Noakhali Science & Technology University

Assignment Title: Algorithm Design and Analysis
Course Code: CSTE 2103

SUBMITTED BY:

Mohammad Borhan Uddin
Roll: ASH2101008M
Year:02 Term:01
Computer Science & Telecommunication Engineering

SUBMITTED TO:

A. R. M. Mahmudul Hasan Rana
Assistant Professor
Computer Science & Telecommunication Engineering
Noakhali Science & Technology University

Date: 17/08/2023

In this following graph, we start traversal from node A. The algorithm uses two queue namely queue1 and Queue2. Queue 1 holds all the nodes that are processed, while Queue 2 holds all the nodes that are processed and deleted from Queue 1.

Step 1: First add A to Queue1 and Null to Queue2.

Queue 1 = {A}
Queue 2 = {NULL}

Step 2: Pop 'A' from Queue 1 and push it into Queue2. pushing all adjacent nodes of 'A' lexicographically. Besides, marking 'A' and adjacents of A as visited node.

Queue1 = {D, G}
Queue2 = {A}

Visited:

A	B	C	D	E	F	G	H
1			1			1	

Step 3: Remove node 'D' from Queue1 and push it and it's adjacent in Queue1. Push 'D' into Queue 2. Marked as visited 'D' and it's adjacent nodes.

Queue 1 = {G, C, F}
Queue 2 = {A, D}

Visited:

A	B	C	D	E	F	G	H
1		1	1		1	1	

Step 4: Pop 'G' from Queue1 and push it into Queue2. Insert all non-visited adjacent of 'G' into Queue1 lexicographically. Moreover, marking all adjacent of G as visite nodes.

Queue 1 = {C, F, E, H}
Queue 2 = {A, D, G}

Visited

A	B	C	D	E	F	G	H
1		1	1	1	1	1	1

Step 5: Delete node 'C' and push it into Queue2. Insert all non-visited adjacent vertices of it into Queue1 and mark them visited.

Queue1 = {F, E, H}

Queue2 = {A, D, G, C}

Visited

A	B	C	D	E	F	G	H
1		1	1	1	1	1	1

Step 6: POP 'F' from Queue1 and push it into Queue2. Add all non-visited neighbours of 'F' into Queue1 and mark them as visited.

Queue1 = {E, H, B}

Queue2 = {A, D, G, C, F}

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

Step 7:

Queue1 = {H, B}

Queue2 = {A, D, G, C, F, E}

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

Step 8:

Queue1 = {B}

Queue2 = {A, D, G, C, F, E, H}

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

Step 9:

Queue1 = { }

Queue2 = {A, D, G, C, F, E, H, B}

Visited:

A	B	C	D	E	F	G	H
1	1	1	1	1	1	1	1

Hence the Queue1 is empty, the algorithm terminates here.

(I) False.

The statement contradicts itself. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n)$ cannot be $\Theta(g(n)^2)$ because that would mean it is both upper and lower bounded by $g(n)^2$, which contradicts the initial assumption.

(II) True.

$2^n + n^2 = O(2^n)$ because, for large n, the exponential term 2^n grows much faster than the quadratic and linear terms.

(III) False.

$2^n + n^2$ grows much slower than 3^n. So, $2^n + n^2 = O(3^n)$ is not true.

(IV) True.

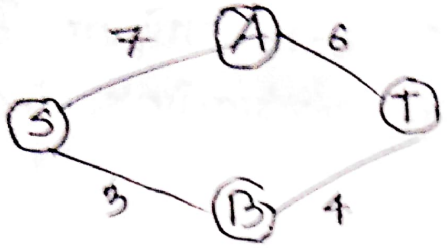
Topological sort is simply DFS with an extra stack. This algorithm scans edges and nodes once to order them. So, time complexity is the same as DFS which is $O(|V| + |E|)$ and it is a linear search time algorithm.

(V) False.

Krushkal's algorithm is not an example of a divide ~~and~~ and conquer algorithm, it's a greedy algorithm.

(VI) False.

Even if no two edges have same weight, there could be two path of same weight



Here, • A to B has two shortest path with equal cost.

$A \rightarrow T \rightarrow B$

$A \rightarrow S \rightarrow B$

(VII) False.

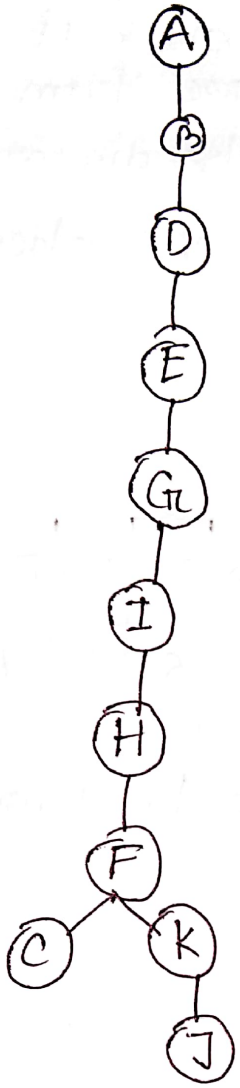
An arbitrary graph with $|E| = |V|$ edges does not necessarily have to be a tree. A tree has $|E| = |V| - 1$ edges.

(VIII) True.

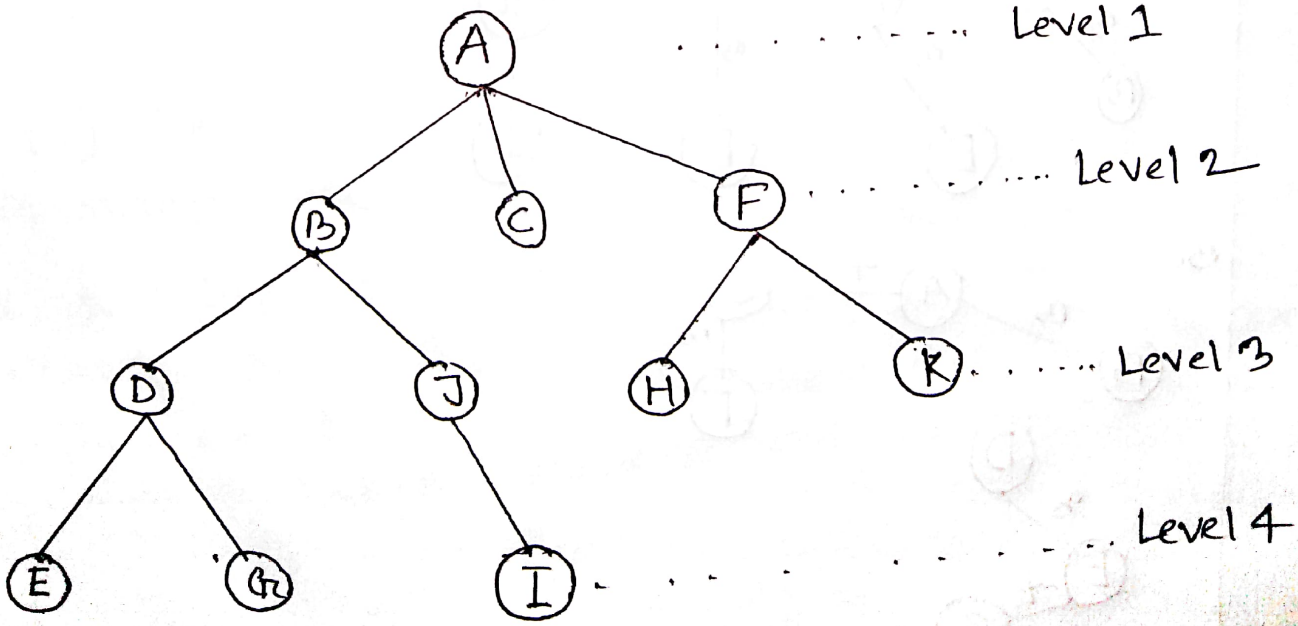
If every node can be reached from vertex 'v' and every node can reach 'v'. then the graph is strongly connected. A DFS will provides the nodes reachable from the vertex 'u'. Since the statement indicates it can be any vertex, then this will hold for all vertices, which means every vertex reachable from every other vertex.

3

(I) Provide the DFS tree starting at node A:



(II) Provide the BFS tree starting at node 'A':



(iii) Use Kruskal's algorithm to derive the MST:

For Kruskal's algorithm:

1. Sort all the edges in increasing order of the weight.
2. Pick the smallest edge and check if it forms a cycle with the spanning tree form. If cycle is not formed include the edge, else discard it.
3. Repeat step 2 until there (V-1) edges in the spanning tree.

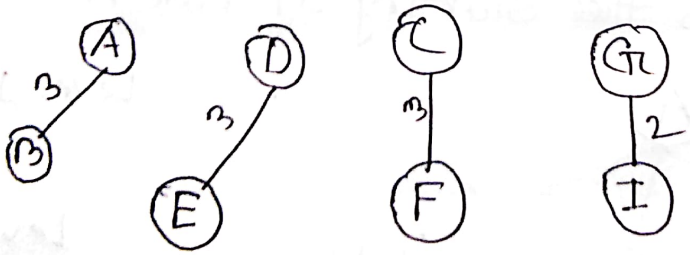
1. After sorting:

G,I	A,B	C,F	D,E	A,C	E,G	A,F	D,G	B,D	D,H	I,J	F,K	H,I	B,J	G,J	F,H	J,K
2	3	3	3	4	4	5	5	6	6	6	7	7	8	8	8	8

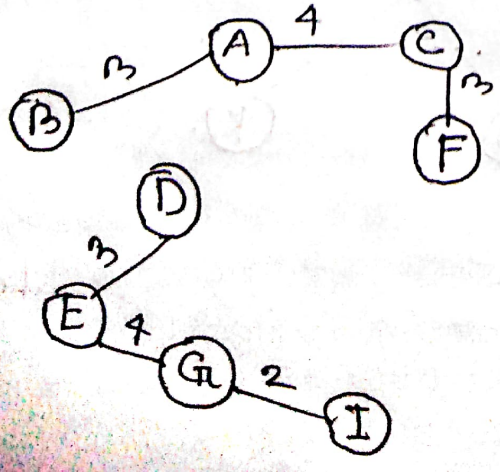
Now pick all edges one by one to form the sorted list of edges.

1.

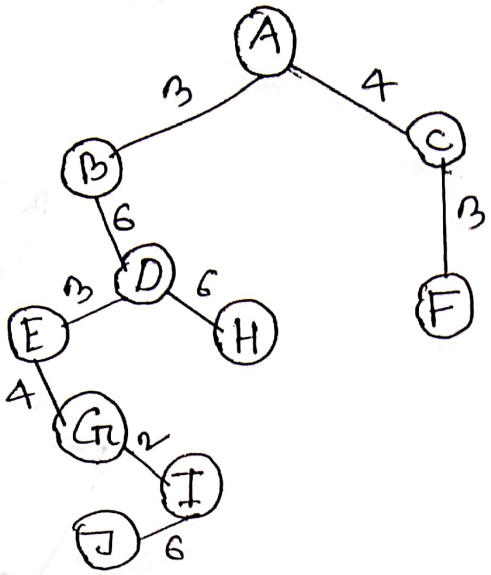
2.



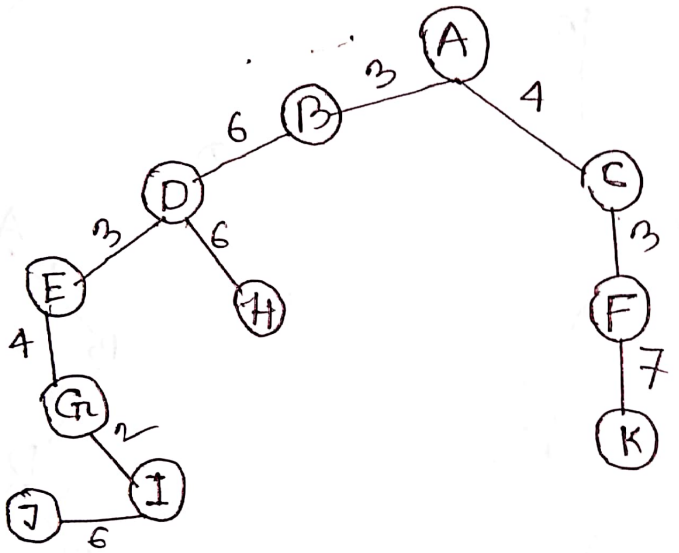
3.



4.



5.



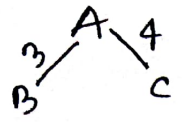
AF, DG, H,I, BJ, GJ, FH, JK form. So, we cannot add them.

(IV) Use prim's algorithm to derive the MST from starting node 'A':

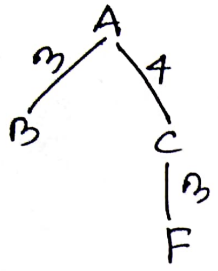
1. Initialize the minimum spanning tree with a vertex chosen at random. Here choose 'A'.
2. Find all the edges that connect the tree to new vertices. Find the minimum and add it to the tree.

1. A —3— B

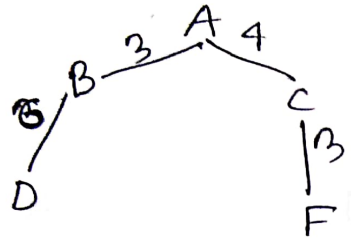
2.


A
3
4
B
C

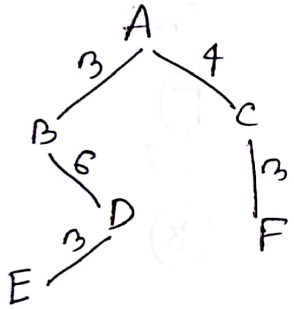
3.


A
3
4
B
C
3
F

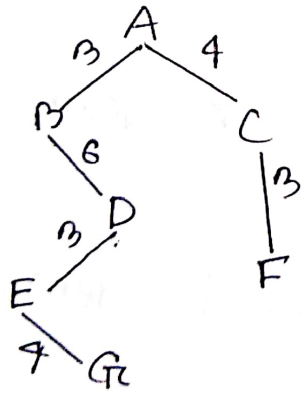
4.


A
3
4
B
C
3
D
F

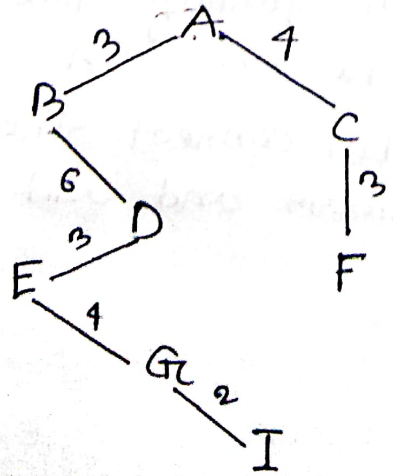
5.


A
3
4
B
C
6
3
D
F
3
E

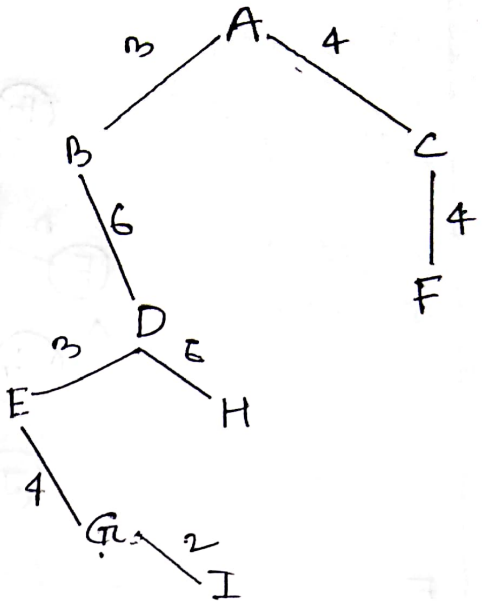
6.


A
3
4
B
C
6
3
D
F
3
E
4
Gr

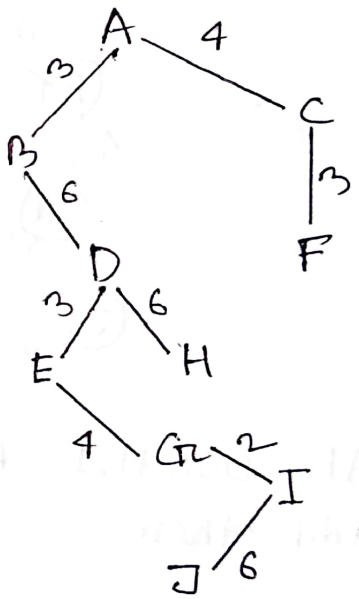
7.


A
3
4
B
C
6
3
D
F
3
E
4
Gr
2
I

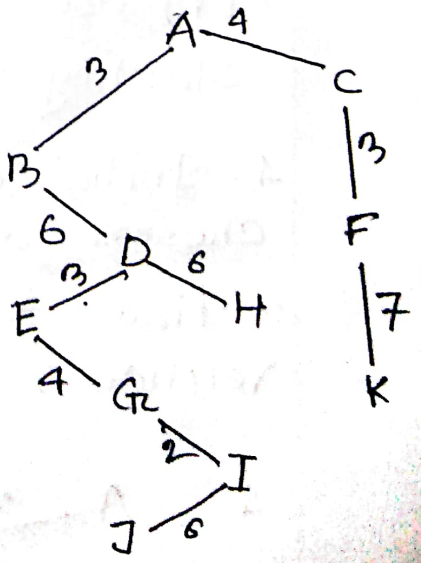
8.


A
3
4
B
C
6
4
F
D
3
6
E
H
4
Gr
2
I

9.


A
4
3
B
C
6
3
F
D
3
6
E
H
4
Gr
2
I
J
6

10.


A
4
3
B
C
3
F
6
D
6
3
E
H
7
4
Gr
K
2
I
J
6

4

Marked Vertex	A	B	C	D	E	F	G	H	S	T
S	∞	∞	∞	∞	∞	∞	∞	∞	0	∞
B	24	[18]	61	∞	∞	∞	∞	∞		∞
A	[24]		60	∞	30	∞	∞	∞		∞
E			58	66	[30]	∞	∞	∞		∞
F			58	66		[41]	∞	77		∞
C			[58]	64			79	76		∞
D				[63]			79	76		∞
G							[75]	76		∞
H								[76]		80
T										[87]

Shortest Route : SBEFHT

Time : 87 minutes.

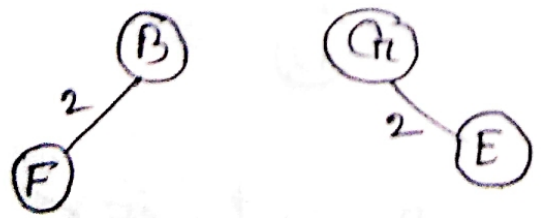
(a) Applying Kruskal's algorithm:

After sorting all edges in their increasing order of weight :-

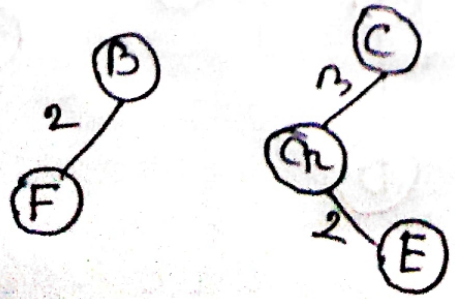
Weight	Edge
2	BF
2	EG
3	CG
4	CE
5	AD
7	AF
13	DF
14	AB
15	BC
15	EF
18	FG
19	BG
21	AC
22	DE

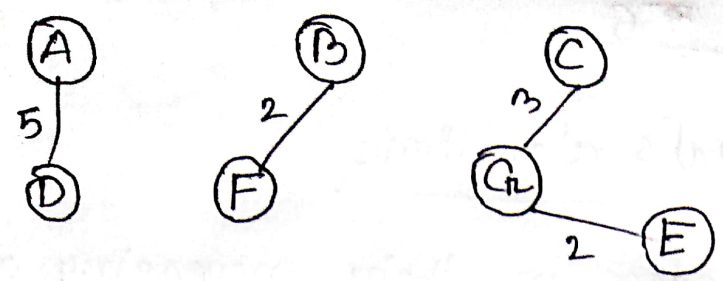
Adding the edge which has the last weight.

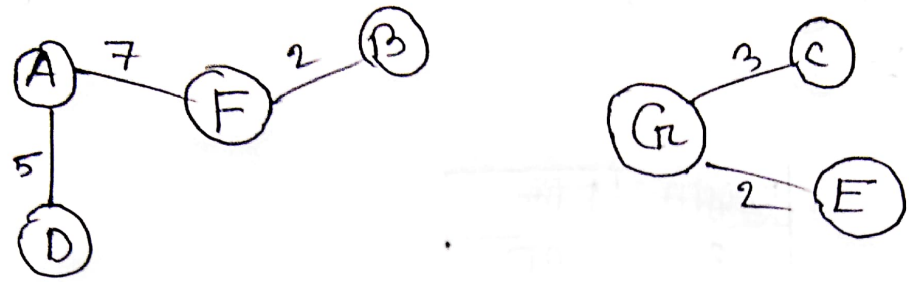
1.

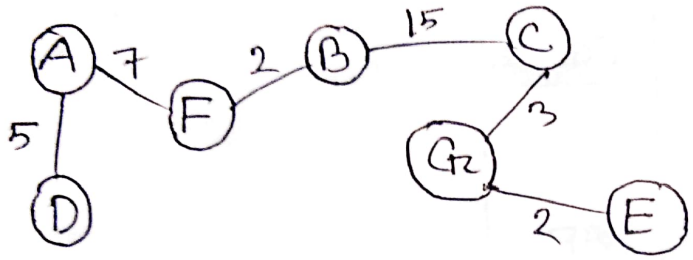


2.



3. 

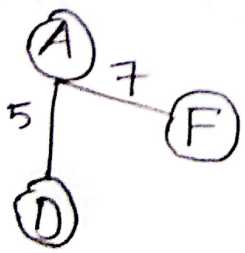
4. 

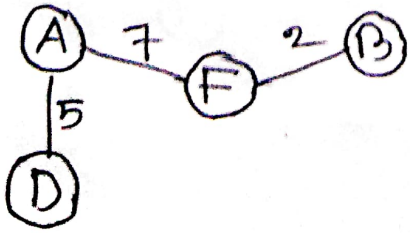
5. 

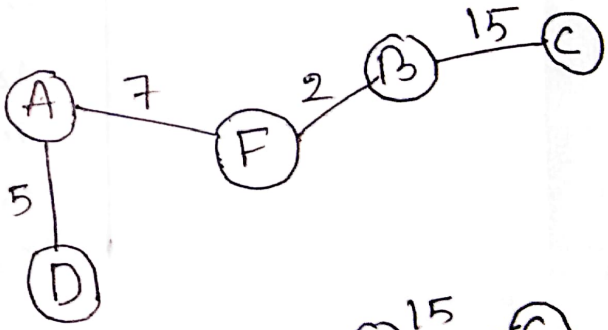
Applying prim's algorithm:

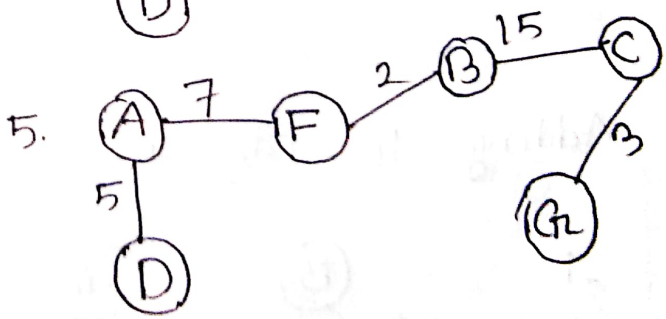
Let starting node is A.

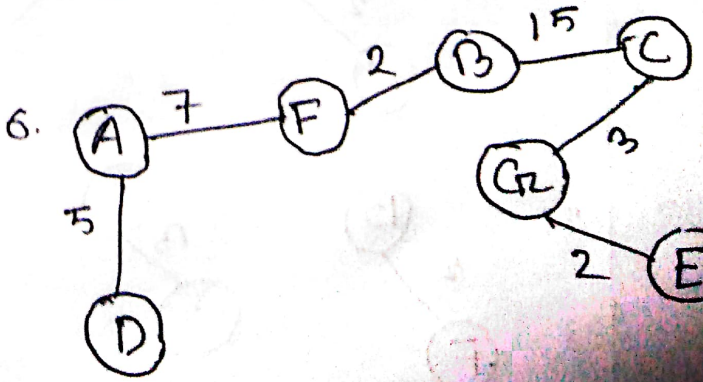
1. 

2. 

3. 

4. 

5. 

6. 

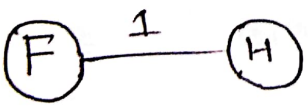
b) Applying Kruskal's algorithm:

After applying sorting all edge in their increasing order of weight:

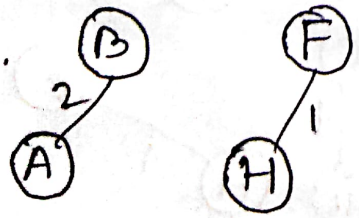
Edge	Weight
FH	1
AB	2
DE	3
EH	4
CG	4
GF	4
AC	5
CD	6
BD	8
CF	9
DH	11
GH	12
AF	20
BF	22

EH = 5 is removed because it's a parallel edge and we take the minimum weight which is 4.

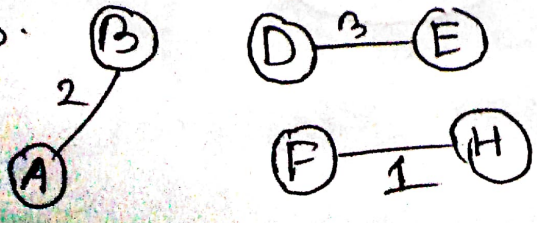
1.



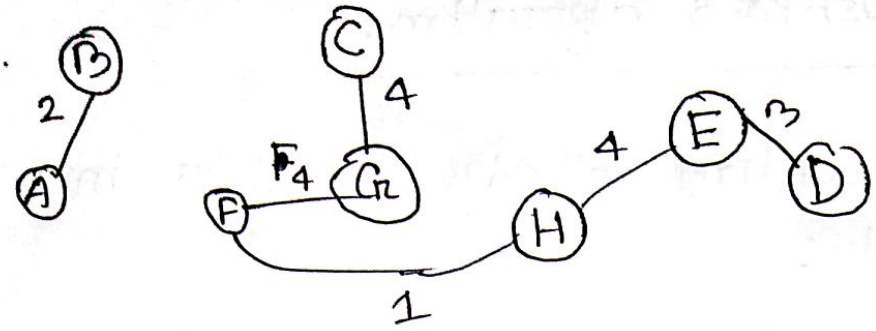
2.



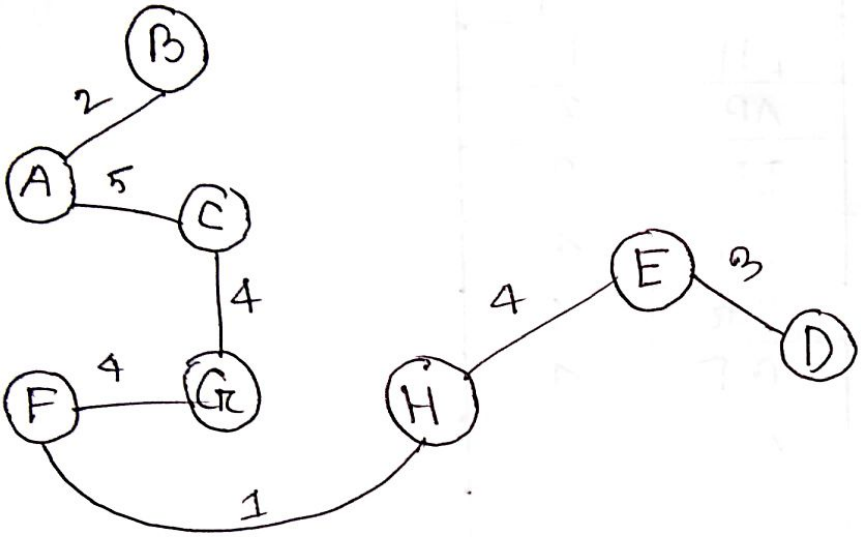
3.



4.

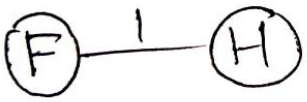


5.

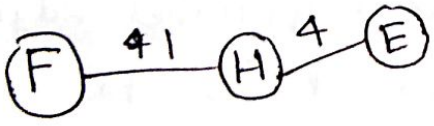


By applying Prim's algorithm:

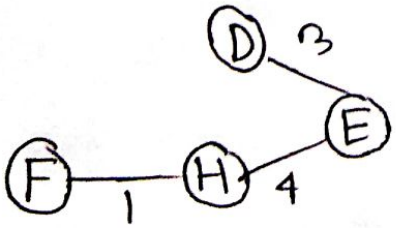
1. Let starting node is F.



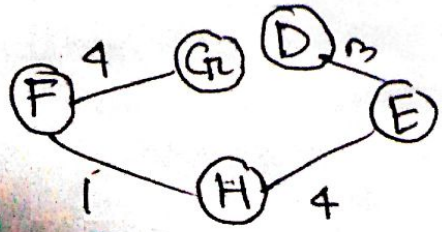
2.



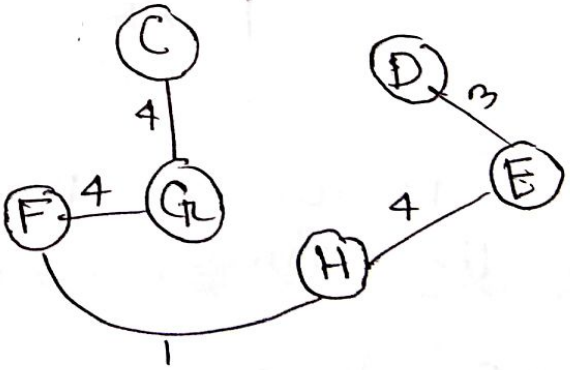
3.



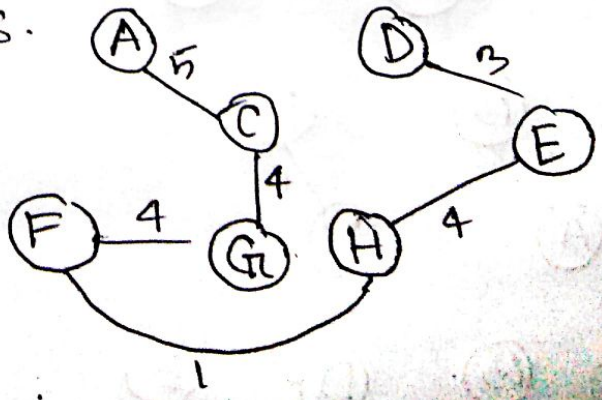
4.



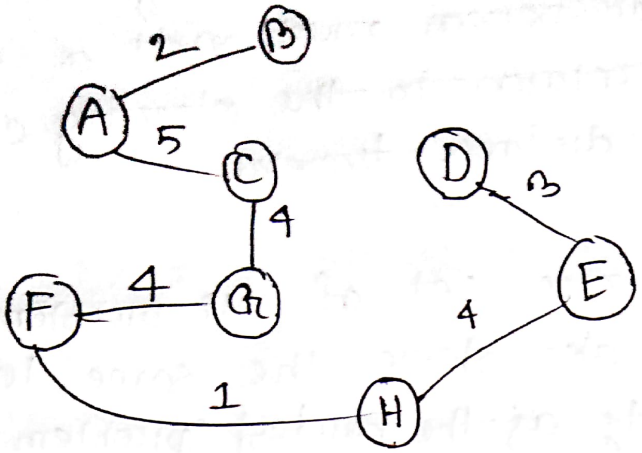
5.



6.



7.



7

(I)

NP hard: A problem 'x' is NP hard if there is an NP complete problem 'Y' such that Y is reducible to x in polynomial time. An example of NP hard problem is the descision subset sum problem.

NP complete: NP complete are those problems which can be solved by a non-deterministic algorithm or Turing machine in polynomial time. To solve this problem, it must be both NP and NP hard problem.

Discussion about NP, NP-hard and NP complete problems:

NP problem are those for which a potential solution can be verified in polynomial time. Such as subset sum problem. Given a ~~sub~~set of integers and a ~~target~~ sum, the question is wheather, the question is wheather there's a subset of the integers that adds up to the ~~target~~ sum.

NP-hard problems are at least as hard as the hardest

problems in NP. For instance, Traveling [illegible] (TSP). In TSP, a salesperson must visit a set of cities exactly once and return to the starting city while minimizing the total distance traveled.

NP complete problems are set of NP problems that are not only in NP but also share the same level of computational complexity as the hardest problem in NP. For example, Boleon Satisfiability Problem (SAT). Given a boleon formula, the question is whether there's an assignment of truth values to variables that make the formula true.

(II) The P versus NP problem is a major unsolved problem in computer science. If a problem is solved in polynomial time, then it is in P, where time require $T(n) = O(c * n^k)$

If a problem verified in polynomial time but solved in exponential time too, then it is in NP. P=NP this term sometimes said to be true or sometimes said to be false

(III) The statement is not valid.

The shortest-path problem is in the class P, because algorithm like Dijkstra's or Bellman-Ford can be solve it in polynomial time. However, it is not NP complete because it can be solved efficiently, which contradicts the inherent difficulty associated with NP complete problems.

(IV)

a) True.

The P in the P class for polynomial time. It is the collection of decision problems that can be solved by a deterministic machine in polynomial time. We can either use BFS or DFS to check whether there is a cycle in undirected graph. The time complexity is O(V+E) which is polynomial.

b) The problem of determining whether there exists a cycle in an undirected graph is in NP.

True.

NP is set of decision problems that can be solved by a non-deterministic polynomial time. P is a subset of NP. The problem of determing whether there exists a cycle in an undirected graph is is P. If a problem is in P, it is definitely in NP.

c) True.

If any NP hard problem is solved by non-deterministic polynomial time, it is called NP complete problem. If one could solve an NP hard problem in polynomial time, then one could solve any NP problem in polynomial time.

d) False

If x can be solved deterministically in polynomial time, then it may be included in P but it does not generate P = NP.

8

$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n$. Yes, it can be solved by master theorem.

From master theorem, we know, if

$$T(n) = aT\left(\frac{n}{b}\right) + n^k \log^p n \quad \ldots (i)$$

then, Master theorem can be applied if

$$a \geq 1 \text{ and } b > 1.$$

Since for the recurrence, $a = 4, \geq 1$

$b = 2 > 1$.

So, master theorem is applicable directly for this.

Comparing the given recurrence with (i),

$a = 4$

$b = 2$

$K = 2$

$P = 1$

Now, $\log_b^a = \log_2^4 = 2$

Since, $\log_b^a = K$ and $P > -1$,

the solution $= \Theta(n^K \log^{P+1} n)$

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

$= O(n^3)$ [Upper bound]

9

According to master method.

$$T(n) = aT\left(\frac{n}{b}\right) + n^k \log^p(n) \quad \ldots\ldots (i)$$

where, $a \geq 1$

$b > 1$

(I) Comparing $T(n) = 2T\left(\frac{n}{4}\right) + 7$ with (i),

$a = 2$

$b = 4$

$k = 0$

$p = 0$

Hence, $\log_b^a = \frac{1}{2}$

Here, $\log_b^a > k$,

the solution $= O\left(n^{\log_b^a}\right) = O(\sqrt{n})$.

(II) Comparing $T(n) = 7T\left(\frac{n}{2}\right) + n^2$ with (i),

$a = 7$

$b = 2$

$k = 2$

$p = 0$.

Hence, $\log_b^a = \log_2^7 = 2.81$

$\log_b^a > k$,

$T(n) = O(n^{2.81})$

$= O(n^3)$ [According to extended Master theorem]

(III) Given,

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

So, $a = 3$, $b = 9$, $K = \frac{1}{2}$, $P = 0$.

Hence, $\log_b^a = \log_9^3 = \log_9 9^{\frac{1}{2}} = \frac{1}{2}$

$\log_b^a = K$ and $P > -1$,

$$T(n) = \Theta(n^{\frac{1}{2}} \log n)$$

(IV) Given, $T(n) = T\left(\frac{9n}{10}\right) + n$

Here, $a = 1$, $b = 10/9$, $K = 1$, $P = 0$.

$$\log_b^a = \log_{10/9} 1 = 0$$

$\log_b^a < K$ and $P \geq 0$,

$$T(n) = \Theta(n^K \log^P n)$$

$$= \Theta(n \cdot) \quad [P = 0]$$

(V) Given, $T(n) = 2T\left(\frac{n}{4}\right) + n \log n$

Here, $a = 2$, $b = 4$, $K = 1$, $P = 1$.

$$\log_b^a = \frac{1}{2}$$

$\log_b^a < K$ and $P \geq 0$,

$$T(n) = \Theta(n^K \log^P n)$$

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

a) The divide and conquer strategy is used to solve complicated problems by breaking them into smaller problem and merge all at last. Dynamic programming is used to efficiently solve a range of problems which are overlapping subproblems and optimal structure. Greedy algorithms make locally optimal choices at each step while divide and conquer divides a problem into and solve each subproblem independently. Divide and conquer is a recursive programming while others are not.

b) Greedy algorithm don't always provide optimal solution because they make locally best choices without considering the bigger picture.
For example, in the coin change problem, choosing the largest coin at each step doesn't guarantee the fewest total coins in all cases.

c) To analyse efficiency or time complexity of a greedy algorithm, one's have to identify the operation and compare them with input size.

d) Dynamic Programming is a problem solving strategy that involves breaking down a complex problem into smaller subproblems and solving each subproblem only once, storing it's solution to avoid redundant.

1. Overlapping subproblems
2. Optimal substructure
3. Memoization or tabulation

(e)

Let, $a = \{1, 0, 0, 1, 0, 1, 0, 1\}$

$b = \{0, 1, 0, 1, 0, 1, 1, 0\}$

$$C[i,j] = \begin{cases} 0, & \text{if } i=0 \text{ or } j=0. \\ C[i-1, j-1]+1, & \text{if } a_i = b_j \\ \max(C[i,j-1], C[i-1][j]), & \text{if } a_i \neq b_j \end{cases}$$

i →		0	1	2	3	4	5	6	7	8	9
↓j		Y	0	1	0	1	1	0	1	1	0
0	X	0	0	0	0	0	0	0	0	0	0
1	1	0	↑0	↖(1)	←1	↑1	↑1	←1	↑1	↑1	←1
2	0	0	↑1	←1	↖(2)	←2	←2	←2	←2	←2	←2
3	0	0	↑1	↑1	↑2	←2	←2	↖3	←3	←3	←3
4	1	0	↖1↑	↑2	↑2	↑3	↖(3)	←3	↖4	↖4	←4
5	0	0	↑1	↑2	↑3	←3	←3	↖(4)	↑4	↑4	↖5
6	1	0	↑1	↑2	↑3	↑4	↑4	↑4	↖(5)	↑5	↑5
7	0	0	↑1	↖2↑	↑3	↑4	↑4	↑5	←5	↑5	↖6
8	1	0	↑1	↑2	↑3	↑4	↑4	↑5	↑5	↑5	↖[6]

LCS: 101010

4

Calculation method for fibonacci sequence using dynamic programming :

Fibonacci number series goes like this -

0, 1, 1, 2, 3, 5, 8, ...

The every number in the series is the sum of previous two numbers except the first two numbers. The first and second number 0 and 1 respectively.

Pseudocode :

```
memo = [ ]
memo [0] = 0
memo [1] = 1

for i in range (2 to n):
    memo[i] = memo[i-1] + memo[i-2]

print memo [n]
```

g) Differentiate between Top-down and bottom up dynamic programming:

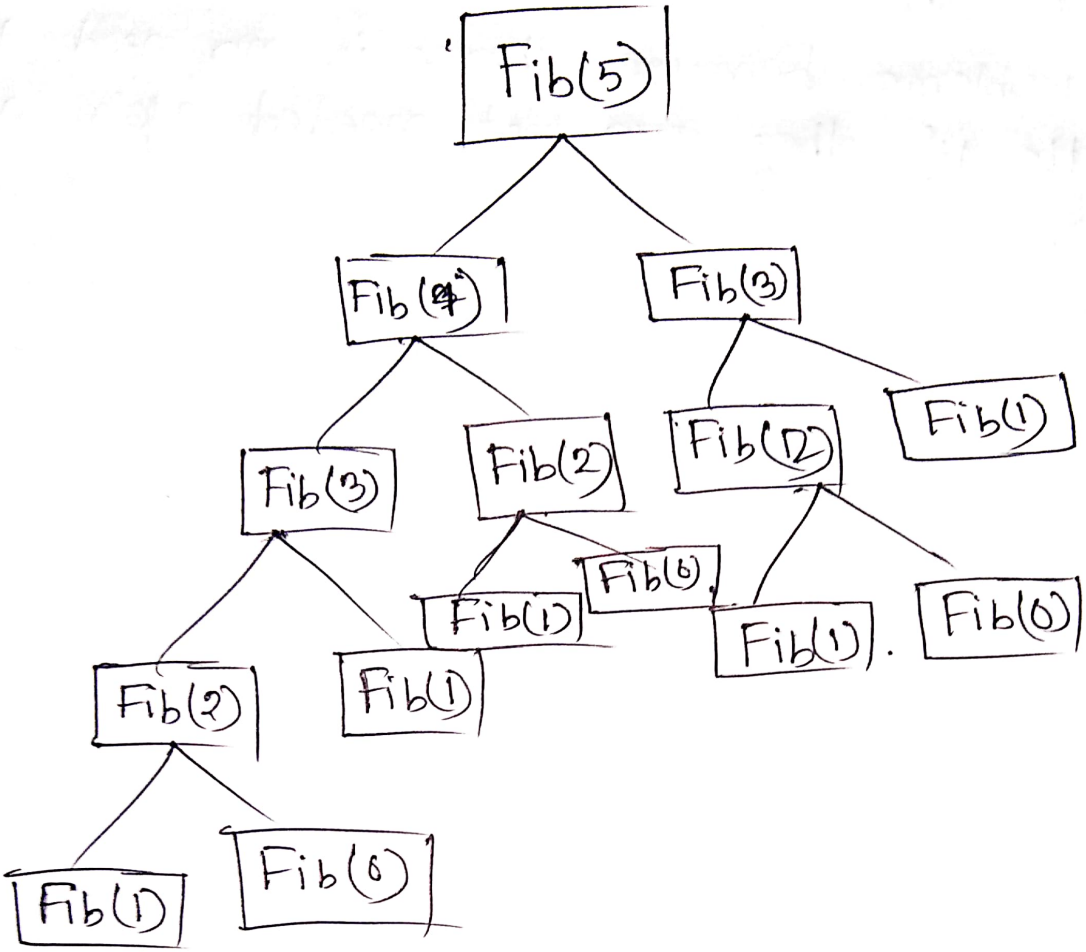
Top down approach	Bottom Up Approach
State-transition relation is difficult to think.	State-transition relation is easy to think.
Code gets complicated when lot of condition required.	Code is easy and less complicated.
Uses memorization method.	Uses tabulation method
Recursive nature.	Iterative nature.

(h)

Subproblems: Dynamic programming is an algorithm paradigm that solves a given complex problem by breaking it into smaller instance of same problem which are called subproblem.

In some problems, there are many small sub problems which are computed many times during finding the solution to the big problem. We compute the same thing again and again. Finding the n-th Fibonacci problem follows the property of having overlapping sub problems. The recursion

tree for fib(5) is:



Here, fib(5) is used for 1 times.
fib(4) is used for 1 times.
fib(3) is used for 2 times
fib(2) is used for 3 times.
fib(1) is used for 5 times.
fib(0) is used for 3 times

So, we see that dynamic programming is used where solution of the same subproblems are needed again and again.

Precomputed results of subproblems are stored in a looked up table to avoid computing same

subproblems again and again. So, dynamic programming is not useful where there are no overlapping subproblems because there is no need to store results if they are not needed again and again.