Sec - 01

1+3; Algorithm: An algorithm is a finite set of instruction that, accomplishes a particular task In addition, all algorithms must satisfy the following criteria -

① Input
② Output
③ Definiteness
④ Finiteness
⑤ Effectiveness

Input: Zero or more quantities are externally supplied.

Output: At least one quantity is produced

Definiteness: Each instruction is clear and unambiguous.

Finiteness: For any input, the algorithm must terminate after a finite number of steps.

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 is not enough that each operation be definite as in criterion definiteness. It also must be feasible.

2: O-notation: Big O notation is used in computer science to describe the performance or complexity of an algorithm. Big O specifically describe the worst-case scenario and can be used to describe the execution time required or the space used by an algorithm.

Big O notation is something that can appear more confusing than it actually is. at it's simplest. Big O notation is a way to represent the relative complexity of algorithm. also use for-

(I) Similar purpose
(II) More complex portion
(III) Worst case scenario
(IV) Machine and Instance Independent
(V) Not a speed comparison.

4. Beast Case: Fastest time to complete, with optimal inputs chosen. for example the best case for a ~~many~~ Sorting algorithm would be data that's already sorted.

worst Case: Slowest time to complete, with pessimal input chosen. for example the worst case for a sorting algorithm might be data that's sorted in reverse order.

Avarage Case: Arithmatic mean. Run the algorithm many times, using many different inputs of size n that come from some distribution that generates these inputs compute the total running time and divide by the number of trials.

Algorithm	Best Case	Average case	Worst case
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n)^2$	$O(n^2)$	$O(n^2)$

5| Assymptotic notation (O, Ω, Θ)

Big-O: Big O commonly written as O, is an Assymptotic Notation for the worst case or ceiling of growth for a given function. It provides us with an asymptotic upper bound for the growth rate of runtime of an algorithm.

$f(n)$ is $O(g(n))$ if for some real constants c ($c>0$) and n_0, $f(n) <= c * g(n)$ for every input size n ($n>n_0$)

Big-Omega: Big Omega commonly written as Ω, is an Asymptotic Notation for the best case, or a floor growth rate for a

given function. It provides us with an asymptotic lower bound for the growth rate of runtime of a algorithm.

f(n) is $\Omega(g(n))$, if for some real constants c ($c > 0$) and n_0 ($n_0 > 0$), f(n) is $>= c * g(n)$, for every input size n ($n > n_0$)

Theta: Theta commonly written as Θ, is a Asymptotic Notation to denote the asymptotically tight bound on the growth rate of runtime of an algorithm.

f(n) is $\Theta(g(n))$, if for some real constants c1, c2 and n_0 ($c1 > 0$, $c2 > 0$, $n_0 > 0$)

$c1 * g(n)$ is $< f(n)$ is $< c2 * g(n)$ for input size n ($n > n_0$)

Set - 2

1. Devide and Conquer Algorithm:

Like Greedy and Dynamic programming, Divide and conquer is an algorithmic paradigm. A Typical Divide and Conquer algorithm solves a problem using following three steps-

① Divide: Divide the given problem into sub-problems of some type.

② Conquer: Recursively sole these subproblem.

③ Combine: Appropriately combine the answer.

Following are some standard algorithms that are Divided and Conquer algorithms.

(I) Binary Search
(II) Quick Sort
(III) Merge Sort
(IV)

Compare between them

Divide and Conquare	Dynamic programming
An algorithm that recursively breaks down a problem into two or more sub-problems of the same type until it becomes simple enough to be solved directly.	An algorithm that helps to efficiantly solve a class of problem that have overlapping subproblems and optimal
Sub-problem are independent of each other	Sub-problems are inter-dependent
Recursive	Non-recursive
More time consuming as it solves each sub-problem	Less time-consuming as it solves problem
Less efficiant	More efficiant
Used by merge Sort, quicksort and binary Search	Used by matrix chain multiplication, optimal binary search tree.

②: Heap Sort Algorithm:

step 1: Start

step 2: Store N integer values in array.

step 3: Construct a Binary tree with given list of element.

step 4: Transform the Binary tree into min Heap.

step 5: Create a new node at the end of heap.

step 6: Assign new value to the node

step 7: Compare the value of this child node with it's parent.

Step 8: If value of parent is less than child, then swap them.

Step 9: Repeat step 7 and step 4 until Heap property holds.

Step 10: Put the deleted element into the sorted list

Step 11: Display the sorted list.

Step 12: Stop.

3| Quick Sort Algorithm:

step 1 : start

step 2 : Store the input Array.

step 3 : Chose the highest index value has pivot.

step 4 : Take two veriables to point left

step 5 : Left point to the high index.

step 6 : Right point to the less index

step 7 : while value at left is less than pivot move right

step 8 : while value at right is greeder than pivot move left

step 9 : If both step 7 and step 8 does not match swap left and right.

step 10 : if left ≥ right, the point where they met is new pivot.

step 11 : Display the Sorted list.

step 12 : stop.

4 Huffman Coding:

example

a	b	c	d	e	f
5	9	12	13	16	45

Solution: There are mainly two major parts in Huffman Coding.

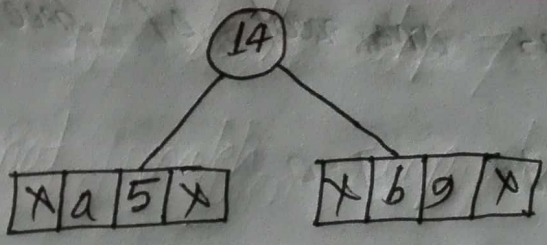
① Build a Huffman tree from input characters

② Traverse the Huffman tree and assign codes to characters.

① Build a Huffman tree:

Step 1: Build a min heap that contains 6 nodes where each node represents root of a tree with single node

Step 2: Extract two minimum frequency nodes from min heap. add a new internal node with frequency 5+9=14

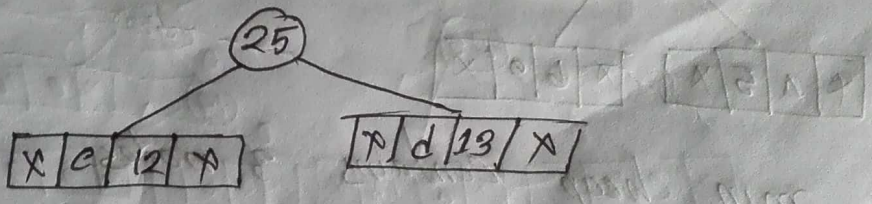


Now min heap contains 5 nodes wher 4 nodes are roots of trees with single element. each and one heap node is root of tree with 3 element.

c	d	IN	e	f
12	13	14	16	45

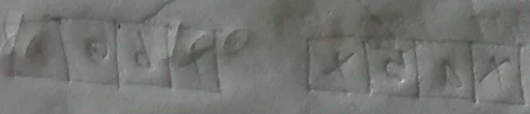
i.e. IN = Internal node.

Step 3: Extract two minimum frequency nodes from heap. Add a new node with frequency 12+13=25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each and

two heap nodes are root of tree with more than one nodes.



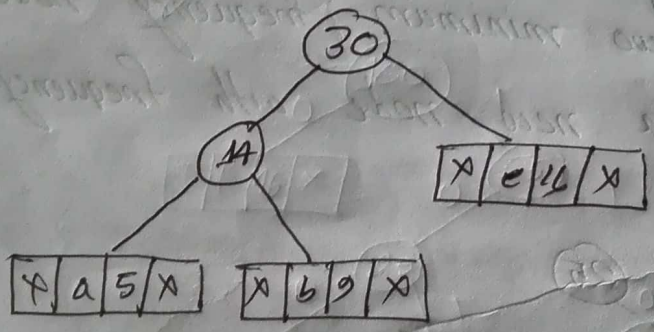
Internal Node 14

e 16

Internal Node 25

f 45

step 4: Extract two minimum frequency nodes. Add new internal node with frequency 14+16=30



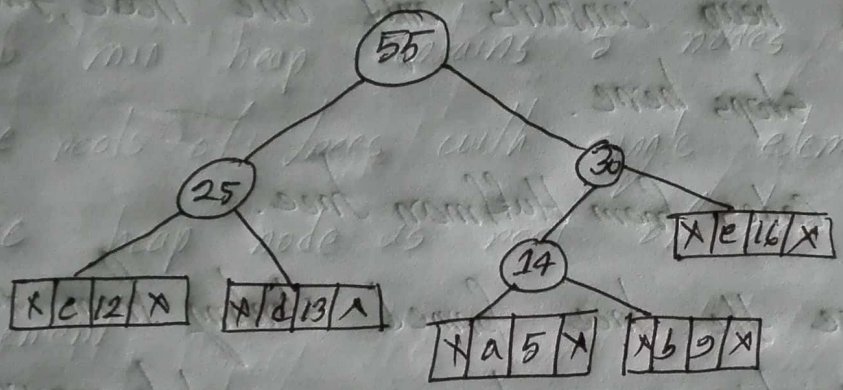
Now min heap contains 3 nodes.

Internal Node 25

Internal Node 30

f 45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25+30=55

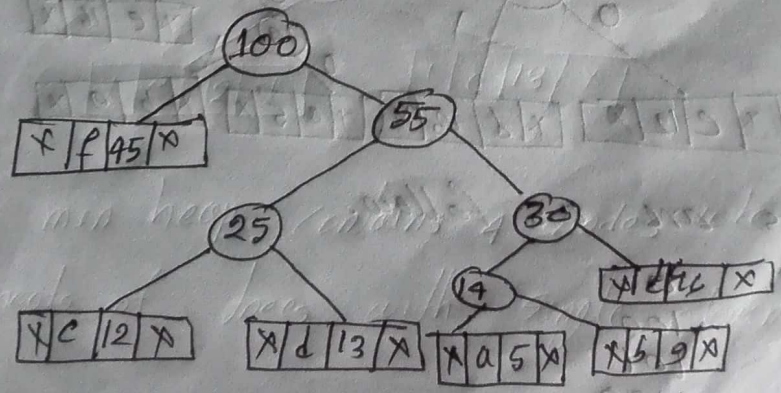


Now min heap contains 2 Nodes

f 45

Internal Node 55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45+55=100



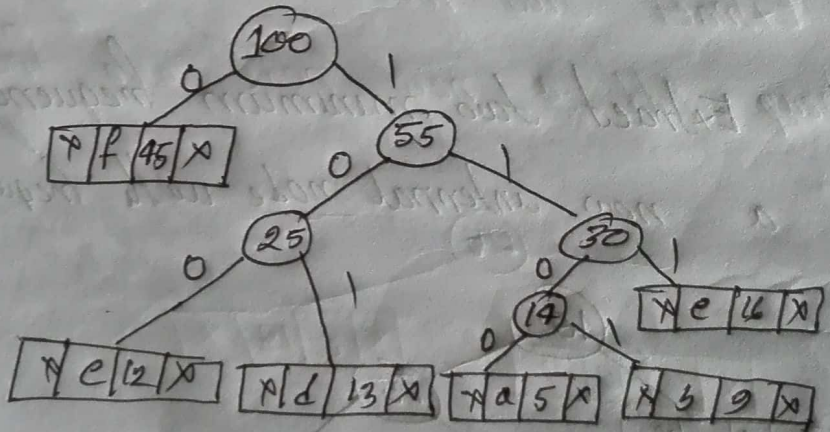
Now min heap contains only one node

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

② print codes from Huffman tree.

Traverse the tree formed starting from the root. Maintain an auxiliary array. while moving to the left child, write 0 to the array. while moving to the right child, write 1 to the array.



The codes are as follow.

f	0
c	100
d	101
a	1100
b	1101
e	111

Algorithm:

Huffman(C)

1. $n = |C|$
2. $Q = C$
3. for $i = 1$ to $n-1$
4. allocate a new node z
5. $z.left = x = Extract\_Min(Q)$
6. $z.right = y = Extract\_Min(Q)$
7. $z.freq = x.freq + y.freq$
8. Insert (Q, z)
9. return $Extract\_Min(Q)$

Set - 3

1) Optimization: optimization is a program transformation technique, which tries to improve the code by making it consume less resource and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below-

(i) The output code must not, in any way, change the meaning of the program.

(ii) Optimization should increase the speed of the program and if possible, the program should demand less number of resources.

(iii) Optimization should itself be fast and should not delay the overall compiling process.

Optimization of Binary Search tree: A Binary Search tree is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically. For each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node.

Search time of an element in a BST is O(n), where as in a balanced-BST search time is O(log n). Again the search time can be improved in optimal cost Binary search tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

2.) Term of optimum Solution:

An Optimal Solution is a feasible Solution where the objective function reaches it's maximum (or minimum) value - for example, the most profit, or the least cost. A globaly optimal solution is one where there no other feasible solutions with better objectives function values.

Two main properties of a problem suggest that the given problem can be solved using Dynamic programming. These properties are overlaping sub-problems and optimal substructure.

Dynamic programming algorithm is designed using the following four steps -

(i) characterize the structure of an optimal solution.

(ii) Recursively define the value of an optimal solution.

(iii) Compute the value of an optimal solution, typically in a bottom-up fashion

(iv) Construct an optimal solution from the computed information.

3.1 Matrix multiplication in dynamic programming:

We shall use the dynamic programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence that we stated at the begining of this chapter.

1. Characterize the structure of an optimal solution.
2. Recursively defines the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

we shall go through these steps in order, demonstrating clearly how we apply each step to the problem.

```
1:  n := length[P] - 1
2:  for i := 1 to n
3:      m[i, i] := 0
4:  for l := 2 to n
5:      for i := 1 to n - l + 1
6:          j := i + l - 1          m[i, j] := ∞
7:          for k := i to j - 1
8:              q := m[i, k] + m[k+1, j] + d_{i-1} · d_k · d_j
9:              if q < m[i, j]
10:                 m[i, j] := q    s[i, j] := k
11: return m and s
```

4.1

Dynamic programming advantage:

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problems, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achive the best solution.

The following computer problems can be solved using dynamic programming approach-

(I) Fibonacci number series

(II) Knapsack problem

(III) Tower of Hanoi

(IV) All pair shortest path by Floyd Warshall

(V) Shortest path by Dijkstra

(VI) Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

5. Longest Common Subsequence :-

We have two string X = ABCD and Y = ABD to find the longest common subsequence.

string X

A	B	C	D

string Y

A	B	D

Making completed LCS table:

step 1: This table is used to store the LCS sequence for each step of the calculation. The first column and first row have been filled in with 0, because when an empty sequence

is compared with a non empty sequence, the LCS is always an empty sequence.

		A	B	C	D
	0	0	0	0	0
A	0				
B	0				
D	0				

Step 2: A row completed

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

Step 3: B row completed

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

Step 4: Complete the D row ~~and~~

	A	B	C	D
0	0	0	0	0
A 0	1 →	1 →	1 →	1
B 0	1 ↓	2 →	2 →	2
D 0	1	2	2	3

Making Trackback table:

	A	B	C	D
0	0	0	0	0
A 0	(1) →	1 →	1 →	1
B 0	1 ↓	(2) →	(2) →	2
D 0	1	2	2	(3)

ABD

Result: The longest string is 'ABD', which length is 3.

Longest Common Subsequence:

```
LCS_Length(X, Y)
1.  m = X.length
2.  n = Y.length
3.  let b[1...m, 1...n] and c[0...m, 0...n] be new table
4.  for i = 1 to m
5.      c[i,0] = 0
6.  for j = 0 to n
7.      c[0,j] = 0
8.  for i = 1 to m
9.      for j = 1 to n
10.         if x_i == y_i
11.             c[i,j] = c[i-1, j-1] + 1
12.             b[i,j] = "↖"
13.         elseif c[i-1, j] ≥ c[i, j-1]
14.             c[i,j] = c[i-1, j]
15.             b[i,j] = "↑"
16.         else c[i,j] = c[i, j-1]
17.             b[i,j] = "←"
18. Return c and b
```

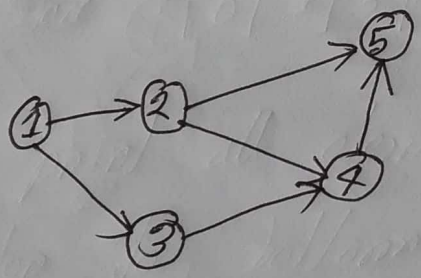
print_LCS (b, X, i, j)

1. if i == 0 or j == 0
2. return
3. if b[i, j] == "↖"
4. print_LCS (b, X, i-1, j-1)
5. print x_i
6. elseif b[i, j] == "↑"
7. print_LCS (b, X, i-1, j)
8. else print_LCS (b, X, i, j-1).

Set – 4+5

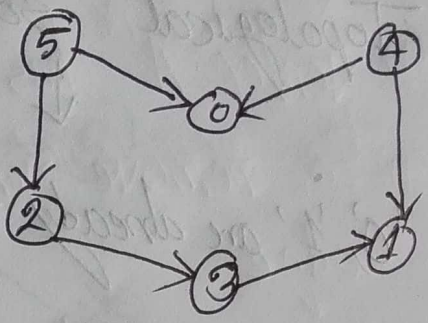
1.1 Topological Sorting:

The topological sort algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to. The ordering of the nodes in the array is called a topological ordering.



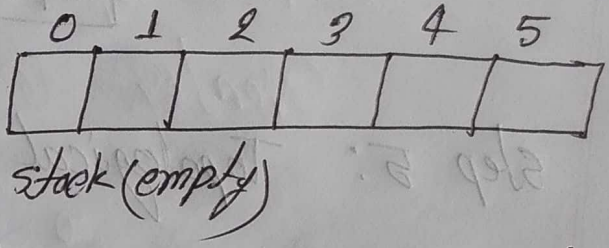
Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

How it is work:



Adjacent list (G)

0 →
1 →
2 → 3
3 → 1
4 → 0,1
5 → 2,0



stack (empty)

Step 1: Topological Sort (0), visited [0] = true

↓

List is empty. No more recursion call

Stack | 0 |

Step 2: Topological Sort (1), visited [1] = true

↓

List is empty. No more recursion call

Stack | 0 | 1 |

step 3: Topological Sort (2), visited [2] = true

↓

Topological sort (3), visited [3] = true

↓

'1' is already visited. No more recursion call

stack | 0 | 1 | 3 | 2 |

Step 4: Topological sort(4), visited[4] = true

↓

'0', '1' are already visited, No more recursion call

Stack

0	1	3	2	4	

Step 5: Topological Sort(5), visited [5] = true

↓

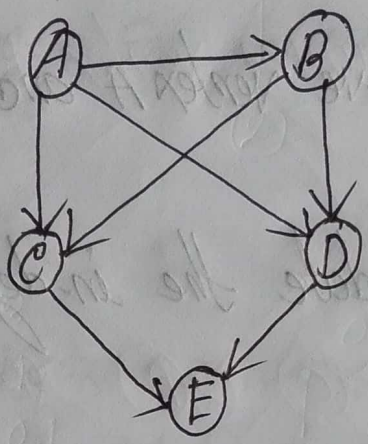
'0' '2' are already visited, No more recursion call

Stack

0	1	3	2	4	5

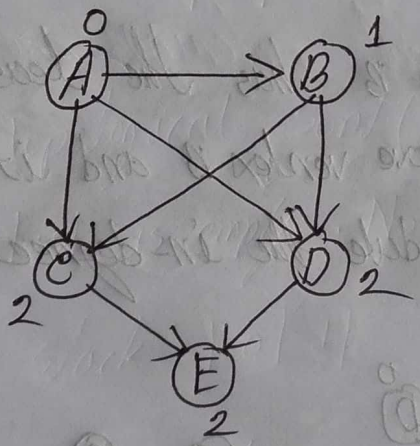
Step 6: print all elements of stack from top to bottom.

2] problem 4:



The topological ordering of the above graph are found in the following steps.—

Step 1: Write in-degree of each vertex.

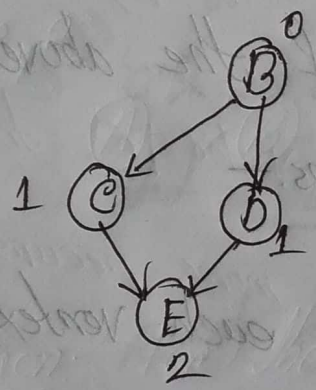


(5)

step 2: (i) Vertex-A has the least in-degree

(ii) So, remove vertex A and it's associated edge

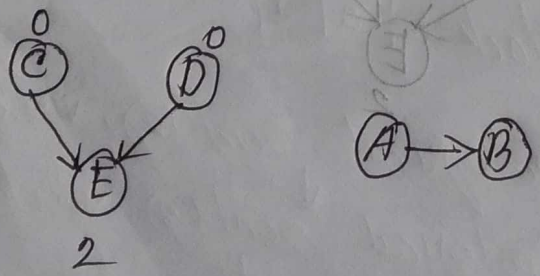
(iii) Now update the in-degree of other vertex



step 3: (i) Vertex B has the least in-degree

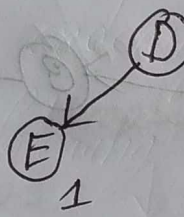
(ii) So, remove vertex B and it's edge

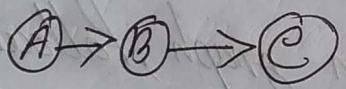
(iii) Now, update the in-degree



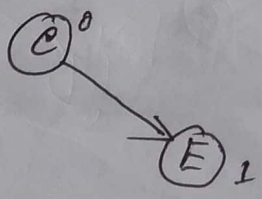
Step 4: There are two vertixs with the least in-degree. So, following two case are possible.

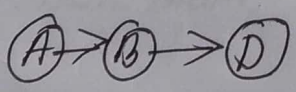
in-case 1: Remove vertex e and it's associated edge. Then update the in-degree.





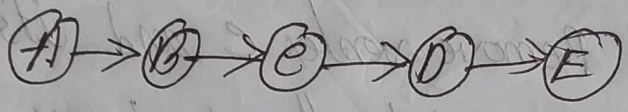
in case 2: Remove vertex D and it's associated edge. Then update the in-degree.



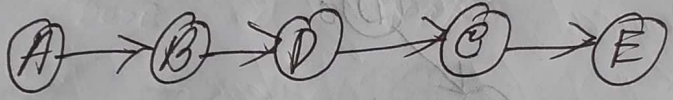


step 5: Now there is 1 vertex E, remove the remaining vertex E.

Case 1:



Case 2:



Conclusion: For the given graph, following 2 different topological orderings are possible.

① ABCDE
② ABDCE

3.1 Minimum Spanning Tree —

Prims Algorithm: Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph.

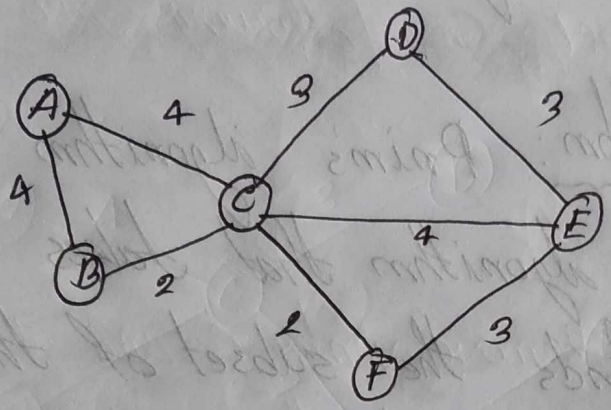
The steps for implementing Prim's algorithm are as follows:

① Initialize the minimum spanning tree with a vertex chosen at random.

② Find all the edges that connect the tree to new vertics, find the minimum and add it to the tree.

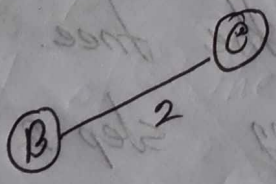
③ Keep repeating step 2 until we get a minimum spanning tree.

Example:

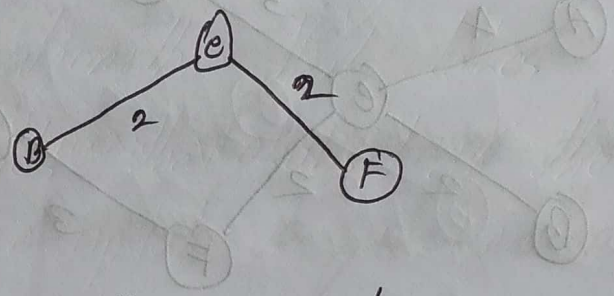


step 1: Chose a vertex

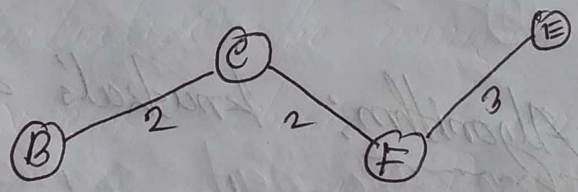
step 2: Choose the shortest edge from this vertex and add it.



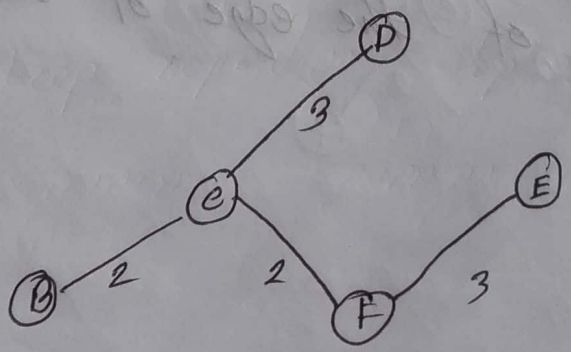
step 3: Choose the nearest vertex not yet in the solution.



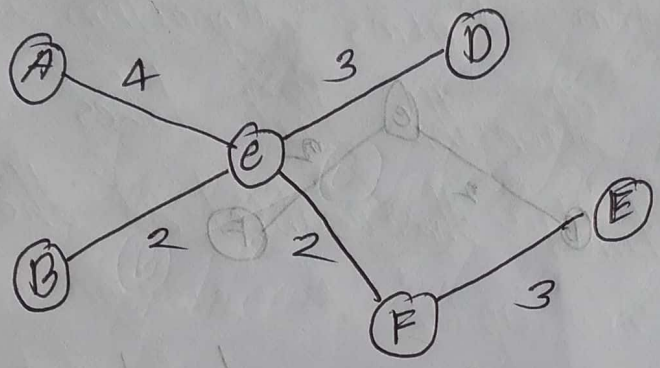
step 4: Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random.



step 5: Choose the nearest edge.



step 6: choose the nearest edge.



Thus graph is represent the minimum spanning tree.

Kruskal's Algorithm: Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and find the subset of the edge of that graph.

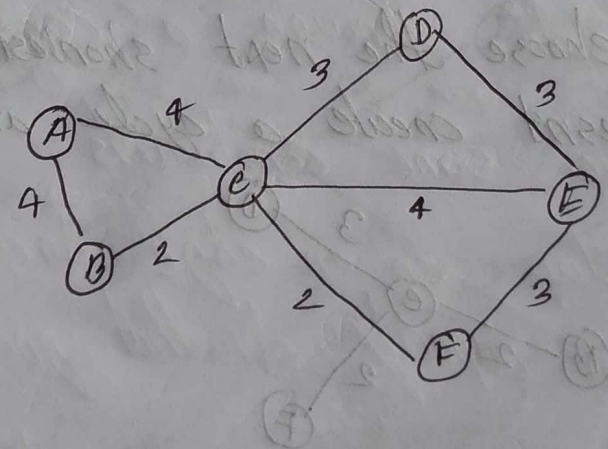
The steps for implementing Knuskal's algorithm are as follows:

① Sort all the edges from low to high

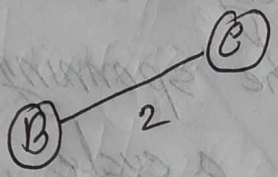
② Take the edge with the lowest weight and add it to the spanning tree. if adding the edge created a cycle, then reject this edge.

③ keep adding edges untill we reach all vertices.

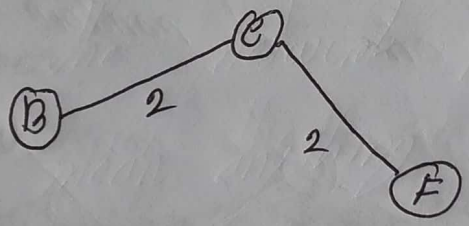
Example:



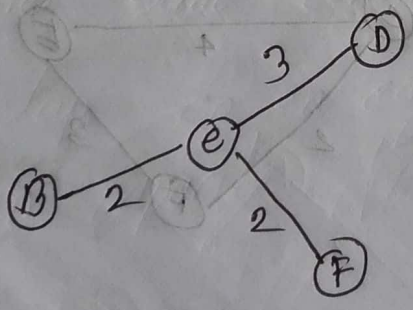
Step 1: Choose the edge with least weight if there are more than 1 choose any one.



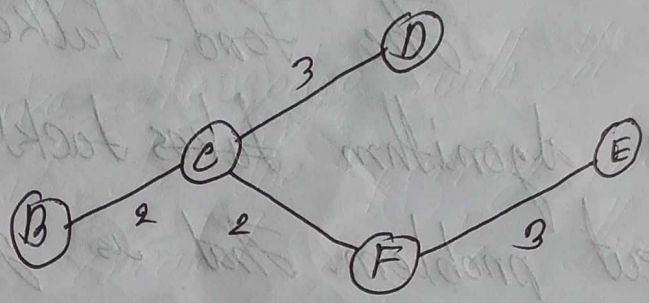
Step 2: Choose the next shortest edge and add it.



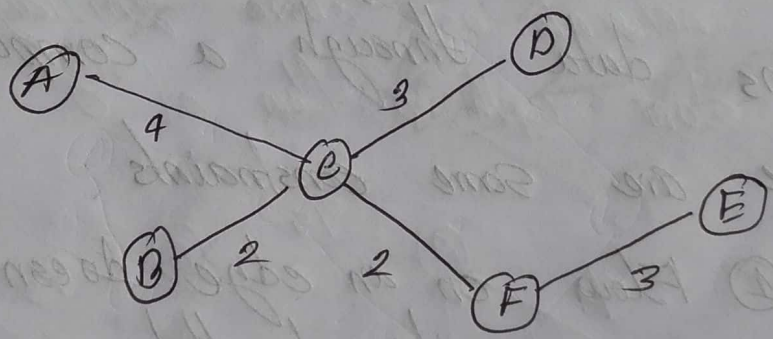
Step 3: Choose the next shortest edge that doesn't create a cycle and add it.



step 4: Choose the next shortest edge that doesn't create a cycle and add it.



step 5: Choose the next shortest edge that doesn't create a cycle and add it.



Thus graph is represent as a minimum spanning tree.

4.1

Ford fulkerson Algorithm:

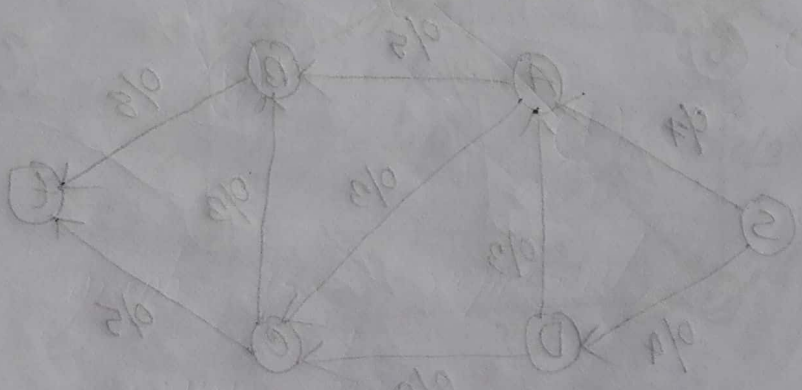
The ford-fulkerson algorithm is an algorithm ~~takes~~ tackles the max-flow min cut problem. That is, given a network with vertices and edges between those vertices that have certain weights, how much flow can the network process at a time? Flow can mean anything, but typically it means data through a computer network. There are some constraints:

① Flow on an edge doesn't exceeded the given capacity of that graph.

② Incoming flow and outgoing flow will also equal for every edge, except the source and the sink.

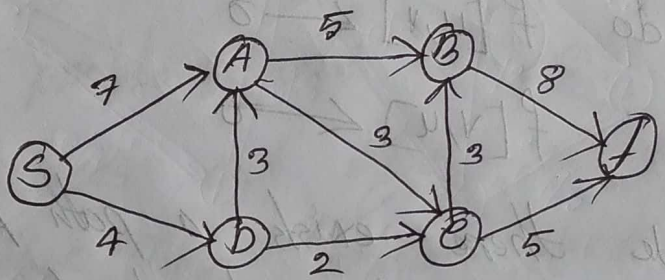
Ford Fulkerson Algorithm:-

```
Ford_Fulkerson (G,s,t)
for each edge (u,v) ∈ E[G]
    do f[u,v] ← 0
       f[v,u] ← 0
while there exists a path p from s to t
in the residual network G_F
    do c_f(p) ← min {c_f(u,v) : (u,v) is in p}
       for each edge (u,v) in p
           do f[u,v] ← f[u,v] + c_f(p)
```

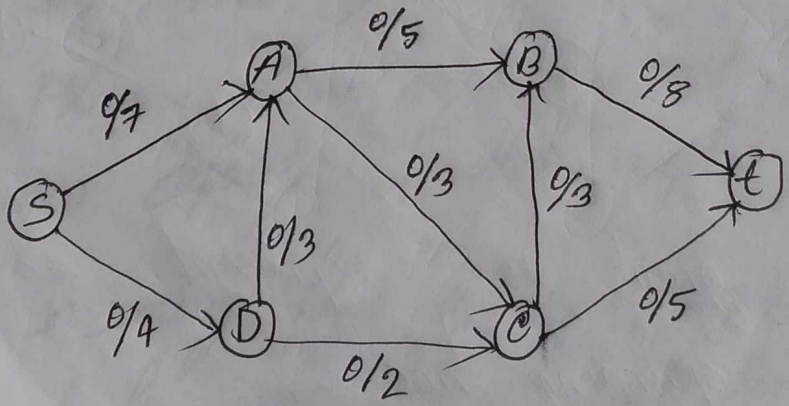


5.1 Find the maximum flow with ford fulkerson's algorithm-

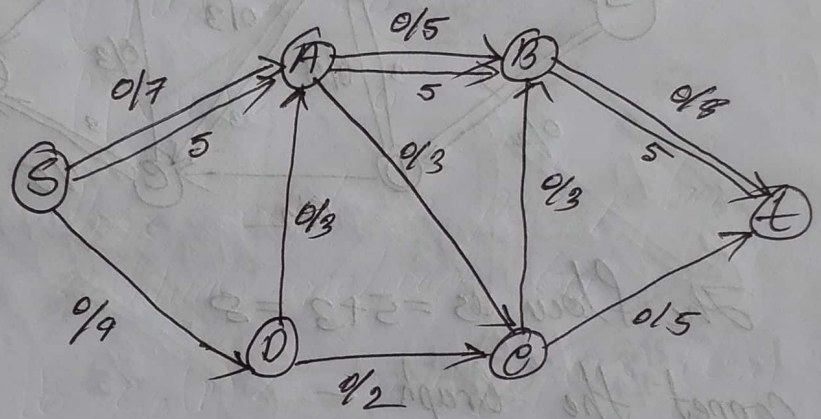
The Given graph-



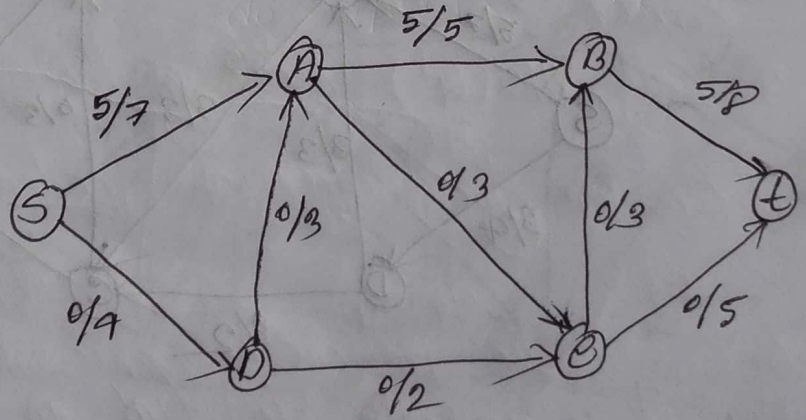
step 1: we use the same flow network as above. Initialy we start with a flow of 0.



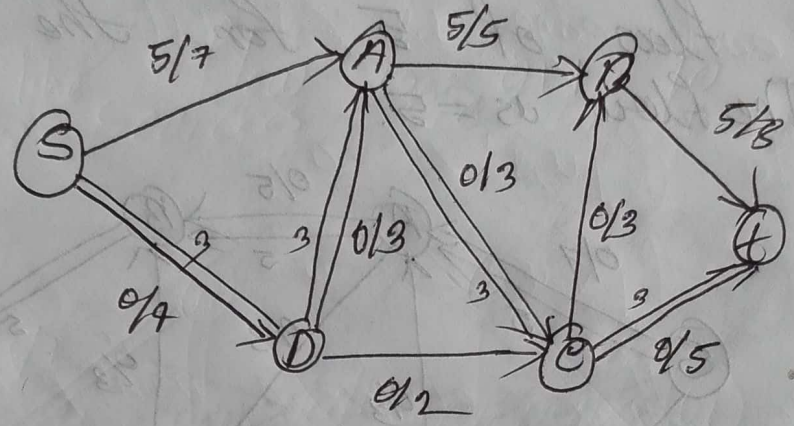
Step 2: We can find the path S→A→B→t with the residual capacities 7, 5, and 8. Their minimum is 5. So this gives a flow of 5 for the network. The flow is = 5.



Step 3: correct the graph –

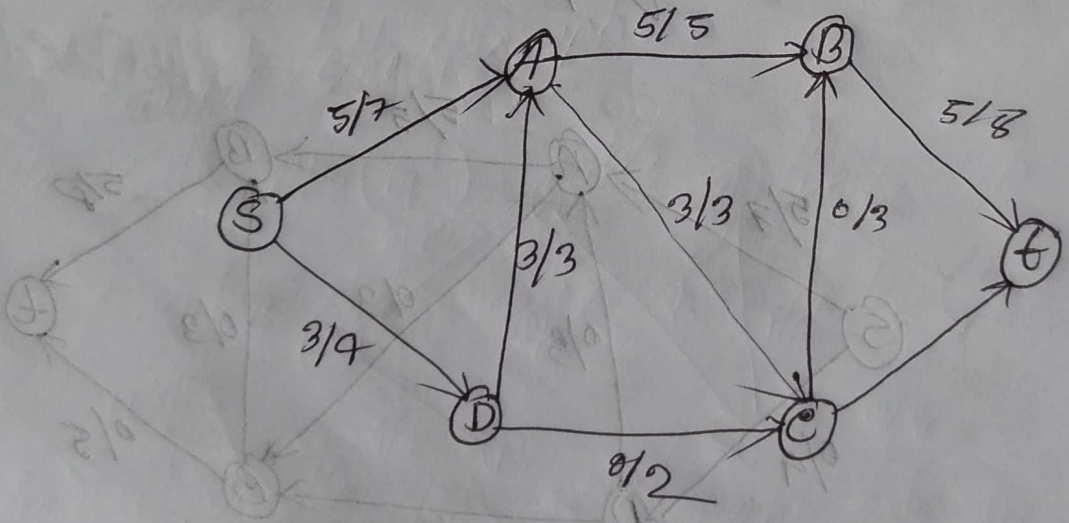


step 3: Again we look for an augmenting path, this time we find S→D→A→C and the minimum flow is 3.



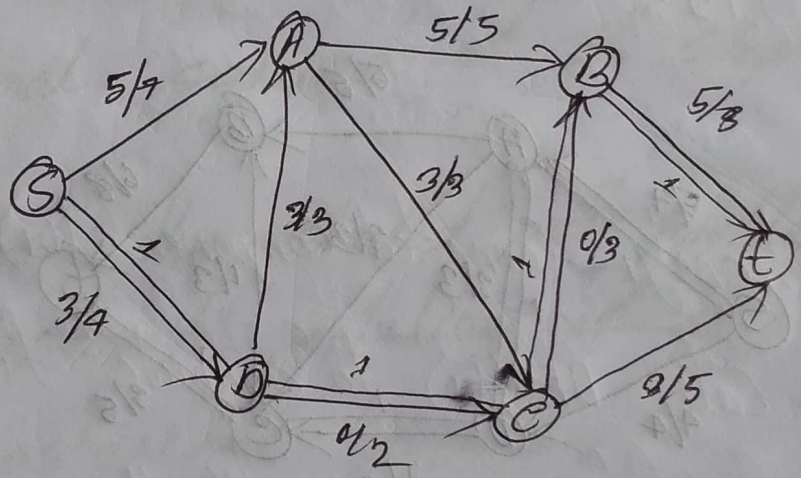
The flow is = 5+3 = 8

correct the graph -

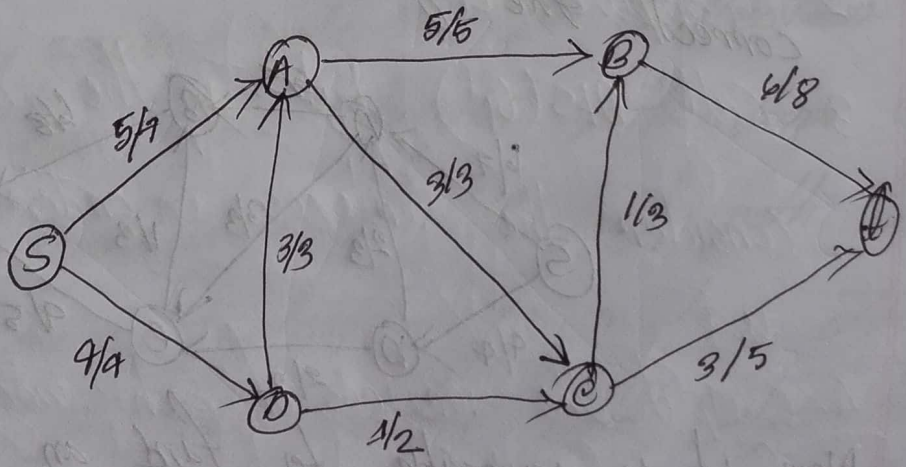


Step-4: This time we took the path S→D→C→E with minimum flow 1.

The flow is = 8+1 = 9

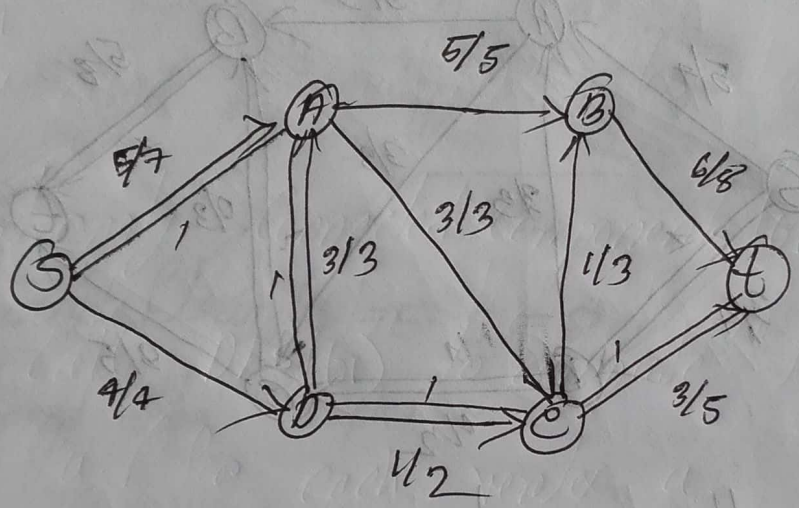


corrected the graph -

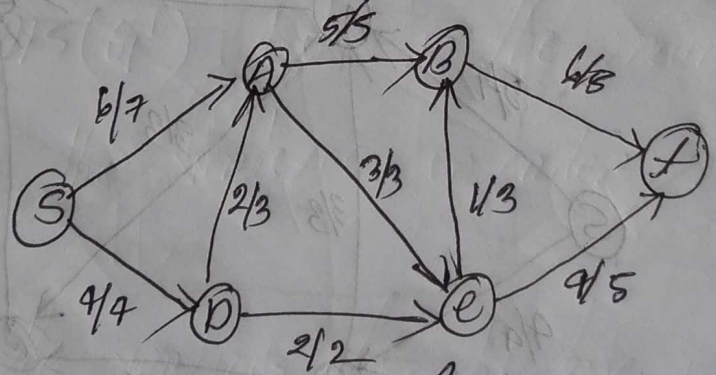


Step 5: This time we find the augmenting path $S \to A \to D \to e \to t$ and the minimum flow is 1.

The flow is $= 9+1 = 10$



correct the graph.



Now it is impossible to find an augmenting path between S and t, therefore this flow of 10 is the maximal possible.

6) Optimal Substructure property:

A given problems has optimal Substructure property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For, example, the shortest path problem has following optimal ~~solution of its subprob~~ lems: Substructure property -

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest from x to v. The standard all pair shortest path algorithms like Floyd-Warshall and Bellman-Ford are typical example of optimal substructure.

7.1 The following linear-time $(V+E)$ algorithm compute the strongly connected components of a directed graph $G=(V,E)$ using two dept-first-searches, one on G and one on G^T.

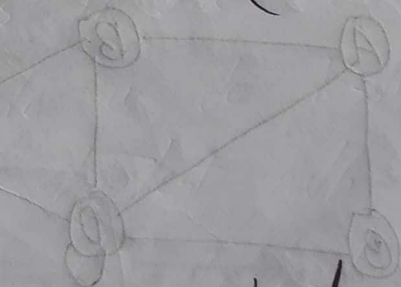
Strongly-Connected-Components (G)

1. Call DFS(G) to compute finishing times $u.f$ for each vertex u.
2. Compute G^T
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$.
4. Output the vertices of each tree in the depth first forest formed in line 3 as a separate strongly connected component.

8.1 The Floyd-Warshall algorithm:

Floyd_Warshall (W)

1. $n = W.rows$
2. $D^{(0)} = W$
3. for $k = 1$ to n
4. let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
5. for $i = 1$ to n
6. for $j = 1$ to n
7. $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
8. Return $D^{(n)}$

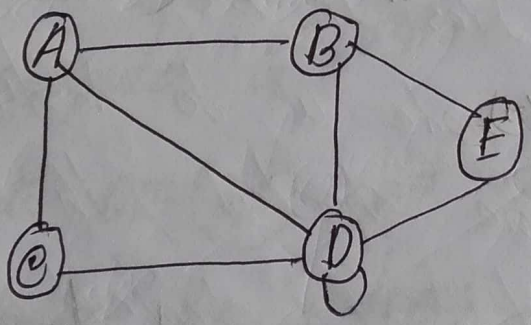


Here we have to discuss about the procedure of Floyd warshall algorithm.

10.1 Adjacency Matrix :-

Adjacency matrix is a sequential Representation. It is used to represent which nodes are adjacent to each other. is there any edge connecting nodes to a graph.

Example: Consider the following graph representation.

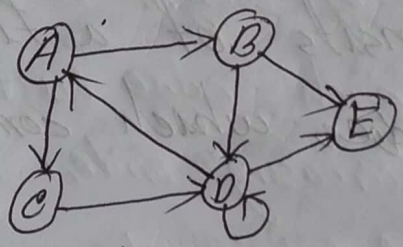


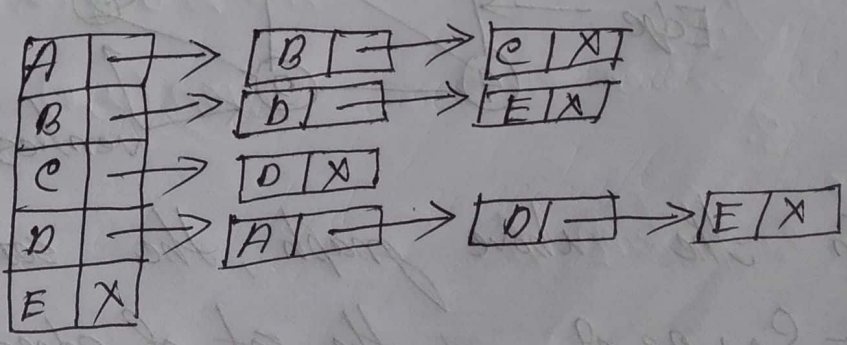
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Adjacency list:-

Adjacency list is a linked representation. In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.

Example:

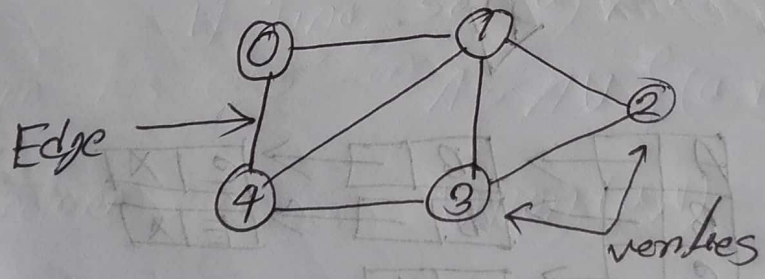




12.] Graph:-

A graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a graph can be defined as,

A Graph consits of a finite set of vertices and set of Edges which connect a pair of nodes.



In the above graph, the set of vertices $V = \{0,1,2,3,4\}$ and the set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$

13.1 Dijkstra's Algorithm:-

```
Dijkstra (G, w, s)
1. Initialize_Single_Source (G, s)
2. S = ∅
3. Q = G.V
4. while Q ≠ ∅
5.         u = Extract_Min(Q)
6.         S = S ∪ {u}
7.         for each vertex v ∈ G.adj[u]
8.                 Relax (u, v, w).
```

3. Kruskal's Algorithm:

MST_kruskal (G, w)

1. $A = \emptyset$
2. for each vertex $v \in G.V$
3. Make_Set (v)
4. sort the edges of G.E into nondecreasing order by weight w
5. for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
6. if find_set (u) ≠ Find_set (v)
7. $A = A \cup \{(u, v)\}$
8. Union (u, v)
9. return A.

prims Algorithm :-

MST_Prim (G, w, r)

1. for each $u \in G.V$
2. $u.key = \infty$
3. $u.\pi = NiL$
4. $r.key = 0$
5. $Q = G.V$
6. while $Q \neq \emptyset$
7. $u = Extract\_Min(Q)$
8. for each $v \in Q$ and $w(u,v) < v.key$
10. $v.\pi = u$
11. $v.key = w(u,v)$

Topological Sort Algorithm :-

L ← Empty list that will contain the sorted elements

S ← Set of all nodes with no incoming edge

While S is non-empty do

remove a node n from S

add n to tail of L

For each node m with an edge e from n to m do

remove edge e from the graph

if m has no other incoming edges then

insert m into S

if graph has edge then

return error.

else

return L.

Set - 7

1.1 Greedy Algorithm:

An algorithm is designed to achive optimum solution for a given problem. In greedy algorithm approach, desisions are made from the given solution problem / domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Approches of greedy algorithm: Most Networking algorithm is use the greedy approch. Here

is a list of few of them —

(i) Travelling Salesman Problem
(ii) Prim's and Kruskal's Algorithm.
(iii) Dijkstra Minimal spanning tree Algorithm.
(iv) Graph - Map coloring
(v) Graph - Vertex Cover
(vi) Knapsack Problem
(vii) Job scheduling problem

There are lots of similar problems that uses the greedy approach to find an optimum solution.

2.1 Huffman Algorithm use:

Huffman coding is an efficient method of compressing data without losing information. Huffman coding provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message. Greedy algorithms are often easier to describe and code up than other algorithms. Greedy algorithms can often be implemented more efficiently than other algorithms.

3.1 Greedy Algorithms work:

When we have a better understanding of greedy algorithm mechanism, let's explore ~~How~~ the core components of a greedy algorithm that sets it apart from other process :—

Candidate set: An Answer is created from this set.

selection function: It selects the best candidate to be included in the solution.

Feasibility function: This section calculates if a candidate can be used to contribute to the solution.

An objective function: It assigns a value to a complete or a partial solution.

A solution function: This is used to indicate if a proper solution has been met.

4.1 Activity Selection:

Given Data,

i	1	2	3	4	5	6	7	8	9	10
s_i	1	2	0	5	3	5	6	8	8	12
f_i	4	5	3	7	5	9	10	11	12	16

Sorted activity:

i.	3	1	2	5	4	6	7	8	9	10
s_i	0	1	2	3	5	5	6	8	8	12
f_i	3	4	5	5	7	9	10	11	12	16

Selected activity:

i.	3	5	4	8	10
s_i	0	3	5	8	12
f_i	3	5	7	11	16

Algorithm:

1. Activity-Selector(A, s, f) :=
2. Sort A by finished time.
2. $S = \{A[1]\}$
4. $k = 1$
5. $n = A.length$
6. for $i = 2$ to n
7. if $s[i] \geq f[k]$
8. $S = S \cup \{A[i]\}$
9. $k = i$
10. Return S.

Set - 8

1.1 Backtracking:

The backtracking is an algorithmic technique to solve a problem by an incremental way. It uses recursive approch to solve the problems. We can say that the backtracking is used to find all possible.

uses —

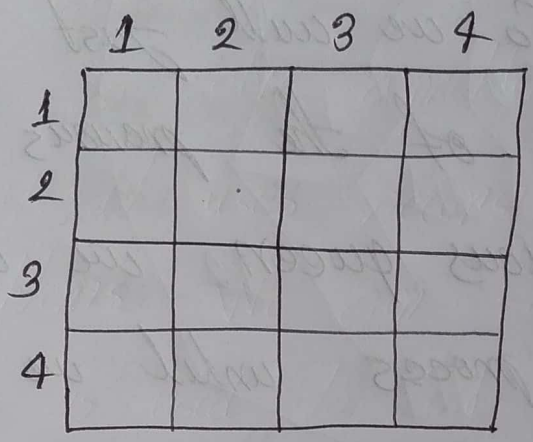
(i) Hamilton cycle
(ii) M-coloring problem
(iii) N-Queen problem.

2.1 4*4 queens:

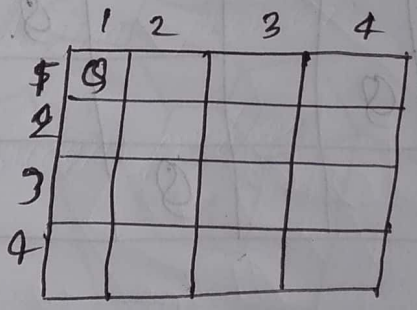
```
Algorithm nQueens(k,n):
{
    for i:=1 to n do
    {
        if place(k,i) then
        {
            x[k]=i;
            if (k==n)
                write (x[1:n]);
            else
                NQueens(k+1, n);
        }
    }
}
```

Given a 4*4 chessboard and number the rows and column of the chesboard 1 through 4

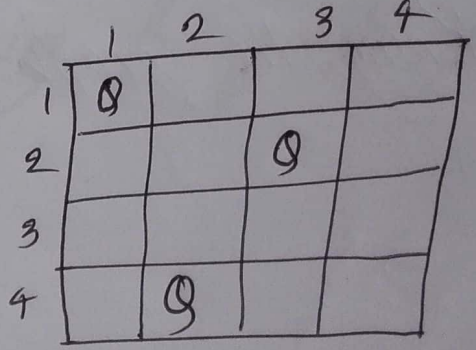
Step-1:



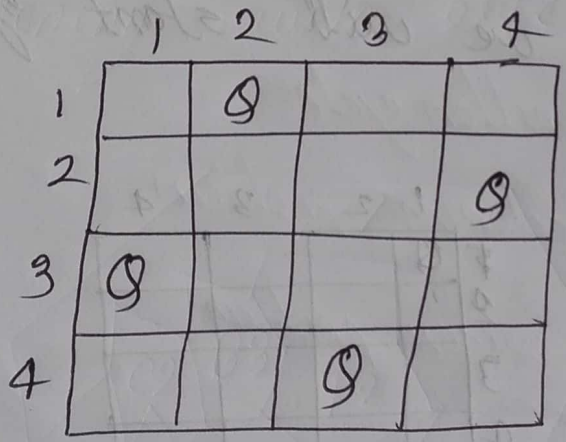
Step 2: Now we will start by placing the first queen



Step 3: place the second queen in a safe position then the third queen.



Step-4: Now we can see that there is no safe place where we can put the last queen. So we will just change the position of the previous position of the previous queen. We will continued this process until we get the solution.



3. Graph Coloring:

Let,

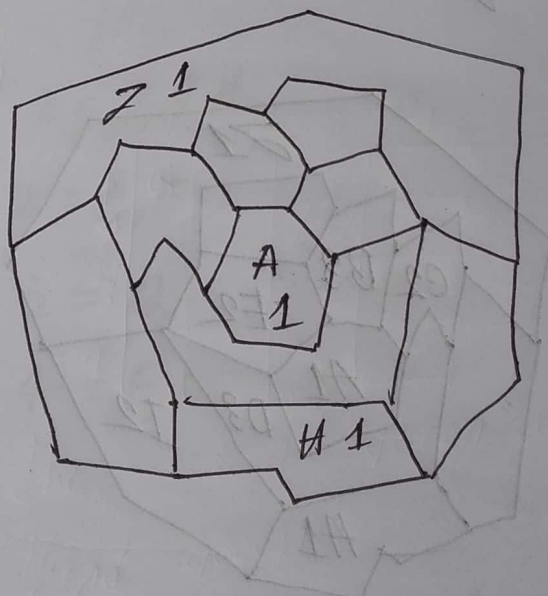
Green = 1

Red = 2

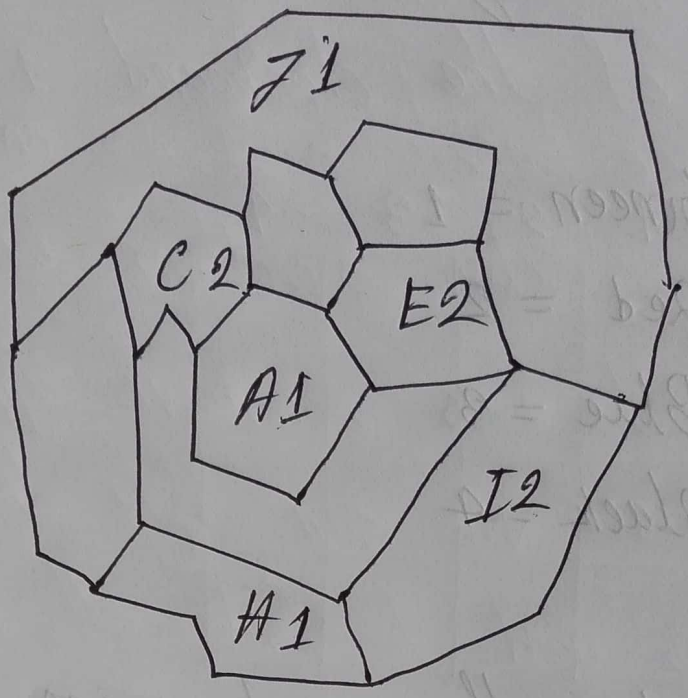
Blue = 3

Black = 4

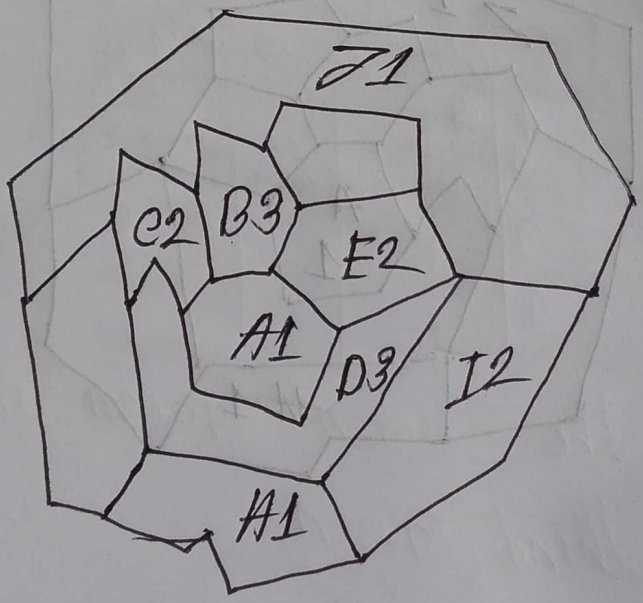
step 1: Coloring the graph using Green = 1 color.



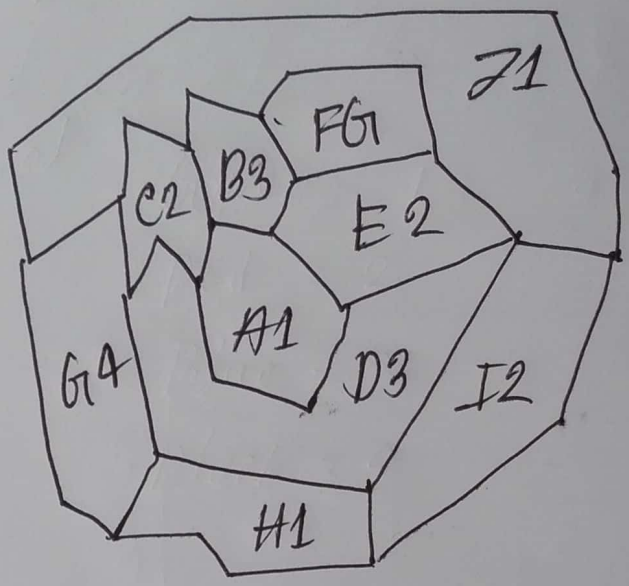
step 2: Coloring the graph using Red = 2 color



Step 3: Coloring with blue = 3 color.



Step 4: Coloring the graph with black = 4 color.



we need to minimum 4 color to fulfill the graph.

1 = Z, A, H

2 = C, E, I

3 = B, D

4 = F, G.