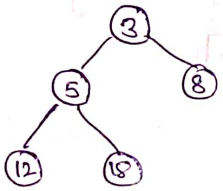
Space complexity ⟶ $O(n)$

↳ for tree created by nodes T

(iii) If L is implemented as a heap (min heap)

first we build the heap with O(n) time

$\langle F_1 - F_5 \rangle = \langle 5, 12, 8, 3, 18 \rangle$



LEAST(L) takes $O(\log n)$ time

INSERT(L, T) takes $O(\log n)$ time

⇒ Time complexity ⟶ $O(n \log n)$

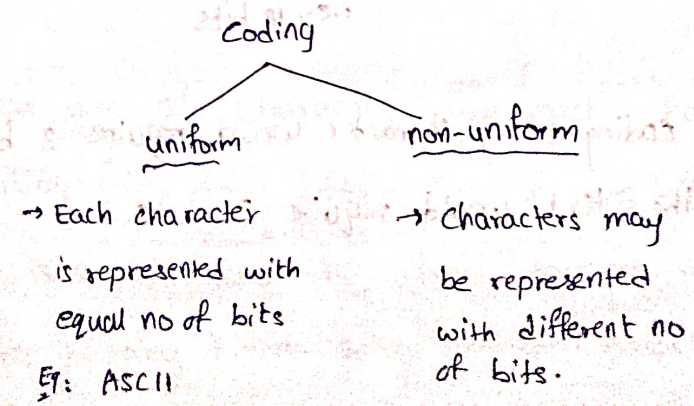
Space Complexity ⟶ $O(n)$

↳ space for heap

Huffman Coding

→ It is an application of optimal merge pattern

→ Huffman coding is a data encoding technique.

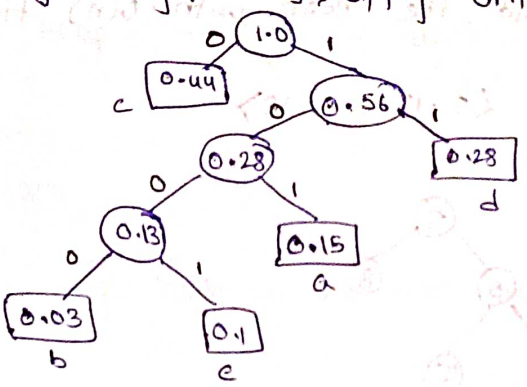


→ Huffman coding is a non-uniform coding.

→ Characters / elements having higher frequencies / probabilities of occurences must be encoded with less no of bits.

Eg: $L = \langle a, b, c, d, e \rangle = \langle 0.15; 0.03; 0.44; 0.28; 0.1 \rangle$

Huffman coding: to get codes, apply OMP algorithm



Mark every left edge as 0 and right edge as 1
(reverse can also be followed)

⇒ a → 101
b → 1000
c → 0
d → 11
e → 1001

Now consider the text ccdccddade

then corresponding binary stream is 0 0 11 0 0 0 11 101 11 1001

i.e., 18 bits

using uniform coding every character would require 3 bits and hence the text would require 30 bits.

Ex: Consider binary stream 10101011111

Find text & considering previous example

Sol:

For i/p traverse the tree until you reach the leaf.

After each leaf, print that character and traverse again from the root using remaining i/p.

∴ In above stream after reading 101 we arrive at leaf 'a'

after reading 0 we arrive at leaf 'c'.

⋮

∴ 101011011111 → acdedd

→ Find avg no of bits used per character in huffman coding.

$(0.15)(3) + (0.05)(4) + (0.44)(1) + (0.22)(2) + (0.1)(4) = 1.97$ bits

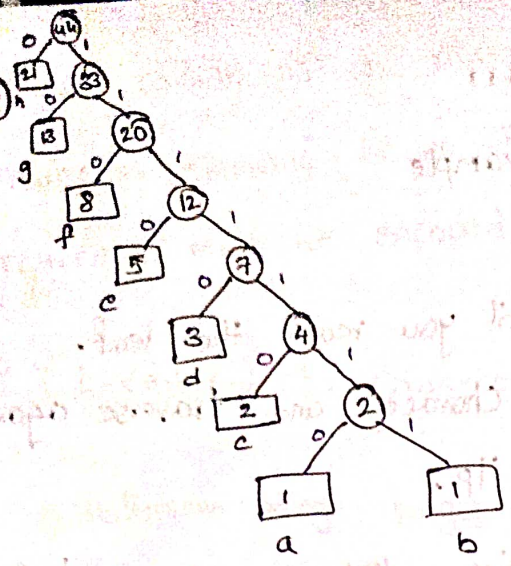
- i.e., $\sum_{i=1}^{n} d_i q_i$

i.e., avg no of bits is given by weighted external path length ÷ sum of internal nodes.

→ Disadvantage of huffman coding is that even a single bit error can make the whole text corrupted.

→ If there are 2^n characters and if each character has equal probability (i.e. $\frac{1}{2^n}$) then avg no of bits req in huffman coding is n. Also the tree formed will be a complete binary tree. with all the characters nodes at leaf position.

H/38



110 111 00 1110 10
f d h e g

∴ fdheg

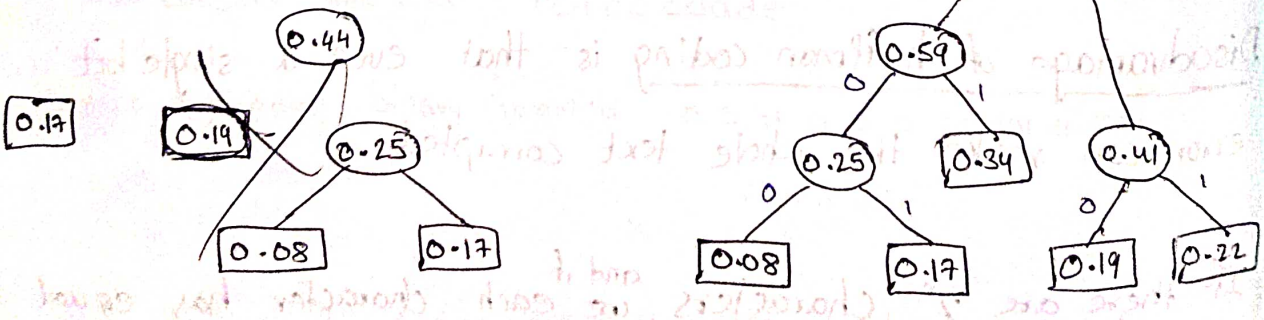
Find avg no of ~~characters~~ bits per characters.

$$= \frac{\sum q_i d_i}{\sum f_i} \quad \Sigma f_i \rightarrow \text{frequencies}$$

$$= \frac{\text{Sum of internal nodes}}{1+1+2+3+5+8+13+21}$$

$$= \frac{122}{54}$$

H/39



avg no of bits = $3(0.08) + 3(0.17) + 2[0.34 + 0.19 + 0.22]$

$= 0.75 + 2(0.75)$

~~1.5 bits~~ $= 2.25$ bits

∴ 225 bits

Spanning Trees:

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

G is weight undirected graph

Spanning tree: A subgraph $T(V, E')$ of a given graph $G = (V, E)$ where $E' \subseteq E$ is a spanning tree iff T is connected acyclic graph (tree)

solution space:

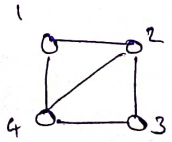
i.e., Maximum no of spanning trees $= n^{n-2}$ → kaley's formula for a complete graph of n vertices.

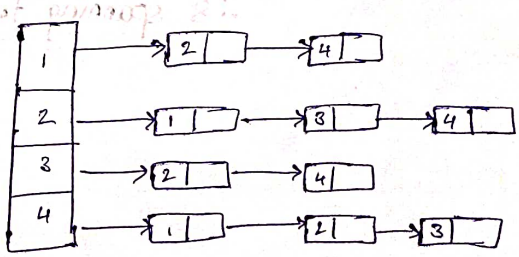
→ ~~No of spanning trees possible for any graph is given by cofactor of any element of the matrix representing the graph~~

Note:

→ Any graph based problem that use adjacency matrix or cost adjacency matrix has time complexity no less than $O(n^2)$

→

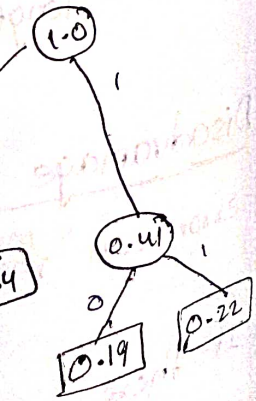




undirected graph → ~~n+2e~~ n+2e nodes.

directed graph → n+e nodes

→ Thus any graph problem which represent graph with adjacency list has atleast $O(n+e)$



→ Max value possible for e is $O(n^2)$

↳ In some cases, $O(n+e) = O(n^2)$

∴ For complete graphs / dense graphs, adjacency matrix representation is desirable.

For sparse graphs adj. list representation is desirable.

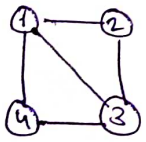
Calculating no of spanning trees for a non-complete graph:

(i) Create adjacency matrix for given graph.

(ii) Replace all diagonal elements with degrees of node and replace all non-diagonal 1's (edges) with -1.

(iii) Now calculate co-factor of any element of the matrix and this gives no of spanning trees.

Eg:



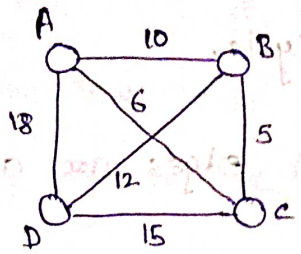
$\longrightarrow$

$$\begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Cofactor of 3 $= \begin{vmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{vmatrix} = 8$

∴ 8 spanning trees.

Minimum cost spanning Tree (MCST):

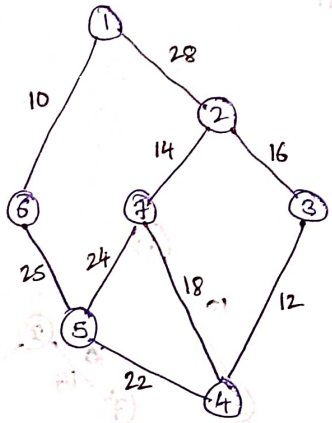


Applications of spanning tree:

→ Used in multicast, broadcast

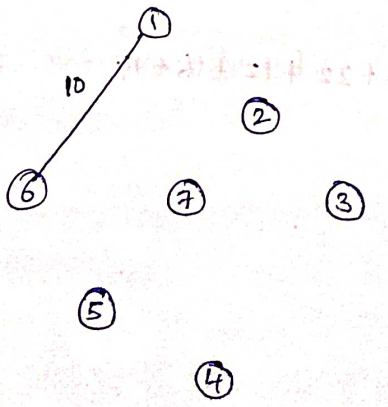
→ MCST is used in electronic & eletrical applications (in circuits)

Construction of MCST:



Prims Algorithm:

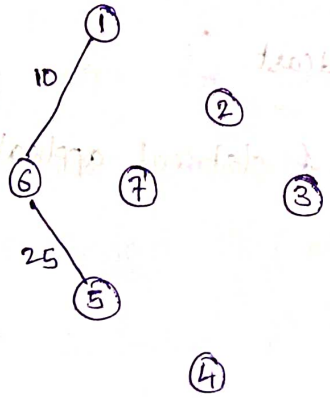
i) choose the least cost edge and add to the solution.



(ii) The next edge to be added, must be adjacent from to the vertices included so far such that it has least cost and it does not form a cycle.

Repeat step (ii) until a total (n-1) edges are added

So add edge 1-6

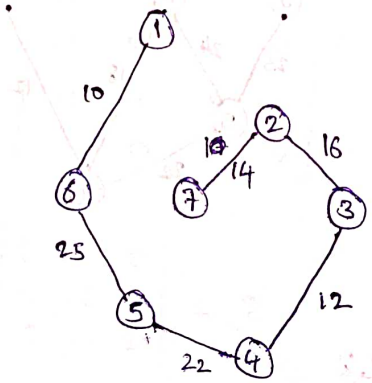


Add 5-4 ... 22

Add 4-3 ... 12

Add 2-3 16

Add 2-7 14



6 edges are added and we finish the process here

∴ Min cost =

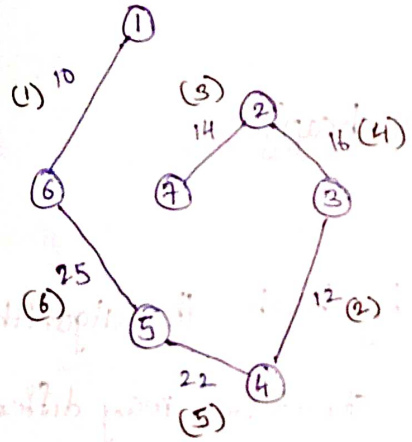
Cost of MCST = 10+25+22+12+16+14

= 99

Kruskal's algorithm

1. Arrange the edges of the graph in increasing order. (min heap)
2. Delete the least cost edge from the list. (delete root of min heap)
3. Add the edge to the soln iff it does not lead to cycle.
4. Repeat (2) & (3) until a total of (n-1) edges are added.

edges → 10, 12, 14, 16, 18, 22, 24, 25, 28



Add 10
Add 12
Add 14
Add 16
Adding 18 leads to a cycle
Add 22
Add 24 forms a cycle
Add 25

The numbering shows the order in which the edges are added.

→ At the time of construction edges 18, 24 are discarded whereas 28 is not considered.

Difference b/w Prim's & kruskal's algorithms

(i) Prims method always maitains tree structured property at every step (~~correct~~ connectedness) where as kruskal's algorithm may or may not.

(ii) Time complexities:

Prim's algorithm :

- adjacency matrix & linear list : $O(n^2)$
- adjacency list & heap : $O((n+e)\log n)$

kruskal's algorithm

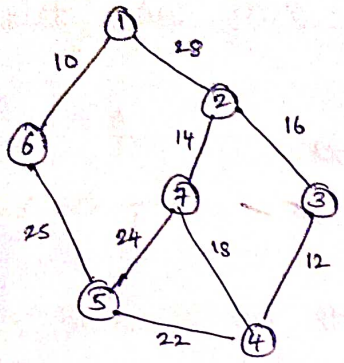
- $O(e \log e)$ [using heap]

(iii) The cost of MCST generated by both the algorithms is same. However, the tree structure may differ.

* The tree structure may differ when there are multiple edges of same weight. (or) ~~when~~. ~~there are two or more spanning trees with same minimum cost~~

* Tree structure generated by prims and kruskal will if all the edge costs are distinct. ~~and there is only one spanning tree with minimum cost~~.

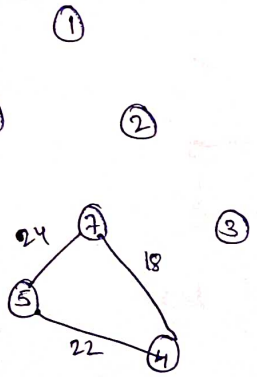
Dijkstra's algorithm:



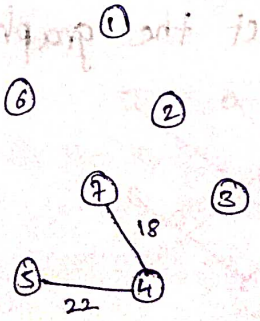
(This algorithm doesn't come totally under greedy)

1. Starting adding any edge and continue adding any edges one by one until a cycle is formed. Once a cycle is formed remove the maximum cost edge from the cycle. Repeat the above step until n-1 edges are added or all the edges are exhausted. (Note that no edge should be added 2nd time after its removal)

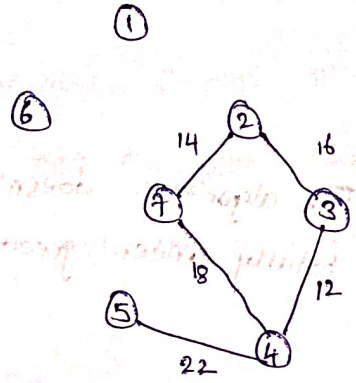
Eg: Consider adding 24, 22, 18.



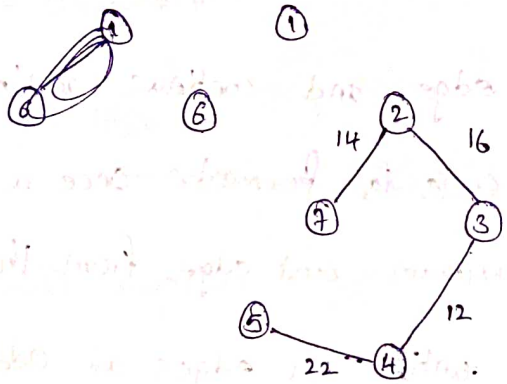
Remove 24 from cycle.



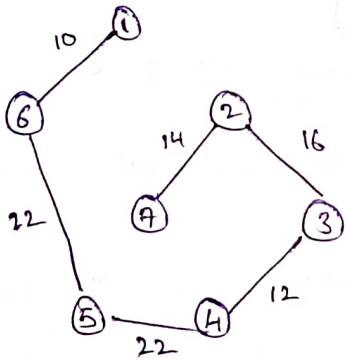
Add 12, 16, 14



Remove 18



Add 25, 10



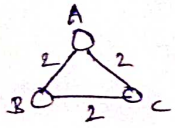
6 edges are added

∴ Stop the process

Calculating no of MCSTs:

→ If all edges costs are distinct the graph will have a unique MCST.

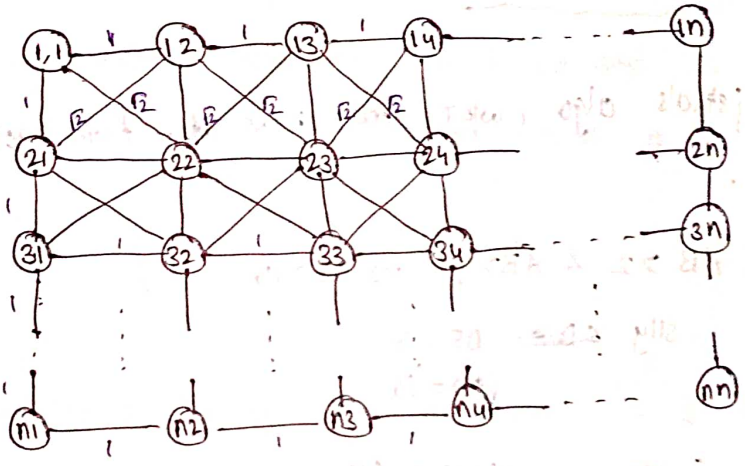
→ If there exist multiple edges of same cost then there may exist more than one MCST with same cost.

Eg: 

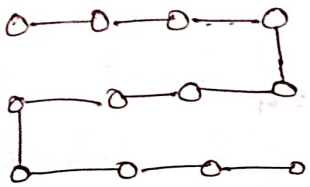
No of MCSTs possible = 3

Note: There is no standard formula for calculating no of MCSTs. we just need to calculate from the graph given using permutations & combinations.

H/40

i) 

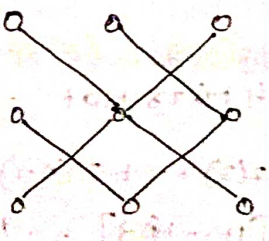
So the min cost spanning tree is



∴ Cost is $(n-1)(n) + (n-1) = (n+1)(n-1) = n^2-1$

ii) Find cost of max cost spanning tree.

take $n=3$



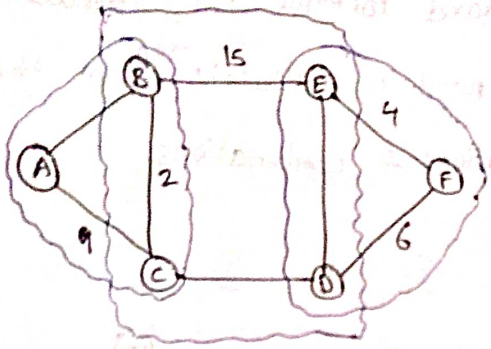
We have added 7 edges ~~add~~ each of cost 12.

Now we can no more add any edge of ~~cost~~ 12.

∴ we add one edge of cost 1.

∴ Cost is $(n^2-2)(12) + 1$

H/41



Using dikjstra's algo (wkT that max wt. from cycle has to removed.

∴ AB > 2 & AB > 9 ⇒ AB = 10

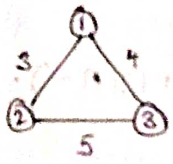
Silly ~~CD~~ DE = 7

ED = 16

∴ 36 + 10 + 16 + 7 = 69

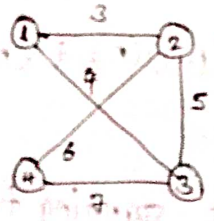
H/42

n = 3



i.e., 3+4 = 7

n = 4



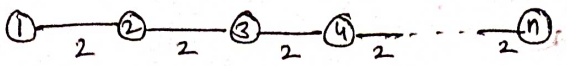
i.e., 3+4+6

∴ the sequence is 3 + 4 + 6 + 8 + 10 + ...

i.e., $3 + 2(2) + 2(3) + \cdots + 2(n-1)$

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

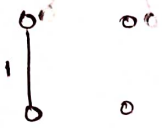
H/43 MCST is



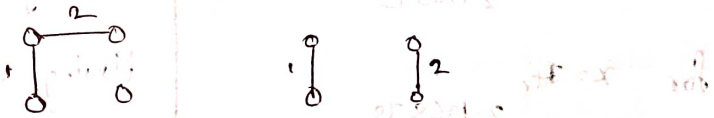
$\therefore 2(n-1)$

H/44 $\{1,2,3,4,5,6\}$

'1' is always parts of MCST



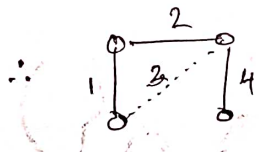
Using kruskal's algorithm, next we add 2 and this doesn't lead to cycle no matter where the edge is



has to

Choosing '3' in first case it could be forming a cycle

So '4' will be part of MCST



~~MIA COS~~

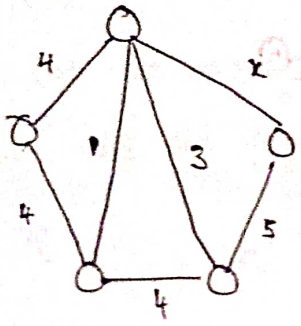
Max possible cost of MCST = 1+2+4 = 7

H/45 After modification,

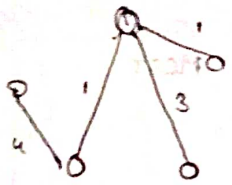
MCST's structure remains same with weights increased.

$\therefore 500 + 99(5) = 995$

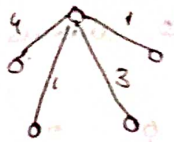
H/46



$x=1$ ⇒



(or)



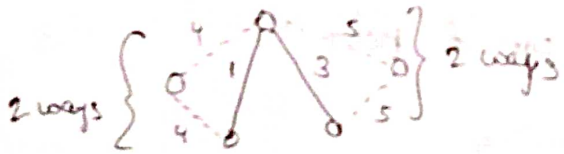
Silly for $x=2$,

2 MCSTs.

for $x=3$, 2 MCSTs

for $x=4$, 2 MCSTs

for $x=5$



$\therefore 2 \times 2 = 4$ ways

For this kind of problems use dijkstra's algo.

Having $x=5$ will add 2 possibilities of removing max cost edge from its corresponding cycle

H/47

(i) ~~False for w-regular graph~~.

(ii) ~~the true~~

(iii)

(i) True, since we can start construction of MCST for that edge (kruskal's algo)

(ii) true

(iii) False in the case of w-regular graph

(iv) true

H/48 I.

12/10/20

Algo PRIM (

// E is set of

// COST(n,n)

either posi

// T[1..n-1, 1

{

1. real co

2. integer

3. (k,l)

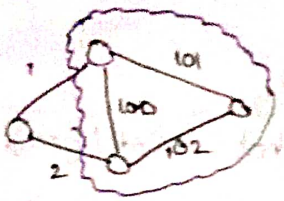
4. mincost

5. (T(1,1)

6. for i←

7. for

(H/48) I.



'100' is lightest, still it is excluded cuz it is heaviest of another cycle

∴ false

II. true from dijkstra's algo

∴ true

12/10/20

Algo PRIM (E, COST, n, T, mincost)

// E is set of edges in G

// COST(n,n) is cost adjacency matrix such that COST(i,j) is either positive or ∞ if there is no edge blw i & j

// T[1...n-1, 1:2] is store set of edges of MCST

{

1. real COST(n,n), mincost;
2. integer NEAR(n), n, i, j, k, l, T(n-1,2);
3. (k,l) ← edge with min cost ..... O(e)
4. mincost ← cost(k,l)
5. (T(1,1), T(1,2)) ← (k,l)
6. for i←1 to n do ..... O(n)
   if COST(i,l) < cost(i,k) then
   NEAR(i) ← l
   else
   NEAR(i) ← k
7. ~~for~~

Problems
algo.
will add
sibilities of
ax. cost edge
corresponding
of MCST

7. NEAR(k) ← NEAR(l) ← 0

8. for i ← 2 to n-1 do ---- O(n)

{

a) let j be an index such that NEAR(j) ≠ 0 and COST(j, NEAR(j)) is minimum. ---------- O(n)

b) (T(i,1), T(i,2)) ← (j, NEAR(j))

c) mincost ← mincost + COST(j, NEAR(j))

d) NEAR(j) ← 0

e) for k ← 1 to n do ---------- O(n)

{

if NEAR(k) ≠ 0 and COST(k, NEAR(k)) > COST(k, j) then

NEAR(k) ← j

}

}

9. if mincost ≥ ∞ then

print ("no spanning tree")

}

Time complexity

- using adjacenty matrix : $O(n^2)$

Using heap / Red black tree:

(i) Creating heap with e edges —— O(e)

(ii) Step 3 —— O(log e)

(iii) Step 8 —— O(n)

Step 8.a) —— O(log n) as using heap with ~~near values~~ cost(j, NEAR(j)) values.

Step 8.e) . NEAR(k) ← j corresponds to decrease key operation and it takes O(log n)

In total se occurs $n^2 \log n$ times (in worst case

However in general it is $e \log n$

$\therefore TC = O(e) + O(e \log e) + \cdots + n \log n + e \log n$

(8.a) (8.e)

i.e., $O((n+e)\log n) \cong O(e \log n)$

↳ in worst case it is $O(n^2 \log n)$

Hence for dense graphs we use only adjacency matrix but if the graph is sparse graph using heap is better.

Space Complexity:

without heap:

S.C = size(T) + size(near)

$\therefore O(n)$

with heap:

S.C = size(heap for edges) + size(heap for near)

$S.C = O(e+n)$

Shortest Paths:

I. Single pair shortest path

II. Single Source Shortest path

Dijkstra's	Bellman-Ford
→ Greedy	→ Dynamic
→ Always works good for +ve wt edges. → May or may not work with -ve edges	→ Always works correctly with -ve wt. edges but not having -ve cycle reachable from source.

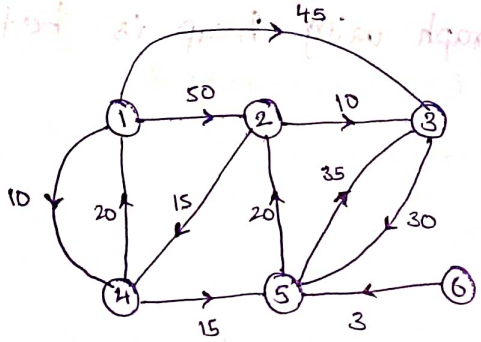
III. All Pair shortest Paths

→ one way is
- we can use single source shortest paths for all vertices

→ Second approach is

Floyd-warshall's algorithim

Single Source ~~Sode~~ Shortest Paths : Dijkstra's Algorithm:



This approach works for both directed & undirected graphs

Consider finding shortest path values from vertex '1'

i) Matrix form:

d(x): The value of vertex 'x' is defined as cost of shortest path known so far

vertex selected	d-values 1	2	3	4	5	6
{1}	-	50	45	(10)	∞	∞
{1,4}	-	50	45	(10)	(25)	∞
{1,4,5}	-	(45)	45	(10)	(25)	∞
{1,4,5,2}	-	(45)	(45)	(10)	(25)	∞
{1,4,5,2,3}	-	(45)	(45)	(10)	(25)	(∞)

→ This process of reducing distance is called relaxation.

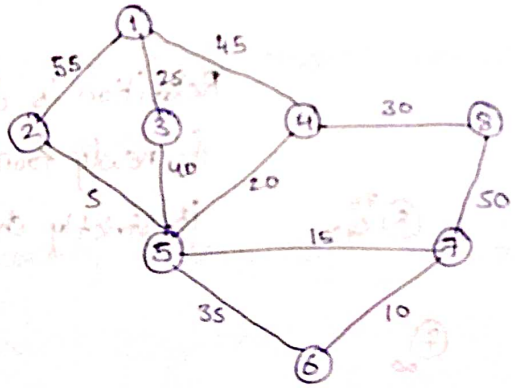
↳ cost of shortest paths.

Consider finding shortest path from vertex 6

Vertex selected	d-values					
	1	2	3	4	5	6
{6}	∞	∞	∞	∞	(3)	-
{6,5}	∞	(23)	38	∞	(3)	-
{6,5,2}	∞	(23)	(33)	38	(3)	-
{6,5,2,3}	∞	(23)	(33)	(38)	(3)	-
{6,5,2,3,4}	58	23	33	28	3	-

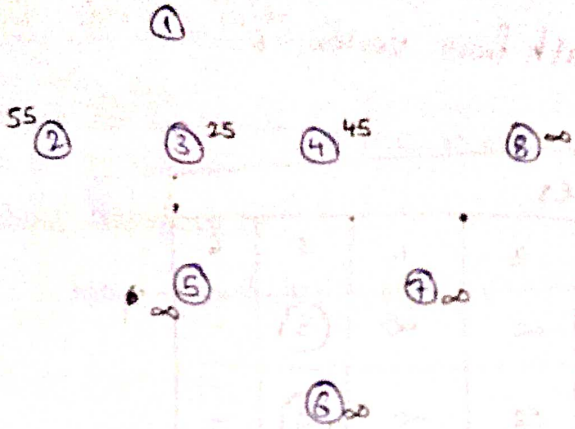
The limitation of this matrix form is that it shows only the cost of the shortest path but not the path.

(ii) Spanning tree approach:

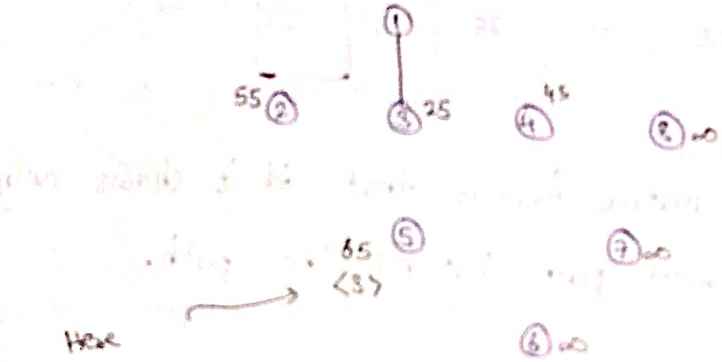


Consider finding shortest path from vertex 1.

∴ write the edges of all nodes from node 1. as shown in the figure.

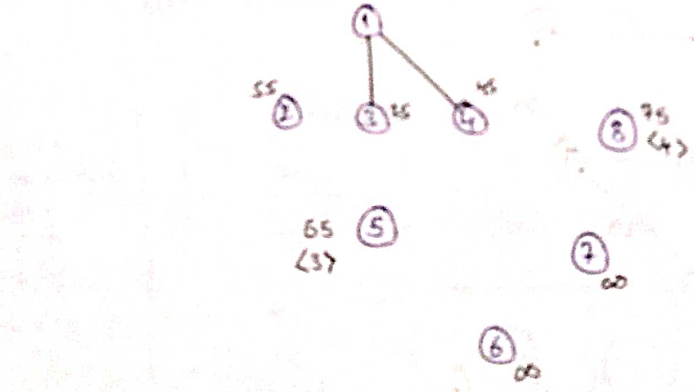


Since '3' is nearest so far, choose vertex 3



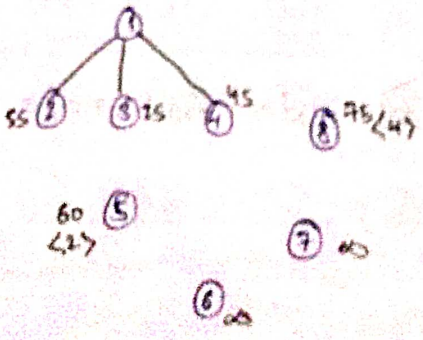
Here vertex 5 is relaxed.

Now choose vertex 4

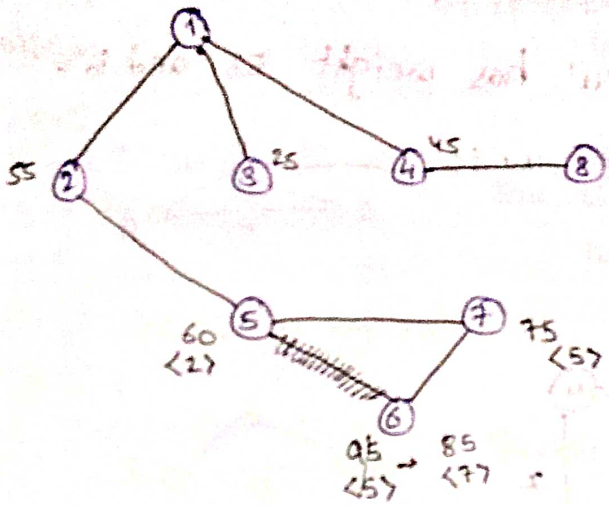


Retrition is done if newly found path is strictly shorter.

Choose vertex '2'



Choose vertex 5 (via 2)



⟹ Single Source Shortest Paths Spanning Tree.

∴ Final costs are

1	2	3	4	5	6	7	8
-	55	25	45	60	85	75	75

1-2 1-3 1-4 1-2-5 1-2-5-7-6 1-2-5-7 1-4-8

Note:

→ MCST and single source shortest paths spanning tree need not to be same.

→ The working / implementation of dijkstra's algorithm is similar to prim's algorithm.

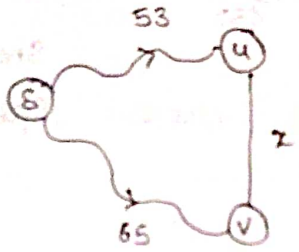
time complexity

- without heap: $O(n^2)$
- with heap: $O((n+e)\log n)$

∴

Q. Consider weighted undirected graph. Let uv be an edge in the graph. It is known that the shortest path from the source vertex 's' to 'u' has weight 53 and to 'v' is 65. Then weight of edge uv is ——

Sol:

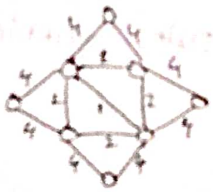


$\Rightarrow 53 + x \geq 65$

$\Rightarrow x \geq 12$

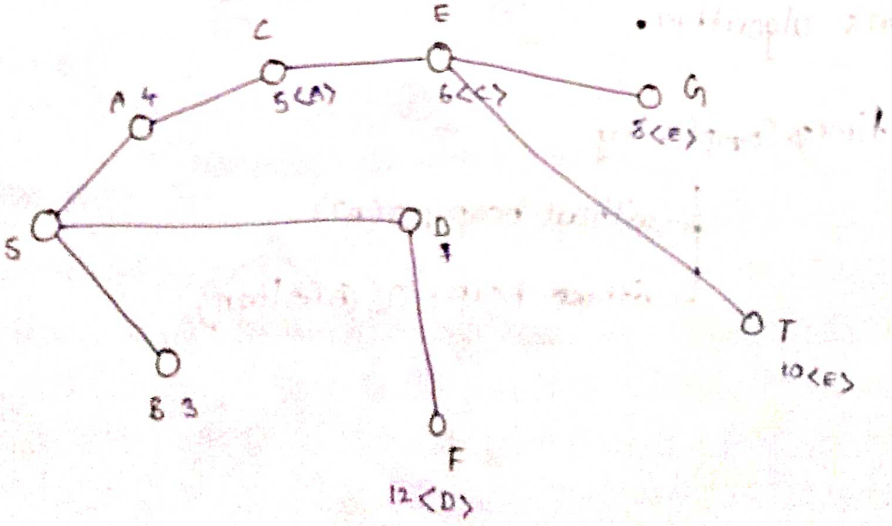
H/49

Q. Find no of MCST possible for below graph



Ans: 64

H/49



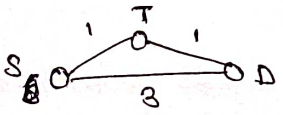
∴ SACET

(H/50)

1. True (∵ doing so will inc cost of every spanning tree by $(n-1)c$)

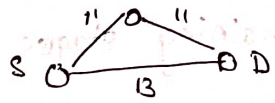
2. False ↳ increased cost.

Eg:



shortest path from S to D is S-T-D

inc weight by 10

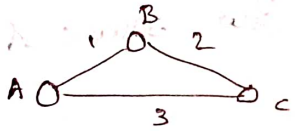


Shortest path : S-D

(H/52)

I. True (concept)

II.



from A to C we have 2 shortest paths

i.e., A - B - C

A - C

∴ False

Dynamic Programming:

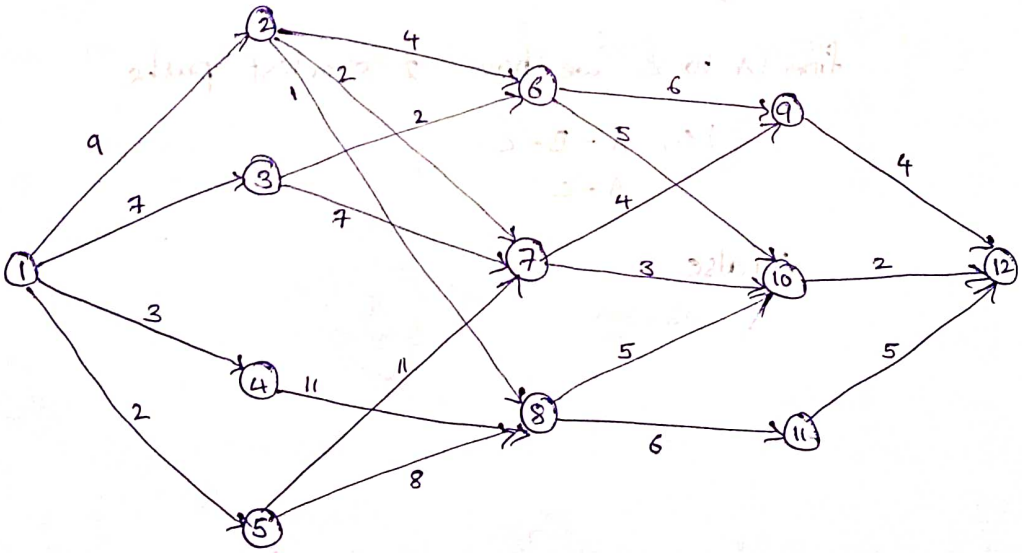
↳ Here programming means act of tabulating the results of subproblems.

Multistage Graph:

→ A k-stage multistage graph has k stages where 1st & last stages have exactly one vertex and remaining stages have atleast one vertex.

→ Every edge is from ~~stage i to~~ a vertex of stage i to a vertex of stage i+1

→ The problem is to find distance btw source & destination



Applying greedy method (choosing least cost at every stage)

1-5-8-10-12 ⇒ 17 ↳ so using greedy method for single pair shortest path may fail

Applying Dijkstra's algorithm

1-3-6-10-12 ⇒ 16 ↳ greedy method works when we need to find shortest paths to all vertices.

→ DP is an algorithm design technique used for solving problems whose solutions are viewed as a result of making a set decisions.

→ One way of solving these problems is by making decisions at every step based on local informations available at that step. This is true for problems solved by greedy method.

→ But for many problems, like multistage graphs, optimal solutions cannot be acheived by making decisions based on local information in a stepwise manner.

→ one way of solving the problems for which optimal solutions cannot be made based on local information, is to enumerate all possible decision sequences and pick up the best. This is known as brute force approach. (enumeration strategy) However for this strategy, the drawback is that it has very large time & space complexities.

→ DP is based on enumeration; but it often tries to curtail the amount of enumeration by removing/avoiding those sequences that are not feasible or sub optimal.

Due this these may be a significant improvement on time complexities.

→ DP makes the sequence of decisions by applying principle of optimality. (global optimality) (optimality substructure)

→ principle of optimality states that whatever ~~whatever~~ the initial state and decision are, the remaining sequence of decision must be optimal with regard to state resulting from the current decision.

→ Any problem that ~~dbp~~ obeys principle of optimality and overlapping subproblems can be solved by DP.

→ The fundamental difference b/w greedy method and DP is that GM generates only one decision sequence whereas DP may generate multiple decision sequences.

→ In DP, the results of subproblems can be tabulated and can be reused. (Memoization)

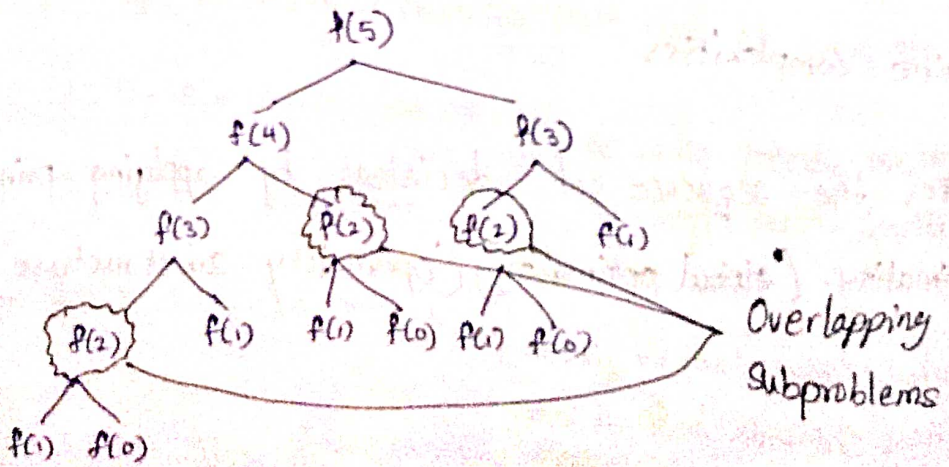
→ Thus DP optimizes time by ~~curtail~~ curtailing decision that lead to non-feasible or sub optimal soln and by tabulated results of subproblems.

Eg: Consider computing nth fibonacci number

fib(0) = 0

fib(1) = 1

fib(n) = fib(n-1) + fib(n-2)

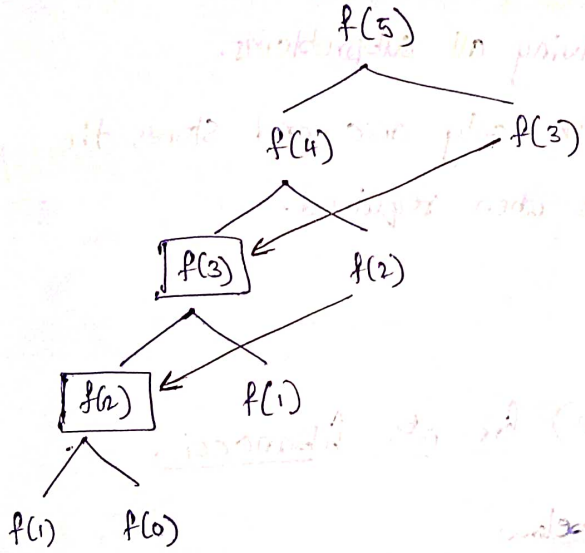


→ Regular recursive implementation

```
int fib(n)
{
    if(n ≤ 1) return n;
    else
        return fib(n-1) + fib(n-2);
}
```

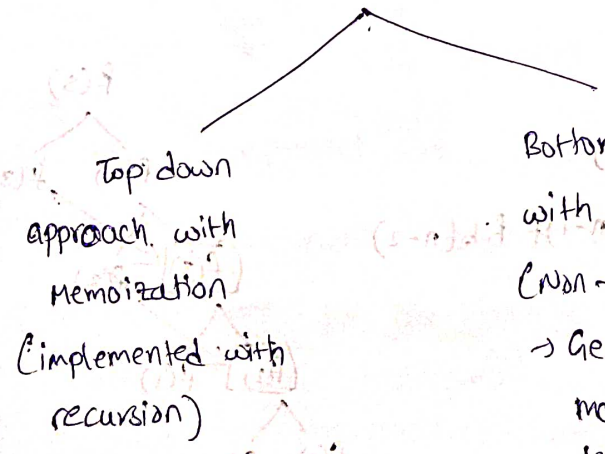
TC: $O(2^n)$. Space. Comp: $O(n)$

→ Top down DP with memoization:



Rather than tree, here a closed graph is formed without cycle.

→ DP problems can be solved in 2 ways:



Note:

Greedy: Builds up a solution increamentally by optimizing at each step some local criteria.

DfC: Break up a problem into seperate subproblems and solve each subproblem independently and combining the solutions to subproblems to get soln of original problem.

DP: Break up a problem into a series of overlapping subproblems and buildup solns to larger and larger subproblems.

Unlike
- DfC, DP involves solving all subproblems.
- DP solves each subproblem only once and stores the results & uses them later when required.

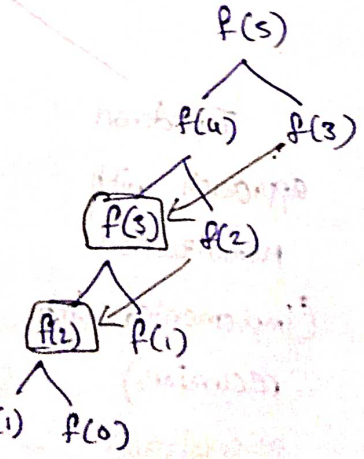
13/10/20

Top down approach (Memoization) for nth fibanocci:

initialize arry f[ ] as shown below

0	1	2	3	4	5	
0	1	NIL	NIL	NIL	NIL	...

```
int fib(n)
{
    if(f[n]==NIL)
        f[n]=fib(n-1)+fib(n-2)
    return (f[n]);
}
```



TC: O(n)

∵ for every number fib(n) is computed only once.

SC: O(n)

i.e., size(array) + size(stack)

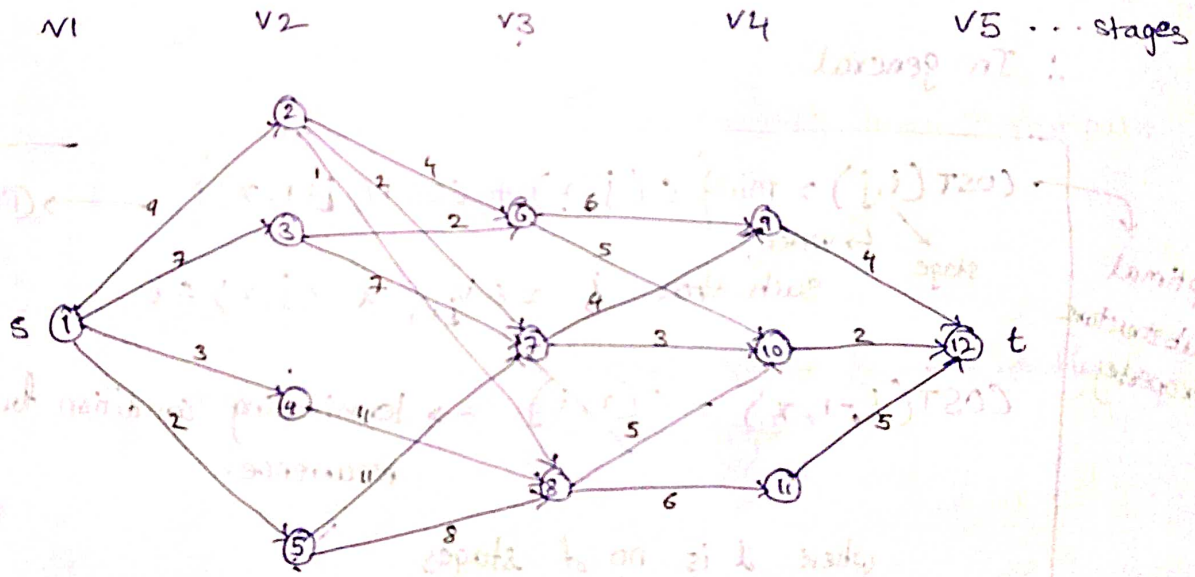
Bottom up approach (Tabulation) for nth fibanocci number

```
f[1000]: int array
f[0]=0; f[1]=1;
int fib(n)
{
    for i=2 to n do
        f[i]= f[i-1]+f[i-2];
    return (f[i]);
}
```

TC: O(n)

SC: O(n) [i.e., size of array]

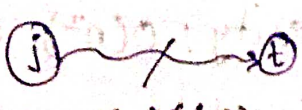
Multistage Graph



Let C[1..n, 1..n] represent cost adjacency matrix.

Let cost(i,j) represent cost of path from vertex 'j' that is present in stage i to the destination vertex 't'.

i.e., stage i



So the target is to find cost(1,1).

from ① we go to vertex x such that $x \in V_2$ & $\langle 1,x \rangle \in E$

↓

i.e., there an edge blw 1 & x.

We choose this x such that (c(1,x) + remaing cost of path from x to t) is minimum.

i.e., $c(1,x) + cost(2,x)$ is minimum

$\therefore$ $COST(1,1) = \min\{c(1,x) + COST(2,x)\}$

such that $x \in V_2$ & $\langle 1,x \rangle \in E$

$\therefore$ In general

$COST(i,j) = \min\{c(j,x) + COST(i+1,x)\}$ → ①

(optimal substructure property)

stage ↙ ↘ vertex

such that $x \in V_{i+1}$ & $\langle j,x \rangle \in E$

$COST(l-1, x) = c(x,t)$ → terminating condition for the recurrence.

where l is no of stages

$D(i,j) =$ 'x'

such 'x' is the value which minizes $COST(i,j)$ in eq①

↳ used to obtain the vertices in the shortest path.

Time complexity : $O(n^2)$ (or) $O(n+e)$

Space complexity : $O(n)$

$COST(1,1) = \min\{\ 9+COST(2,2),\ 7+COST(2,3),\ 3+COST(2,4),\ 2+COST(2,5)\}$

$2 = \{2,3,4,5\}$

so we start from back

$Cost(4,9) = c(9,t) = 4$

$Cost(4,10) = c(10,t) = 2$

$Cost(4,11) = c(11,t) = 5$

Now we compute costs of stage 3 vertices

$COST(3,6) = \min\{c(6,9)+COST(3+1,9),\ c(6,10)+COST(3+1,10)\}$

$= \min\{6+COST(4,9),\ 5+COST(4,10)\}$

$= \min\{6+4,\ 6+2\} = 7$

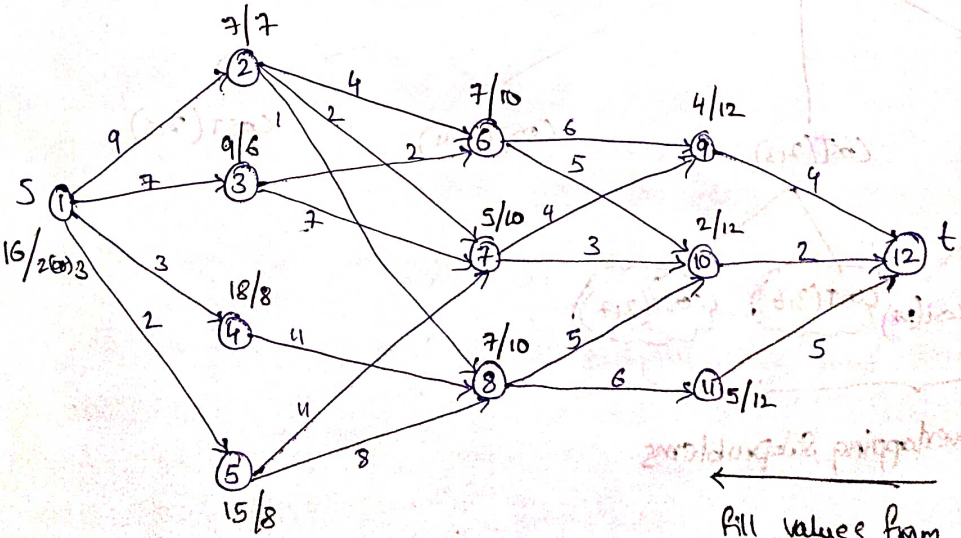
also $D(3,6) = 10$

↳ the vertex which gives shortest path from vertex 6.

$COST(3,7) = 5$

$D(3,7) = 10$

} These value can be found directly from the graph



∴ Min cost path's cost = 16

path can be found by using array D.

From graph we find path using the labels assigned to each vertex

∴ path: 1 - 2 - 7 - 10 - 12

1 - 3 - 6 - 10 - 12

Formal way for path construction:

For an l-stage graph we need to make l-2 decisions

path: 1 -

D(1,1) = 2

path: 1 - 2 -

↑ 1st decision

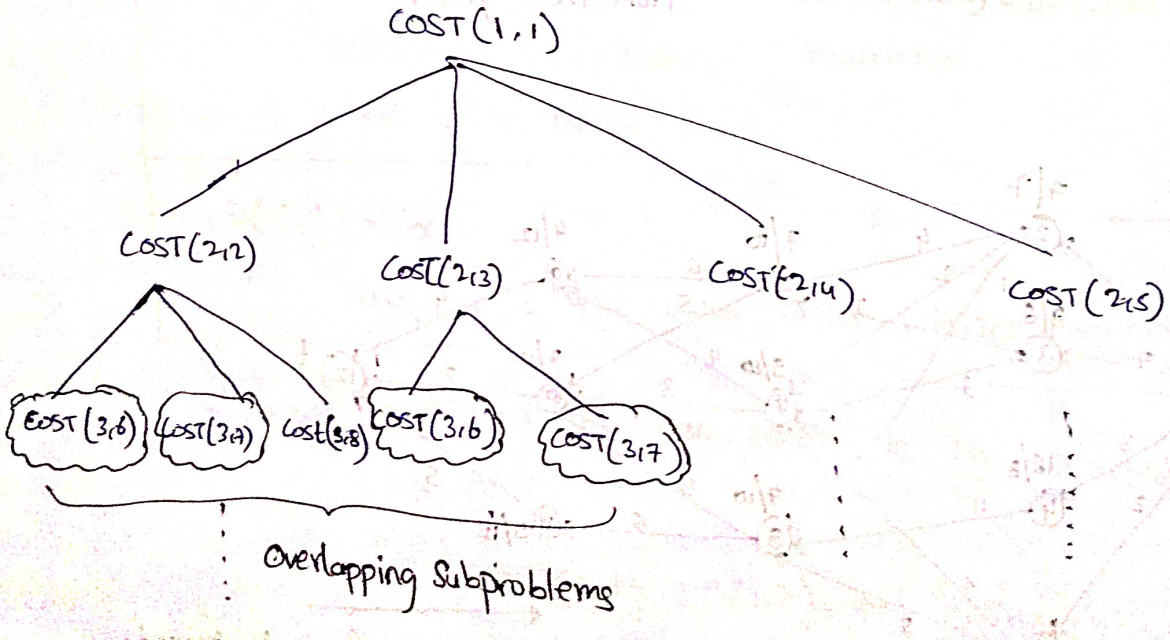
D(2, D(1,1)) = D(2,2) = 7

path: 1 - 2 - 7 -

↑ 2nd decision

D(3, D(2, D(1,1))) = D(3, D(2,2)) = D(3,7) = 10

path: 1 - 2 - 7 - 10 - 12

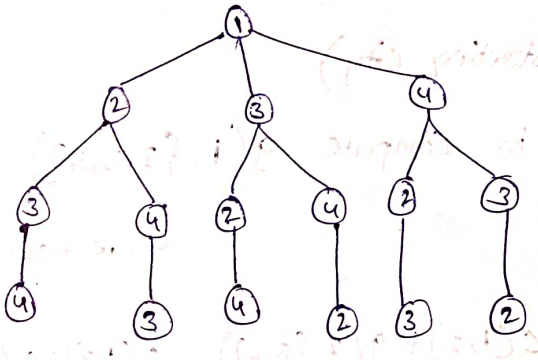


Travelling Salesman Problem

Problem Defn:

The TSP has to organize a tour in such a way that all the cities (except the starting city) must be visited exactly once and return to the home (starting city) with an objective of minimizing the cost of the tour.

(8) Thus if there are 4 cities (including starting city) brute force approach will be



$\longrightarrow (n-1)!$

$\therefore$ TC of brute force $= (n-1)! = O(n^n)$

The path is like $1 \text{——} x \text{——} 1$

optimality substructure.

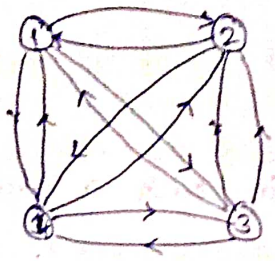
From tree, we can show overlapping subproblems.

Recurrence formula for TSP:

Let $e[i,j]$ be a cost adjacent matrix

let $g(i,s)$ represent cost of the tour from vertex i, visiting all the vertices in the set s exactly once and terminating the tour at v_0 (home city)

Consider



Let cost adjacency matrix C be

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

for $v_0 = 1$ (starting city)

we need to compute $g(1, \{2,3,4\})$

↳ we need to minimize this.

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

∴ In general,

$$g(i,S) = \min\{c(i,x) + g(x, S-\{x\})\}$$

such that $x \in S$ & $\langle i,x\rangle \in E$

$$g(i,S) = c(i,v_0) \quad \text{if } S = \phi$$

↳ $i \neq v_0$

↳ termination condition for recursion.

we use $J(i,S)$ to compute the path

$J(i,S) = x$

Now let us calculate $g(1,\{2,3,4\})$

$|S|=0$:

$g(2,\phi)=5$

$g(3,\phi)=6$

$g(4,\phi)=8$

$|S|=1$:

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

$g(2,\{4\}) = \min\{c(2,4)+g(4,\phi)\} = 10+8=18$, $J(2,\{4\})=4$

$g(3,\{2\}) = \min\{c(3,2)+g(2,\phi)\} = 13+5=18$, $J(3,\{2\})=2$

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

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

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

$|S|=2$:

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

$= \min\{9+20, 10+15\}$

$= 25$

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

$g(3,\{2,4\}) = \min\{13+18, 12+13\}$

$= 25$

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

$g(4,\{2,3\}) = \min\{8+15, 9+18\}$

$= 23$

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

$|S| = 1$:

$g(1, \{2,3,4\})$

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

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

$= 35$

~~$J(1,\{3,3$~~ $J(1,\{2,3,4\}) = 2$

Path construction :

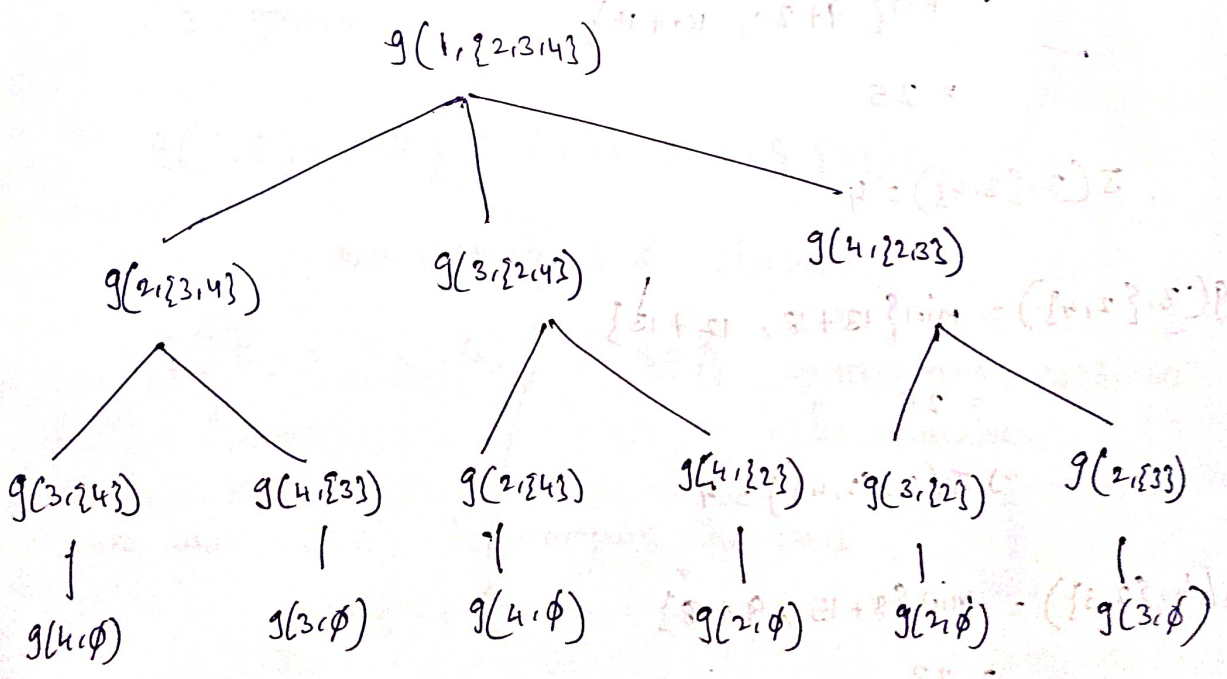
If $J(i,s) = x$ then from ~~x~~ x we go to $J(x, S-x)$

$J(1,\{2,3,4\}) = 2$

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

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

$\therefore$ path : 1 - 2 - 4 - 3 - 1



For larger problems we can see overlapping subproblems.

→ The algorithm for TSP is Held & Carp Algorithm

Time Complexity : $O(n^2 \cdot 2^n)$

∵ we have 2^n subsets and for every subsets we need to check for possible edges (Max possible edges = n^2)

Space Complexity : $O(n \cdot 2^n)$

↳ To store values of g

because for every possible subset s and for every vertex except start vertex we calculate and store

$g(v, s)$ ∴ $O(n \cdot 2^n)$

n 2^n

$(x, S-x)$

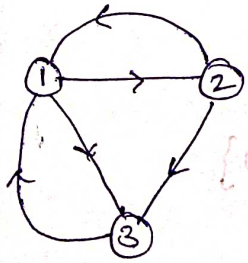
→ TSP is an intractable Problem

∵ it doesn't have a polynomical TC algorithm so far.

∴ TSP is an NP problem (NP-complete).

16/10/20

All Pairs Shortest Paths (Flyod Warshall's Algorithm)



c	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$C(i,j) \longrightarrow$ edge cost

Let $A^k(i,j)$ represent the cost of the path from vertex 'i' to vertex 'j' NOT going through the intermediate vertex greater than k.

Assume vertices are sequentially numbered from 1 to n $\langle 1,2,3,\ldots n \rangle$

$\therefore A^k(i,j) = \min\{$ Path goes through k, path doesn't go through k $\}$

If path goes through k

now here highest intermediate vertex possible = k-1

$\therefore$ If path goes through k, it will be

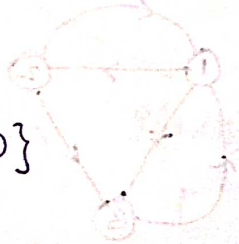
$$A^{k-1}(i,k) + A^{k-1}(k,j)$$

$$\therefore A^k(i,j) = \min\{ A^{k-1}(i,k) + A^{k-1}(k,j),\ A^{k-1}(i,j) \}, \quad k \neq 0$$

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

Eg:

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0



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

$= 4$

$A^1(1,3) = 11 \quad (1 \to 1 \to 2)$

$A^1(2,1) = 6 \quad (2 \to 1 \to 1)$

$A^1(2,3) = \min\{ A^0(2,1) + A^0(1,3),\ A^0(2,3) \} = \min\{17, 2\} = 2$

$A^1(3,1) = 3$

$A^1(3,2) = \min\{3+4, \infty\} = 7$

$A^1(3,3) = 0$

$\therefore$

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

$\Rightarrow$

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

Algo Floyd-Warshall (G, n, e, C, A)

// C[1..n, 1..n] --- cost adjacency matrix

// A[1..n, 1..n] --- path cost adjacency matrix

{

1. for i ← 1 to n
   for j ← 1 to n
   A[i,j] = C[i,j];

TC: $O(n^3)$

SC: $O(n^2)$

2. for k ← 1 to n
   for i ← 1 to n
   for j ← 1 to n
   A[i,j] = min{A[i,j], A[i,k] + A[k,j]}

}

→ ~~The transi~~

Transitive closure:

→ Let A be the adjacency matrix of a directed/undirected graph. The transitive closure A^* of A be a matrix with the property that $A^*(i,j) = 1$, iff G has a directed path/path containing atleast one edge from vertex i to vertex j, otherwise $A^*(i,j) = 0$

→ Thus, transitive closure of a matrix, represent graph with n vertices, can be computed in $O(n^3)$ time.

Reflexive Transitive Closure:

Let 'A' be adjacency matrix of graph G and matrix A^+ be reflexive transitive closure of graph G defined as.

$A^+(i,j) = 1$ iff G has a path containing 0 or more edges from i to j

$A^+(i,j) = 0$, otherwise

0/1 knapsack:

Knapsack capacity: m

n-object: $\langle o_1, o_2, \cdots o_n \rangle$

- weight (w_i)
- Profit (P_i)
- Decision (x_i)
- Explicit Constraint: $\sum w_i > m$

Objective Func: Maximize the total profit ~~that we~~ subjected to the condition that total weight being put into the knapsack doesnot exceed its capacity.

i.e., Maximize $\sum_{i=1}^{n} P_i x_i$ subjected to $\sum_{i=1}^{n} w_i x_i \leq M$

where $x_i = 0/1$

Approach I:

Let $f_n(m)$ represent the profit with n-objects and a knapsack capacity of 'M'.

$f_n(m) = \max\{$ profit by including the object, profit by excluding the object $\}$

$$\therefore f_n(m) = \max\left\{ \underset{(x_n=1)}{P_n + f_{n-1}(m-w_n)}, \underset{(x_n=0)}{f_{n-1}(m)} \right\}$$

$f_0(x) = 0;$

Eg: $n=3;\ M=6;\ \langle P_1, P_2, P_3 \rangle = \langle 1, 2, 5 \rangle$

$\langle w_1, w_2, w_3 \rangle = \langle 2, 3, 4 \rangle$

$\langle x_1, x_2, x_3 \rangle = ?$

Tuple method:

Let f_n be represented by $S^n = \{(P, w)\}$

$P \rightarrow$ Profit

$w \rightarrow$ weight put into the knapsack.

$\therefore S^0 = \{(0,0)\}$.

S_1^1 is obtained by adding $P_1 \& w_1$ to S^0

i.e. $S_1^1 = \{(1,2)\}$

Now we merge S^0 & S_1^1 to obtain S^1.

$$S^1 = \{\underset{x_1=0}{(0,0)}, \underset{x_1=1}{(1,2)}\}$$

S_1^2 is obtained by ~~action~~ adding (P_2, w_2) to S^1

$$S_1^2 = \{(2,3)(3,5)\}$$

$x_1 x_2$: 00 01 10 11

$$S^1 \quad S_1^2 \rightarrow S^2 = \{(0,0)\ (1,2)\ (2,3)\ (3,5)\}$$

not feasible to add (5,4)

not feasible to add (5,4)

Now

$$S_1^3 = \{(5,4)\ (6,6)\}$$

$$S^2 \quad S_1^3 \rightarrow S^3 = \{(0,0)\ (1,2)\ (2,3)\ (5,4)\ (6,6)\}$$

↳ Here we have discarded (3,5) since (5,4) gives max profit with less weight.

This is called Purging rule.

∵ (3,5) will never lead to optimal soln.

Purging rule:

Let (P_i, w_i) & (P_j, w_j) be tuple to be merged

if $(P_i < P_j)$ and $(w_i > w_j)$ then the tuple (P_i, w_i) may be purged (removed).

Computing values of x_i's:

we got (6,6) as answer

$(6,6) \in S^3$ but $(6,6) \notin S^2$

⇒ we added O_3 i.e., (5,4) is added

$\therefore x_3 = 1$

$(6,6) - (5,4) = (1,2)$

Now $(1,2) \in S_2^2$ & $(1,2) \in S_2^1$ & $(1,2) \notin S^0$

$\Rightarrow x_1 = 1$

$x_2 = 0$

$\therefore (x_1 x_2 x_3) = (1,0,1)$

∴ If $(P,w) \in S^i$ & $(P,w) \notin S^{i-1}$ then $x_i = 1$ & $(P,w) \leftarrow (P-P_i, w-w_i)$

if $(P,w) \in S^i$ & $(P,w) \in S^{i-1}$ then $x_i = 0$

Eg: $n=4$; $\langle P_1,P_2,P_3,P_4 \rangle = \langle 60, 28, 20, 24 \rangle$

$M=11$; $\langle w_1,w_2,w_3,w_4 \rangle = \langle 10, 7, 4, 2 \rangle$

I. Greedy fractional knapsack:

$P_i/w_i : \langle 6, 4, 5, 12 \rangle$

∴ $x_4 = 1 \Rightarrow m = 11-2 = 9$

$x_1 = \frac{9}{10} = 0.9$

∴ $P = (0.9)(60) + (1)(24)$

$= 54 + 24 = 78$

II. 0/1 knapsack with greedy approach

$P_i/w_i : \langle 6, 4, 5, 12 \rangle$

$x_4 = 1 \Rightarrow m = 11-2 = 9$

$w_1 = 10$ not possible to include

∴ $x_1 = 0$

$x_3 = 1 \Rightarrow m = 9-4 = 5$

$w_2 > m$ ∴ $x_2 = 0$

∴ $P = 0 + 20 + 24 = 44$

III : 0/1 knapsack using DP

$S^0 = \{(0,0)\}$

$S^1_1 = \{(60,10)\}$

$S^1 = \{(0,0)\ (60,10)\}$

$S^2_1 = \{(28,7)\}$ — Merge → $S^2 = \{(0,0)\ (28,7)\ (60,10)\}$

$S^3_1 = \{(20,4)\ (48,11)\}$ — (+) →

$S^3 = \{(0,0)\ (20,4)\ (28,7)\ (60,10)\}$

$S^4_1 = \{(24,2)(44,6)(52,9)\}$

$\Rightarrow S^4 = \{(0,0)(24,2)(44,6)(52,9)(60,10)\}$

$\therefore$ (60,10) is optimal

$(60,10) \in S^4$; $(60,10) \in S^3$; $(60,10) \in S^2$; $(60,10) \in S^1$; $(60,10) \in S^0$

$x_4 = 0$; $x_3 = 0$; $x_2 = 0$; $x_1 = 1$

$\therefore P = (1)(60) = \underline{60}$

Approach II : (tabulation / bottom up approach)

KNAP → $k(n, m)$

n → no of obj ; m → knapsack capacity

$k(n,m) = k(n-1,m)$, $w[n] > m$

$= \max\{k(n-1,m), k(n-1, m-w_n)+P_n\}$, $w[n] \leq m$

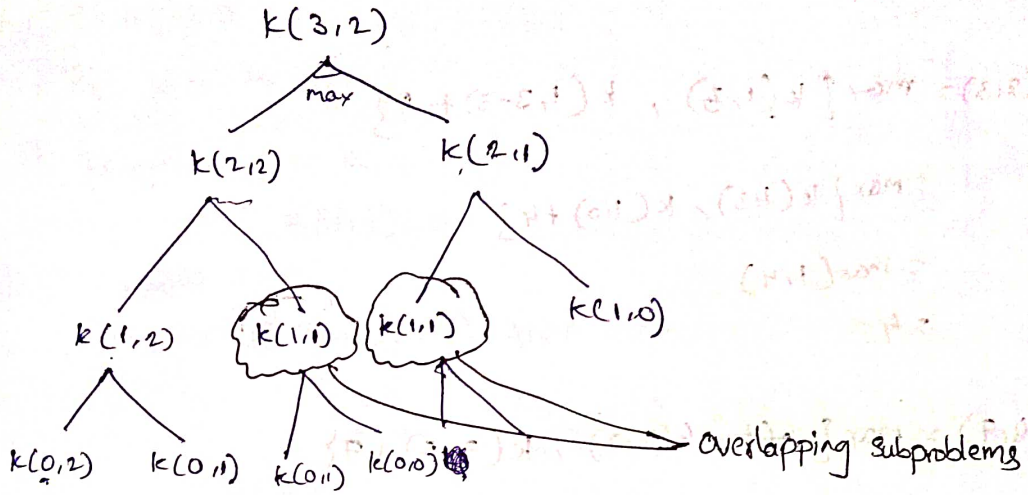
($x_n = 0$) ($x_n = 1$)

optimality substructure.

Eg: n=3; m=2;

w[ ] = {1, 4, 1}

P[ ] = {5, 10, 15}



Boundary conditions:

$$k(0,x) = 0; \quad k(y,0) = 0;$$

Eg: m=7; n=4; $P_i = \langle 1,4,5,7 \rangle$ $w_i = \langle 1,3,4,5 \rangle$

we can solve this problem using a 2-dimensional matrix.

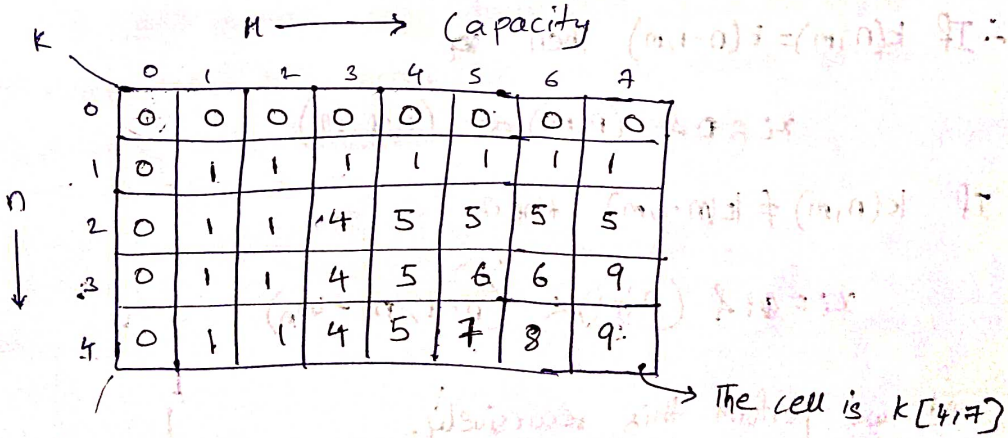
↓

k[n+1, m+1]

~~index starts from~~

m=7, n=4 ⟹ k[5,8]

~~Fabool~~



k(1,1) = max{ k(0,1), k(0,0)+1 } = 1

k(1,2) = 1

$k(2,1) = \max\{ k(0,1), k(1,-2)\}$

(not possible)

$= k(1,1)$

$= 1$

$k(2,3) = \max\{ k(1,3), k(1,3-3)+4\}$

$= \max\{ k(1,3), k(1,0)+4\}$

$= \max(1,4)$

$= 4$

$k(4,7) = \max\{ k(3,7), k(3,2)+7\}$

$= \max\{ 9, 8\}$

$= 9$

Calculating x_i values from table

$k(n,m)$ is obtained from either $k(n-1,m)$ cell or $k(n-1, m-w_n)$ cell.

If obtained from $k(n-1,m)$ then $k(n,m) = k(n-1,m)$

If obtained from $k(n-1, m-w_n)$ then $k(n,m) = k(n-1, m-w_n) + P_n$

∴ If $k(n,m) = k(n-1,m)$ then

$x_i = 0$ & $(n,m) \leftarrow (n-1,m)$

If $k(n,m) \neq k(n-1,m)$ then

$x_i = 1$ & $(n,m) \leftarrow (n-1, m-w_n)$

and we perform this recursively.

```
Algo KNAP(n, m, w[], P[])
{
    declare k[n+1, m+1];
1.  for i ← 0 to n
        for j ← 0 to m
            if (i == 0 or j == 0)
                k[i, j] = 0
            else if (w[i] ≤ j) then
                k[i, j] = max{ k[i-1, j], k[i-1, j-w[i]] + P_i }
            else
                k[i, j] = k[i-1, j];
}
```

→ Time Complexity : $O(n*m)$

→ Time complexity with brute force : $O(2^n)$

However for large values of m, $O(n*m)$ is too high.

For example if $m = 2^n$

then TC of tabulation method = $O(n \cdot 2^n)$

and in this case brute force would be better.

Thus tabulation method is good for smaller values of n.

→ Space Complexity with tabulation method : $O(nm)$

17/10/20

Longest Common Subsequence:

Substring : grp of one or more characters taken from the string that are contiguous

Subsequence : group of one or more characters may not be contiguors nevertheless the relative order is same.

→ Every substring is a subsequence

→ For a string of length n,

no of substrings → $O(n^2)$

no of subsequences → $O(2^n)$

→ A subsequence that is common to given two string is called common subsequence.

The longest of all these common subsequences is called longest common subsequence.

Applications of LCS :

(i) Web search

(ii) DNA matching

(iii) Software matching

(iv) Plagiarism

Solution :

Given two strings X & Y of length 'n' & 'm' respectively. It is required to determine a subsequence of longest length that is common to both X & Y

$X = \langle x_1 x_2 \ldots x_n \rangle$ $Y = \langle y_1 y_2 y_3 \ldots y_m \rangle$

→ Let 'i' & 'j' be indices into the strings X & Y as shown.

→ Let L[i,j] represent the length of common subsequence of string X & Y.

Case I: Last characters are same

Eg: X = <G T T C C T A A T A>
0 1 2 3 4 5 6 7 8 9

Y = <C G A T A A T T G A G A>
0 1 2 3 4 5 6 7 8 9 10 11

so we need to find L[9,11]

if X[i] = Y[j] then

$$L[i,j] = 1 + L[i-1, j-1]$$

Case II: Last characters don't match

X = <G T T C C T A A T A>

Y = <C G A T A A T T G A G>

If X[i] ≠ Y[j]

∴ This means that LCS may terminate with last character of X or last character of Y or neither.

i.e., $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$

∴ if X[i] = Y[j]

$$L[i,j] = 1 + L[i-1, j-1]$$

if X[i] ≠ Y[j]

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

$$L[-1,j] = 0$$

$$L[i,-1] = 0$$

Tree approach to solve LCS:

X = ⟨ABBAB⟩ Y = ⟨ACBAB⟩

("ABAB") ↑ u ("ABB")

LCS("ABBAB", "ACBAB")

("ABB") 3 ↑ (1+)(B)

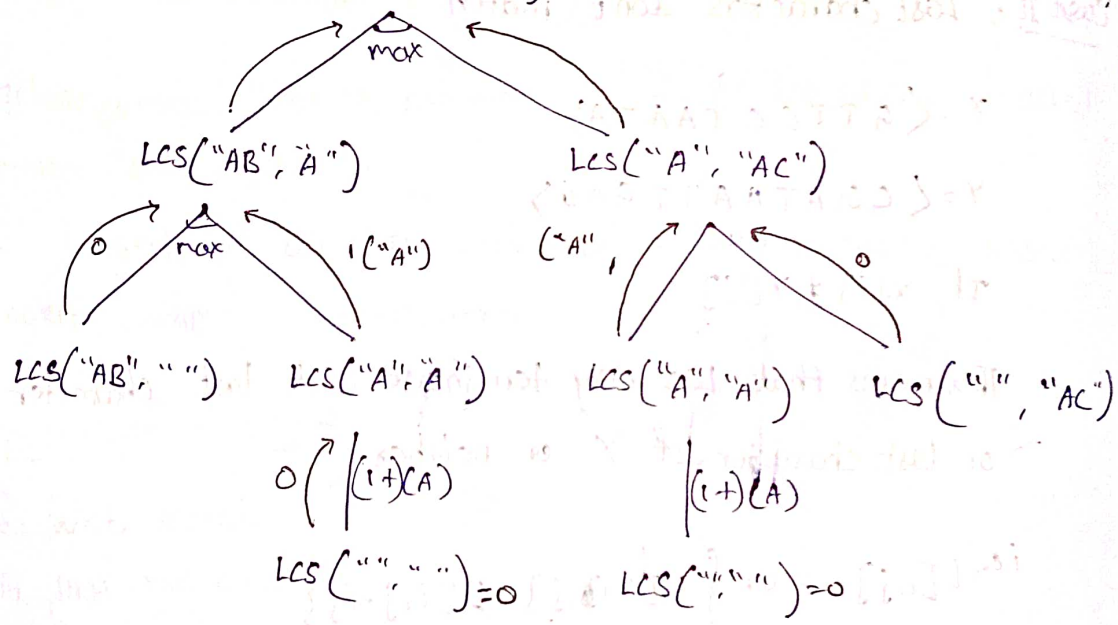
LCS("ABBA", "ACBA")

("AB") 2 ↑ (1+)(A)

LCS("ABB", "ACB")

("A") 1 ↑ (1+)(B)

LCS("AB", "AC")



Tabulation method (bottom up):

X = ⟨A B C B D A B⟩ (0 1 2 3 4 5 6)

Y = ⟨B D C A B A⟩ (0 1 2 3 4 5)

		-1	0 (B)	1 (D)	2 (C)	3 (A)	4 (B)	5 (A)
-1		0	0	0	0	0	0	0
0	A	0	0 ←	0 ←	0 ←	↖ 1	← 1	↖ 1
1	B	0	↖ 1	1 ←	← 1	← 1	↖ 2	← 2
2	C	0	↑ 1	← 1	↖ 2	← 2	← 2	← 2
3	B	0	↑ 1	← 1	↑ 2	← 2	↖ 3	← 3
4	D	0	↑ 1	↖ 2	← 2	← 2	↑ 3	← 3
5	A	0	↑ 1	↑ 2	← 2	↖ 3	← 3	↖ 4
6	B	0	↖ 1	↑ 2	← 2	↑ 3	↖ 4	← 4

→ This is the soln i.e., L[6,5]

$L[0,0] = \max\{L[-1,0], L[0,-1]\} = 0$

$L[0,1] = \max\{L[-1,1], L[0,0]\} = 0$

If case "II" applies then we take max {upper cell, left side cell}

If case I apples we take 1 + upper left diagonal cell's value.

$L[0,3] = 1 + L[-1,2] = 1+0 = 1$

$\therefore L[6,5] = 4$

To get the subsequence we backtrack while backtracking add corresponding symbol whenever you move diagonally upward.

$\therefore$ Lcs: B D A B

Algorithm LCS (x, y, n, m)

L[n+1, m+1] : array

{

1. for i ← 0 to n-1

~~for j ← 0 to m-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 m-1

if (x[i] = y[i])

L[i, j] = 1 + L[i-1, j-1];

else

L[i, j] = max {L[i-1, j], L[i, j-1]}

}

T.C: $O(n * m)$

S.C: $O(n * m)$

TC of LCS with brute force is $O(2^n \cdot m)$ or $O(n \cdot 2^m)$

Matrix Chain Product:

→ Given two square matrices of order n

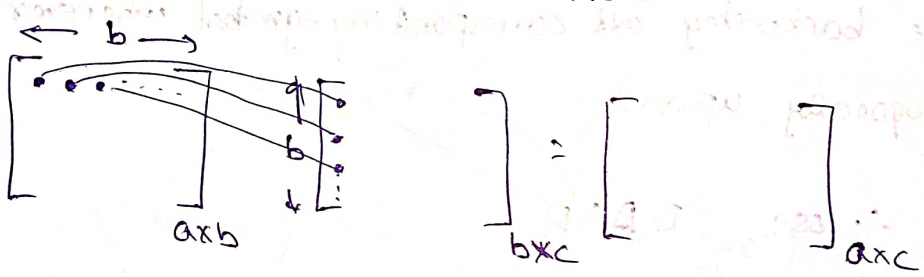
time to multiply: $O(n^3)$

operation: scalar multiplication

i.e., for product of two square matrices n^3 multiplications are required.

→ However two multiply two non-square matrices, the matrices have to be compatible.

i.e., no of colums of 1st matrix = no of rows of 2nd matrix



we need to compute axc element and computations of each element require b scalar multiplications.

⇒ we nee to perform axbxc no of multiplications

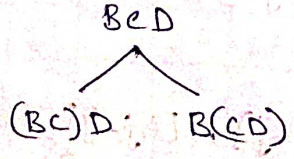
$\therefore TC = O(axbxc)$

~~Problem Statement:~~

Given a chain of matrices, it is required to multiply them to get a resultant product matrix.

- Eg: Consider A = BCD

B: 2x10; C: 10x50; D: 50x20;



(BC)D < BC : 2×10×50 = 1000 ; (BC)D : 2×50×20 = 2000 } = 3000

(BC) D
/ \
2×50 50×20

B(CD) < CD : 10×50×20 : 10000 ; B(CD) : 2×10×20 : 400 } = 104000

Objective function:

Minimizing the number of scalar multiplications.

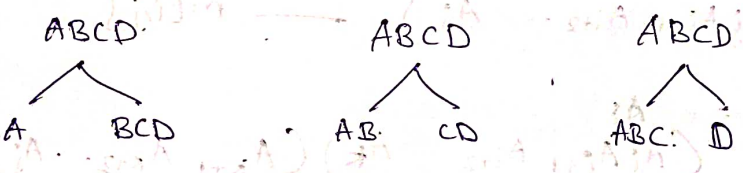
Problem stmt:

Given a chain of matrices $\langle A_1 A_2 A_3 \ldots A_n \rangle$ where matrix A_i is of order $P_{i-1} \times P_i$. It is required to fully parantesize the given chain of matrices such that the no of scalar multiplications is minimum.

ABC can be paranthesized in 2 ways

i.e., (AB)C & A(BC)

ABCD can be paranthesized in 5 way



∴ (A(BC(D))) ((AB)(CD)) ((AB)C)D

A((BC)D) (A(BC))D

→ Observing this we can say brute force method will have exponential time complexity.

Applications

(i) In mathematics

(ii) Satellite communication

(iii) Physics

Eg: $A_1 \rightarrow 3 \times 5 \longrightarrow P_0 \times P_1$

$A_2 \rightarrow 5 \times 8 \longrightarrow P_1 \times P_2$

$A_3 \rightarrow 8 \times 6 \longrightarrow P_2 \times P_3$

$A_4 \rightarrow 6 \times 2 \longrightarrow P_3 \times P_4$

$A_5 \rightarrow 2 \times 3 \longrightarrow P_4 \times P_5$

$A_6 \rightarrow 3 \times 7 \longrightarrow P_5 \times P_6$

$\langle A_1 A_2 \cdots A_6 \rangle \longrightarrow P_0 \times P_6$

→ Let $A_{i..j}$ be the matrix that results from multiplication of matrices $\langle A_i A_{i+1} A_{i+2} \cdots A_j \rangle$

Eg: $A_{1..6} = \langle A_1 A_2 A_3 A_4 A_5 A_6 \rangle$

→ Let $m[i,j]$ be the no of scalar multiplications needed to get the matrix $A_{i...j}$

→ Any optimal parenthesization must split the given chain of matrix about matrix A_k such that the no of scalar multiplications is minimum

$(A_i A_{i+1} A_{i+2} \cdots A_j) \longrightarrow m[i,j]$

$= (A_i A_{i+1} A_{i+2} \cdots A_k)(A_{k+1} A_{k+2} \cdots A_j)$

$P_{i-1} \times P_k \qquad P_k \times P_j$

Here k value ranges from i to j-1

i.e., j-i splits are possible.

∴ If split at A_k

$\therefore m[i,j] = m[i,k] + m[k+1,j] + (P_{i-1} * P_k * P_j)$

$$\therefore m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + (P_{i-1} * P_k * P_j) \}$$

$$m[i,i] = 0$$

Tabulation method: (bottom up)

Eg: $A_1 \rightarrow 3 \times 5$

$A_2 \rightarrow 5 \times 8$

$A_3 \rightarrow 8 \times 6$

$A_4 \rightarrow 6 \times 2$

so we need to compute $m[1,4]$

So in tabulation method we compute

j-i=0

$m[1,1] = m[2,2] = m[3,3] = m[4,4] = 0$

j-i=1

$m[1,2] = 3 \times 5 \times 8$

$m[2,3] = 5 \times 8 \times 6$

$m[3,4] = 8 \times 6 \times 2$

j-i=2

$m[1,3] = \min\{ m[1,2] + m[3,3] + 3*8*6 , m[1,1] + m[2,3] + 3*5*6 \}$

Silly compute $m[2,3]$

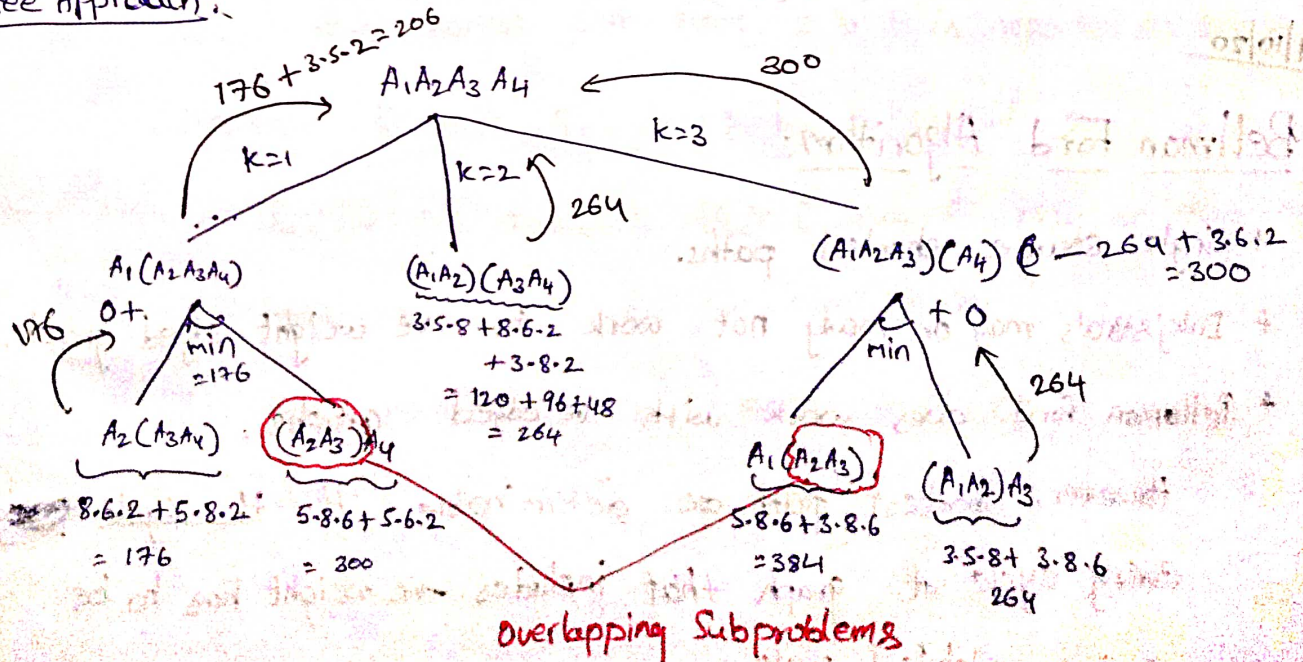
j-i=3

$m[1,4] = \min\{ m[1,1] + m[2,4] + 3*5*2,$

$m[1,2] + m[3,4] + 3*8*2,$

$m[1,3] + m[3,4] + 3*6*2 \}$

Tree Approach:



∴ Minimum cost = 206

optimal paranthesization = $(A_1 (A_2(A_3 A_4)))$

Given a chain of n matrices $\langle A_1, A_2 \cdots A_n \rangle$

The time complexity to multiply them with optimal paranthesization

$= O(n^3)$

Proof:

In tabulation method

$j-i = n-1 \longrightarrow$ no of comparisions $= n-1$

no of instances $= 1$

$j-i = n-2 \longrightarrow$ no of comparisions $= n-2$

no of instances $= 2$

$j-i = n-3 \longrightarrow$ no of comparisions $= n-3$

no of instances $= 3$

⋮

no of comparisions =

$$(n-1)(1) + (n-2)(2) + \cdots + (n-k)(k) + \cdots + (n-(n-1))(n-1)$$

$= O(n^3)$

19/10/20

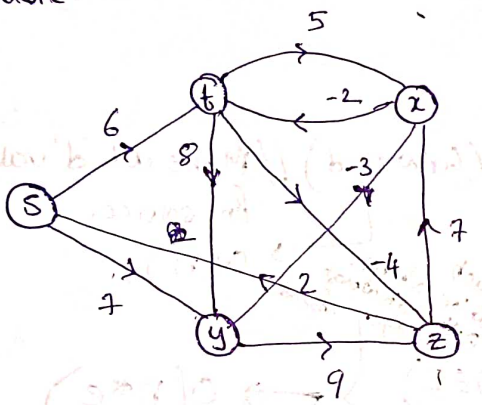
Bellman Ford Algorithm:

* Single source shortest paths.
* Dikjstra's may or many not work for -ve weight edged graph.
* Belloman Ford does work with -ve edged graphs, however, shortest path are determinable if every cycle of graph that includes -ve weight has to be a tve weighted cycle.

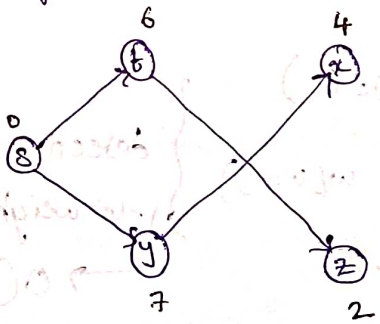
If the cycle containing negative edge is negative weighted then shortest path is not determinable.

Infact it is not determinable with any algorithm.

→ The shortest path of all the vertices that are reachable through negative weighted cycle is not determinable. there could be vertices that are not part of negative weighted cycle but still have their shorted path not determinable.



Using Dikjstra's algorithm



Here shortest path from s to t is reported as 6.

However the path s-y-x-t has a cost of 2.

Thus we say dijkstra may or may not have correct soln for -ve weight graph.

(∵ re-relaxation of path s-t (6) is never done in dijkstra's algorithm)

Bellman ford approach:

An unrestricted shortest path b/w a pair of vertices in a graph having n-vertices cannot have more than (n-1) edges assuming cycle/loop free path.

Algo BellmanFord(G, V, E, w, s)

// w[1...n][1...n] – cost adjacency matrix

// d[1...n] – result array

{

1. initialize – single_src(G, w, s, d) // Make all d value to ∞ except for source. → O(n)

2. for i ← 1 to (n-1) → In each iteration shortest path with atmost i edges is found.

   for (each edge (u, v) ∈ E)

   relax(u, v, w, d);

   → O(n * e)

3. for (each edge (u, v) ∈ E)

   if [d[v] > d[u] + w[u, v]]

   return FALSE

   → detecting the presence of -ve weight cycle. → O(e)

4. return TRUE // No -ve weight cycle reachable from source.

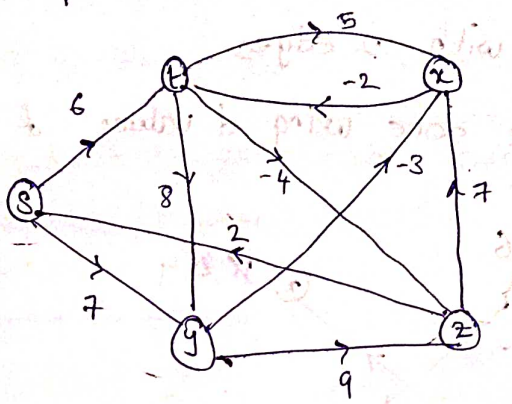
}

∴ BellmanFord algo detects -ve weight cycle if ~~exists from src~~ it is reachable from source.

→ Time complexity: O(n * e) → adjacency list
adjacency matrix → O(n³)

→ Space complexity: O(n)

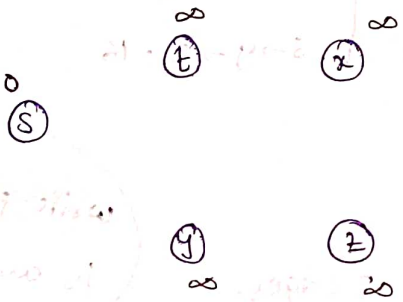
Consider the previous example again



~~Trace of point (2) in algo~~

Step 1:

Initialize d values:

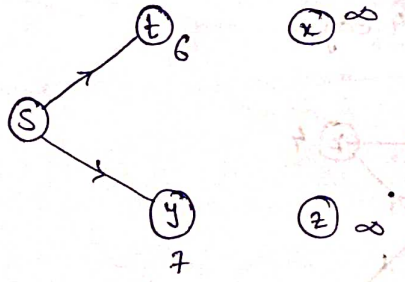


Refer Sahani for optimization on Bellmanford

Step 2: (Point '2' in algo - 1st iteration) (~~edge~~ paths with 1 edge)

→ no relaxation can be done using ~~value of~~ d values of t, x, y, z (∵ it is infinite)

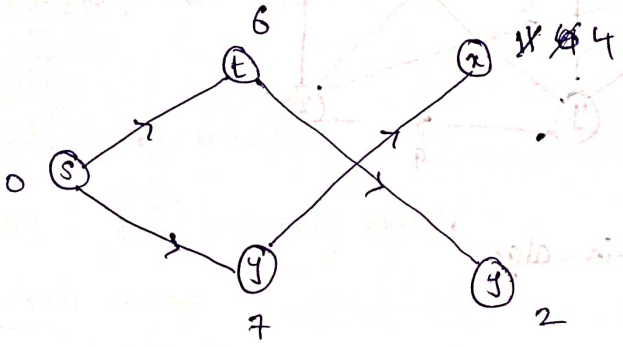
→ In this step relaxation can be done only from vertex s.



⇒ we perform this relaxation for every edge.

Step 3 :

* finding shortest paths with 2 edges
* Now relaxation can be done using d values of s,t,y



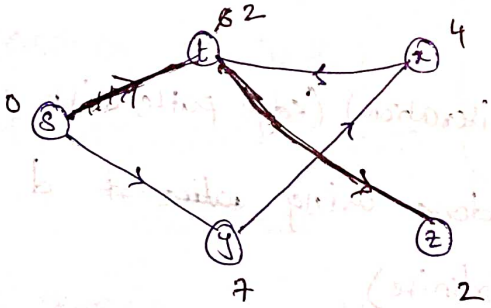
$S \to t \to x : 11$ | $S \to t \to z : 2$

$S \to y \to x : 4$ | $S \to y \to z : 16$

Step 4 :

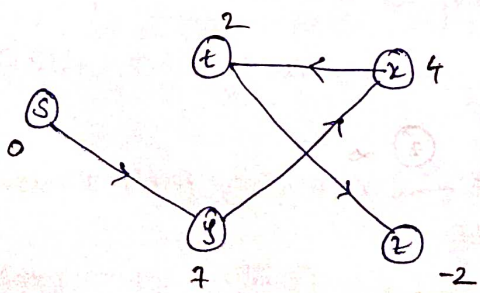
→ Finding shortest paths with 3 edges.

write path names to avoid confusion



Step 5 :

→ Find shortest paths with 4 edges



Derivation of formula:

Let c[i,j] be edge cost

$d^l[x]$ = cost of the path from source 's' to vertex 'x' with atmost 'l' edges.

$$\therefore d^l[x] = \min\left\{ d^{l-1}[x],\ \min_{\forall k \in V}\{ d^{l-1}[k] + c[k,x]\}\right\}$$

$d^{l-1}[x]$ → without l-1 edges

$d^l[x] = c[s,x]$

S ..... k — x

l-1 edges, 1 edge

$d[*]$

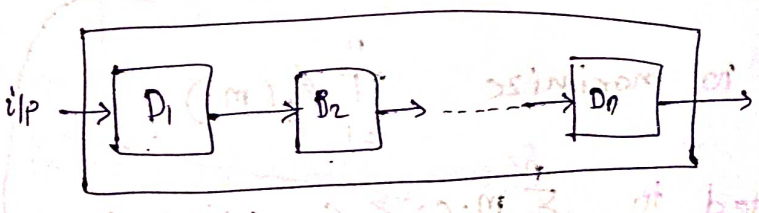
~~Reliability Design~~:

Reliable System Design

Design of an n-stage system

Here, design is in cascade

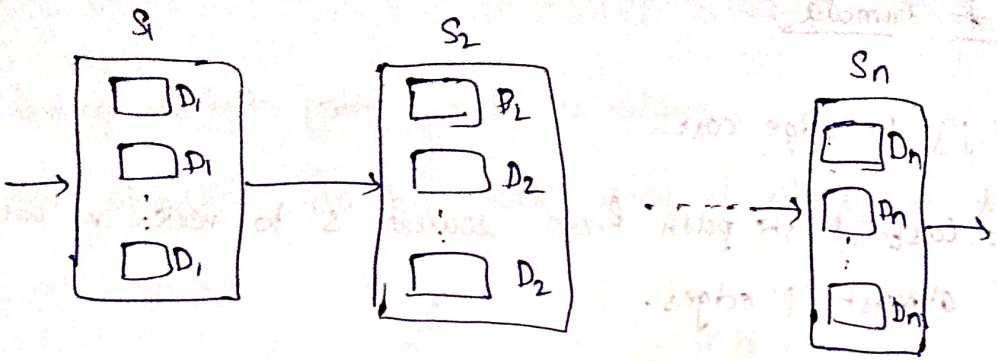
i.e., one design take o/p of another device as i/p



Every device Di has a cost c_i and reliability r_i

→ So the problem is to maximize the ~~size~~ reliability of the system such that cost of the design is less than given cost.

→ we increase reliability by adding more devices at each stage.



Let each stage 'S_i' has m_i copies of device D_i such that

$1 \leq m_i \leq u_i$

where u_i is an upper bound.

→ Total cost of the system = $\sum_{i=1}^{n} m_i c_i$

→ Reliability of the system :

reliability of S_1

$= 1 - (1 - r_1)^{m_1}$

reliability of S_2

$= 1 - (1 - r_2)^{m_2}$

∴ Reliability of the design = $\prod_{i=1}^{n} [1 - (1 - r_i)^{m_i}]$

Problem Defn:

Thus we need to maximize $\prod_{i=1}^{n} \phi_i(m_i)$

subjected to $\sum_{i=1}^{n} m_i c_i \leq C$ → given cost

where $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$ & $1 \leq m_i \leq u_i$

$$u_i = \left\lfloor \frac{C - \sum c_j + c_i}{c_i} \right\rfloor$$

Eg: Consider 3 stage system with

$$C = 105 ; \quad \langle c_1 c_2 c_3 \rangle = \langle 30, 15, 20 \rangle ; \quad \langle r_1 r_2 r_3 \rangle = \langle 0.9, 0.8, 0.5 \rangle$$

→ Let $f_n(C)$ represent reliability of an n-stage system with a total project cost of C;

$$f_n(C) = \max_{1 \le m_n \le u_n} \{ \phi_n(m_n) * f_{n-1}(C - c_n m_n) \}$$

$$f_0(x) = 1$$

Eg: $n = 3$; $C = 105$;

$\langle c_1 c_2 c_3 \rangle = \langle 30, 15, 20 \rangle$

$\langle r_1 r_2 r_3 \rangle = \langle 0.9, 0.8, 0.5 \rangle$

$$u_1 = \left\lfloor \frac{105 - 15 - 20}{30} \right\rfloor = 2$$

$$u_2 = \left\lfloor \frac{105 - 30 - 20}{15} \right\rfloor = 3$$

$$u_3 = \left\lfloor \frac{105 - 30 - 15}{20} \right\rfloor = 3$$

Let $f_n \sim S^n = \{(\text{reliability}, \text{cost})\}$

∴ initially we have

$$S^0 = \{(1, 0)\}$$

For calculation S^1 we computes S_1^1 & S_2^1

where S_j^i represents i^{th} stage & j^* copies of stage i devices.

$S_1' = \{(0.9, 30)\}$

$S_2' = \{(0.99, 60)\}$

$1-(1-0.9)^2$

$1-(0.1)^2$

$= 0.99$

using S_1' & S_2', we compute S'

$S' = \{(0.9, 30)\ (0.99, 60)\}$

Computing S^2:

we compute S^2 by computing S_1^2, S_2^2, S_3^2

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$ → removed in purging rule

$\phi_2(m_2=2) = 1-(0.8)^2 = 1-0.64 = 0$

$= 1-(1-0.8)^2 = 1-0.04 = 0.96$

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$

(X) ↳ from here we cannot add even 1 copy of D_3. so this path is not explored.

$\phi_2(3) = 1-(1-0.8)^3 = 1-(0.2)^3$

$= 1-0.008 = 0.992$

$S_3^2 = \{(0.8928, 75), (\quad)\}$

↳ infeasible

$\therefore S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Computing S^3 :

→ First we calculate S^3_1, S^3_2, S^3_3

$$S^3_1 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$$

$$\phi_3(2) = 1 - (1-0.5)^2 = 1 - 0.25 = 0.75$$

$$S^3_2 = \{(0.54, 85), (0.648, 100), (\ )\}$$

↳ infeasible

$$\phi_3(3) = 1 - (1-0.5)^3 = 1 - 0.125 = 0.875$$

$$S^3_3 = \{(0.63, 105)\}$$

↳ eliminated using purging rule

$$\therefore S^3 = \{(0.36, 65), (0.432, 80), (0.54, 80), (0.648, 100)\}$$

∴ Ans : (0.648, 100)

Computing no of copies at each stage

(0.648, 100) is present in S^3_2

$\Rightarrow m_3 = 2$

Now subtract $2 * C_3 = 40$ from 100

$100 - 40 = 60$

Now search for tuple $(-, 60)$ in S^2

i.e., $(0.864, 60)$

It is found in S^2_2 $\Rightarrow m_2 = 2$

Now search for $(-, 60-(2)(15)) = (-, 30)$ in S^1

i.e., found in S_1^1

$\therefore m_1 = 1$

$\therefore (m_1, m_2, m_3) = \langle 1, 2, 2 \rangle$

$\therefore$ reliability $= (0.9) * (1-(1-0.8)^2) * (1-(1-0.5)^2)$

$= 0.648$

Optimal Cost Binary Search Tree:

→ BST is used in compilers to maintain the keyword table. (contains all the keywords)

~~Cost of~~

Computing cost of BST:

Let T be a BST

Cost(T) = cost(successful searches) + cost(unsuccessful searches)

$$\text{Cost(successful searches)} = \sum_{i=1}^{n} \text{cost}(a_i)$$

where cost(a_i) is cost for searching a_i in T.

$\therefore$ cost(a_i) $\propto$ level(a_i)

where level(a_i) is the level of a_i in T

cost(a_i) = P_i * level(a_i)

& where P_i is probability of using a_i in the application.

$$\therefore \text{Cost(succ. searches)} = \sum_{i=1}^{n} P_i * \underbrace{\text{level}(a_i)}_{\text{element comparisions}}$$

→ Now let us compute the cost of unsuccessful searches.

Unsuccessful searches and of null links.

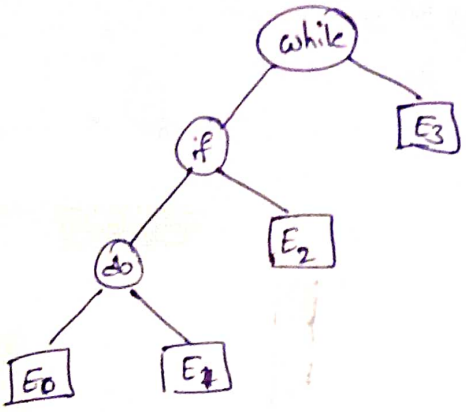
In a binary tree of 'n' nodes ~~no of~~

total links = 2n

no of used links = n-1

∴ no of null links = 2n-(n-1) = n+1

Ey: In BST with 3 nodes ~~wit~~ we have 4 null links.



whenever there it unsuccessful search it ends at ~~E0 or~~ E0/E1/E2/E3

Unsuccessful search ends at E0 if search word is < do

unsuccessful search ends at Ei if search word is (< if) and (> do)

~~∴ cost of an~~

$$\therefore \text{cost(unsuccessful search)} = \sum_{i=0}^{n} \text{cost}(E_i)$$

$$\text{cost}(E_i) \propto [\text{level}(E_i) - 1]$$

$$\text{Cost}(E_i) = q_i * (\underbrace{\text{level}(E_i) - 1}_{\hookrightarrow \text{no of element comparisions.}})$$

where q_i is ~~pro~~ probability of ~~se~~ searching a string from set E_i

$$\therefore \text{Cost (unsuccessful search)} = \sum_{i=1}^{n} q_i * (\text{level}(E_i) - 1)$$

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

$\sum_{1}^{n} P(i) + \sum_{0}^{n} q(i) = 1$

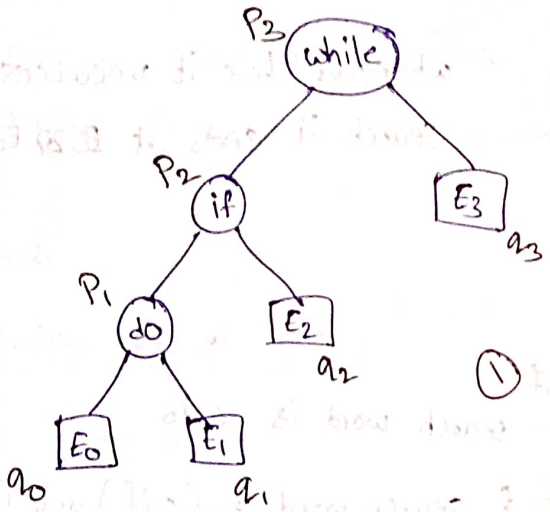
Formally $q(i)$ is probability of searching key x such that $a_i < x < a_{i+1}$

Eg: $n=3$; $\langle a_1 a_2 a_3 \rangle = \langle$ do, if, while $\rangle$

$\langle P_1 P_2 P_3 \rangle = \langle 0.5, 0.1, 0.05 \rangle$

$\langle q_0 q_1 q_2 q_3 \rangle = \langle 0.15, 0.1, 0.05, 0.05 \rangle$

Compute the cost below BST



Each E_i corresponds to class of string 'x' such that $a_i < x < a_{i+1}$ also we assume $a_0 = -\infty$; $a_{n+1} = +\infty$

Sol:

The level of root is 1.

$$\text{Cost(suc. search)} = \sum_{i=1}^{n} \text{level}(a_i) * P_i$$

$= P_1*1 + P_2*2 + P_3*3 = P_1*3 + P_2*2 + P_3*1$

$= 0.5 + 0.2 + 0.15 = (0.5)(3) + (0.1)(2) + (0.05)(3)$

$= 0.85 = 1.5 + 0.2 + 0.05$

$= 1.75$

$$\text{Cost(unsuc. search)} = \sum_{i=0}^{n} (\text{level}(E_i - 1)) * q_i$$

$= (0.15)(3) + (0.1)(3) + (0.05)(2) + (0.05)(1)$

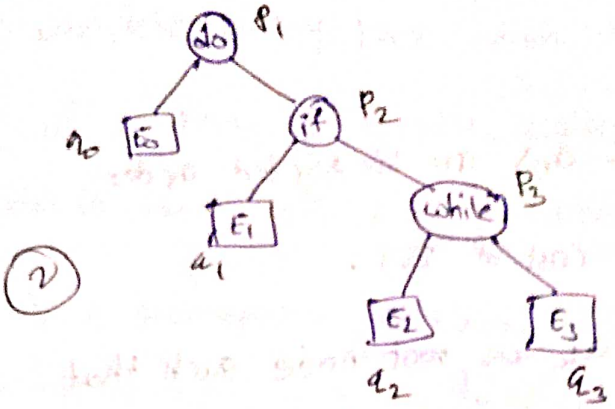
$= 0.45 + 0.3 + 0.1 + 0.05$

$= 0.9$

1.75
1.85

$\therefore$ Cost(BST) = ~~0.85~~ + 0.9 = ~~2.75~~ = 2.65

Eg: Compute cost of below tree



(2)

If Tree in previous example is considered, then ~~order of~~ arrangement of E_0, E_1, E_2, E_3 has to fixed in 2nd tree

Cost(suc) = $1 * 0.5 + 2 * 0.1 + 3 * 0.05$

= $0.5 + 0.2 + 0.15$

~~= 1.85~~ = 0.85

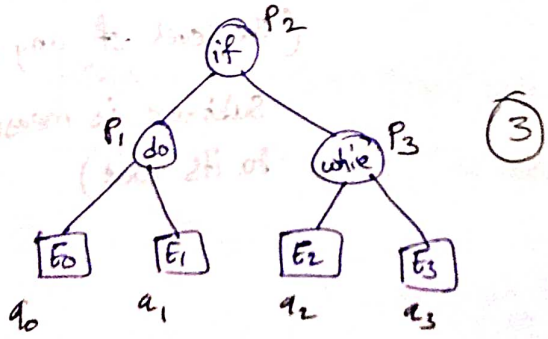
Cost(unsuc) = $(0.15)(1) + (0.1)(2) + (0.05)(3) + (0.05)(3)$

= $0.15 + 0.2 + 0.15 + 0.15$

= 0.65

$\therefore$ Cost(T) = 0.85 + 0.65 = 1.5

Eg: Compute cost of below tree



(3)

Cost(suc) = $(0.5)(2) + (0.1)(1) + (0.05)(2) = 1 + 0.1 + 0.1 = 1.2$

Cost(unsuc) = $(0.15)(2) + (0.1)(2) + (0.05)(2) + (0.05)(2)$

= $0.3 + 0.2 + 0.1 + 0.1 = 0.7$

Cost(T) = 1.9

even through Tree 3 is Complete it has higher cost than Tree 2.

Construction of BST of optimal cost:

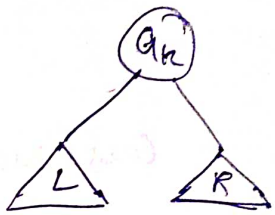
Let us consider n identifiers ~~with~~ $\langle a_1 \cdots a_n \rangle$ associated with probabilities $\langle P_1 \cdots P_n \rangle$. Let probabilities of external nodes be $\langle q_0 \cdots q_n \rangle$.

Assume $\{a_1\ a_2\ a_3 \cdots a_n\}$ are in sorted order.

Let cost(T) denote cost of BST.

Now we select a_k as root node such that

$$Cost(T) = \min_{1 \le k \le n} \{ P_k + COST(L) + COST(R) \}$$



Mathematical modelling:

$$C(0,n) = \min_{k=1 \text{ to } n} \{ P_k + c(0,k-1) + C(k,n) \}$$

$+\Delta_1$ $+\Delta_2$

~~However the~~

The value calculated are calculate for subtree considering it as a tree. So level number will be less. So we add Δ_1 & Δ_2 (The level of any identifier within subtree is measured with respect to its root)

Heap Algorithms:

Heap is used in the implementation of priority queue.

Defn: It is a complete binary tree with the property that the value of a node is greater(smaller) than the values of the nodes in their left & right subtrees.

→ In general it is implemented as binary heap. However we may use 3-ary or 4-ary or k-ary heap.

Operations:

(i) Create
(ii) Insert
(iii) Delete
(iv) Inc/Dcr key

Insertion:

(i) Inserting elements one ~~order~~ after other. (Max Heap)

Time complexity:

- Best case : O(n) (Dec order)
- worst case : O(nlogn) (Inc order)

* *
* Framework:
*

→ The max no of nodes at any level 'i' is 2^{i-1}. (root at level 1)

→ The no of level comparisions/movements of a node being inserted at level 'i' = i-1

→ The total no of level comparision for all nodes at level i

$$= (i-1)\,2^{i-1}$$

→ Time for all nodes at all levels

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

where k is last level

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

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

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

$$= \frac{1}{2}\left[2^{k+1}(k-2) + 4\right]$$

$$= 2^k (k-2) + 2$$

$\because \sum_{i=1}^{n} i \cdot 2^i = (n-1)\,2^{n+1} + 2$

For a full binary tree / complete binary tree

if 'n' is no of nodes

$$2^k = O(n) \;\&\; k = O(\log n)$$

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

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

22/10/20

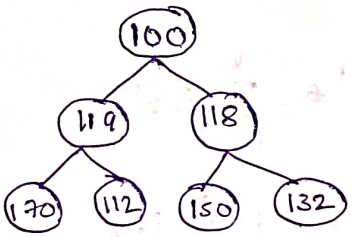
Build Heap (Heapify):

Heapify: The process of transforming a given complete binary Tree (with n elements) to a heap using method of adjust.

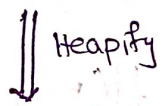
Adjustment procedure is carried out level by level starting from level k-1 then k-2 upto level 1.

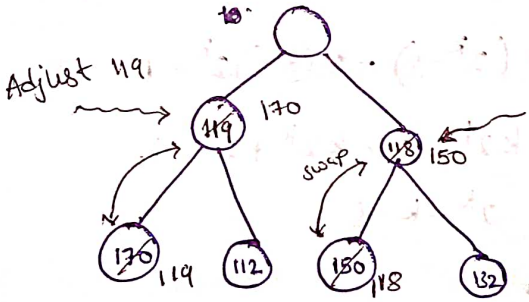
where k is last level

Eg: A: <100, 119, 118, 170, 112, 150, 132>



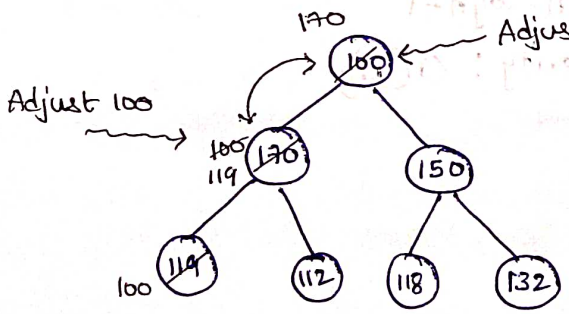
100	119	118	170	112	150	132

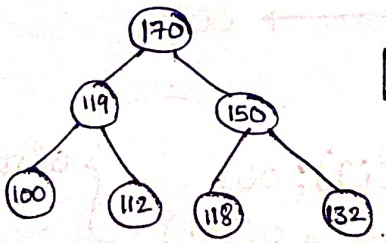




Adjust on 118

get largest child of 118 and compare with 118





170	119	150	100	112	118	132

Time Complexity:

→ Max no of nodes at any level is 2^{i-1}

→ Max no of level comparisions for a node that is getting adjusted at level i is $k-i$

where k is height (or) last level

→ for all node at level i, total comparisions $= (k-i)\,2^{i-1}$

→ Time Complexity = total comparision for all nodes at all levels

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

$$= k \sum_{i=1}^{k} 2^{i-1} - \sum_{i=1}^{k} i \cdot 2^{i-1}$$

$$= k \cdot \frac{2^k - 1}{2-1} - \frac{1}{2} \sum_{i=1}^{k} i \cdot 2^i$$

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

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

$$= k2^k - k - k2^k + 2^k - 1$$

$$= 2^k - k - 1$$

k = height $= O(\log n)$

$= n - \log n - 1$

∴ time complexity: $O(n)$

Heap Sort:

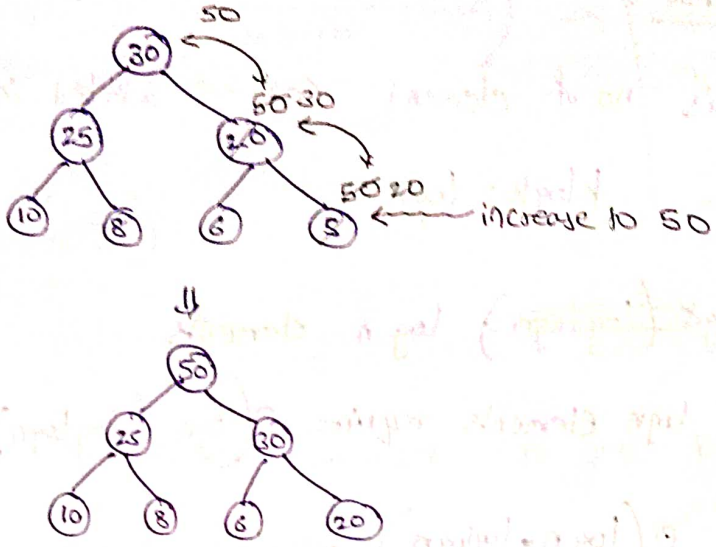
```
Algo Heap_Sort(n)
{
   1. Heapify(n)  ⟶ O(n)
   2. for i ← n downto 2
      {
         swap(A[1], A[i]); O(1)
         Adjust(A[1]); ⟶ O(log n)
      }
}
```

} $O(n \log n)$

∴ Time complexity of heap sort : O(n log n)

Increasing/ Decreasing value of a Node :

Consider Max heap



Thus if it is increase operation, the node may go from ~~root~~ leaf to root in worst case and if it is delete operation, the node may go from root to leaf.

∴ TC for inc/dec operation : O(log n)

(H/61)

1st smallest element cannot be below 1st level

2nd smallest element cannot be below 2nd level

:

Silly 7th smallest cannot be below 7th level.

∴ we need to search all 1st 7 levels.

∴ $2^7 - 1 = 128 - 1 = 127$ nodes have to be searched

∴ O(1)

(H/63) ⓓ

options are

a) $\log n$ b) $\sqrt{\log n}$ c) $O(1)$ d) $\frac{\log n}{\log \log n}$

option verification :

If k is no of element can be sorted in $O(\log n)$ then

$$k \log k = \log n$$

a) ~~$\log n (\log \log n)$~~ $\log n$ elements

$\log n$ elements require $O(\log n (\log \log n))$ time.

$$O(\log n \cdot \log \log n) > O(\log n)$$

$\therefore$ false

b) $\sqrt{\log n}$ elements

$\sqrt{\log n}$ elements require $O(\sqrt{\log n} (\log \sqrt{\log n}))$ time

$$O\left(\frac{1}{2} \sqrt{\log n} \cdot \log \log n\right) \neq O(\log n)$$

$\therefore$ false.

c) ~~Constant time is~~.

$O(1)$ --- constant elements.

to sort constant no of elements constant time is required

$$O(1) \neq O(\log n)$$

d) to sort $\frac{\log n}{\log \log n}$ we need $O\left(\frac{\log n}{\log \log n} \cdot \log \left(\frac{\log n}{\log \log n}\right)\right)$

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

$$= O\left(\log n\left(1 - \frac{\log(\log\log n)}{\log\log n}\right)\right)$$

$$= O\left(\log n - \underbrace{\frac{(\log n)(\log\log\log n)}{\log\log n}}_{< \log n}\right)$$

$$= O(\log n)$$

→ $\log\log\log n < \log\log n$

$\Rightarrow \frac{\log\log\log n}{\log\log n} < 1$

$\Rightarrow \frac{\log n \cdot (\log\log\log n)}{\log\log n} < \log n$

H/64 (i) Take 1st element from each list and build a heap heap. H

this takes $\log\log n$ time.

(ii) Delete an element from this heap and put it in the array (the array in which we want to store the sorted result) --- $O(\log\log n)$.

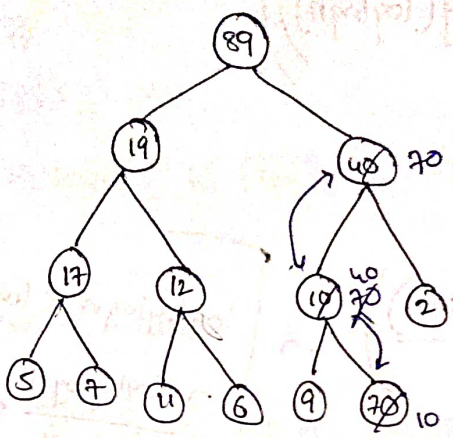
(iii) Now take and element from ~~array~~ list ~~to~~ whose element got added in the array and insert into heap H

--- $O(\log\log n)$.

Repeat ~~step~~ step (ii) & (iii) until all the elements ~~are~~ from the list are finished i.e., n times

~~i.e., $n\log\log n$~~

$\therefore TC : O(n\log\log n)$



∴ 2 interchanges

Graph Techniques (Traversals):

Traversal: Visiting all the nodes of the tree/graph in an order & processing the information only once is traversal.

Tree traversal vs Graph traversal:

→ The traversal of tree (inorder or preorder or postorder) is unique where as graph traversal need not to be unique.

Framework:

1. **Status of a node:**

During the traversal, any node must be in one of the 3 states

a) **E-Node:**

The node that is currently under exploration.

There is only one E-node during traversal.

b) Live - Node :

The Node which has not yet been explored fully or all of whose children has not yet been explored.

- stack in DFS & queue in BFS are used to store these node

c) Dead - Node :

The node which has been explored fully.

2. Timing values during traversal :

→ every node is associated with 2 two timing value

d = discovery time (Prev_value)

f = finishing time (Post_value)

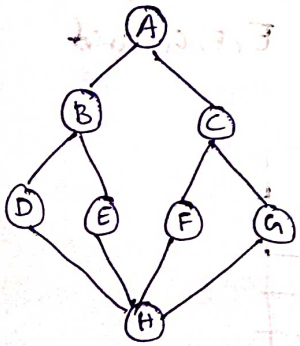
d & f > 0

d: the time at which node is first visited.

f: the time at which node becomes dead.

DFS :

1) DFS in undirected connected graph :



when node is currently under exploration (E-node) it is not pushed onto the stack. Node pushed only after it becomes fire node

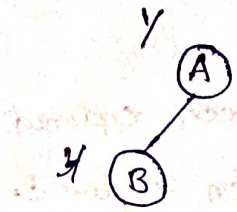
Let us start DFS from A

A node become E-node only after it is popped

So initially we give discovery time of A = 1

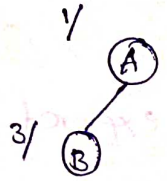
(However we can have any value greater than 0)

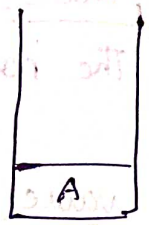
Now current A is E-Node.

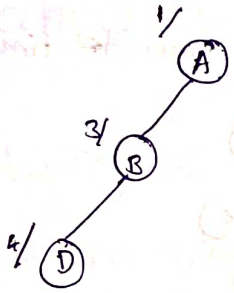


A ; B

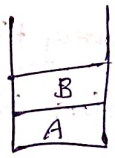
Now B becomes E-node, & A becomes live node and A is stored in stack



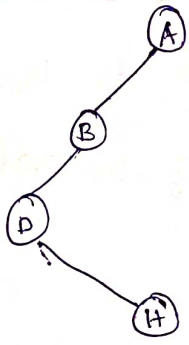




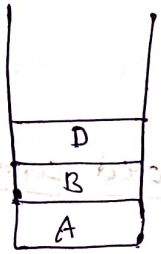
A ; B ; D



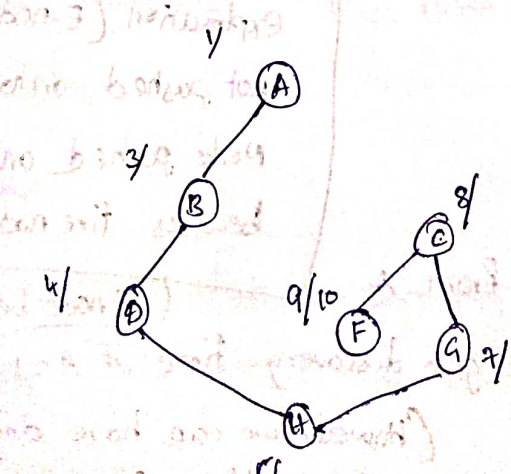
} live nodes

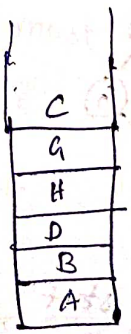


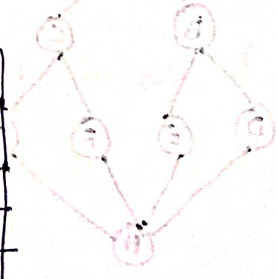
A ; B ; D ; H



Here D is parent of H and E, F, G are children of H







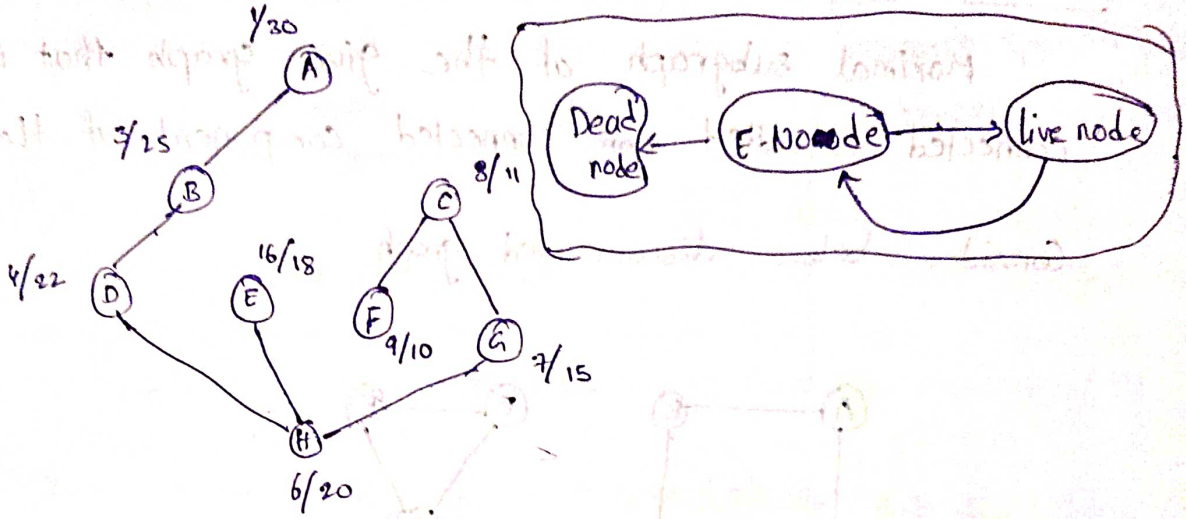
A ; B ; D ; H ; G ; C ; F

Now F 'b' has no adjacent unexplored notes.

So we assign it finish time and 'F' becomes dead node.

Now pop off 'c' from the stack.

Now c becomes the E-node



In this traversal 'H' becomes a E-node thrice

∴ traversal : A, B, D, H, G, C, F, E

Note :

→ ~~The graph~~ is ..

→ The undirected graph is connected

⇔ finish time of first node is the highest

⇔ ~~traversal finishes when the stack become empty for first time.~~

⇔ traversal finishes when the first node becomes dead node

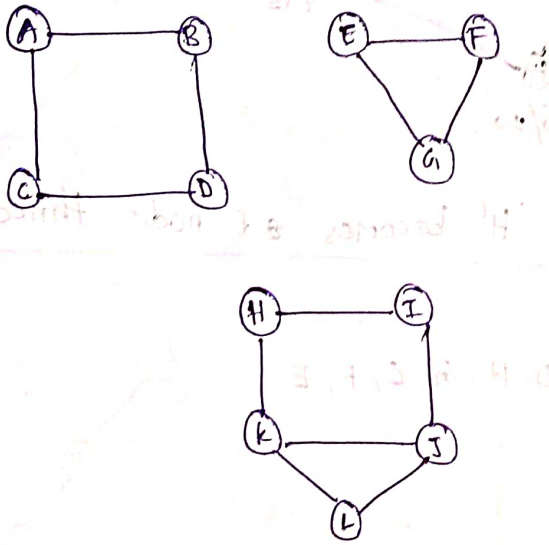
2) DFS in undirected disjoint (disconnected graph):

→ DFS when carried out on disconnected graph is called Depth First Tree (DFT)

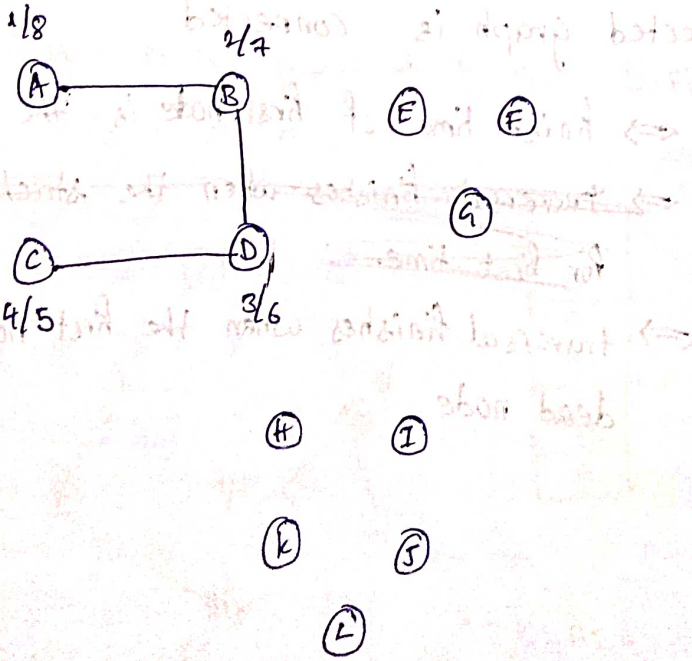
Connected Component:

Maximal subgraph of the given graph that is connected is called connected component of the graph.

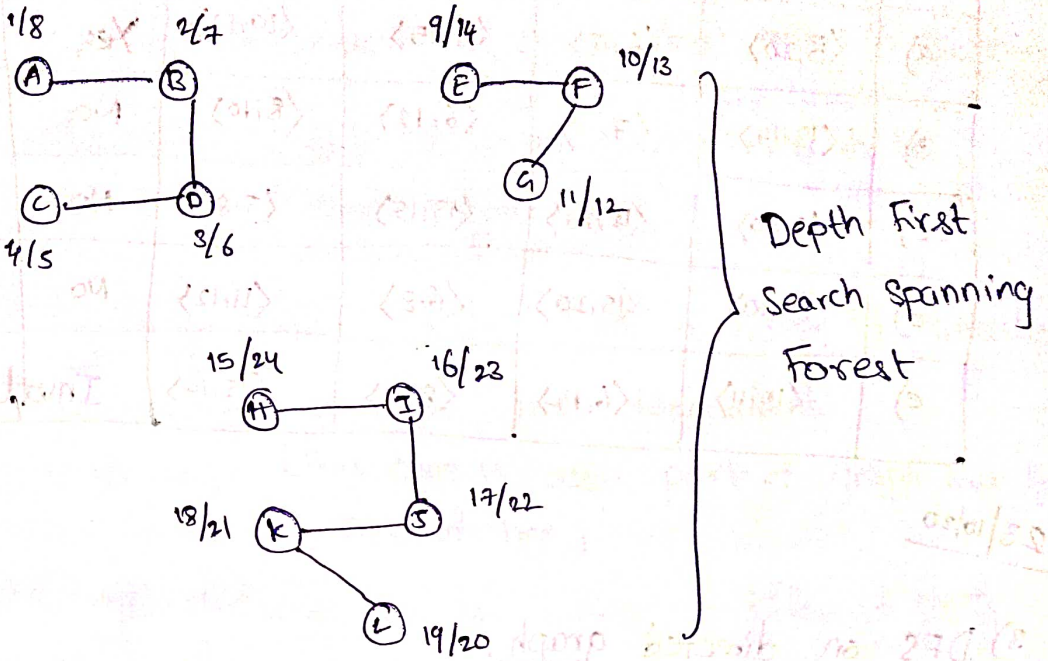
Consider below disconnected graph



Start traversal from any of the unexplored node



Now stack has become empty but still traversal is not finished since we have unexplored nodes.



Here finishing time of first node (A) is not the highest finishing time

∴ disconnected

Also graph is disconnected if there are more than one spanning trees.

no of disconnected components = no of spanning trees in depth first traversal.

Q. Given an undirected graph with 4 vertices (P, Q, R, S) and their (d, f) values as follows. Indicate for each option whether the graph is connected or disconnected and no of connected components.

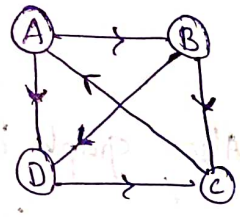
	Vertices				Connected	no of Connected components
	P	Q	R	S		
	<d,f>	<d,f>	<d,f>	<d,f>		
a)	<5,20>	<6,18>	<8,15>	<10,12>	Yes	1
b)	<13,14>	<7,11>	<6,12>	<8,10>	No	2 (QRS) (P)
c)	<3,10>	<18,20>	<13,15>	<5,8>	No	3 PSR Q
d)	<9,10>	<15,20>	<6,8>	<11,12>	No	4 RPSQ
e)	<10,11>	<6,12>	<8,13>	<15,14>	Invalid	

23/10/20

3) DFS on directed graph:

→ DFS when carried out over a directed graph leads to the following types of edges

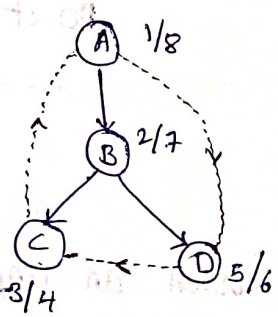
a) Tree edge:



The edges that are part of depth first search tree / forest

Eg: AB, BC, BD

b) Forward Edge:



It is an edge that leads from a node to its non-child descendent. (This edge is part of graph but not of spanning tree / forest)

Eg: AD is the only forward in the spanning tree

(∵ D is non-child descendent of A)

c) Back Edge:

It is an edge that leads from a node to its ancestor (This edge is present in graph but not in spanning tree). Eg: CA

Self loops are considered to be back edges

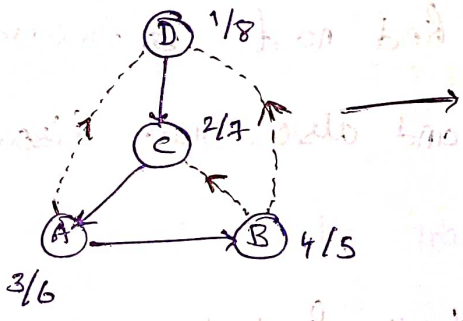
d) Cross Edge:

It is an edge that leads from a node to another node which is neither ancestor nor descendent.

Eg: DC (This edge is also part of graph but not of tree)

→ Cross edge may even be b/w vertices of different DFTs

Eg: Consider below below DFS on node 'D'

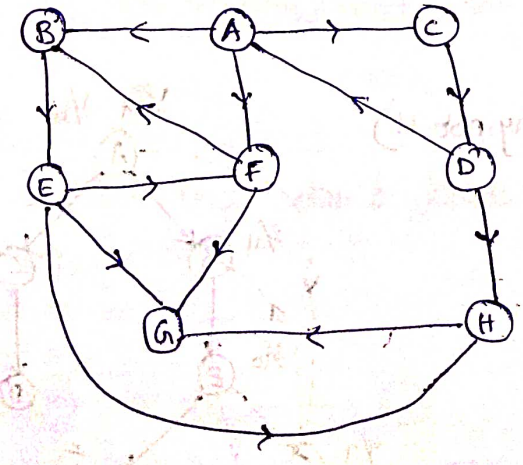


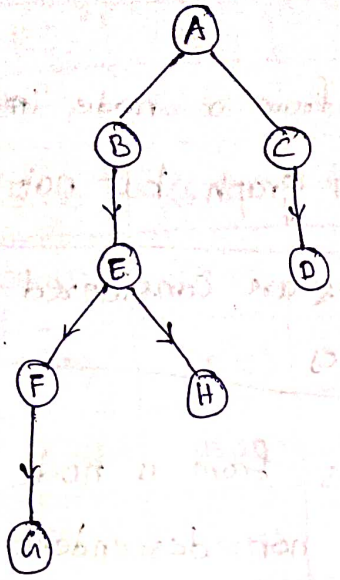
→ Fully decorated DFS spanning tree.
(The DFS spanning tree in which all the types of edges are drawn)

DC, CA, AB are tree edges

AD, BC, BD are back edges

Eg:





For above given graph and a spanning tree, check whether the given ~~spann~~ DFT tree is valid or not.

If it is valid find no of forward edges, backward edges and cross edges. and also mark discovery & finishing times.

Sol:

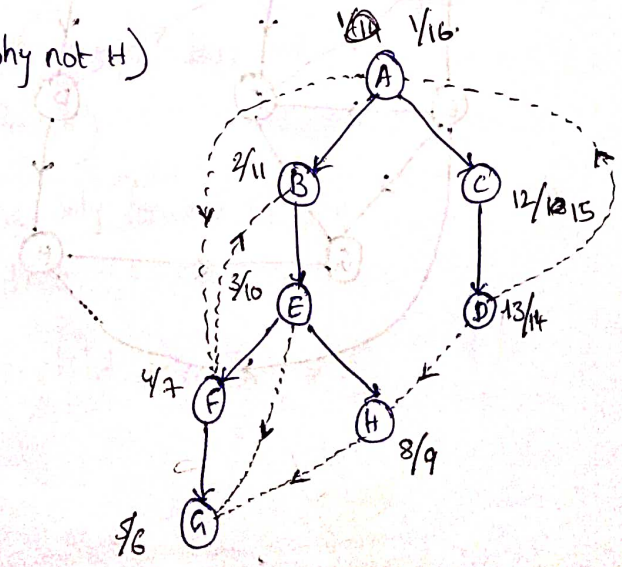
It is a valid depth first traversal spanning tree.

The order in which nodes are visited is

A; B; E; F; G; H; C; D

(Think why not C)

(Think why not H)



Forward Edges: AF; EG;

Back Edges: DA; FB;

Cross edges: DH; HG;

Parenthesis Theorem:

In any DFS of a directed graph $G=(V,E)$, for any two vertices 'u' & 'v' having an directed edge from u to v; exactly one of the condition holds

inclusive

I. `[ [ ] ]` → The (d/f) time interval of vertex 'v' is enclosed within (d/f) time interval of vertex 'u'; Then edge uv is either a forward edge (or) a tree edge

d[u] d[v] f[v] f[u]

II. `[ [ ] ]` → Same as previous, case but Here uv is formes a back edge.

d[v] d[u] f[u] f[v]

exclusive

III. `[ ] [ ]` → uv is a cross edge

d[u] f[u] d[v] f[v] (or)

d[v] f[v] d[u] f[u]

valid

backward edge

hing times.

4) DFS on DAG (Directed Acyclic Graph) (Precedence Graph)

Every DAG is charecterized by two kinds of vertices

i) Source vertices:

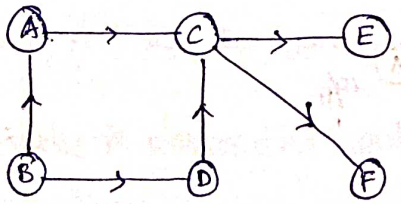
Vertices that doesn't have any precedences.

i.e., vertices that don't have any incoming edges

ii) Sink vertices:

Vertices that don't have any out bounded (outgoing) edges.

Eg:



B → source vertex

E, F → Sink vertices.

→ One of application of DAG is to perform topological sort (or) topological ordering.

→ Topological sort:

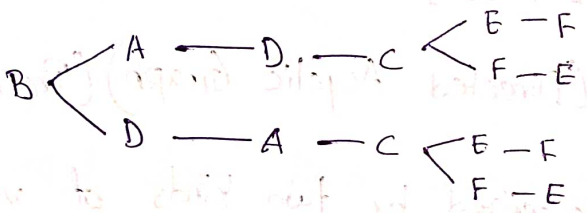
It is a linear ordering that exhibits the precedences of the vertices in the graph.

DAG can be viewed as task where edges represents dependencies or precedences and vertices represent activities.

Eg: In the example graph

- Activity C is should be done only after finishing activity A & D.

Eg: For example graph taken topological order is



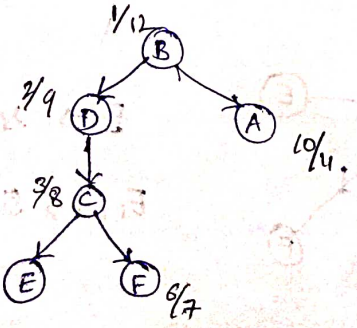
∴ topological sorts possible are

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

∴ no of topological orders possible = 4

Finding topological sort from DFS:

Consider starting DFS from 'B'

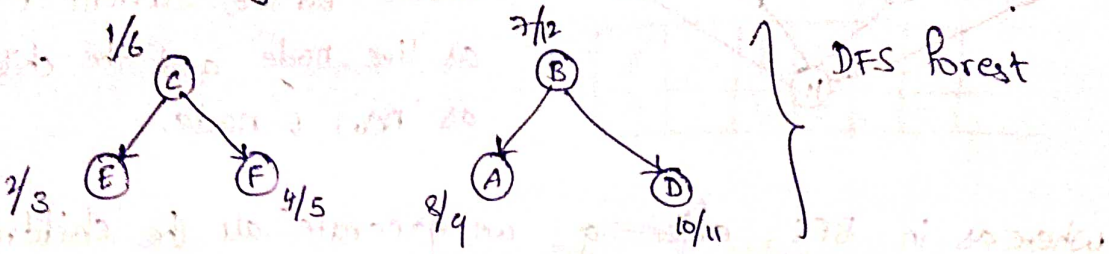


The decreasing order of finishing times gives topological sort.

i.e., BADCFE for this traversal

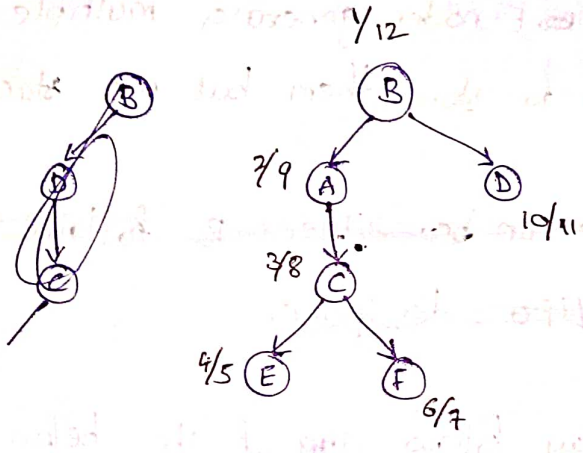
Note that to obtain topological sort we can start from any vertex.

Consider starting from vertex 'C'



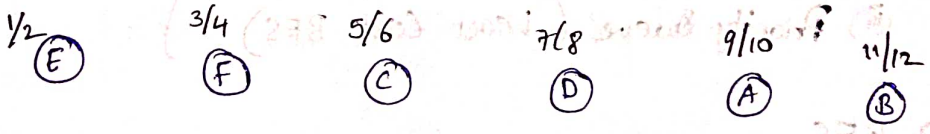
∴ Topological sort : BDACFE

Consider another traversal



∴ topological sort : BDACFE

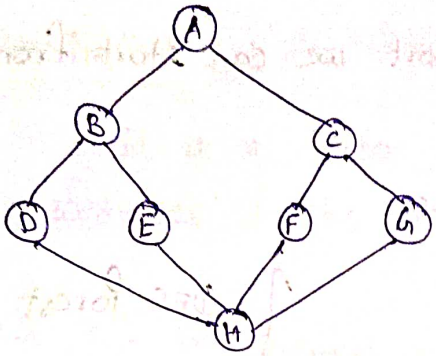
Consider another traversal



topological sort : BADCFE

```
Algo topological_sort()
{
    DFS(v); // v can be any vertex.
    Print nodes in descending order of their finishing times
}
```

Breadth First Search (BFS)



In DFS, when we explore a node, we generate a child node of current E-node and make the current E-node as live node and the child node as new E-node.

whereas in BFS, we generate all the children of current E-node and the unexplored nodes of these nodes are made live nodes (stored in the queue)

Since a single E-nodes generates multiple live nodes, we must use queue to store them but not stack.

~~→ The BFS algo can be either a follow either a FIFO discipline or LIFO discipline.~~

→ The Queue may follow any of the below 3 desciplines.

(i) FIFO (FIFO BFS) → Taken as default

(ii) LIFO (LIFO BFS or Dsearch)

(iii) Priority Queue (Least Cost BFS)

} Corresponding BFS

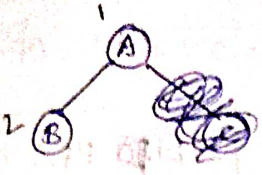
FIFO BFS

(A)

Q

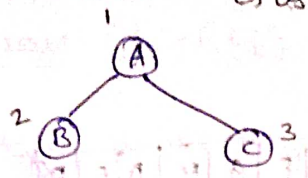
Live	B	C	
Parent	A	A	

In BFS, the 1st vert never gets onto Q. The vertices in Q are visited but unexplore vertices.



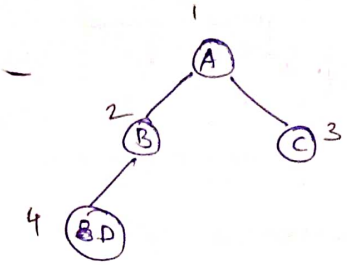
live	~~B~~	C	~~D~~ D	E
parent	A	A	B	B

Now obtain C from queue and add to tree
↳ as E-node



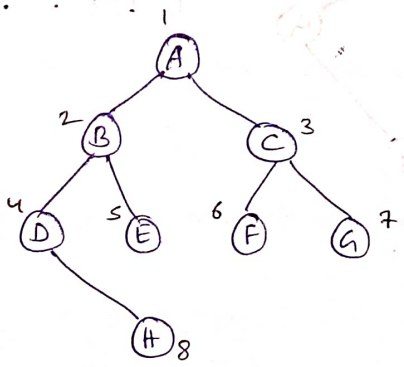
~~B~~	~~C~~	~~D~~ D	E	F	G
A	A	B	B	C	C

Now ~~cho~~ obtain D as next E-Node



~~B~~	~~C~~	~~D~~	E	F	G	H
A	A	B	B	C	C	D

Choose E as next E-Node



~~B~~	~~C~~	~~D~~	~~E~~	~~F~~	~~G~~	~~H~~
A	A	B	B	C	C	D

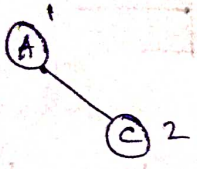
Breadth First Search Spanning tree

This is FIFO BFS and the order of visiting will be the order in which they are pushed in queue

Note:

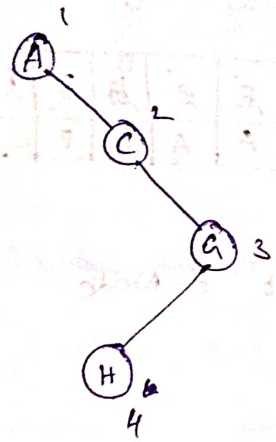
BFS can be used to find shortest cycle containing given vertex v.
(Think how)

LIFO BFS:



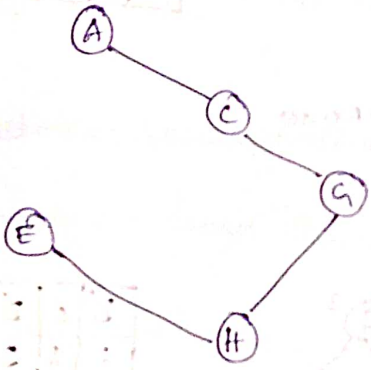
B	~~C~~	F	G	
A	A	C	C	

LIFO BFS can be implemented using a stack also



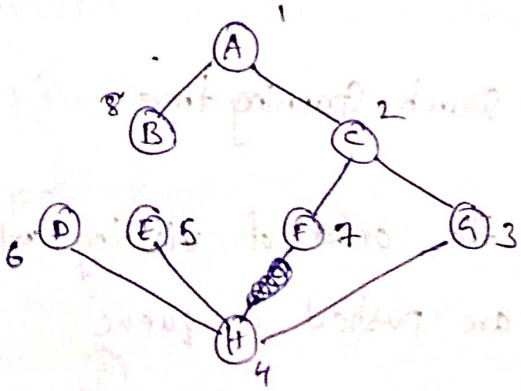
B	~~C~~	F	~~G~~	~~H~~	D	E	
A	A	C	C	G	H	H	

Choose E



B	~~C~~	F	~~G~~	~~H~~	D	~~E~~
A	A	C	C	G	H	H

Choose D



B	~~C~~	~~F~~	~~G~~	~~H~~	~~D~~	~~E~~
A	A	C	C	G	H	H

At every node we move to depth.

Hence we call it D-search (depth-search)

D-search sequence is A C G H E D F B

↳ (This is invalid DFS)

Note:

D-search need not to be a depth first search.

Priority Queue (LC-BFS)

→ This is used in gaming algorithms

Eg: Consider 15-puzzle problem

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial arrangement

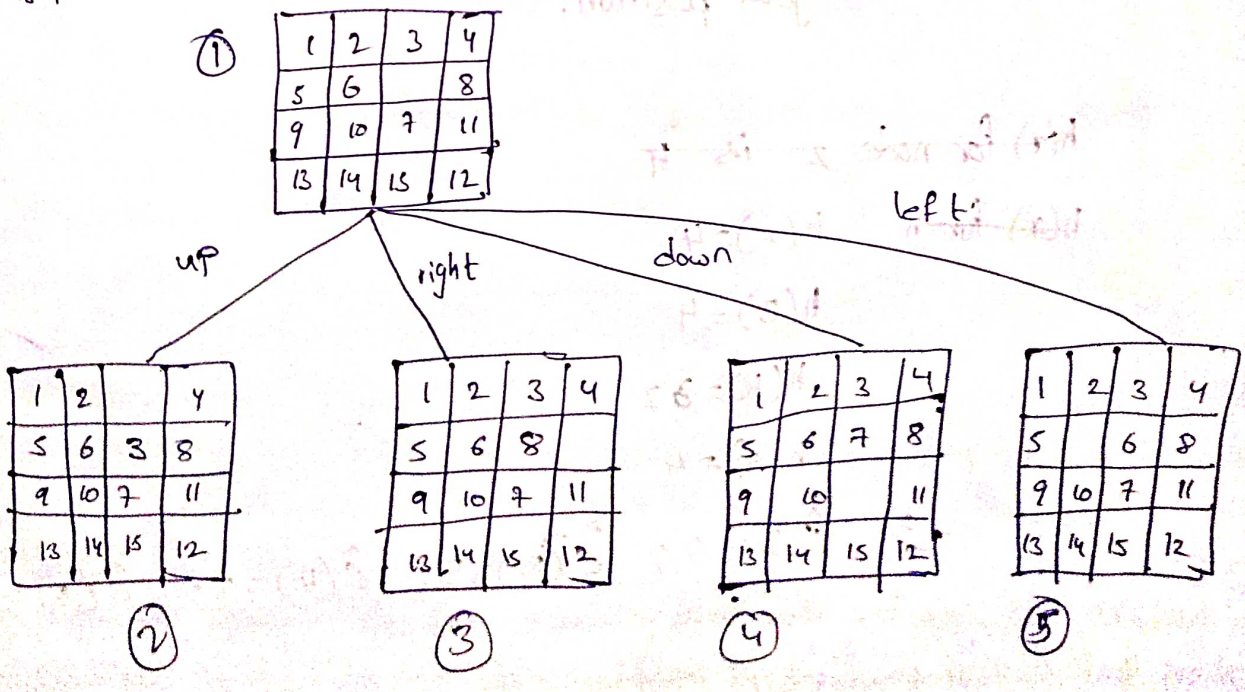
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal Arrangement

The objective is to reach goal arrangement with min possible moves.

We LC-BFS for this

Based empty slot we may have 2 (or) 3 (or) 4 possible moves at each step



→ Using FIFO or LIFO, here, is a blind search.

→ So we need to prioritize one of the live nodes and make it as E-node.

Thus to perform every LC-BFS, every live node is associated with a cost.

we select a node with minimum cost and hence the priority queue is min-heap

→ Cost function of live node is indicated as $\hat{c}(x)$

$$\hat{c}(x) = f(x) + h(x)$$

$f(x)$ → cost of reaching node 'x' from root.

$h(x)$ → estimated cost of reaching goal state from node x.

Formulating $h(x)$ may vary from node to node.

→ One way to estimate $h(x)$ in this problem is

$h(x)$ = no of non-blank tiles that are not in goal position.

~~$h(x)$ for node 2 is 4~~

~~$h(x)$ for n~~ $h(2) = 4$.

$h(3) = 4$

$h(4) = 2$

$h(5) = 4$

$\Rightarrow \hat{c}(2) = 5;\ \hat{c}(3) = 5;\ \hat{c}(4) = 3;\ \hat{c}(5) = 5;$

Thus we explore ~~node~~ node 4.

i.e., ~~that~~ make 4 as E-node.

→ After exploring node '4' we get 3 more live node and total live nodes at this point = 6.

Now we choose least cost node and continue the process.

Time Complexities:

→ The time complexity of both DFS & BFS with

(i) adjacency matrix ---- $O(n^2)$

(ii) Adjacency list ---- $O(n+e)$

DFS on undirected graph has either tree edge or back edge or forward edge

BFS on undirected graph contains either tree edge or cross edge

Applications:

→ Both BFS & DFS can be used to determine the presence of cycle in graph.

→ either using DFS are BFS

The presence of backedge in directed (back/forward in undirected) confirms the presence of cycle.

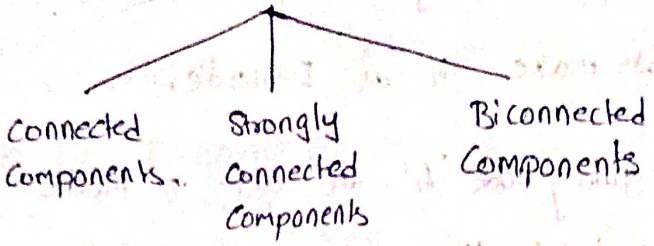
→ DFS & BFS can be used to know existence of a path btw a given pair of vertices.

→ DFS is used as a searching strategy in backtracking design strategy.

→ BFS is used as a searching strategy in branch & bound strategy.

→ For undirected graph, the BFS spanning tree will act as the solution for the single source shortest path problem by considering unit wt. cost graph. (optimal ~~graph~~ algorithm)

→ DFS is used to find ~~conne~~ components & Articulation points



Components:

(i) Connected Components:

Maximal subgraph that is connected is called connected component of given graph.

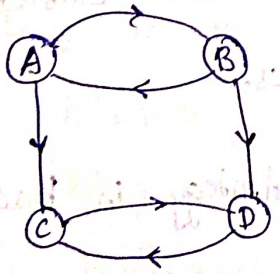
This concept applies only of undirected graphs.

(ii) Strongly Connected Components (This concept applied to directed graphs)

Two vertices 'u' & 'v' in a directed are said to be connected iff there is a path from 'u' to 'v' and 'v' to 'u'.

→ This relation of being connected, partitions the vertex set 'V' of the graph into maximal disjoint sets known as strongly connected components.

Eg:

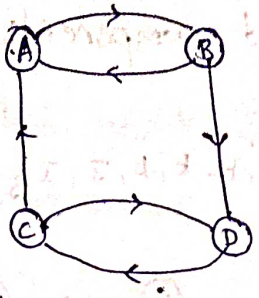


$G=(V,E)$

The above graph has 2 strongly connected components

i.e., {A,B} {C,D}

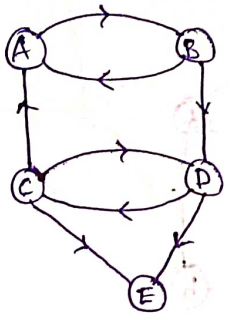
Eg:



no of strongly connected components = 1

i.e., {A, B, C, D}

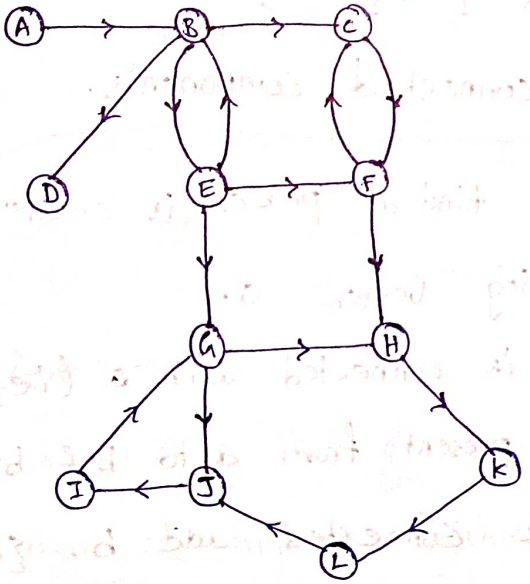
Eg:



no of strongly connected components = 2.

i.e., {A, B, C, D} & {E}

Eg:



Sol:

A has 1 out going edge & D has only 1 incoming edge

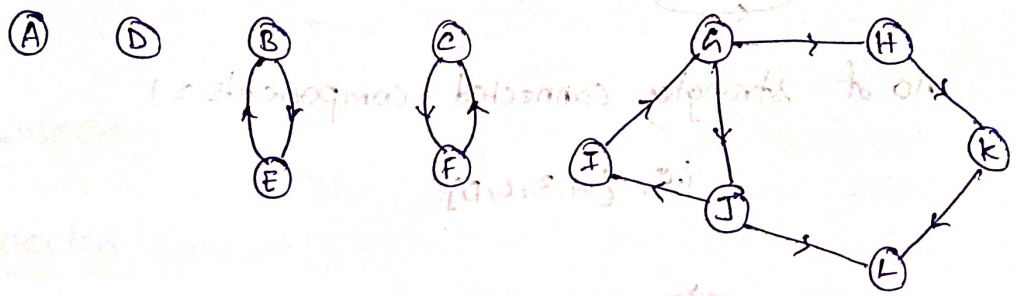
∴ {A} {D} are 2 strongly connected components

G H K L J I is a cycle.

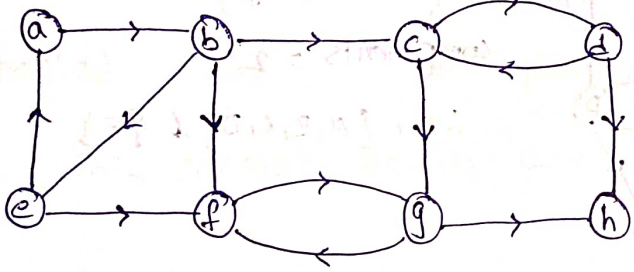
∴ {G, H, K, L, J, I} is a strongly connected component

∴ we have 5 strongly connected components

{A} {D} {B,E} {C,F} {G,H,K,L,J,I}



Eg:



{a,b,e} {f,g} {c,d} {h}

∴ 4 strongly connected components.

Approach for this kind of problem is create a set initially containing vertex a.

Now check if 'b' is connected with 'a' (i.e, path ~~present b to from~~ present from a to b & b to a.)

Since b & a are connected add b. {a,b}.

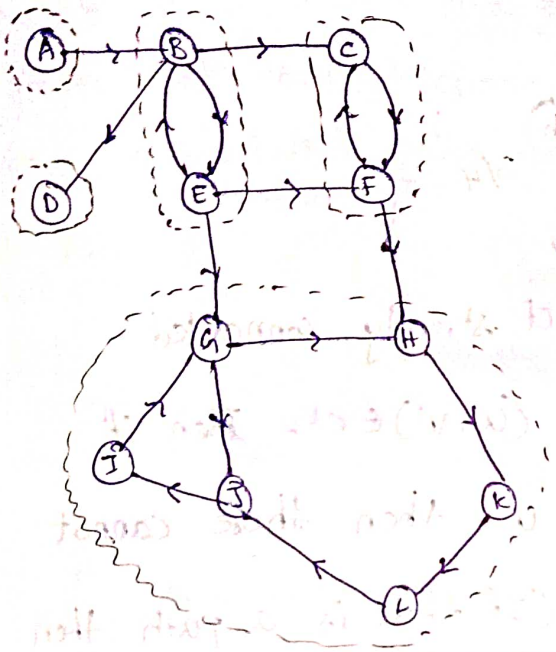
Now check with c.

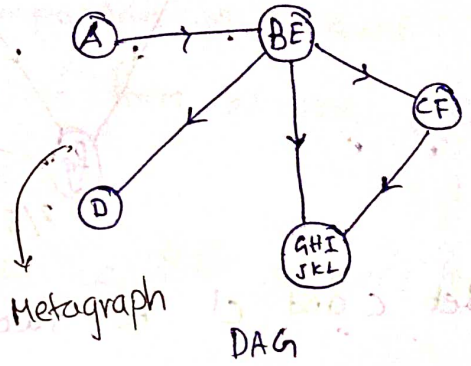
~~C~~ C is not connected with 'a'. So create a new set containing c.

Continue this process for all the vertices.

Properties:

1) Every directed graph is a DAG of its strongly connected components.





Metagraph

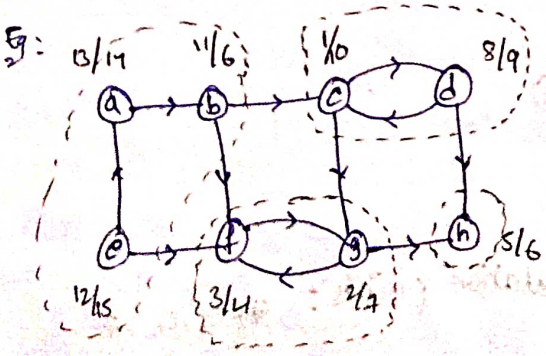
DAG

Metagraph: If $G=(V,E)$ is graph then $G'=(V,E')$ such that $E' \subseteq E$ is called metagraph i.e. Metagraph is a subgraph with all the vertices in graph

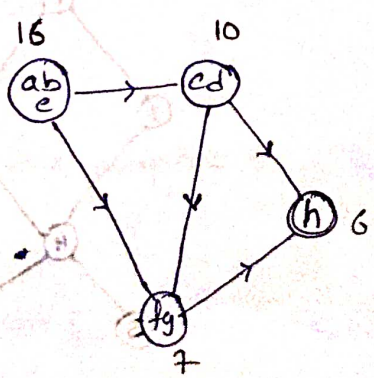
Proof:
Cuz if there is a cycle blw any two strongly connected components then the two components would become one.

2) Let c & c' be two strongly conneted components. If there is a path from a vertex in c to a vertex in c', then the highest finishing time of 'c' will always be greater than the highest finishing time (of any vertex) of c'. (Think why)

Eg:

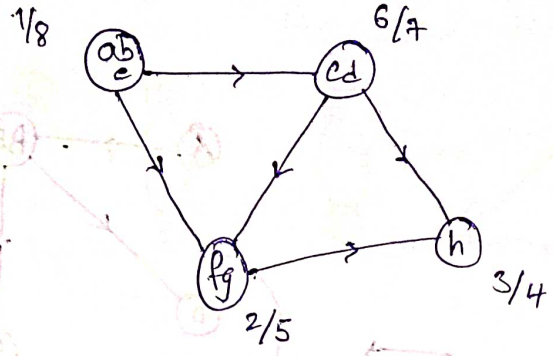


Metagraph



The property can be verified from above graph

Consider carrying DFS on DAG of connected components of a directed graph. The same property holds again.



3) Let c and c' be two distinct strongly connected components. Let $(u,v) \in c$ and $(u',v') \in c'$. Then if there is a path from u to u', then there cannot be a path from v' to v. (If there is a path then the whole thing forms a single connected component)

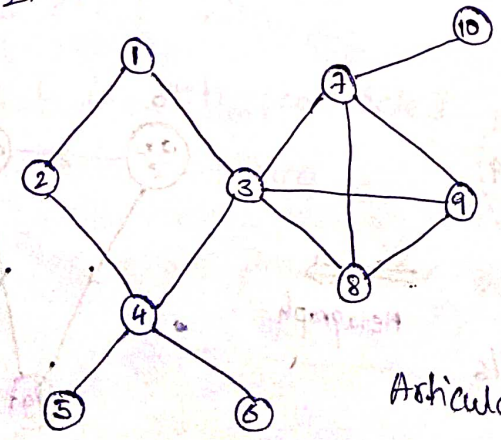
24/10/20

Articulation Point (Cut vertex): & Biconnected Components

The vertex whose removal makes a connected undirected graph disconnected is called articulation point.

→ The graph is said to be biconnected if it does not have articulation point.

Eg:



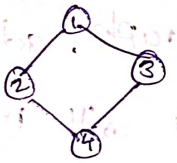
Articulation points: 3, 4, 7

Biconnected Components:

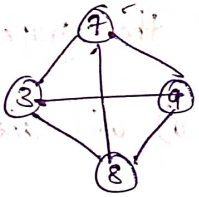
Maximal subgraph that is biconnected is called biconnected component of graph G.

→ To find no of biconnected ~~graph~~ components in a given graph note that biconnected points form at the point where articulation points are present

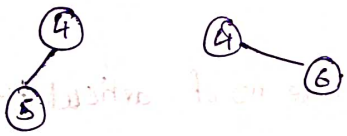
i.e., for 3 & 4 draw all the vertices adjacent to 3 & 4



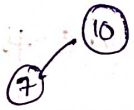
Now choose 3 again '7' also gets added since it is adjacent to 3.



Choose '4'

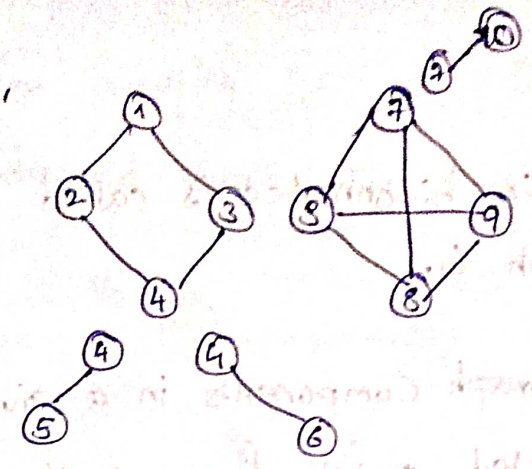


Choose '7':



∴ no of biconnected components for this graph = 5

i.e.,



→ Minimum no of edges that required to be added to make a graph biconnected ≤ no of articulation points.

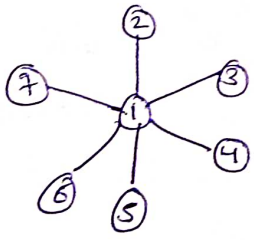
Eg: In the previous graph, adding edges (1,10) (2,5) (8,6) will make the graph biconnected.

However adding only two edges (5,10) & (6,8) will will also make the graph biconnecte.

(or) adding (1,5) & (6,10) will also make the graph biconnected.

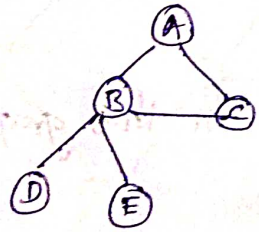
→ This formula generally applies to all graph except to star graph or graph of that kind.

Eg:



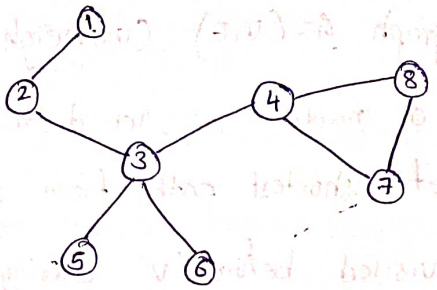
Here no of articulation points = 1
However adding (≤1) edges won't make the graph biconnected.

Eg:



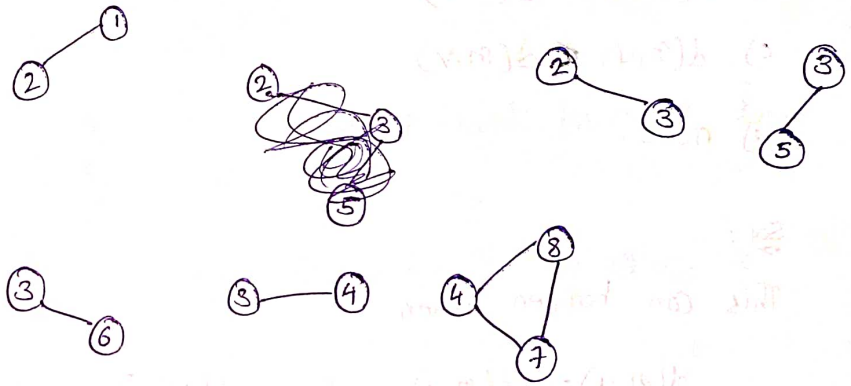
B is the articulation point.
no of articulation points = 1.
However min no of edges req to make the graph biconnected = 2

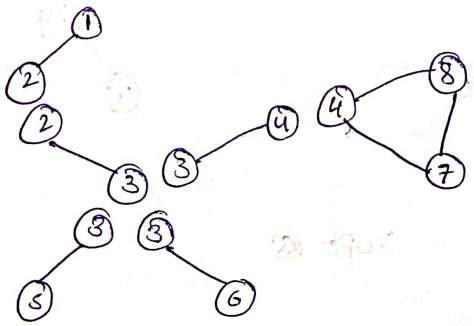
Eg: Identify the articulation points and draw the biconnected components of graph. and find min of edges to be added to make the graph biconnected.



Articulation points → 2, 3, 4

Biconnected Components





∴ 6 biconnected Components

Min no of edges to be added = to make the graph biconnected = 3

i.e., Add (2,5) (6,7) (1,4)

Bridge:

The edge whose removal makes the graph disconnected.

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

a) $d(r,u) \leq d(r,v)$

b) $d(r,u) > d(r,v)$

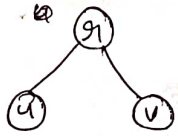
c) $d(r,u) \leq d(r,v)$

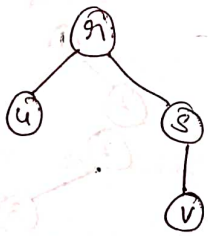
d) none.

Sol:

This can happen when

$d(r,u) = d(r,v)$ (or) $d(r,u) < d(r,v)$





∴ opt @

Q) 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 ——

a) k b) k+1 c) n-k+1 d) n-k

If a forest of n vertices have k edges then no of trees in (components) in forest = n-k.

(8) A DFS is performed on a DAG. d(x) is discovery time and f(x) is finishing time of a vertex 'x'. which is always true for all edges (u,v) in the graph?

a) d[u] < d[v] b) d[u] < f[v]

c) f[u] < f[v] d) f[u] > f[v]

Sol:

opt (a) & (b) & (c) fails if we start traversal from v.

opt (d) will be successful in any case.

Sorting Techniques:

Sorting techniques are classified ~~into two types~~: as

(i) Stable vs unstable

(ii) Internal vs external

(iii) Inplace vs not inplace

(iv) Comparision based vs non-comparision based.

(v) Recursive vs non-recursive.

Comparision baged:

The sorting algorithm that use comparision blw elements is called comparision based sort.

Eg: Quick sort, heapsort etc.

Non-Comparision baged:

The sorting technique that doesn't used comparision.

Eg: radix sort, counting sort, address calculation sort.

→ No comparision based sorting takes time less than O(nlogn) for any best case.

Inplace vs Not-inplace:

Sorting technique is said to be inplace if its space complexity is O(1). However for recursive sorting technique it can be O(logn) for recursion stack.

Otherwise it is said to be not inplace.

Eg: Inplace → Quicksort

not inplace → Mergesort.

Internal vs External:

The sorting technique which puts a constraint that all elements must be present in memory before the sorting begins.

Where as external sorting technique allows partial no of elements to be present in memory or allows

new elements to be added to the list of the numbers to be sorted.

Eg: Insertion sort behaves like external sorting. But it is not completely external sorting.

Stable vs Unstable

A sorting method is said to be stable iff the relative order of non-distinct elements is maintained in the sorted list.

Otherwise it is unstable.

Eg: Merge sort → stable

Quick sort, heap sort → unstable.

Generally stable sorting techniques are more desirable.

Note:

→ Time complexity of a comparision based sort depends on no of comparisons and no of swaps.

∴ time complexity = $O(\max\{\text{no of comparision}, \text{no of swaps}\})$

→ No of swaps depends on inversions in the list to be sorted.

if (i<j) and if A[i] > A[j] then the pair (i,j) is known as inversion of the array.

~~Thus no of inversions forms a lower bound for no of swaps.~~

Bubble Sort

```
Algo BS (A,n)
// A[1...n]: array
{
    for  pass ← n  downto 1
    {
        for i ← 1 to (pass-1)
        {
            if (A[i] > A[i+1])
                swap(A[i], A[i+1]);
        }
    }
}
```

Eg: A: 85 65 25 15 40 5

1st pass:

i=1 : 65 85 25 15 40 5
i=2 : 65 25 85 15 40 5
i=3 : 65 25 15 85 40 5
i=4 : 65 25 15 40 85 5
i=5 : 65 25 15 40 85 85

2nd pass:

i=1 : 25 65 15 40 5 85
i=2 : 25 15 65 40 5 85
i=3 : 25 15 40 65 5 85
i=4 : 25 15 40 5 65 85

3rd pass:

i=1 : 15 25 40 5 65 85
i=2 : 15 25 40 5 65 85
i=3 : 15 25 5 40 65 85

4th pass:

i=1 : 15 25 5 40 65 85

i=2 : 15 5 25 40 65 85

5th pass:

i=1 : 5 15 25 40 65 85

Note:

→ No of comparision in bubble sort = $\frac{n(n-1)}{2}$ (irrespective of data set)

→ No of swaps = no of inversions

↳ $[(n-1)+(n-2)+\cdots+1]$

∴ Time complexity : $O(n^2)$

→ Even if the elements are in increasing order, we still make $\frac{n(n-1)}{2}$ comparision.

To overcome this we keep track of whether swap has occured in a pass or not. If no swap has occured then the algorithm terminates. The below is the modified algorithm

```
for pass ← n downto 1
{
    flag=0;
    for i←1 to (pass-1)
    {
        if (A[i] > A[i+1])
        {
            swap(A[i], A[i+1]);
            flag=1;
        }
    }
    if(flag==0) break;
}
```

→ Now for this modified B.S

best case TC = O(n)

worst case TC = $O(n^2)$

The disadvantage is that it requires flag and this is overcome by insertion sort.

→ The recurrence relation for ~~no of comparisions~~ TC of BS is given by

$$T(n) = T(n-1) + O(n)$$

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

Selection Sort:

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

This algorithms selects ith smallest element in ith pass and places it in ith place. So we repeat this for n-1 times.

Eg: A: 80 60 20 15 40 10

i=1: (10) 60 20 15 40 80

i=2: (10) (15) 20 60 40 80

i=3: (10) (15) (20) 60 40 80

i=4: (10) (15) (20) (40) 60 80

i=5: (10) (15) (20) (40) (60) 80

Time complexity:

- Time = no of comp + no of swaps

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

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

→ Selection sort requires least no of swaps compared to any other comparision based sorting technique.

→ Rec. relation:

$$T(n) = T(n-1) + O(n)$$

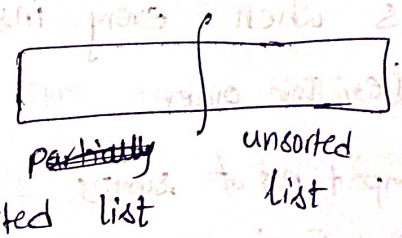
$$\Rightarrow T(n) = O(n^2)$$

Insertion Sort:

working: Insert one element at a time from an unsorted list to a partially sorted list.

→ It works in (n-1) passes

→ At every pass, we have the below scenerio



Take an element from unsorted list and put it in the sorted list in its position

Eg: A: 4 3 8 6 9 2

initially size of sorted list = 1

A: ⟨4 | 3 8 6 9 2⟩

Now insert 3 into partially sorted list

pass 1 → A: ⟨3 4 | 8 6 9 2⟩

pass 2 → A: ⟨3 4 8 | 6 9 2⟩

pass 3 → A: ⟨3 4 6 8 | 9 2⟩

pass 4 → A: ⟨3 4 6 8 9 | 2⟩

pass 5 → A: ⟨2 3 4 6 8 9⟩

From this we say insertion sort works as external sorting.

Since at any point of time insertion sort doesn't bother about where the unsorted list is stored.

Time Complexity:

(i) Best case:

Best case happens when: every insertion finishing with one comparision. i.e., Inc order

⇒ Time = no of comp + no of swaps

= θ(n-1) + 0

⇒ T.C = O(n)

ii) Worst Case:

Worst case happens when elements are in descending order.

time = no of comp + no of swaps = $O(n^2)$

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

→ In general, it is found that TC of insertion sort depends on no of inversions.

⇒ TC = $O(n+d)$

where d is no of inversions.

Thus if d=0 (i.e., no inversions)

TC = $O(n)$

Eg: Consider a list with n inversions. Then TC for insertion sort is _____?

Sol:

TC = $O(n+d)$

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

25/10/20

Non-Comparision bayed sorting techniques

i) Radix Sort:

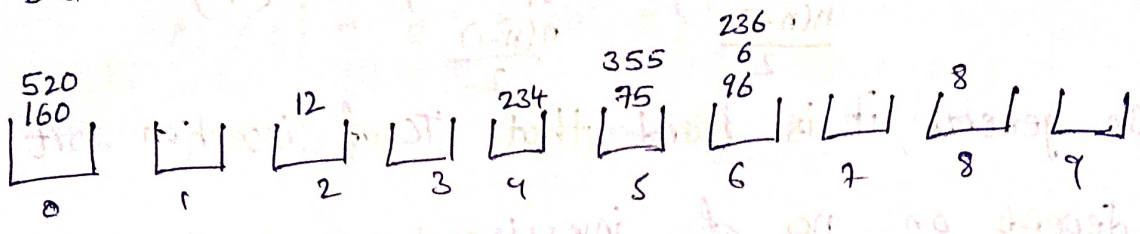
→ Here the term radix indicates base of the system.

→ If base is 'b' we take 'b' no of buckets.

Eg: A: 234 60 96 12 8 75 6 355 520 236

Pass 1: *

Distribute the numbers into buckets based on unit digit.

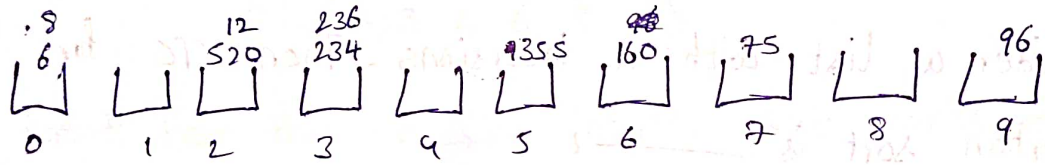


∴ Elements after pass 1

160 520 12 234 75 355 96 6 236 8

Pass 2:

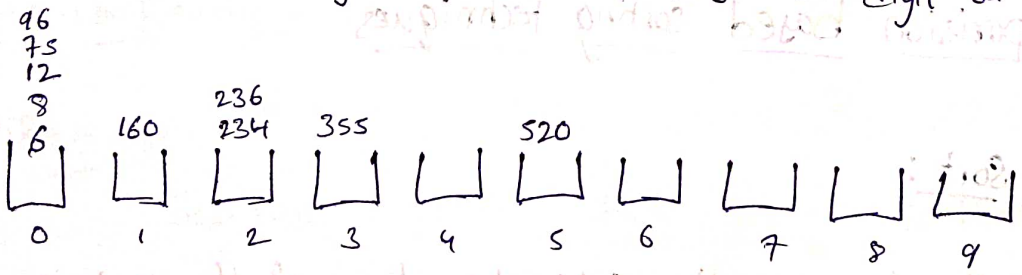
Distribute the digits into buckets based on digit at 10's place



∴ 0 6 8 520 12 234 236 355 160 75 96

Pass 3:

Distribute the digits into buckets based on digit at 100's place.



∴ 6 8 12 75 96 160 234 236 355 520

Drawback:

→ Special implementation is needed if we need to sort negative numbers.

→ Sorting fractional numbers is not possible.

Time complexity :

T.C = $O(d(n+b))$ ⟶ since 'b' is const we can also write $T_C = O(dn)$

d → max digits in a number

n → size of list

b → base

If 'x' is max number possible in the base. then

$d = \lceil \log_b x \rceil$

$TC = O((n+b)\log_b x)$

$= O(n \log_b x)$

let $x \leq n^c$

c is a constant

$T.C = O(n \log_b n)$

Thus radix sort performs good when $b \geq n$

If $b \geq n$

$O(n \log_b n) = O(n)$

(H/82) Method 1 :

we ~~heap~~ use heap for this (similar to problem (H/64))

we have mn elements.

size of heap = no of lists = n

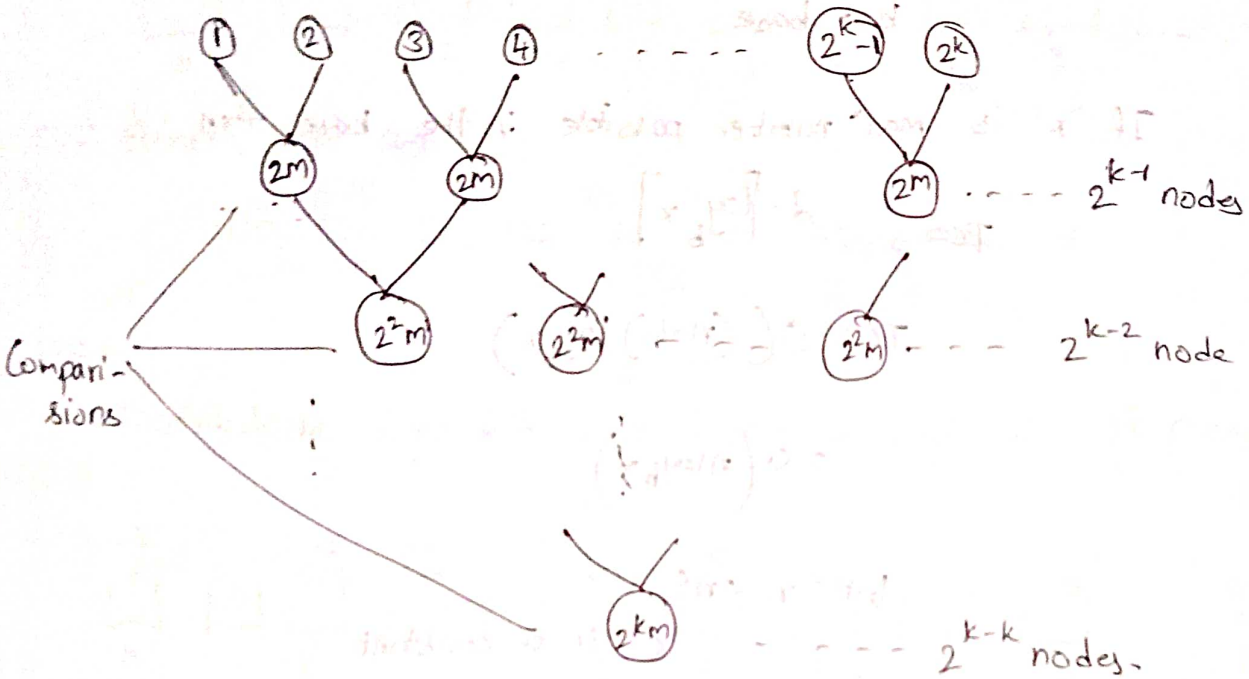
∴ del & insert on heap → log n

∴ $O(mn \log n)$

Method 2:

This can also be solved by considering optimal merge pattern.

let $n = 2^k$



∴ total comparisions requied

$$(2^1m)(2^{k-1}) + (2^2m)(2^{k-2}) + \cdots + (2^km)(2^{k-k})$$

$$= (2^km + 2^km + \cdots + 2^km)k$$

$$= nmlogn$$

(H/83) Let b be d base.

no of digits $d = \log_b n^k = k\log n$

TC of radix sort $= O(d(n+b))$

$= O(dn)$

$= O(n \cdot k\log n)$

$= O(n\log n)$

H/88 If assume base is n

then TC of radix sort = O(n)

Disjoint Set Data structure

Operations:

(i) Create(): Create the set will required element

(ii) Union(): Combining the two sets into one.

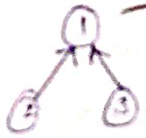
(iii) Find(): returns the representative element of the set.

Every set has a representative element

Representation:

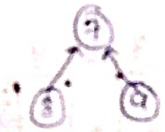
(i) forest:

Eg: $S_1 = \{1,2,3\}$ → representative element

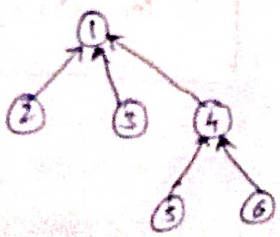


$S_2 = \{4,5,6\}$

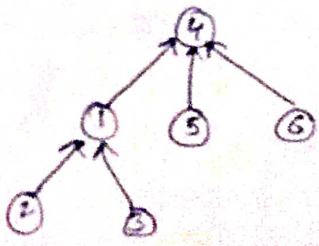
$S_3 = \{7,8,9\}$



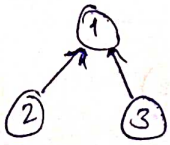
$S_1 \cup S_2$: This can be done by choosing one of ~~two~~ the representative element as new representative element for $S_1 \cup S_2$



(or)

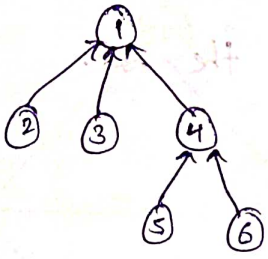


Find operation:



Here Find(1) = 1
Find(2) = 1
Find(3) = 1

Here Find(4) = 4
Find(5) = 4
Find(6) = 4



Here Find(1) = 1
Find(2) = 1
~~Find~~ Find(3) = 1
Find(4) = 1
Find(5) = 1
Find(6) = 1

TC of find depends on ~~highest~~ hieght of the tree.

(ii) Array based Representation:

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

Let 1, 4, 8 be corresponding representative elements.

Array:

	1	2	3	4	5	6	7	8	9
Parent element	-1	1	1	-1	4	4	4	-1	8
	1	2	3	4	5	6	7	8	9

with amortized analysis it is found that TC of both union & find is O(1)

Kruskal's algorithm for MCST

```
Algo kruskal(E, COST, n, T, mincost)
{
  //E is edge set in G, G has n vertices
  //COST[1..n, 1..n] is cost matrix
  //T is set of edges in MCST
1. real mincost COST(1..n, 1..n);
2. integer Parent(1:n), T(1:n, 1:2), n;
3. Construct a heap out of edge cost using heapify ——— O(e)
4. Parent ← -1
5. i ← mincost ← 0
6. while i<n-1 and heap not empty ——————— O(e)
     a) delete a min cost edge (u,v) from the heap } → O(loge)
        and reheapify using ADJUST
     b) j ← FIND(u); k ← FIND(v);
     c) if j≠k then i ← i+1
                    T(i,1) ← u; T(i,2) ← v
                    mincost ← mincost + cost(u,v)
                    call UNION(j,k)                  → O(1)
        endif
   repeat
7. if i≠n-1 then print ("no spanning tree") end if
8. return;
}
```

∴ TC = O(eloge)

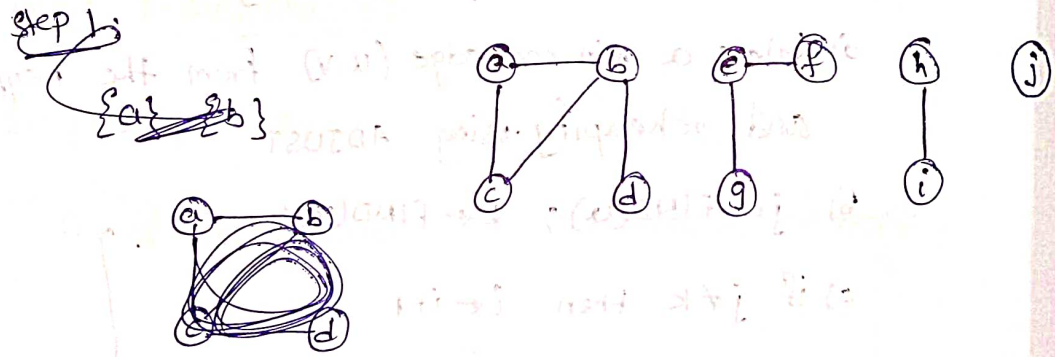
S.C = O(n) [i.e., space for parent array]

Algorithm to find connected components:

```
CONNECTED COMPONENTS (G)
{
    1. for each vertex v ∈ V[G]
           do MAKE-SET(v)
    2. for each edge (u,v) ∈ E[G]
           if find(u) ≠ find(v) then
               union(u,v)
}
```

Eg Tr

Eg: Consider below graph & its tracing



Step 1. -

{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

Step 2.

edge (a,b)

{a,b} {c} {d} {e} {f} {g} {h} {i} {j}

edge (b,c)

{a,b,c} {d} {e} {f} {g} {h} {i} {j}

edge {b,d}

{a,b,c,d} {e} {f} {g} {h} {i} {j}

edge (a,c):

find(a) = find(c)

∴ no union will be perform

edge (e,f) & edge (e,g)

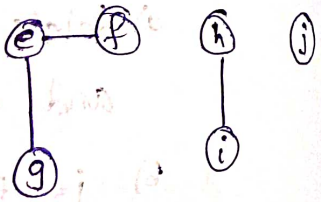
{a,b,c,d} {e,f,g} {h} {i} {j}

edge (h,i)

{a,b,c,d} {e,f,g} {h,i} {j}

∴ 4 connected components.

TC = O(e) [assuming that find & union can be done in O(1)]



Sum of Subsets (SOS):

→ Given a set of n-elements and another positive integer 'M' It is required to determine whether there exists a subset of given numbers wher sum equal M.

→ TC of brute force — $O(2^n)$

$SOS(n,M) = True$ if $M=0, n\geq 0$

$= False$ if $n=0, M>0$

$= [SOS(n-1,M) \;||\; SOS(n-1, M-a[n])]$

↳ only if $A[n] \leq M$

Tabulation Method for SOS

n=5 ; A = ⟨2, 8, 4, 11, 9⟩ ; M = 6

S[0...n, 0...M]

	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
1	T	F	T	F	F	F	F
2	T	F	T	F	F	F	F
3	T	F	T	F	T	F	T
4	T	F	T	F	T	F	T
5	T	F	T	F	T	F	T

→ soln i.e., SOS(5,6)

SOS[1,1] = SOS(0,1) || SOS(0,-1)

= F

SOS[1,2] = SOS(0,1) || SOS(0,0)

= F || T

= T

∴ SOS[5,6] = T

Think how to backtrack and obtain the subset required

Note:

If S(i,j) is true then S(k,j) is true ∀ k ≥ i

```
Algo SOS(A, n, M)
{
    for i ← 0 to n
    {
        for j ← 0 to m
        {
            sos(i,j) ← implement code for recursive eqn
        }
    }
}
```

TC: O(nM)

However if value of M is too large like 2^n (or) n^n then

TC will be $O(n \cdot 2^n)$ (or) $O(n \cdot n^n)$

and this case brute time complexity is better.

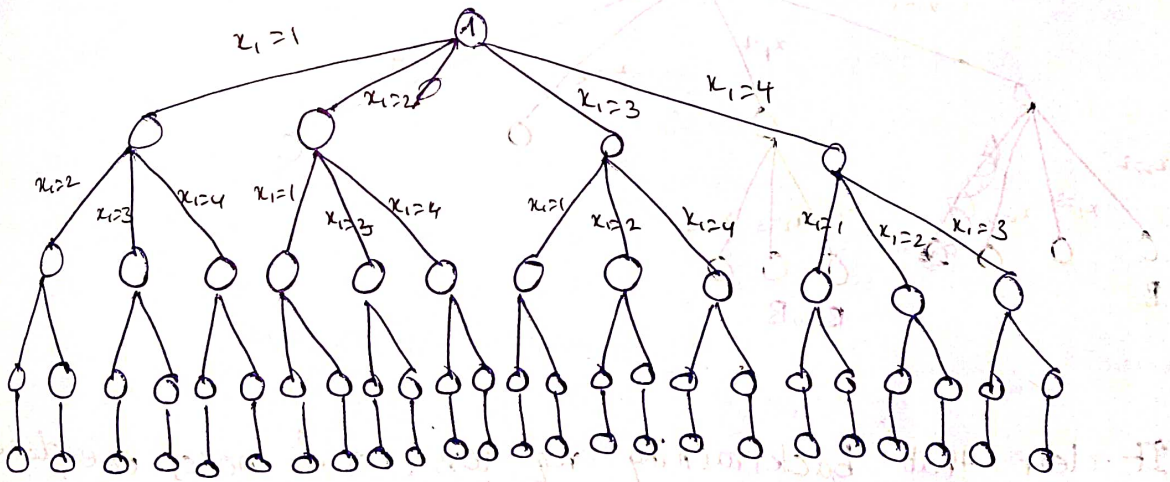
→ The time complexities like $O(n \cdot M)$ are called pseudo-polynomical TCs.

Introduction to Backtracking and Branch & Bound

Backtracking (BFS) is an algorithm design strategies that uses depth first searching & BFS to find feasible/optimal soln in solution space (state space tree)

Eg: n-queens

Consider 4 queens problem

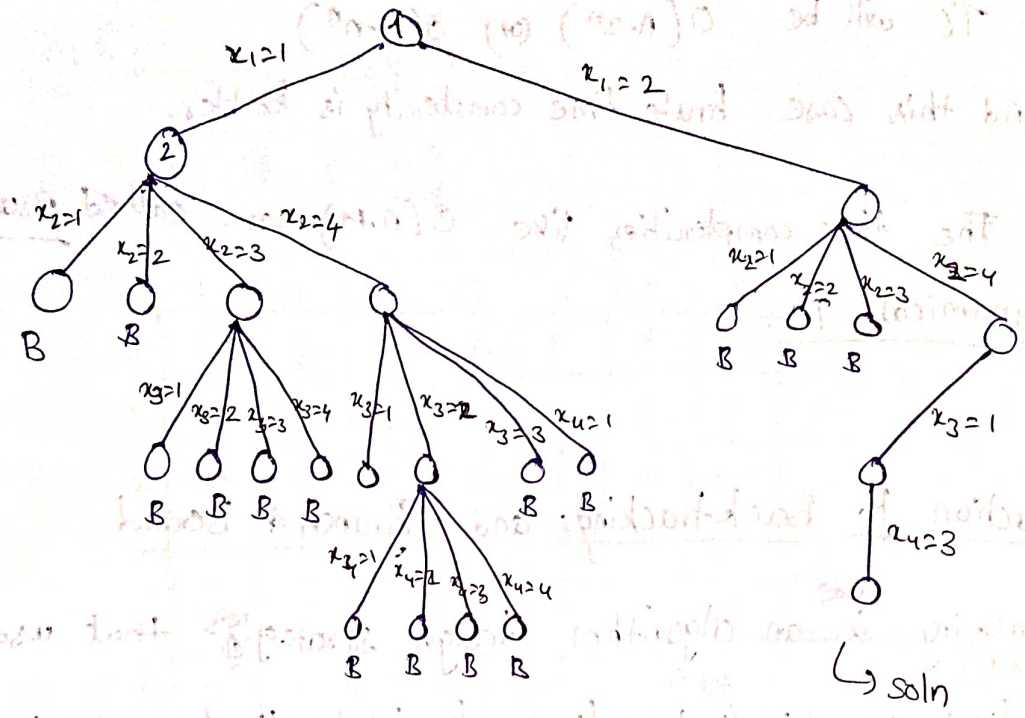


State Space tree

If DFS is used to search for soln in state space tree then it is called Backtracking.

If BFS is used then it is called Branch & Bound.

DFS : Backtracking

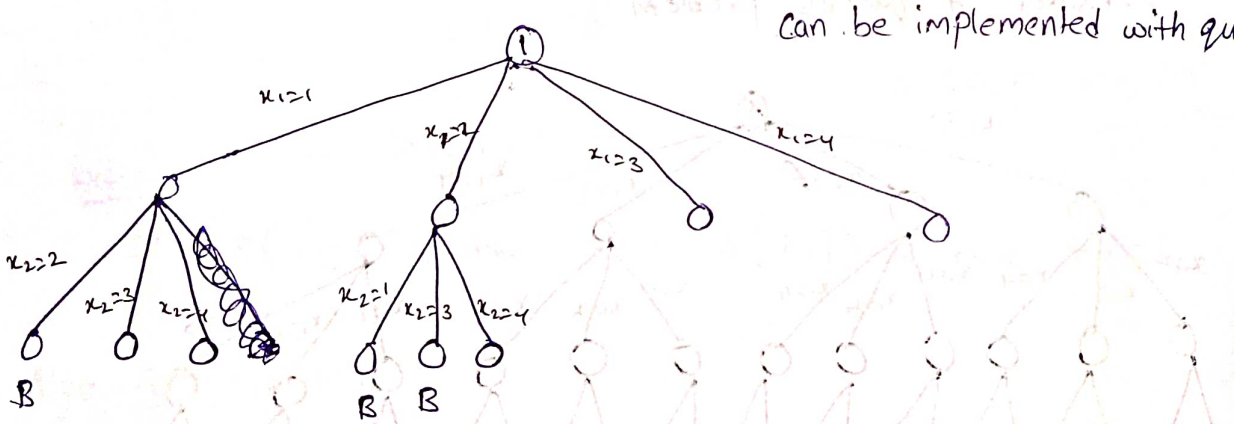


B → Bound

(Soln is not feasible

BFS : Branch & Bound :

Can be implemented with queue



→ It clear that backtracking req less no of nodes explored compared to branch & bound.

Thus n-queens can be efficiently implemented using backtracking.

→ similarly we can also solve backtracking in this approach.

→ Also TSP can be more efficiently solved using branch & bound.

kadane's algorithm for max sum sub array

→ Bruteforce TC : $O(n^2 * n) = O(n^3)$

↙ all subarray ↘ computing sum

→ DP approach (kadane's algo)

~~we maintain 3 variable~~

we perform n iterations (n is no of elements)

we maintain 3 variables max_sum_so_far, current_sum, index.

~~Index says from which iteration we start summing~~

~~Thus n iterations correspond to index value from 1 to n.~~

Index says till which index max sum is obtained.

```
Algo kadane ( )
{
   // start_index, end_index // soln.
   // Max Sum

   for  i=1  to  n
   {
        temp_start = i;  Max_sum_sofar=0;  curr_sum=0;  index=-1;
        for j=i to n
        {
             curr_sum += a[j];
             If (curr_sum > max_sum_so_far)
             {
                  Max_sum_so_far = curr_sum;
                  index=j;
             }
        }
        if ( max_sum_sofar > MaxSum)
        {    MaxSum = Max_sum_so_far;
```

```
            start_index = i;
            end_index = index;
        }
    }
    return [start_index, end_index];
}
```

Tc : $O(n^2)$

LIS :

→ one approach is solving this using LCS.

→ Let A be given ~~arrg~~ array and store the sorted array of A in another array B.

Now LCS(A, B) will be the LIS(A)

→ Also we can use pure DP approach and solve this.