2K18

1.a. An algorithm is a finite set of instructions that if followed, accomplishes a particular task.

The basic criteria that must be satisfied by an algorithm is given below:

i. Input: Zero or more quantities of input(s) are supplied.

ii. output: At least one quantity is produced.

iii. Definiteness: Each instruction is clear and unambigious.

iv. Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps

v. Effectiveness: Every instruction must be very basic so that it can be carried out in principle by a person using only pencil and paper. It's not enough that each operation be definite as in criterion 3; it also must be feasible.

b. Memory space

~~For~~ i → 2 bytes

j → 2 bytes

m3 → (3+3×2) bytes
= 18 bytes

m2 → (3×3×2) bytes
= 18 bytes

m1 → (3+3+2) bytes
= 18 bytes.

Total memory space = 18+18+18+2+2
= 58 bytes

Time Complexity

```
int ** matrix_sum(int **m1, int **m2)
{
    int **m3 = malloc(sizeof(int*) * 3);          ---> 1
    for(i=0; i<3; i++)                            ---> 3
    {
        for(j=0; j<3; j++)                        ---> 3
        {
            m3[i][j] = m1[i][j] + m2[i][j];
        }
    }
    return m3;                                    ---> 1
}
```

Total time = 1 + (3×3×1) + 1

= 1 + 9 + 1

= 11 unit ta

= 11 secs

C. Bigoh : The function f(n) = O(g(n)) iff ∃ +ve. constant C and n_0 such that $f(n) \leq C * g(n)$ $\forall n \geq n_0$.

(i) log(n!)

$n! = 1 \times 2 \times 3 \times \dots \times n$

$f(n) = \log(n!) = \log(1 \times 2 \times 3 \times \dots \times n) \leq$

$\log(1 \times 1 \times 1 \times \dots \times 1) \leq \log(1 \times 2 \times 3 \times \dots \times n) \leq$

$\log(n \times n \times n \times \dots \times n)$

$\log(1^n) \leq \log(n!) \leq \log(n^n)$

$\log(1) \leq \log(n!) \leq n \log n$

$\therefore O(n \log n)$

b. $6 \cdot 2^n + n^2$

$f(n) = 6 \cdot 2^n + n^2$

$g(n) = 2^n$

$f(n) \leq C \cdot g(n)$

$6 \cdot 2^n + n^2 \leq 6 \cdot 2^n + 2^n$

$6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$

$f(n) \leq c \cdot g(n)$

$\therefore O(2^n)$

iii. $1000n^2+100n-6$

$f(n) = 1000n^2+100n-6$

$g(n) = n^2$

$1000n^2+100n-6 \leq 9 \cdot 1000n^2$

$1000n^2+100n-6 \leq 9000n^2$

$f(n) \leq c \cdot g(n)$

$\therefore O(n^2)$

iv. $\frac{9n^3}{\log(n+1)}$

$f(n) = \frac{9n^3}{\log(n+1)}$

$g(n) = n^3$

$\therefore \frac{9n^3}{\log(n+1)} \leq \frac{9n^3}{\log(n+1)}$

$\frac{9n^3}{\log(n+1)} \leq 9n^3$

$f(n) \leq c \cdot g(n)$.

$\therefore O(n^3)$

2.a- $T(n) = \begin{cases} a & : n \leq 1 \\ 2T(n/2)+cn & : n > 1 \end{cases}$

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

$T(n) = 2\left[2T(n/4)+cn/2\right]+cn$

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

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

$T(n) = 2^2\left[2T(n/8)+cn/2^2\right]+2cn$

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

$T(n) = 2^k T(n/2^k)+kcn$

Assume $T(n/2^k) = T(1)$

$n = 2^k$

$k = \log n$

$T(n) = an + cn\log n$

$T(n) = an + cn\log n$

Here, $f(n) = an + cn\log n$

Let $g(n) = n\log n$

$\therefore an + cn\log n \leq an\log n + cn\log n$

$an + cn\log n \leq (a+c)n\log n$

$\therefore f(n) \leq c \cdot g(n)$

$\therefore \Omega(g(n)) = O(n\log n)$

b. Branch and Bound vs Backtracking

Branch and Bound	Backtracking
(i) Branch and Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution that the pre solution leads to, it abandons that pre solution. It completely searches the state space tree to get optimal solution	(i) Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes that the last choice by backing it up. It searches the state space tree until it has found a solution for the problem
(ii) Branch and Bound traverse the tree in any manner, DFS or BFS	(ii) Backtracking traverses the state space tree by DFS manner
(iii) Branch and Bound involves a bounding function	(iii) Backtracking involves feasibility function
(iv) Branch and Bound used for solving optimisation problem.	(iv) Backtracking is used for solving decision problem
(v) In Branch and Bound as the optimum solution may be present any where in the state space tree, so the tree need to be searched completely	(v) In backtracking, the state space tree is searched until the solution is obtained.
(vi) Branch and Bound is less efficient	(vi) Backtracking is more efficient.

Backtracking vs Brute force

Backtracking	Brute Force
(i) A backtracking algorithm is a problem solving algorithm that uses a brute force approach for finding the desired output	(i) The Brute force approach tries out all the possible solutions and chooses the desired/best solutions
(ii) It's faster than Brute force algorithm	(ii) It's slower than Backtracking algorithm,
(iii) It's having to perform redundant work	(iii) It's having to calculate overall solution

C. A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can take, it represents some partial information about the variables of interest.

Constraints are useful in Backtracking algorithm. If the partial solution doesn't satisfy the constraint, it'll not be explored further. The algorithm backtracks from that point and explores the next possible candidate. Such processing is convenient to represent using a state space tree.

There are two types of constraints are mostly used.

(i) Implicit constraint: It's a rule in which how each element in a tuple is related.

(ii) Explicit constraint: It's the rules that restrict each element to be chosen from the given set.

Constraints of N queens problem is no two queens on the same column. This constraint is implicit in the definition of queens. Since no two elements of queens can have the same index, no two queens can be in the same column.

$Q_i \neq Q_j$ for all $i \neq j$ & $|Q_i - Q_j| \neq i - j$

Recursive Backtracking Algorithm to solve N queen problem:

i. Start in the leftmost column.

ii. If all queen are placed return true

iii. Try all rows in the current column. Do following for every tried row

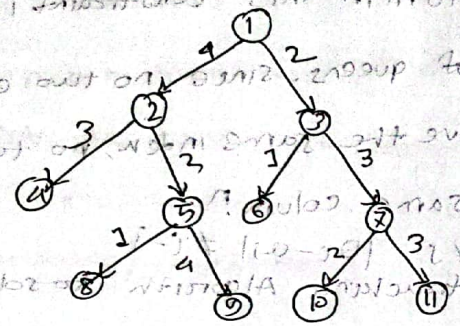
a. If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

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

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

iv. If all rows have been tried and nothing worked

return false



Here leaf nodes 4 8 9 6 10 11

$d(4) = d(6) = d(8) = d(9) = d(10) = d(11) = 0$

$c(5) = \{8, 9\}$

$d(5) = \max_{v \in 8,9} \begin{cases} d(8) + w(5,8) \\ d(9) + w(5,9) \end{cases}$

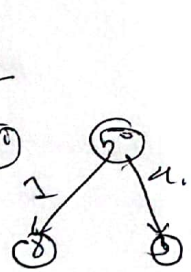
$= \max \begin{cases} 0+1 \\ 0+4 \end{cases} = 4 < 5$

$d(5) = 4$

$c(2) = \{4, 5\}$

$d(2) = \max_{v \in 4,5} \begin{cases} d(4) + w(2,4) \\ d(5) + w(2,5) \end{cases}$

$= \max \begin{cases} 0+3 \\ 4+2 \end{cases} = 6 > 5$



nodes 5 has to be splitted

$d(5) = 0$

$c(7) = \{10, 11\}$

$d(7) = \max_{v \in 10,11} \begin{cases} d(10) + w(7,10) \\ d(11) + w(7,11) \end{cases}$

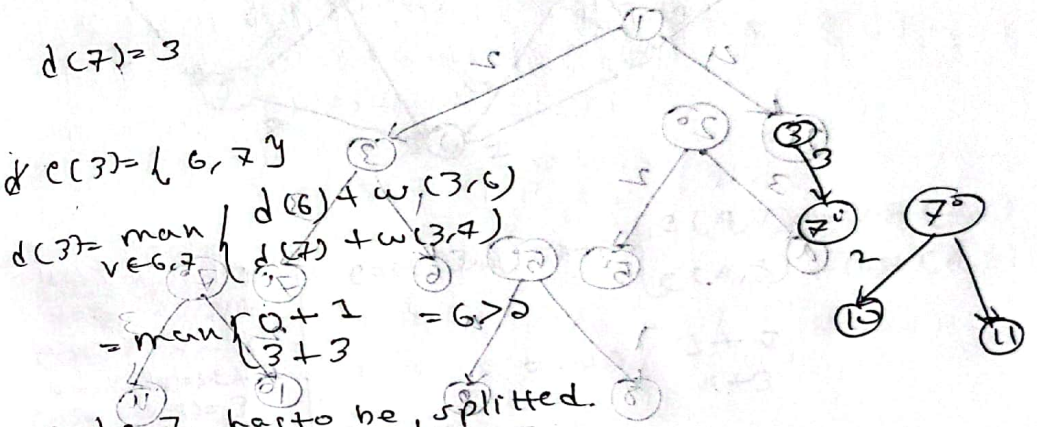
$= \max \begin{cases} 0+2 \\ 0+3 \end{cases} = 3 < 5$

$d(7) = 3$

$c(3) = \{6, 7\}$

$d(3) = \max_{v \in 6,7} \begin{cases} d(6) + w(3,6) \\ d(7) + w(3,7) \end{cases}$

$= \max \begin{cases} 0+1 \\ 3+3 \end{cases} = 6 > 5$



node 7 has to be splitted. $d(7) = 0$

$c(1) = \{2, 3\}$

again. $d(2) = \max \begin{cases} d(5) + w(2,5) \\ d(4) + w(2,4) \end{cases}$

$= \max \begin{cases} 0+2 \\ 0+3 \end{cases} = 3 < 5 \quad d(2) = 3$

$d(3) = \max \begin{cases} d(6) + w(3,6) \\ d(7) + w(3,7) \end{cases} \quad d(3) = 3$

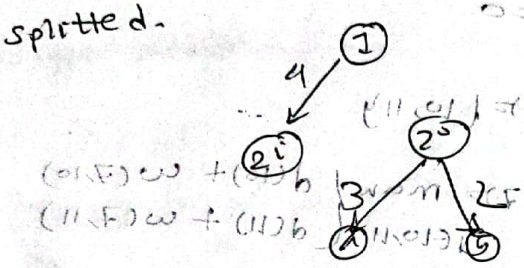
$= \max \begin{cases} 0+1 \\ 0+3 \end{cases} = 3 < 5$

now $d(1) = \max_{v \in 2,3} \begin{cases} d(2) + w(1,2) \\ d(3) + w(1,3) \end{cases} = \max \begin{cases} 3+4 \\ 3+2 \end{cases}$

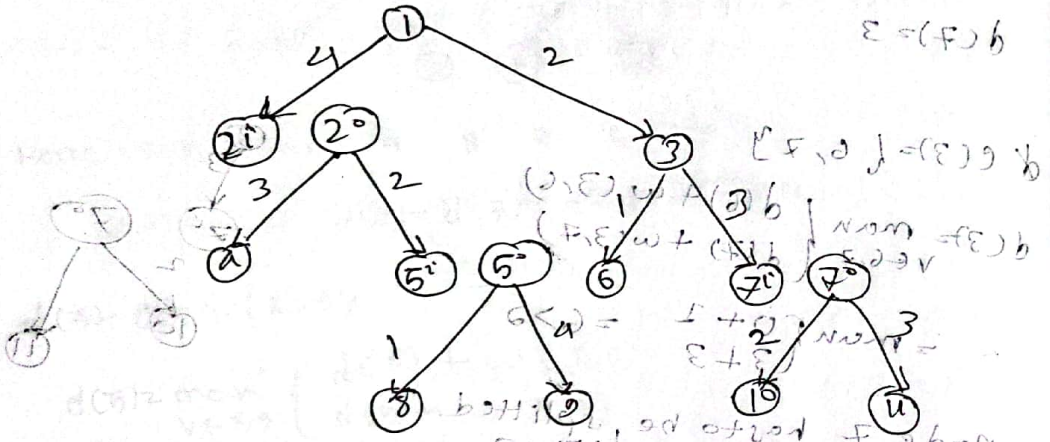
$= 7 > 5$

Node 2 has to be splitted.

d(2)=0



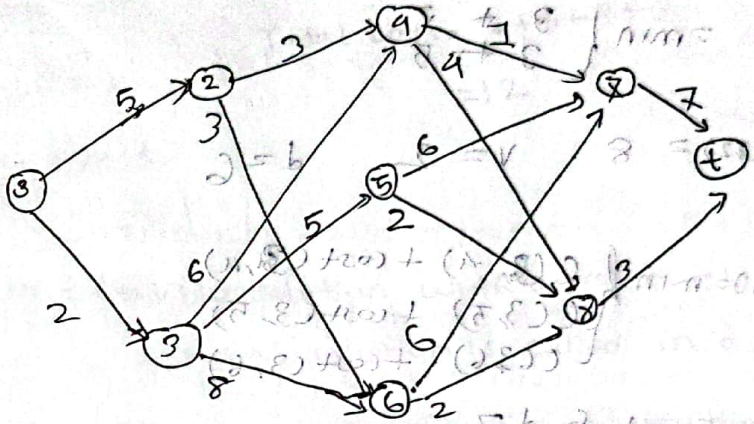
So, the splitted graph is given below



b. Control abstraction of greedy method

```
Algorithm Greedy(a, n)
// a[1:n] contains the inputs
{
    solution = // Initialize solution
    for i = 1 to n do
    {
        x := select(a);
        if Feasible(solution, x) then
            solution = Union(solution, x);
    }
    return solution;
}
```

c. V_1 V_2 V_3 V_4



$cost(5,t) = 0$, $v = 7$, $d = 1$

$cost(4,7) = 7$, $v = 7$, $d = 1$

$cost(4,8) = 3$, $v = 8$, $d = 1$

$$cost(3,4) = \min \begin{cases} c(4,7) + cost(4,7) \\ c(4,8) + cost(4,8) \end{cases}$$

$$= \min \begin{cases} 1+7 \\ 4+3 \end{cases}$$

$cost(3,4) = 7$; $v = 4$, $d = 8$

$$cost(3,5) = \min \begin{cases} c(5,7) + cost(4,7) \\ c(5,8) + cost(4,8) \end{cases}$$

$$= \min \begin{cases} 6+7 \\ 2+3 \end{cases}$$

$cost(3,5) = 5$; $v = 5$ $d = 8$

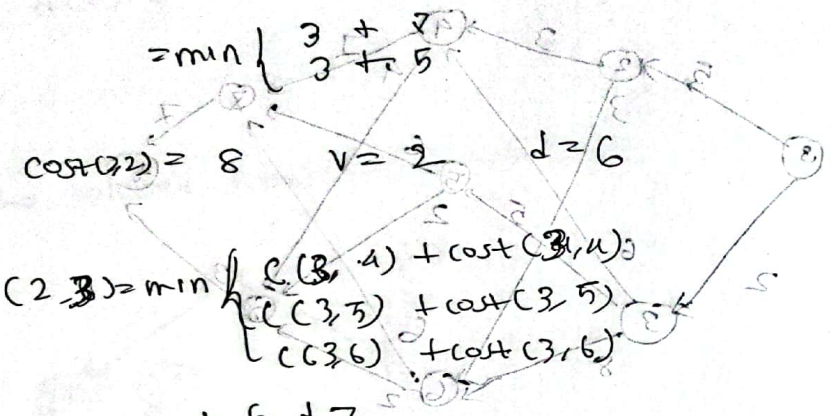
$$cost(3,6) = \min \begin{cases} c(6,7) + cost(4,7) \\ c(6,8) + cost(4,8) \end{cases}$$

$$= \min \begin{cases} 6+7 \\ 2+3 \end{cases}$$

$cost(3,6) = 5$; $v = 5$ $d = 8$

cost (2, 7) = min { c(2,4) + cost(3, 4)
c(2,6) + cost(3, 6)

= min { 3 + 7
3 + 5



cost(2,2) = 8 v = 2 d = 6

cost(2,3) = min { c(3,4) + cost(3,4)
c(3,5) + cost(3,5)
c(3,6) + cost(3,6)

= min { 6 + 7
5 + 5
8 + 5

cost(2,3) = 10 ; v = 3 ; d = 5

cost(1,5) = min { c(5,2) + cost(2,2)
c(5,3) + cost(2,3)

= min { 5 + 8
2 + 10

cost(1) = 12 v = 5 d = 3

v	5	2	3	4	5	6	7	8	t		
cost	12	8	10	7	5	5	7	3	0		
d	3	6	5	8	8	8	t	t	t		

d(1,5) = 3 5 → 5 → 3 → 8 → t

d(2,3) = 5

d(3,5) = 8

d(4,8) = t

Total cost = 2+5+2+3
= 12

4. a. A feasible solution with a value close to the value of an optimal solution is called an approximate solution. An approximation algorithm for an optimization problem is an algorithm that generates approximate solution for the problem.
There are three types of approximation algorithm : Let F*(I) = optimal value
F̂(I) = close to optimal value (may be greater or less)

(i) absolute approximation:
A is an absolute approximation algorithm for problem P if and only if for every instance I of P, |F*(I) − F̂(I)| ≤ K for some const K.

(ii) f(n) approximation:
A is an f(n)-approximation algorithm for problem P if and only if for every instance I of size n, |F*(I) − F̂(I)| / F*(I) ≤ K for F*(I) > 0.

(iii) ε-approximation:

An ε-approximation algorithm is an $f(N)$-approximate algorithm for which $f(N) \leq \varepsilon$ for some constant ε.

Importance of approximation algorithm.

i. Problems that ~~prob~~ optimal solution ~~is~~ is not found in a polynomial time i.e. set cover, bin packing .etc, in these places approximation algorithm is used.

ii. It also needs to find a near optimal solution, in polynomial time and ~~if heuristic~~ This gives us a guarantee approximation ratio.

iii. It guarantees to run in polynomial time though it does not guarantee the most effective soln

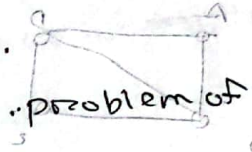
iv. It guarantees to seek out high accuracy and top quality soln.

v. ~~These are used to get an answer~~

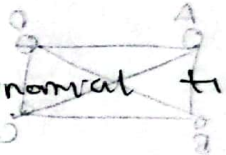
b. The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by non-deterministic machine in polynomial time.

NP-hard problem.

A problem X is NP hard if there is an NP complete problem Y, such that Y is reducible to X in polynomial time. NP-hard problems are as hard as NP-complete problems. NP-hard problem need not be in NP class.



If every problem of NP can be polynomial time reduced to it called as NP hard.



Example: Hamiltonian cycle, optimization problem, Shortest path

NP complete problem:

A problem X is NP complete if there is an problem Y, such that Y is reducible to X in polynomial time. NP-complete problems are as hard as NP problems.

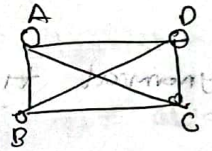
A problem is NP complete if it is a part of both NP and NP-Hard problem. A non deterministic turning machine can solve NP-complete problem in polynomial time. A problem is np complete when

it is both NP and NP hard combines together. This means NP complete problems can be verified in polynomial time.

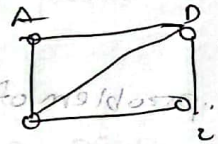
Example: Decision problems, Pulsar graphs

Clique problem is NP-complete.

Clique: A clique in an undirected graph $G=(V,E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. The size of a clique is the number of vertices in it.



This is a clique.



Not a clique

B and C not connected

The clique problem is an opt-

Clique = { (G,k): G is a graph containing a clique of size k}

Proof:

Clique $\in$ NP

Now, we have to prove 3-CNF-SAT $\leq_p$ Clique.

The reduction algorithm begins with an instance of 3-CNF-SAT.

Let $\phi = C_1 \wedge C_2 \wedge \dots \wedge C_k$ be a boolean formula in 3-CNF with k clauses.

For C_r = L_1^r, L_2^r, L_3^r are its three distinct literals

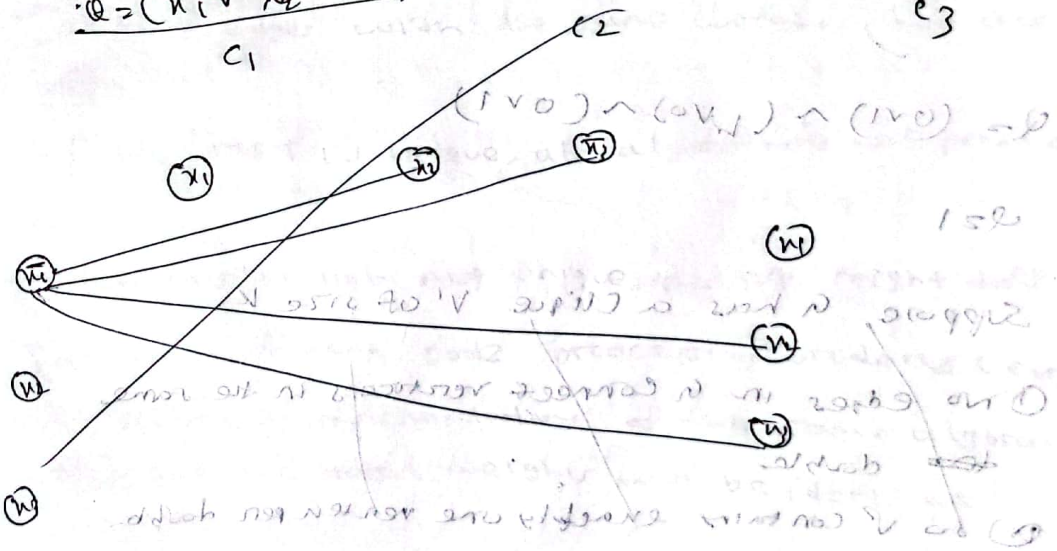
We'll construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

First we need to construct the graph G=(V,E) as follows. For each clause $C_r = (L_1^r \vee L_2^r \vee L_3^r)$ in ϕ we place a triple of vertices v_1^r, v_2^r, v_3^r as follows.

Example:

$\phi = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

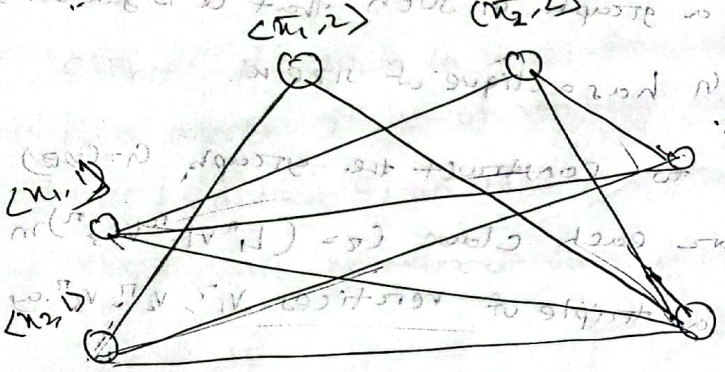
C_1 C_2 C_3



Here, k_2

$Q_1 = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_1) \wedge (x_1 \vee x_3)$

C_1 C_2 C_3



x_1 x_2 x_3

0 1 1

$Q_2 = (0 \vee 1) \wedge (1 \vee 0) \wedge (0 \vee 1)$

$Q_2 = 1$

Suppose, G has a clique V' of size K

① No edges in G connect vertices in the same ~~set~~ double

② So V' contains exactly one vertex per double

So Clique is a NP complete problem

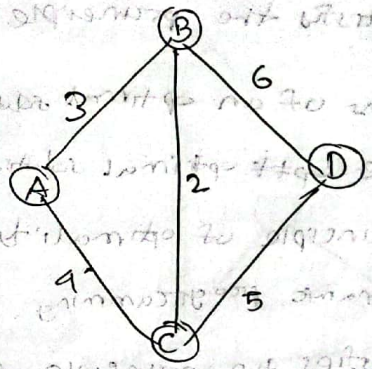
3.(c) A problem is said to satisfy the principle of optimality if the subsolutions of an optimal solution of the problem are themselves ~~the~~ optimal solutions for their subproblems. The principle of optimality is the basic principle of dynamic programming. The shortest path problem satisfies the principle of optimality

4.d. Prims and Kruskals algorithms will always return the same minimum spanning tree. A graph where every edge weight is unique (there are no two edges with the same weight) has a unique MST.

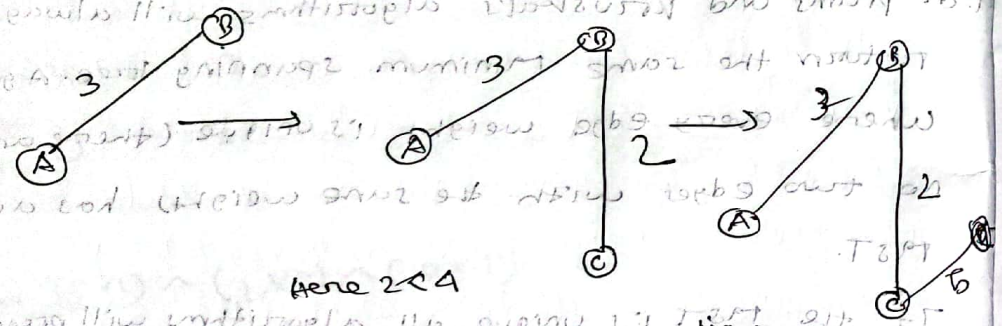
If the MST is unique, all algorithms will produce it.

If the MST is not ~~not~~ unique, the o/p might differ due to different node processing orders (even two distinct implementations of the same algorithm can), but the total weights will be identical

Example:



Using prims Algorithm: (Source A)

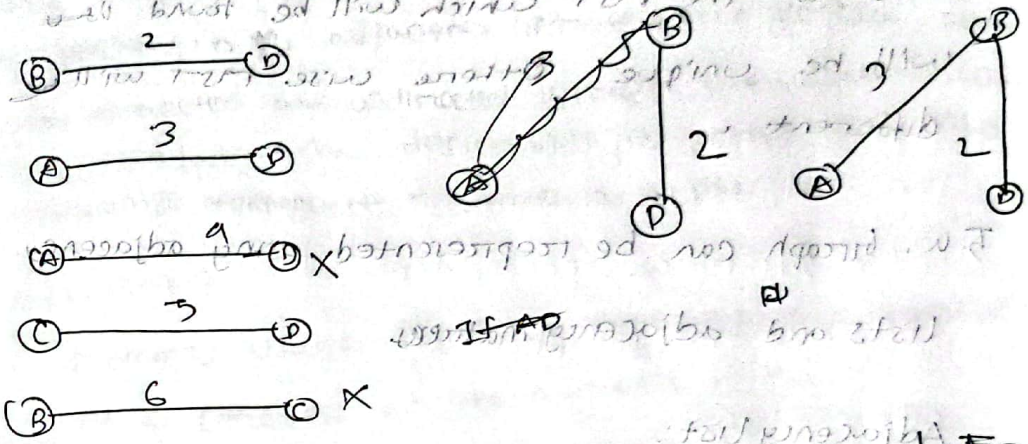


Total weight = 3+2+5 = 10

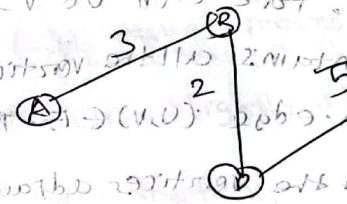
If A is taken the cycle will be formed

For this it's not taken

Using Kruskal's Algorithm

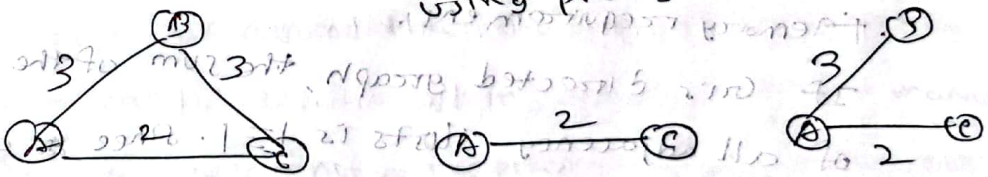


If AD is taken then cycle will be formed. For this it's not taken.



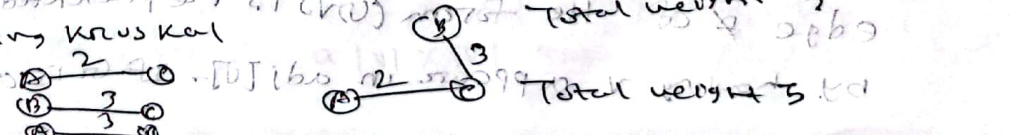
Total weight: 3+2+5

Using prims



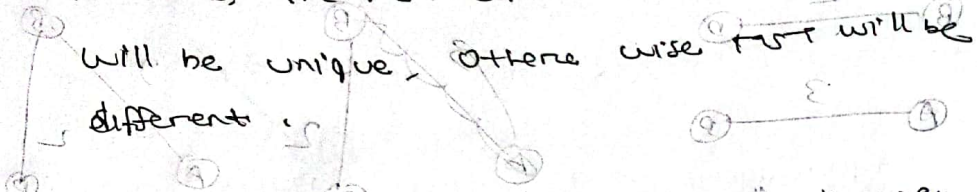
Total weight = 5

Using Kruskal

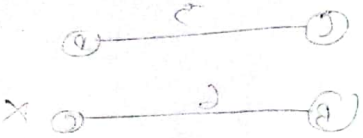


Total weight = 5

If weight is unique then using prim's and Kruskals, the MST which will be found is will be unique, otherwise MST will be different.



5. a. Graph can be represented using adjacency lists and adjacency matrices



Adjacency list:

The adjacency list representation of a graph $G(V,E)$ consists of an array Adj of $|V|$ lists one for each vertex in V. For each $u \in V$ the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u,v) \in E$. That is $Adj[u]$ consists of all the vertices adjacent to u in G.

Memory requirement:

If G is directed graph, the sum of the lengths of all adjacency lists is $|E|$. Since an edge of the form (u,v) is represented by having v appear in $Adj[u]$. If G is an undirected graph, the sum of lengths of all the adjacency lists is $2|E|$. Since if (u,v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graph the adjacency list representation has desirable property that the amount of memory it requires is $\theta(V+E)$

Advantages:

i. Memory usage depends more on the number of edges and less on the number of nodes.

ii. It's fast to iterate over all edges

iii. It's fast to add/delete a node.

Disadvantages:

i. It takes more time to find specific edge.

ii. It's slightly slower than matrix.

Adjacency matrix:

The adjacency matrix representation of a graph $G(V,E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. Then the adjacency representation of a graph consists of a $|V| \times |V|$ matrix.

A = (a_ij) such that,

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Memory requirements:

The adjacency matrix of a graph requires $\theta(V^2)$ memory, independent of the number of edges in the graph.

Advantages:

i. It's fast to look up and check for presence or absence of a specific edge between any two nodes

ii. Adding and deleting an edge in const. time

Disadvantages:

i. It takes more memory

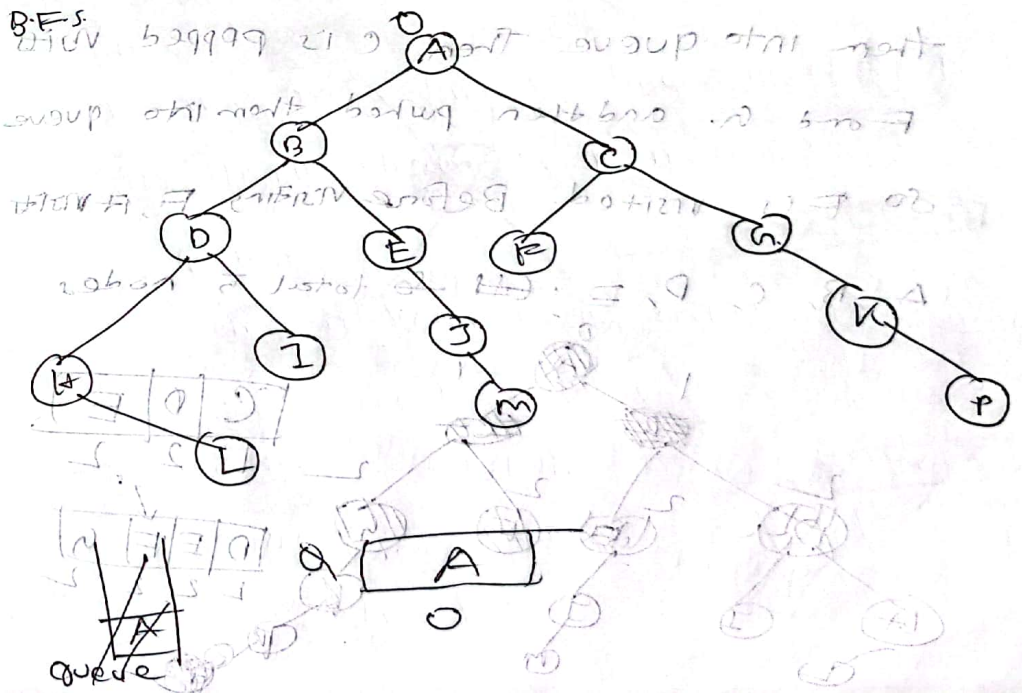
ii. It's slow to iterate over all edges

iii. Traversing takes more time.

b. If the graph, which is given in question, starting from the node A at the top, BFS will visit the least number of nodes before visiting the node.

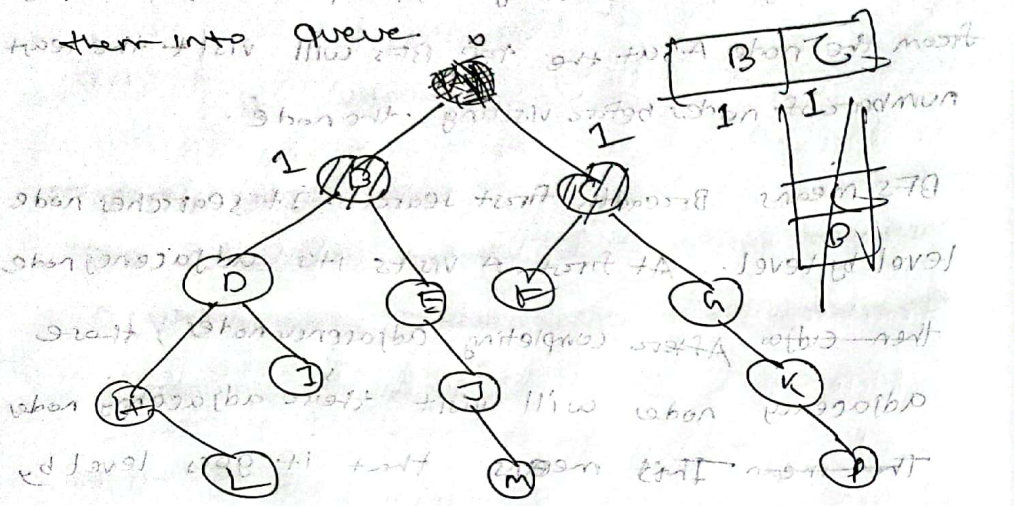
BFS means Breadth first search. It searches node level by level. At first A visits it's adjacency node. After completing adjacency nodes, those adjacency nodes will visit their adjacency nodes that it goes level by level.

B.F.S.



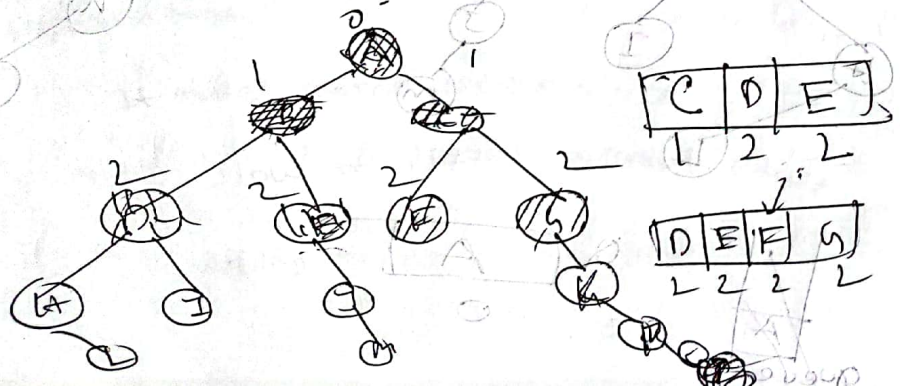
is pushed into queue. and the popped

At first A visits ~~A and f~~ B and C. ~~and~~ push them into queue



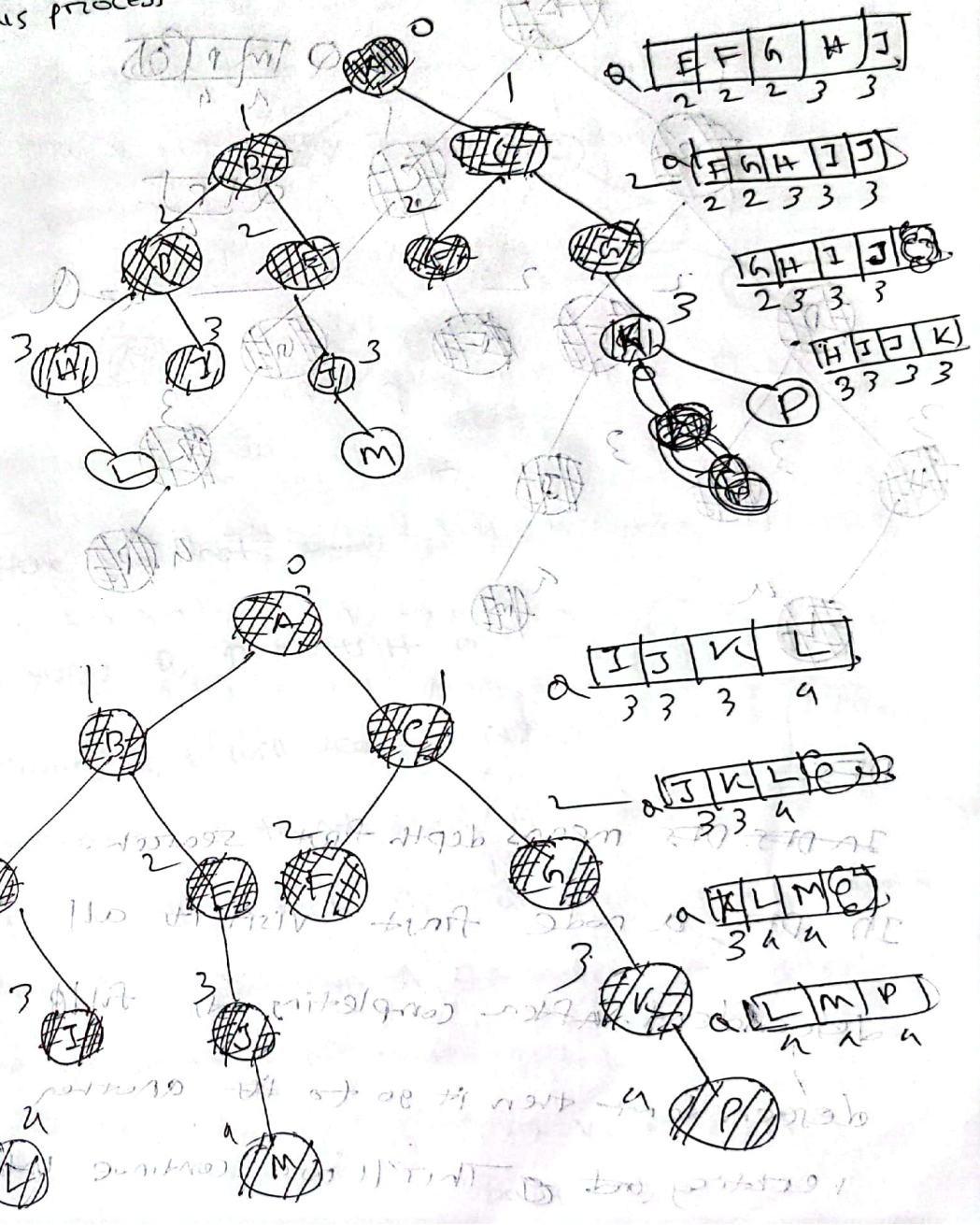
is popped from queue

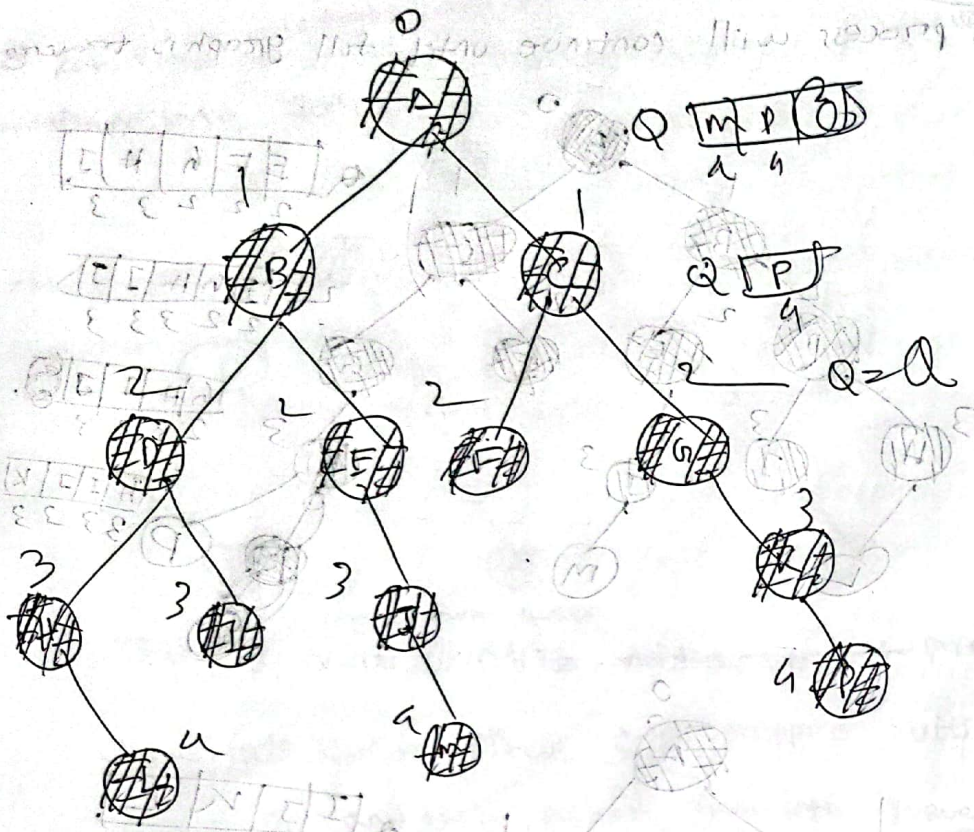
Then B visits D and E. ~~After visit~~ ~~the~~ pushed them into queue Then C is popped visits F and G. and then pushed them into queue so. F is visited. Before visiting F, it visits A, B, C, D, E : ~~(1)~~ ~~so~~ total 5 nodes



~~no need~~

This process will continue until full graph is traversed



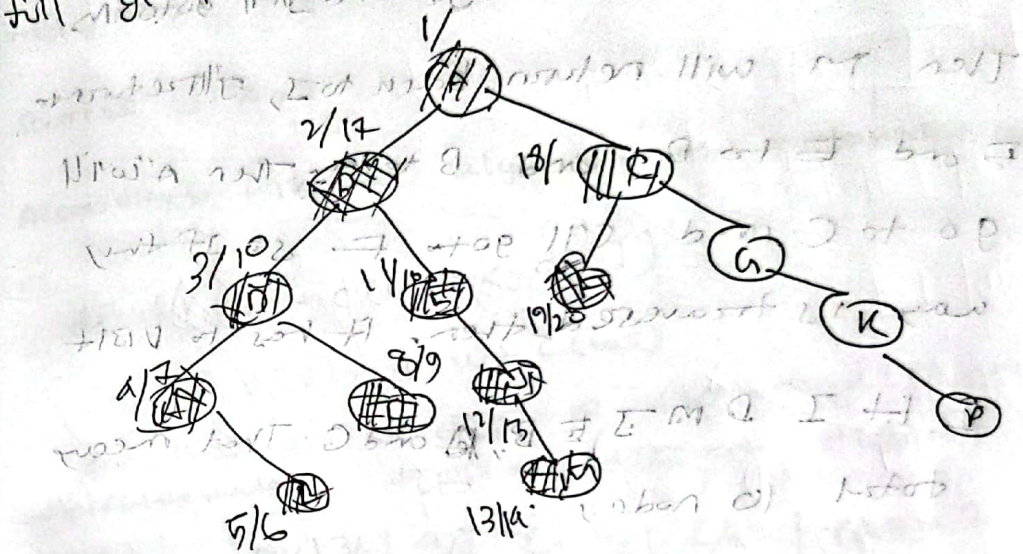


DFS.

~~IA DFS.~~ DFS means depth first search.

In DFS, a node first visit its all descendent. After completing its full descendent then it go to ~~its~~ another ~~recursive~~ ~~of~~ This'll ~~can~~ continue until

full graph is traversed.



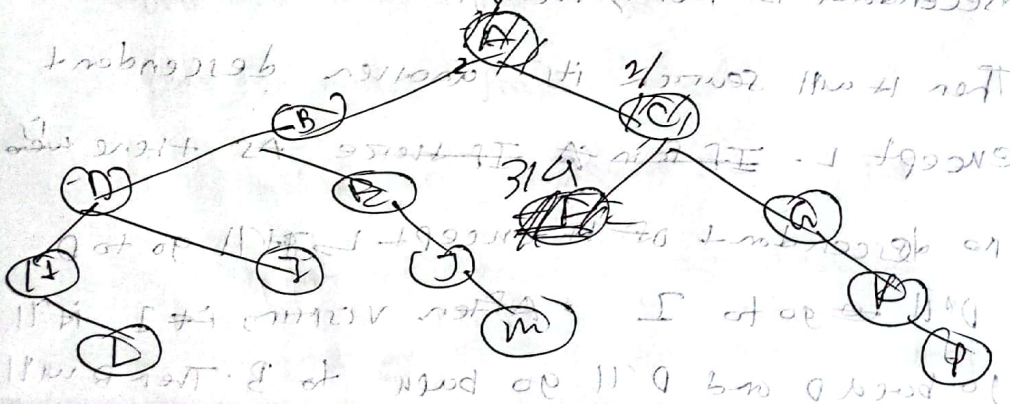
Here ~~At first~~ ~~A~~ A will visits B, then B visits D, D visits H and H visits L. ~~After~~ visiting L, L will search its descendant. If no dsecendant is found, then it'll go to its ancestor. Then H will search its another descendant except L. ~~If found A If there~~ As there ~~is no~~ no descendant of H except L, it'll go to D. D'll ~~go~~ go to I. ~~After~~ visiting I, it'll go back D and D'll go back to B. Then B will

go to E, E'll go to J, and J'll goto M

Then M will return back to J, J'll return to E and E to B and B to A. Then A'll go to C and C'll goto F so, if they way J is traversed, then it has to visit L, I, J, D, M, J, E, B, A and G. That means total 10 nodes.

But if it's traversed from ~~left~~ A Right side

That means ~~at first~~ A'll go to C and C'll go to F. Then F will be visited after two nodes visiting



6.a.

① Let A=1 B=2 C=3 D=4, E=5, F=6, G=7.

Source, S=A=1

According to Dijkstra algorithm the relaxation condition is,

if(d[U] + c(U,V) < d[V])

d[V] = d[U] + c(U,V)

At first, d[A]=0, d[B]=∞ d[C]=∞, d[D]=∞ ~~selected vertex~~ d[E]=∞, d[F]=∞ d[G]=∞

Selected vertex	B/2	3/C	4/D	5/E	6/F	7/G
B	②	7	12	∞	∞	∞
D	②	7	④	∞	∞	∞
F	②	7	④	∞	⑥	∞
C	②	⑦	④	∞	⑥	8
G	②	⑦	④	9	⑥	⑧
E	②	⑦	④	⑨	⑥	⑧
	2	7	4	9	6	8

C(A,B)= 2
C(A,C)=7
C(A,E)=12
C(B,D)=2
C(C,B)=3
C(C,D)=1
C(C,E)=2
C(D,F)=2
C(E,G)=-7
C(E,A)=-4
C(F,G)=2
C(G,D)=1

d[B] = ∞ d[A]+C(A,B) = 0+2.

2 < ∞ By the same way

d[B]= 2 d[C]= 7, d[E]= 12

$d[B] = 2$

$d[D] = 12$ $\quad d[B] + C(B,D)$

$\geq 2+2$

$= 4$

$d[D] < d[B] + C(B,D)$

$d[D] = 4$

$d[F] = \infty$

$d[D] + C(D,F) = 4+2 = 6 < \infty$

$d[F] = 6$

$d[G] = \infty$

$d[F] + C(F,G) = 6+2 = 8 < \infty$

$d[G] = 8$

$d[B] = 2$

$d[D] + C(C,B) = 2+3 > 2$

$d[D] = 4$

$d[C] + C(C,D) = 2+4 > 4$

$d[E] = \infty$

$d[C] + C(C,E) = 7+2 = 9 < \infty$

$d[E] = 9$

$d[D] = 4$

$d[G] + C(G,D) = 8+1 = 9 > 4$

$d[G] = 8$

$d[E] + C(E,G) = 9-7 = 2 < 8$

$d[A] = \infty$

$d[E] + C(E,A) = 9-4 = 5 > \infty$

~~Here~~

According to Dijkstra Algorithm this ~~A~~ is the shortest path. But here in last stage value of $d[G]$ is changed. If it's again relaxed then this might give another answer.

selected vertex	B	C	D	E	F	G.
G	2	7	4	9	6	12
D	2	2	3	9	6	12
F	2	2	3	9	5	12

$d[D] = 4$

$d[G] + C(G,D) = 2 + 1 = 3 < 4$

$\therefore d[D] = 3$

$d[F] = 6$

$d[D] + C(D,F) = 3 + 2 = 5 < 6$

$d[F] = 5$

$d[G] = 2$

$d[F] + C(F,G) = 5 + 2 = 7 > 2$

(ii) For, vertices D, F & G, we got wrong path shortest path from DFG from A are 3, 5 2 respectively. The correct path for D, F, G should have been 4, 6, 8 respectively

(iii) At first for edge (E, G) we got the wrong path. If it's removed then Dijkstra might be given correct answers If that's removed the table will look like

vertex	B	C	D	E	F	G.
shortest path from A	2	7	4	9	6	8

b.

vertex	A	B	C	D	E	F	G	H
Indegree	1	0	1	1	1	1	1	1

P.Queue: {B}

If B is removed.

vertex	A	B	C	D	E	F	G	H
Indegree	0	0	0	1	1	1	1	1

P.Queue = {A, C}

ordering = {B}

vertex	A	B	C	D	E	F	G	H
Indegree	0	0	0	1	0	1	1	1

P.Queue = {C, E}

ordering = {B, A}

vertex	A	B	C	D	E	F	G	H
Indegree	0	0	0	0	0	1	1	1

P.queue {C, E, F}

ordering {B, A}

vertex	A	B	C	D	E	F	G	H
Indegree	0	0	0	0	0	0	1	1

P.queue {F, H}

ordering {B, A, C, D, E}

vertex	A	B	C	D	E	F	G	H
Indegree	0	0	0	0	0	0	0	1

P.queue {H}

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

P.queue ϕ

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

Except Lexiographical ordering: B A C D E F G H

But if lexiographical ordering then Queue is used in place of P.Queue.

c. The algorithm follows

1. Initialize the flow in all the edges to 0

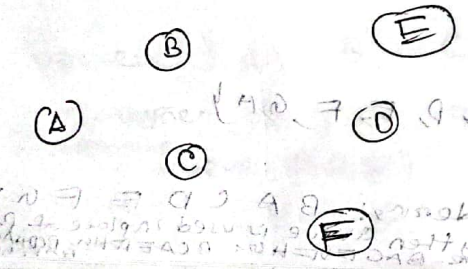
2. While there is an augmenting path between the source and the sink add this path to the flow

3. Update the residual graph

We can also consider reverse path if required because if we don't consider them we may never find a max. flow.

4. a there if bold ages form minimum spanning trees then x<0.

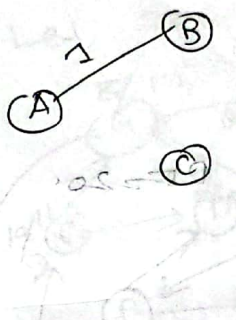
If x>0 then in place of DF, CD is selected to form mst so, the range of x is x<0



Here from A

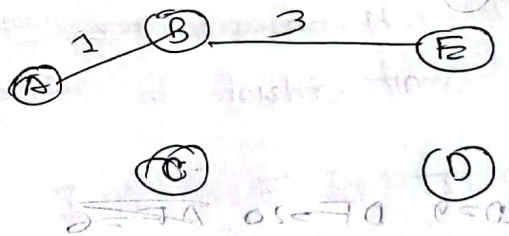
AB=1 AC=10 AF=6.

From them, AB is le less so AB edge is selected



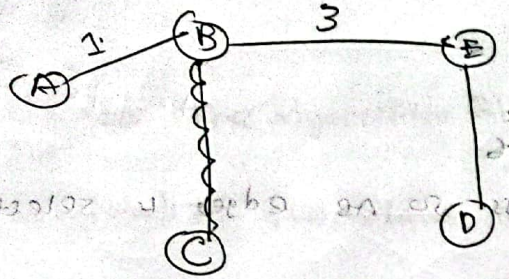
Now, AC=10, AF=6, BC=5, BE=3

All of them, BE is less. so BE edge is selected



AC=10, AF=6 BC=5 ED=3.

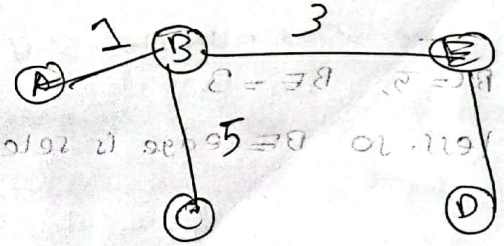
All of them ED is less. so ED edge is selected.



F

Now, AC=10, AE=6, BC=3, ED=9, DF=20

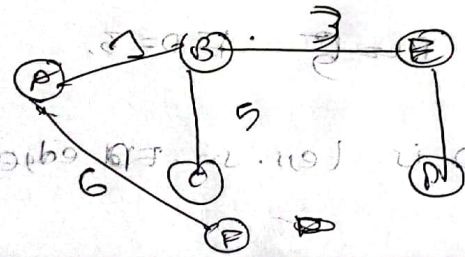
Here, BC is less



F

Now, AC=10, AF=6, CD=9, DF=20, ~~AB=6~~

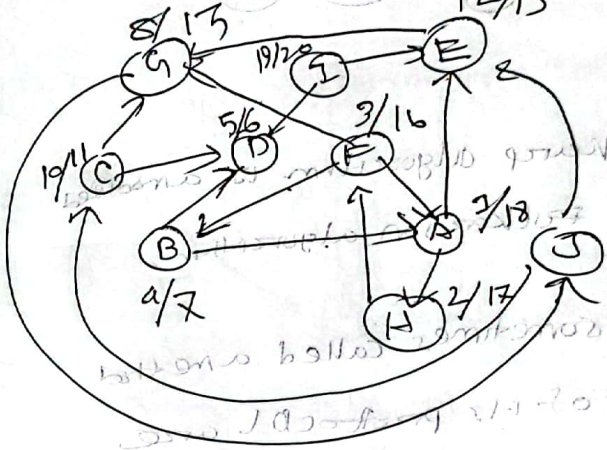
Here AF is less.



So, Cost: 1+3+5+5+6 = 20

Edges. AB BE BC ED AF

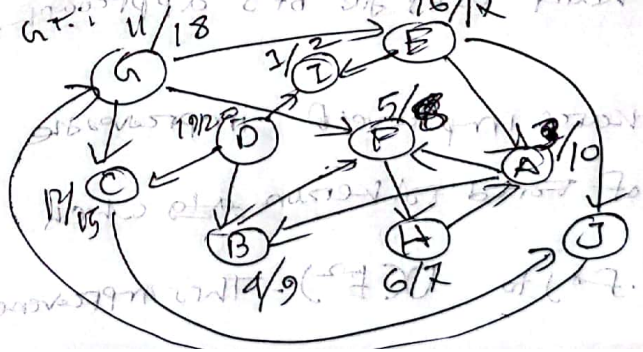
b. Let, DFS is applied in the graph



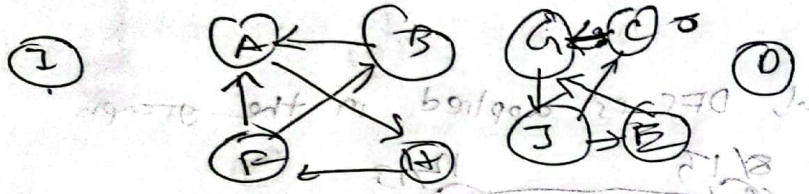
Now, re-arrange it according to decreasing order of finishing time

J I A H F G D E C B D

Now G.T.:



{I} {A B F H} {h, c, J, E} {D}

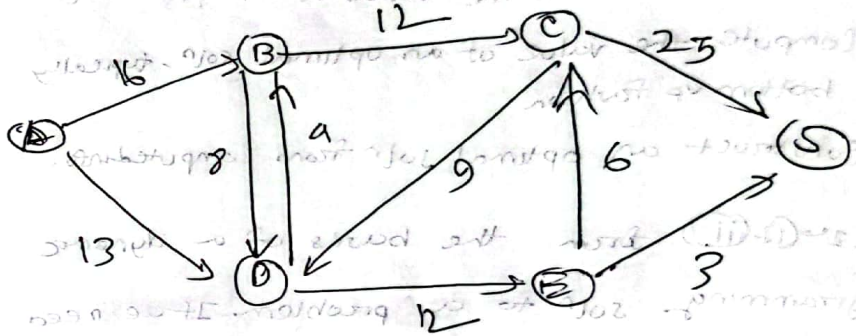


8. b The Edmonds Karp algorithm is an edited form of the Ford Fulkerson algorithm.

Ford Fulkerson is sometimes called a method because some parts of its protocol are left unspecified. On the other hand Edmonds Karp provides a full specification.

- Ford Fulkerson uses DFS approach and Edmonds Karp uses the BFS approach.

Edmonds Karp improves it improves the runtime of Ford Fulkerson algorithm which is $O(E)$ to $O(E^2)$. This improvement is important because it makes the runtime of Edmonds Karp independent of the max flow of the network.



$A \xrightarrow{12/16} B \xrightarrow{12/12} C \xrightarrow{12/23} S$

$A \xrightarrow{3/13} D \xrightarrow{3/12} E \xrightarrow{3/3} S \quad 3$

$A \xrightarrow{9/13} D \xrightarrow{9/12} E \xrightarrow{6/6} C \xrightarrow{18/23} S \quad 6$

21

- max flow: 21.

2k18

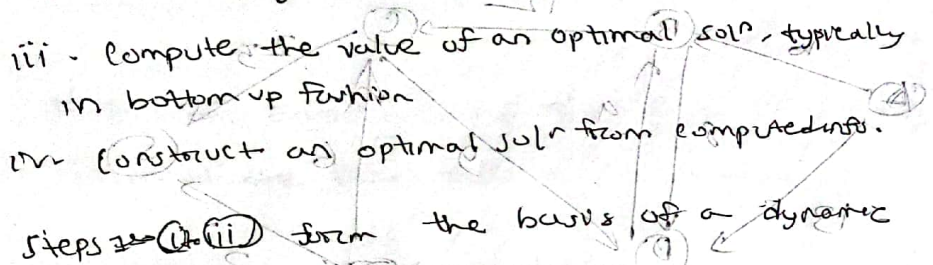
1.(b). When developing a dynamic programming algorithm we follow a sequence of four steps.

i. Characterize the structure of an optimal soln

ii. Recursively define the value of an optimal soln

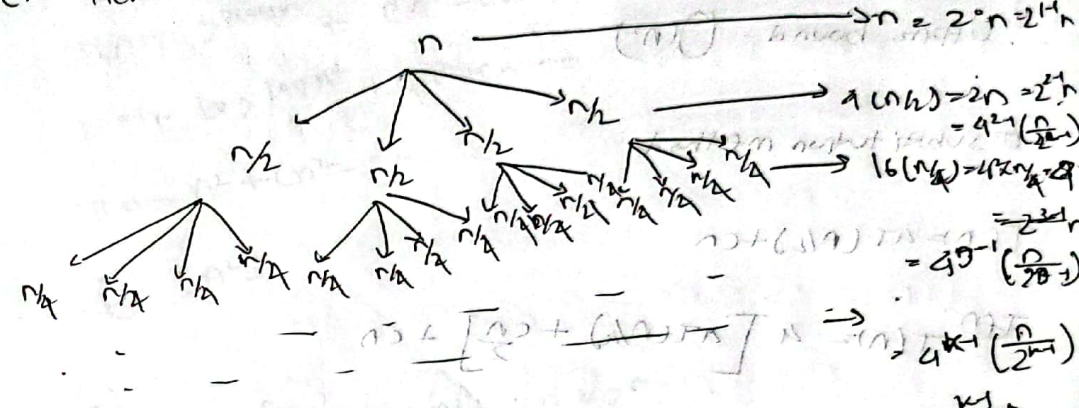
iii. Compute the value of an optimal soln, typically in bottom up fashion

iv. Construct an optimal soln from computed info.



Steps (i) (ii) (iii) form the basis of a dynamic programming soln to a problem. If we need only the value of an optimal soln and not the soln itself, then we can omit the step (iv). When we do perform step (iv) we sometimes maintain additional information during step (iii) so that we can easily construct an optimal soln

c. Here $T(n) = 4T(n/2) + cn$



$T(n) = 4^k T(\frac{n}{2^k}) + cn(1 + 2 + 2^2 + \dots + 2^{k-1})$

$T(n) = 4^{k-1} T(\frac{n}{2^{k-1}}) + cn(1 + 2 + 2^2 + \dots + 2^{k-1})$

Let, $\frac{n}{2^{k-1}} = 1.$

$n = 2^{k-1}$

$\log n = k-1$

$k = \log n + 1$

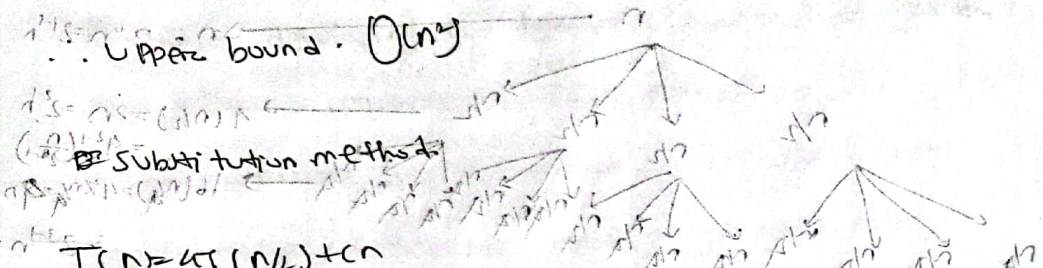
$T(n) = 4^{\log n} T(1) + cn \cdot \frac{2^{k-1}}{2-1}$

$T(n) = 2^{2\log n} + cn \cdot \frac{2^{\log n} - 1}{1}$ or $2^{2\log n} + \frac{cn(n-1)}{1}$

$T(n) = 2^{\log n^2} + cn(n-1)$

$T(n) = n^2 + cn^2 - cn$.

∴ Upper bound. $O(n^2)$



Substitution method:

$T(n) = 4T(n/2) + cn$

$T(n) = 4\left[4T(n/4) + \frac{cn}{2}\right] + cn$

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

$T(n) = 4^2\left[4T(n/8) + cn/8\right] + 3cn$

$T(n) = 4^3 T(n/8) + 7cn$

$T(n) = 4^3\left(4T(n/16) + c \cdot n/8\right) + 7cn$

$T(n) = 4^4 T(n/16) + 15cn$

⋮

$T(n) = 4^k T\left(n/2^k\right) + (2^k - 1)cn$

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

$n = 2^k$

$\log n = k$

$T(n) = 4^{\log n} T(1) + (n-1)cn$

$T(n) = 2^{2\log n} + cn^2 - cn$

$T(n) = 2^{\log n^2} + cn^2 - cn$

$T(n) = n^2 + cn^2 - cn$.

∴ $O(n^2)$

2. a. (i) $f(n) = 1000n^2 + 16n + 2^n$

Let $1000 \cdot 2^n + 16 \cdot 2^n + 2^n = 1017 \times 2^n$.

$g(n) = 2^n$.

$\therefore f(n) \leq c \cdot g(n)$.

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

(ii) $f(n) = 5n + n\log(n) + 1000\log n$.

$f(n) = 5n + 2n\log n + 1000\log n$

Let $5n\log(n) + 2n\log(n) + 1000n\log n$

$\therefore g(n) = 1052 n\log(n)$

$g(n) = n\log(n)$

$\therefore f(n) \leq c \cdot g(n)$.

$\therefore O(g(n)) = n\log n$.

(iii) $\log(n) + 10000 = f(n)$

Now, $\log(n) + 10000\log(n)$

$\therefore 10001 \cdot \log(n)$

$g(n) = \log(n)$.

$f(n) \geq c \cdot g(n)$.

$\therefore O(g(n)) = O(\log n)$.

(iv) $2^{20} + 37 = f(n)$

$g(n) = 1$.

$2^{20} + 2^{20} = 2 \cdot 2^{20} = 2^{21}$.

$\therefore f(n) \geq 2^{21} \cdot 1$.

$f(n) \geq c \cdot g(n)$

$O(g(n))$

b. ~~Pseudocode~~ pseudo code for m coloring problem

At first graph is represented by its boolean adjacency matrix G[1:n, 1:n]. All assignments of 1, 2, ..., m to the vertices of the graph such that vertices are assigned distinct integers are printed.

k is the index of the next vertex to color.

mcoloring(k):

1. Generate all legal assignments for x[k].

2. Assign x[k] a legal color.

3. if (x[k] = 0) then no new color is possible

4. If (k == n) then at most m colors have been used to color the n vertices.

5. write (x[1:n]);

6. else, mcoloring(k+1);

It'll continue until no coloring is possible.

C. Difference between performance analysis and performance measurement

Performance Analysis	Performance measurement
1. It's mainly algorithm dependent	1. It's mainly programming dependent
2. It's independent of language	2. It's depends on language.
3. It's Hardware independent.	3. It's hardware dependent.
4. Time & space function	4. watch time and bytes
5. It's calculated before running	5. It's calculated after running
6. It's mainly approximation.	6. It's the exact testing

3 → Knapsack capacity, m=10

index	1	2	3
total weight	10	3	5
profit	40	20	30
profit/weight	4	6.66	6

According to greedy method, profit/weight is arranged according to non decreasing order

index	2	3	1
total weight	3	5	10
profit	20	30	40
Profit/weight	6.66	6	4

so taking in the stack

Indent	weight	profit
2	3 (10-3)=7	20
3	5 (7-5)=2	30
1	2 (2-2)=0	8

Total weight=10 profit: 58

But in 0/1 Knapsack problem we can't break an object. so, this approach is not correct.

According to dynamic programming.

V[i,w] = max { V[i-1,w], V[i-1, w-wt[i]] + pt[i] }

Tabulation method:

Pi	Wi	0	1	2	3	4	5	6	7	8	9	10
0		0	0	0	0	0	0	0	0	0	0	0
20	3	1	0	0	0	20	20	20	20	20	20	20
30	5	2	0	0	0	20	20	30	30	30	50	50
40	10	3	0	0	0	20	20	30	30	30	50	50

Tabulation method.

i	w_i	0	1	2	3	4	5	6	7	8	9	10
1	10	0	0	0	0	0	0	0	0	0	0	10
2	20	0	0	0	0	20	20	20	20	20	20	40
3	30	0	0	0	20	20	30	30	30	50	50	50
4	5	0	0	0	20	20	30	30	30	50	50	50

x_1 x_2 x_3 ~~10 − 5 = 5~~ ~~5 − 5 = 0~~ 30

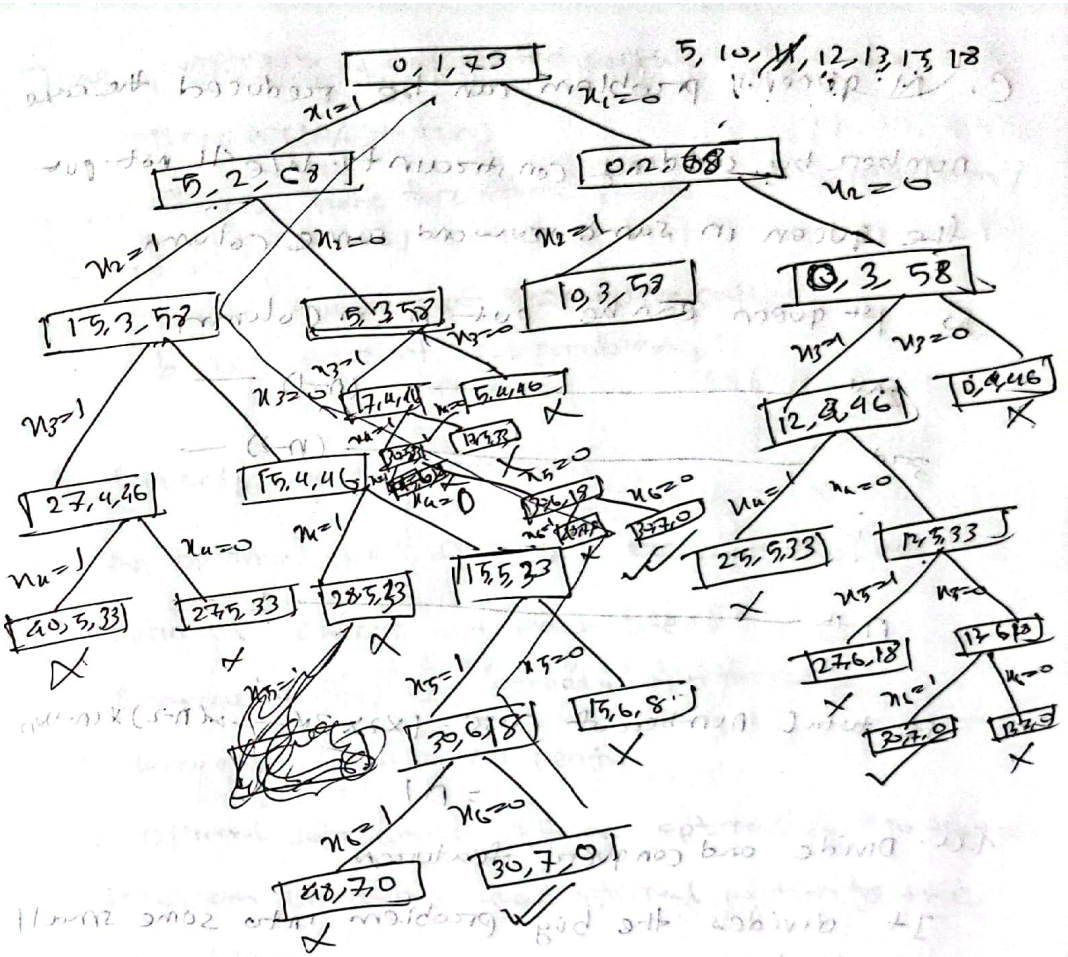
0 1 1 ~~5 = 5~~ 20

∴ object (2,3) are selected

b- Sum of subset problem with six items (n=6) and sum(m)=30. The items are $w[1..6] = \{5, 10, 12, 13, 15, 18\}$

Here, initially, s=0, k=1 and r = ∑w.

A state space tree is drawn in next page ~~and sum of subsets~~ and by backtracking solution is also given in next page.



x_1 x_2 x_3 x_4 x_5 x_6

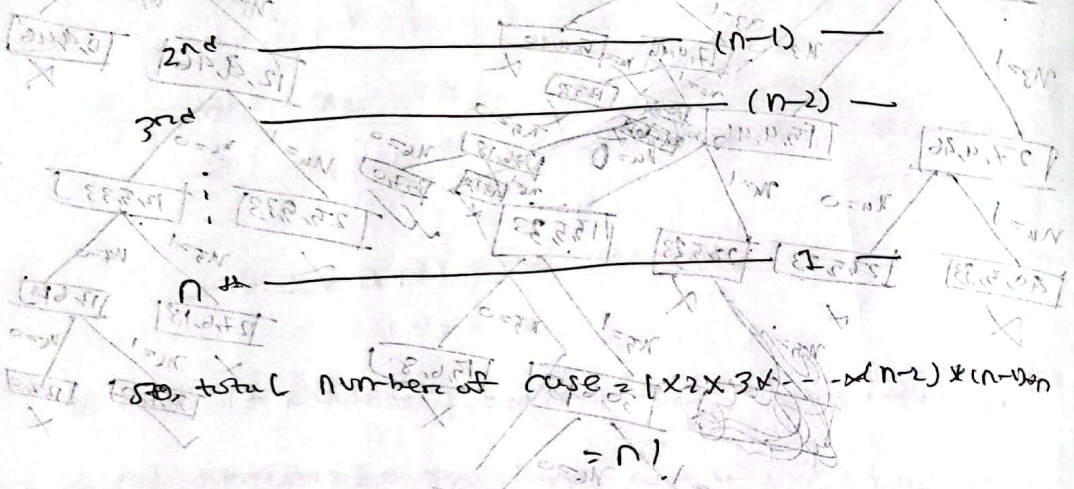
1 0 0 1 0

0 0 1 0 0 1

1 0 1 1 0 0

c. N queens problem can be reduced the case number by adding constraints. We'll not put the queen in same row and same column.

So, 1st queen can be put in n columns.



So, total number of case = 1x2x3x---(n-2)*(n-1)*n

= n!

4.a. Divide and conquers featured.

It divides the big problem into some small problem then try to solve those small problem. If those problem is also too big then those are also cuts into small problems. Recursively this process will continue until the program can solve the problem. Then it'll combine the sub solution and make the result.

Time complexity is calculated with

$T(n) = aT(n/b) + f(n)$

Here $f(n)$ is time for other parts of algorithm / function.

a is number of recursive calls.

b is size of subproblems

b. Greedy choice property:

An optimal soln can be reached by choosing optimal choice at each steps.

Characteristics of greedy optimization

1. Greedy choice property.
2. Optimal subroutine: If an optimal soln to the total problem contains the optimal solution to the subproblems

so greedy will choose an optimal soln in each stage and thus the problem would be solve.

In other word, we can make a decision based on the previous result and then solve the subproblems arised later. Greedy algo.

does not depend on future subproblem choices.

Algorithm: GreedyFractionalKnapsack(w[1..n], p[1..n], W)

for i = 1 to n

do x[i] = 0

weight = 0

for i = 1 to n

if weight + w[i] ≤ W then

x[i] = 1

weight = weight + w[i]

else

x[i] = (W - weight) / w[i]

return x

C. For obst C[i,j] = min_{i≤k≤j} {C[i,k-1] + C[k+1,j]} + (C[i,j] + w[i,j])

$W = \{1, 2, 3, 4, 5\}$

$P = \{0.25, 0.20, 0.05, 0.20, 0.30\}$

	0	1	2	3	4	5
0	$w_{00}=0$ $c_{11}=0$ $r_{11}=0$	$w_{11}=0$ $c_{22}=0$ $r_{22}=0$	$w_{22}=0$ $c_{33}=0$ $r_{33}=0$	$w_{33}=0$ $c_{44}=0$ $r_{44}=0$	$w_{44}=0$ $c_{55}=0$ $r_{55}=0$	w_{55} c_{5-}
1	$w_{12}=0.25$ c_{12} r_{12}	$w_{23}=0.20$ c_{23} r_{23}	$w_{34}=0.05$ c_{34} r_{34}	$w_{45}=0.20$ c_{45} r_{45}		
2	w_{13} c_{13} r_{13}	w_{24} c_{24} r_{24}	w_{35} c_{35} r_{35}			
3	w_{14} c_{14} r_{14}	w_{25} c_{25} r_{25}				
4	w_{15} c_{15} r_{15}					

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

$w[5,5]=0$

$c_{11}=0$ $c_{22}=0$ $c_{33}=0$ $c_{44}=0$ $c_{55}=0$

$r_{11}=0$ $r_{22}=0$ $r_{33}=0$ $r_{44}=0$ $r_{55}=0$

$w[i,j] = w[i,j-1] + p_j$

$w[1,2] = w[1,1] + p_1$ $w[2,3] = w[2,2] + p_2$

$w[3,4] = w[3,3] + p_3$ $w[4,5] = w[4,4] + p_4$

K=	1	2	3	4	5
P(a)	0.25	0.20	0.05	0.20	0.30

j-i	0	1	2	3	4	5
0	$w_{00}=0$ $c_{00}=0$ $r_{00}=0$	$w_{11}=0$ $c_{11}=0$ $r_{11}=0$	$w_{22}=0$ $c_{22}=0$ $r_{22}=0$	$w_{33}=0$ $c_{33}=0$ $r_{33}=0$	$w_{44}=0$ $c_{44}=0$ $r_{44}=0$	$w_{55}=0$ $c_{55}=0$ $r_{55}=0$
1	$w_{01}=0.25$ $c_{01}=0.25$ $r_{01}=1$	$w_{12}=0.20$ $c_{12}=0.20$ $r_{12}=2$	$w_{23}=0.05$ $c_{23}=0.05$ $r_{23}=3$	$w_{34}=0.20$ $c_{34}=0.20$ $r_{34}=4$	$w_{45}=0.30$ $c_{45}=0.30$ $r_{45}=5$	
2	$w_{02}=0.45$ $c_{02}=0.65$ $r_{02}=1$	$w_{13}=0.25$ $c_{13}=0.30$ $r_{13}=2$	$w_{24}=0.25$ $c_{24}=0.70$ $r_{24}=4$	$w_{35}=0.50$ $c_{35}=0.70$ $r_{35}=5$		
3	$w_{03}=0.5$ $c_{03}=0.33$ $r_{03}=1/2$	$w_{14}=0.45$ $c_{14}=0.75$ $r_{14}=2/4$	$w_{25}=0.55$ $c_{25}=0.80$ $r_{25}=4$			
4	$w_{04}=0.7$ $c_{04}=1.05$ $r_{04}=4$	$w_{15}=0.75$ $c_{15}=0.75$ $r_{15}=2$				
5	$w_{05}=1$ $c_{05}=1.63$ $r_{05}=4$					

$w[i,j]=w[i,j-1]+p_j$

$c[i,j]=\min_{i<k\le j}\{c[i,k-1]+c[k,j]\}+w[i,j]$

$c[0,1]=\min_{0<k\le 1}\{c[0,0]+c[1,1]\}+w[0,1]$

$=\min\{(0+0)+0.25\}$

$=0.25$

$c[1,2]=\min_{1<k\le 2}\{c[1,1]+c[2,2]\}+w[1,2]$

$=\min\{(0+0)\}+0.20$

$=0.20$

$c[2,3]=\min_{2<k\le 3}\{c[2,2]+c[3,3]\}+w[2,3]$

$=0.05$

same way $c[3,4]=0.20$ $c[4,5]=0.30$

$c[0,2]=\min_{0<k\le 2}\{c[0,0]+c[1,2], c[0,1]+c[2,2]\}$
$+w[0,2]$

$=\min\{0+0.20, 0.25+0\}+0.45$

$=0.65$ (1)

$c[1,3]=\min_{1<k\le 3}\{c[1,1]+c[2,3], c[1,2]+c[3,3]\}$
$+w[1,3]$

$=\{0+0.05, 0.20+0\}+0.25=0.30$

$$c[2,4] = \min\{c[2,2]+c[3,4],\ c[2,3]+c[4,4]\} + w(2,4)$$

$$= \min(0+0.20,\ 0.05+0) + 0.25$$

$$= 0.30$$

$$c[3,5] = \min\{c[3,3]+c[4,5],\ c[3,4]+c[5,5]\} + w(3,5)$$

$$= (0+0.30,\ 0.20+0) + 0.30$$

$$= 0.70$$

$$c[0,3] = \min_{1 \le k \le 3} \begin{cases} c[0,0]+c[1,3] \\ c[0,1]+c[2,3] \\ c[0,2]+c[3,3] \end{cases} + w(0,3)$$

$$= \min(0+0.30,\ 0.25+0.05,\ 0.65+0) + 0.5$$

$$= 0.85$$

$$c[0,4] = \min\{c[1,1]+c[2,4],\ c[1,2]+c[3,4],$$
$$c[1,3]+c[4,4]\} + w(1,4)$$

$$= \min\{0+0.30,\ 0.20+0.20,\ 0.30+0\} + 0.43$$

$$= 0.73$$

$$c[2,5] = \min\begin{cases} c[2,2]+c[3,5] \\ c[2,3]+c[4,5] \\ c[2,4]+c[5,5] \end{cases} + w(2,5)$$

$$= \min\begin{cases} 0+0.70 \\ 0.05+0.30 \\ 0.30+0 \end{cases} + 0.30$$

$$= 0.80$$

$$c[0,4] = \min\begin{cases} c[0,0]+c[1,4] \\ c[0,1]+c[2,4] \\ c[0,2]+c[3,4] \\ c[0,3]+c[4,4] \end{cases} + w(0,4)$$

$$= \min\begin{cases} 0+0.73 \\ 0.25+0.30 \\ 0.65+0.20 \\ 0.85+0 \end{cases} + 0.2$$

$$= 1.05$$

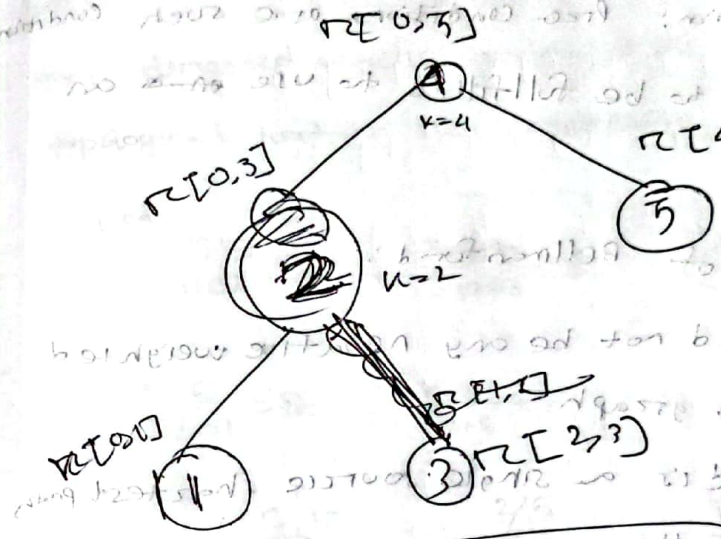
$$C[1,5] = \min \begin{cases} C[1,1] + C[3,5] \\ C[1,2] + C[3,5] \\ C[1,3] + C[4,5] \\ C[1,4] + C[5,5] \end{cases} + w[1,5]$$

$$= \min \begin{cases} 0 + 0.80 \\ 0.20 + 0.60 \\ 0.30 + 0.30 \\ 0.75 + 0 \end{cases} + 0.75$$

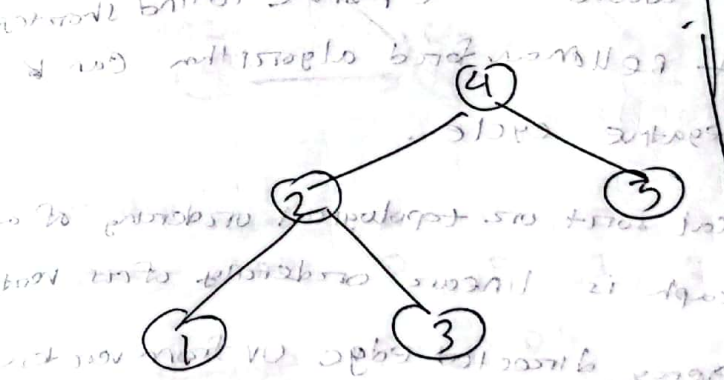
$$C[0,3] = \min \begin{cases} C[0,0] + C[1,3] \\ C[0,1] + C[2,3] \\ C[0,2] + C[3,3] \\ C[0,3] + C[4,3] \end{cases} + w[0,3]$$

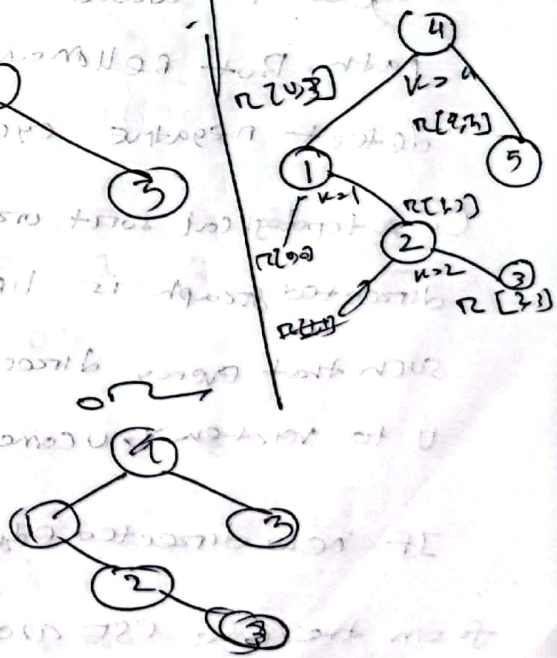
$$= \min \begin{cases} 0 + 1.35 \\ 0.25 + 0.80 \\ 0.65 + 0.70 \\ 0.75 + 0.30 \\ 1.05 + 0 \end{cases} + 1$$

$$= 1.65$$



so the trees are.





5.a. Pre condition: Pre conditions are such conditions which needed to be fullfilled to use an algorithm.

Preconditions of Bellman ford:

This should not be any negative weighted cycle in the graph.

Bellman ford is a single source shortest path algorithm. If there is a negative weighted cycle it would not be posible to find shortest path. But Bellman ford algorithm can detect negative cycle.

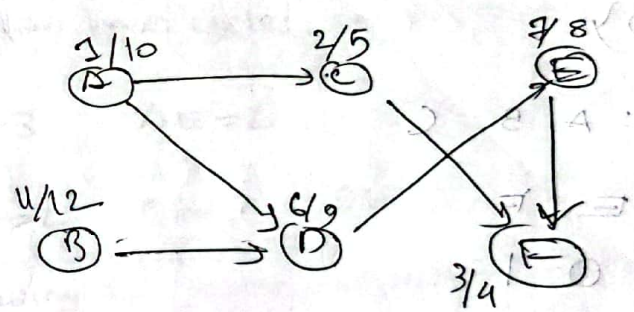
C: A topological sort or topological ordering of a directed graph is linear ordering of its vertices such that every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

If new directed edge is inserted in the graph from the node CSE 4203 to CSE 2173, the cycle will be formed. Topological sort is applicable for directed acyclic graph. As there is cycle so topological sort is not applicable for this.

Let

CSE 1107 = A CSE 1203 = B CSE 2231 = C CSE 2173 = D

CSE 3207 = E CSE 4203 = F



vertex	A	B	C	D	E	F
Indeg	0	0	1	2	1	2

Q → {A, B}

P C

vertex			D	E	F
Indeg			1	1	2

Q → {D, C}

Q = { } Precedence ordering = A

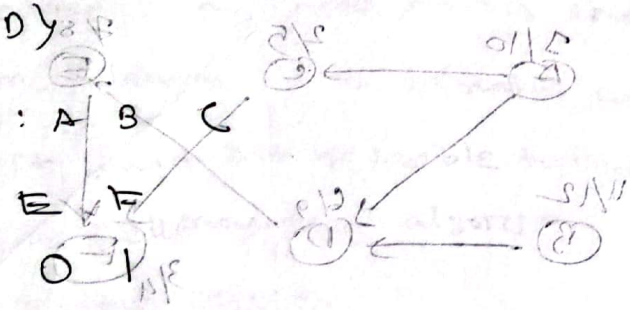
Vertex	C	D	E	F
Indegree	0	0	1	2

Q = {C, D}

Ordering: A B

Vertex	D	E	F
Indegree	0	1	1

Q = {D}



Ordering: A B C

Vertex	E	F
Indegree	0	1

Q = {E}

Ordering: A B C D

Vertex F

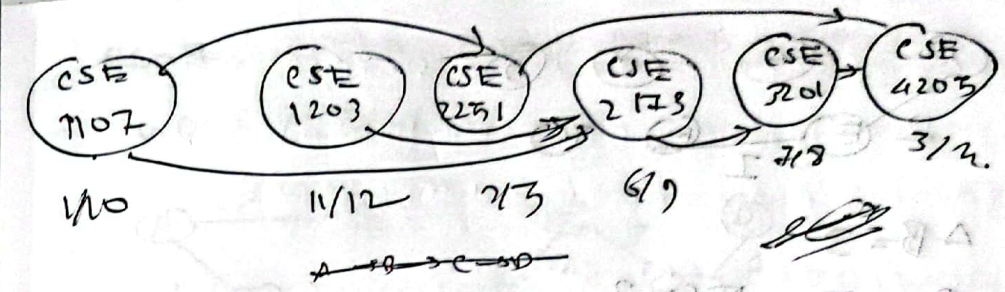
Indegree 0

Q = {F}

Ordering A B C D E

Q = Q

Ordering A B C D E F



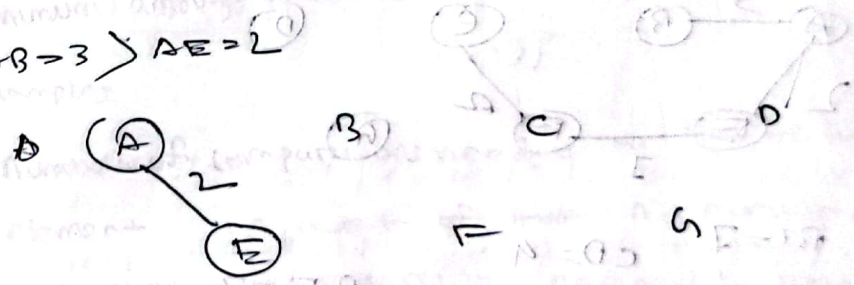
6.a A spanning tree is a subset of graph G which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it can't be disconnected.

AB = 3 AE = 2 BC = 7 BE = 5 CD = 4

CF = 2 CG = 3 DG = 1 EF = 1 FG = 9

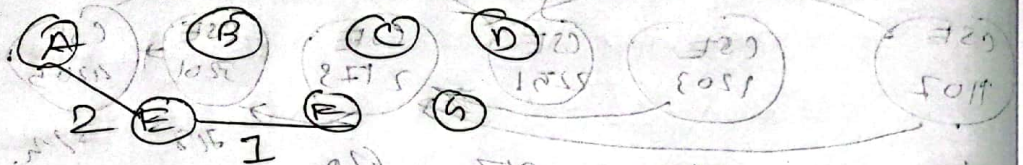
According to prims algorithm A is source.

AB = 3 > AE = 2



AB = 3 EF = 1 BE = 5

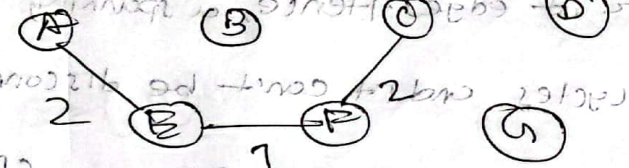
EF is less.



AB = 3

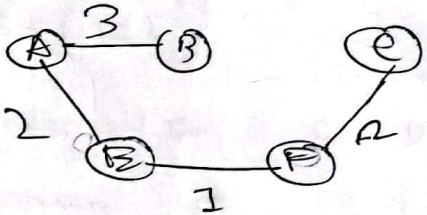
BE = 5 FC = 2

Here FC is less



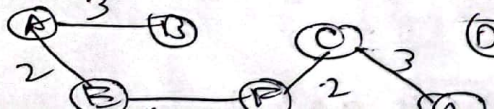
AB = 3 BE = 5 FG = 9 CG = 3 CD = 4

AB is less.



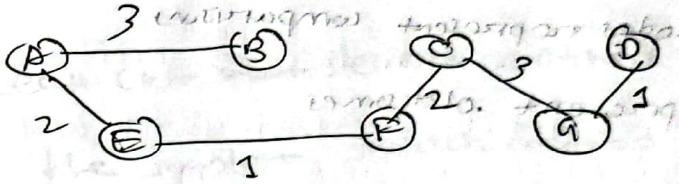
BC = 7 CD = 4

CD is less



BC = 7 CD = 4 FG = 9 BE = 5 GD = 1

GD is less



(i). A to D

A → E → F → C → G → D

2 1 2 3 1 = 9

(ii) Cost of MST

3 + 2 + 1 + 2 + 3 + 1 = 12

b. Lower bound: Lower bound is an estimate on a minimum amount of work needed to solve a given problem.

Examples:

i. Numbers of comparisons needed to find the largest element in a set of n numbers

ii. Numbers of comparisons needed to sort an array of size n.

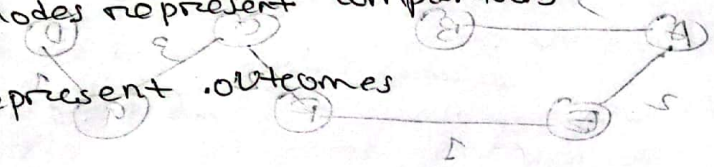
iii. Number of comparisons necessary for searching in a sorted array

iv. Number of multiplications needed to multiply two n by n matrices.

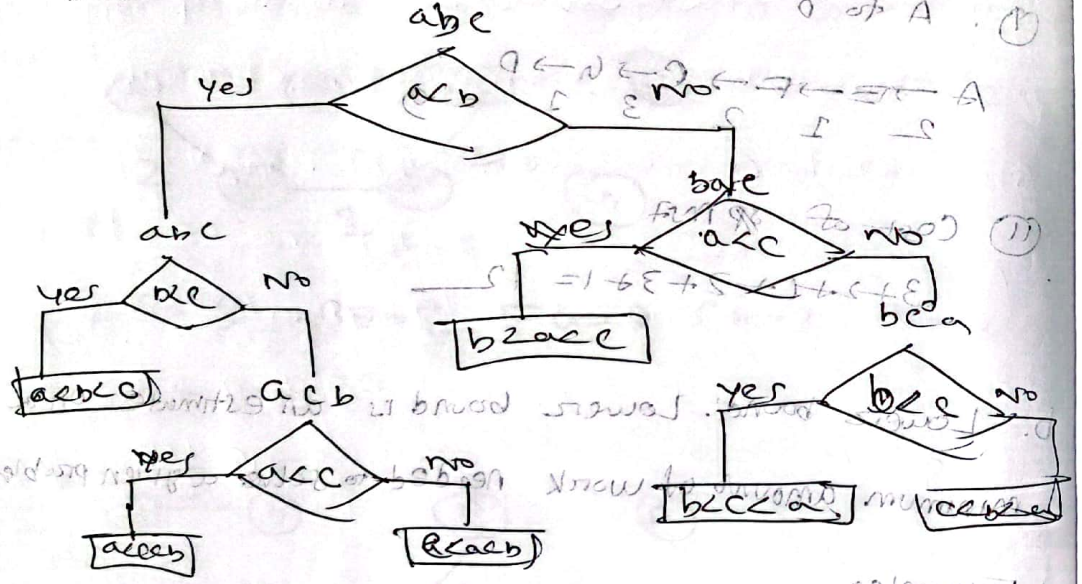
Decision/comparison tree: A convinient model of algorithms involving comparisons in which

i. internal nodes represent comparisons

ii. leaves represent outcomes



Decision tree for 3 element insertion sort.



A comparison based sorting algorithm can be represented by a decision tree. Number of leaves (outcomes) $\geq n!$ Height of a binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$

Min. no. of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$

for any comparison-based sorting

$\lceil \log_2 n! \rceil \approx n \log n$.

C. Least cost search method is one of the simplest search method that is guaranteed to find min cost path, which is similar to bfs, but instead of expanding a path with the fewest number of arcs it selects a path with the lowest cost.

Least cost search is similar to DFS and BFS

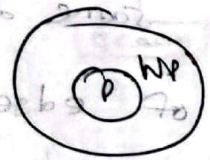
If from the graph cost is removed, then BFS and DFS will be same as the LC search.

By removing cost of edges, LC search can be converted into BFS & DFS

P(h) problems are problems that can be solved in polynomial time. Example: multiplication, searching & sorting.

NP problems are problems that are solved by non deterministic algorithm in exponential time but their solution can be verified easily by polynomial time. Ex: TSP, knapsack 0/1.

NP problems are solved by non deterministic algorithms, which means we don't know the statements inside. But we can discover them later on and reduce their time complexity to solve in polynomial time. Which means the problem is now reduced to P class so all NP problems can be a part of P class.



C. Bin packing problem involves assigning n items of different weights and bins each of capacity C to a bin such that number of total uses bins is minimized. It may be assumed that all items have weights smaller than bin capacity

Here, Bin size B=10

Items 7 2 5 1 9 4 3 6

1. First fit:

Here every time check the first bin, if it's able to take any items then it'll take it, otherwise it'll pass to its next bin

7 is placed in bin 1

2 is placed in bin 1

Here, (7+2)=9 & remaining (10-9)=1

so, 5 doesn't take place in bin 1

5 takes place in bin 2

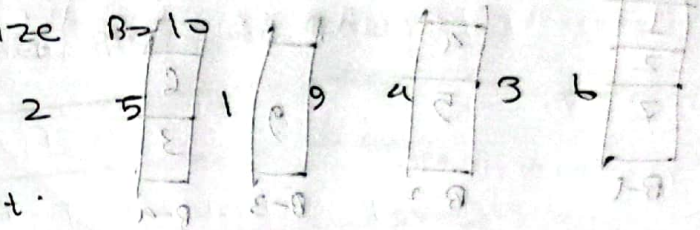
1 " " " bin 1

now bin 1 is fullfilled. Bin 2 remains (10-5)=5

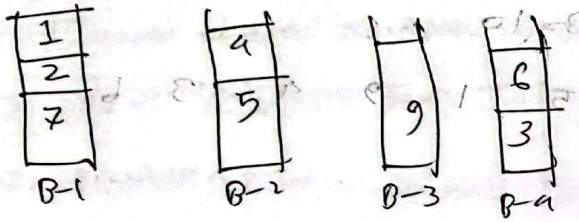
9 takes place in bin 3

4 " " " 2

Bin 2 remains (5-4) = 1.



3 & 6 takes place in bin 4.



M=4.

Best fit:

7 At first takes place in Bin 1

2 takes place in Bin 1

5 takes place in Bin 2

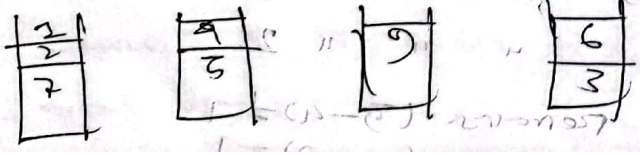
1 takes place in Bin 1

9 takes place in Bin 3

4 takes place in Bin 2

3 takes place in Bin 4

6 takes place in Bin 4.



iii. Next fit:

Here, if it's not fit in the current bin, then a new bin is created.

so 7 takes place in Bin 1

2 takes place in Bin 1

5 takes place in Bin 2

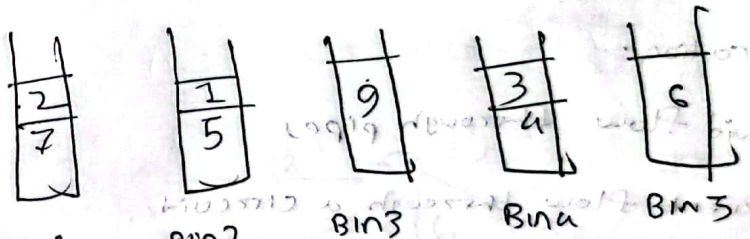
1 takes place in Bin 2

9 takes place in Bin 3

4 takes place in Bin 4

3 takes place in Bin 4

6 takes place in Bin 3



8. a. Maximum flow network is the maximum amount of flow that the network would allow to flow from source to sink.

The maximum flow problem can be seen as a

Speial case of more complex network flow problems such as the circulation problem. Multiple algorithm exists in solving the max flow problem.

Characteristics:

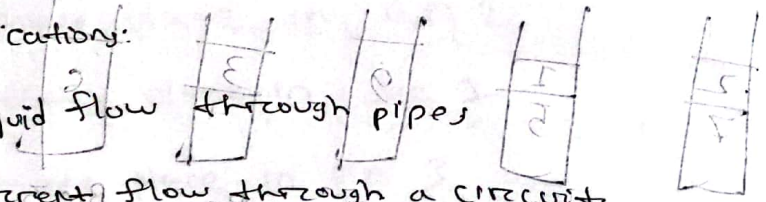
i. source: materials are produced at a steady rate

ii. Sink: materials are consumed at the same rate

iii. flows through conduits are constrained to max values

Applications:

1. Liquid flow through pipes

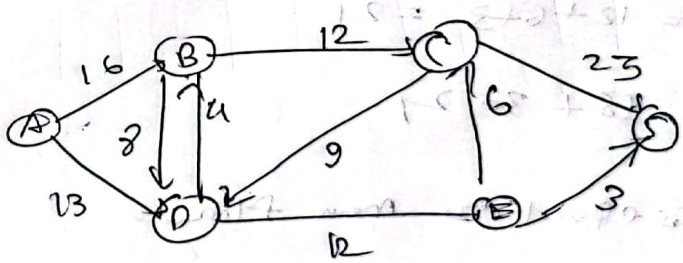


ii. Current flow through a circuit

iii. Information in a communication network

b. Difference between Ford Fulkerson and Edmond Karp algorithm

Ford Fulkerson	Edmond Karp
i. $O(E * F)$	i. $O(V * E^2)$
ii. Uses DFS	ii. Uses BFS
iii. Randomly selects a path for augmenting	iii. only selects a shortest path if the residual capacity not zero
iv. We can select longer augmenting paths.	iv. Always selects the shortest augmenting path.
v. Does depend on the capacity value of any edge	v. It doesn't depend on the capacity value of any edge

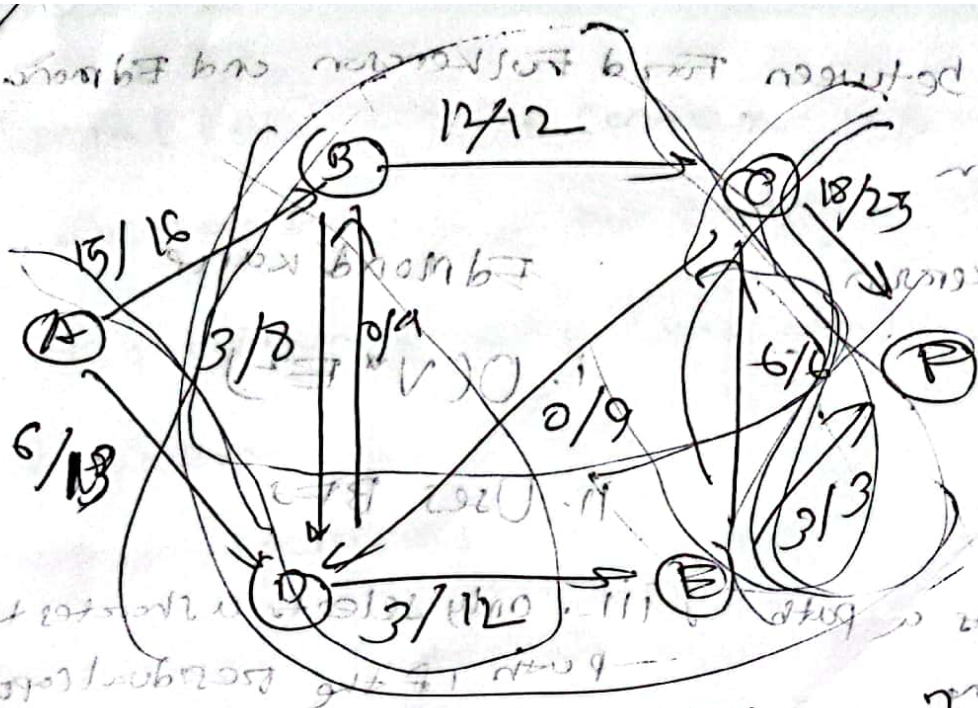


source = A
sink = S

A → B → C → S ⟶ 12

A → B → D → E → S ⟶ 3

A → D → S, E → S, C → S ⟶ 6



(i) max flow = 12+3+6 = 21

(ii) min cut = 15+6 = 21

~~which is eq~~

min cut = 12+3+3+3 = 21

min cut = 12+6+3 = 21

min cut = 18+3 = 21

which is equal to max flow