JGi JAIN DEEMED-TO-BE UNIVERSITY

DAA – Question Bank

Unit - 3

Q.No	Questions	Mark
1.	Define 'Greedy algorithm'? Explain the general method of Greedy Method.	7

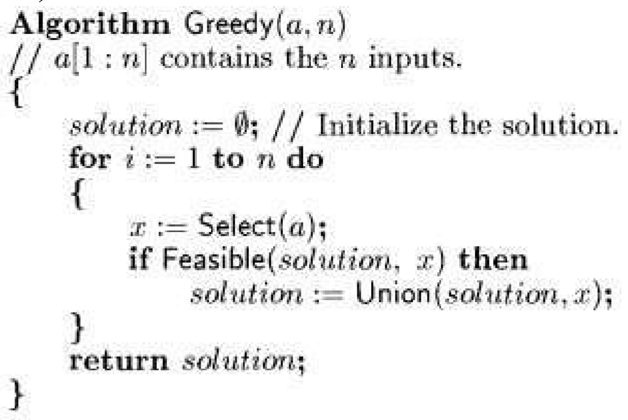
Answer:

Greedy algorithm:
A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

General method
The greedy method is the straight forward design technique applicable to variety of applications.
The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step the choice made must be:

- **feasible**, i.e., it has to satisfy the problem's constraints
- **locally optimal**, i.e., it has to be the best local choice among all feasible choices available on that step
- **irrevocable**, i.e., once made, it cannot be changed on subsequent steps of the algorithm

As a rule, greedy algorithms are both intuitively appealing and simple. Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem. What is usually more difficult is to prove that a greedy algorithm yields an optimal solution (when it does).

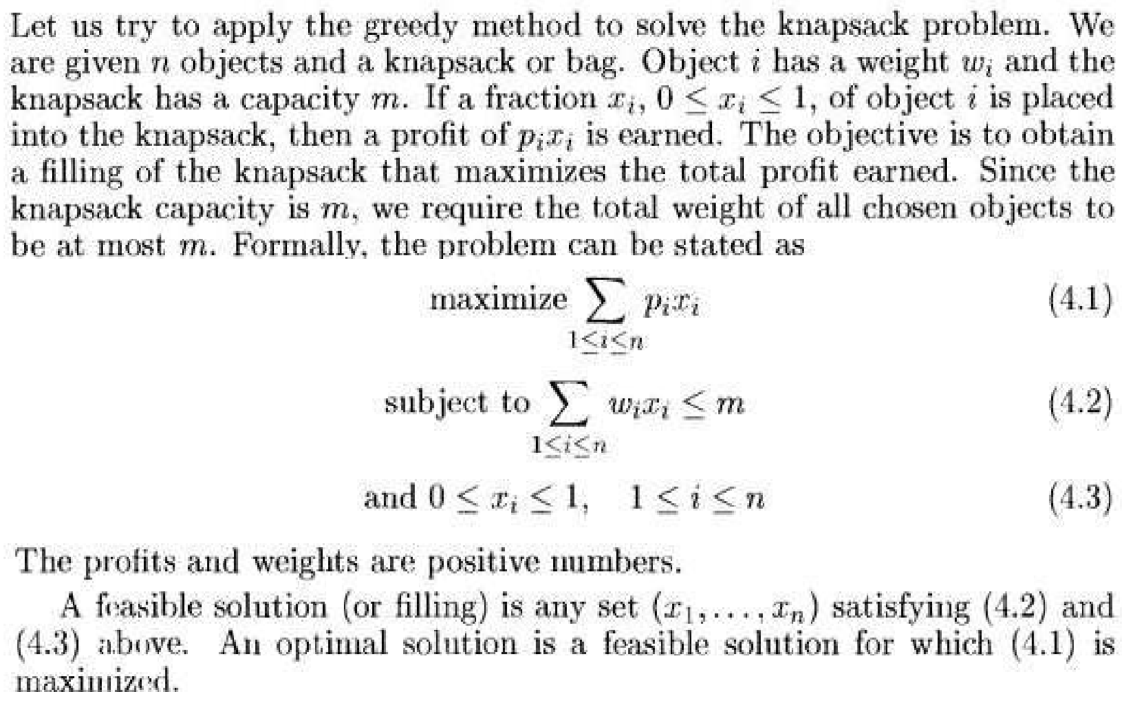


```
Algorithm Greedy($a, n$)
// $a[1:n]$ contains the $n$ inputs.
{
    $solution := \emptyset$; // Initialize the solution.
    for $i := 1$ to $n$ do
    {
        $x :=$ Select($a$);
        if Feasible($solution, x$) then
            $solution :=$ Union($solution, x$);
    }
    return $solution$;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

Greedy method control abstraction for the subset paradigm

2. Write an algorithm knapsack problem .Give example — 14

Answer:



Let us try to apply the greedy method to solve the knapsack problem. We are given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m. Formally, the problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \tag{4.1}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \tag{4.2}$$

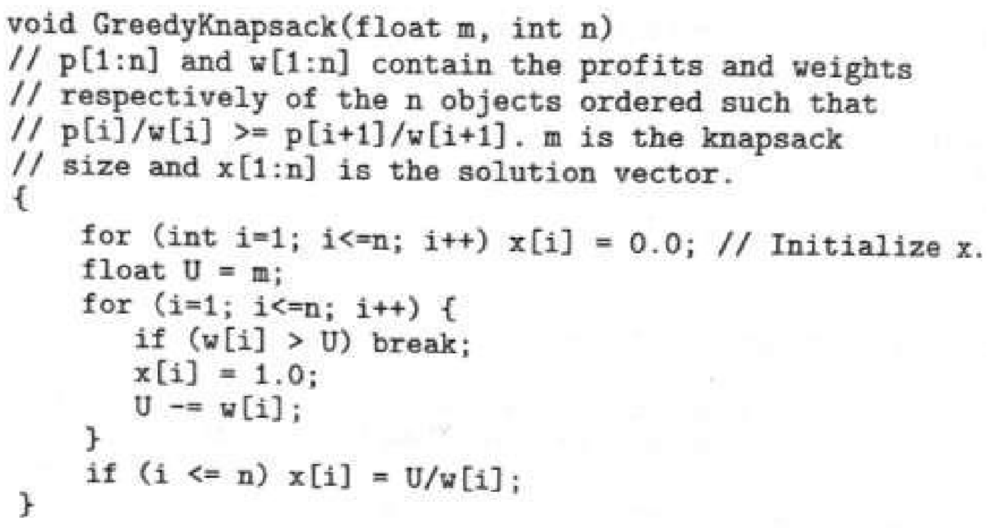
$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \tag{4.3}$$

The profits and weights are positive numbers.

A feasible solution (or filling) is any set $(x_1, \ldots, x_n)$ satisfying (4.2) and (4.3) above. An optimal solution is a feasible solution for which (4.1) is maximized.

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.



```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
     for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
     float U = m;
     for (i=1; i<=n; i++) {
         if (w[i] > U) break;
         x[i] = 1.0;
         U -= w[i];
     }
     if (i <= n) x[i] = U/w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n) time.

Example:
Consider the following instance of the knapsack problem:
n=3, m=20, (p_1, p_2, p_3) =(25, 24, 15), (w_1, w_2, w_3) =(18, 15, 10)

There are several greedy methods to obtain the feasible solutions. Three are discussed here
a) At each step fill the knapsack with the object with **largest profit -** If the object under consideration does not fit, then the fraction of it is included to fill the knapsack. This method does not result optimal solution. As per this method the solution to the above problem is as follows;

Select Item-1 with profit p_1=25, here w_1=18, x_1=1. Remaining capacity = 20-18 = 2 Select Item-2 with profit p_1=24, here w_2=15, x_1=2/15. Remaining capacity = 0
Total profit earned = 28.2.
Therefore optimal solution is (x_1, x_2, x_3) = (1, 2/15, 0) with profit = 28.2

b) At each step fill the object with **smallest weight**

Select Item-3 with profit p_1=15, here w_1=10, x_3=1. Remaining capacity = 20-10 = 10 Select Item-2 with profit p_1=24, here w_2=15, x_1=10/15. Remaining capacity = 0
Total profit earned = 31.
Optimal solution using this method is (x_1, x_2, x_3) = (0, 2/3, 1) with profit = 31
Note: Optimal solution is not guaranteed using method a and b

c) At each step include the object with **maximum profit/weight ratio**
Select Item-2 with profit p_1=24, here w_2=15, x_1=1. Remaining capacity = 20-15=5 Select Item-3 with profit p_1=15, here w_1=10, x_1=5/10. Remaining capacity = 0 Total profit earned = 31.5
Therefore, optimal solution is (x_1, x_2, x_3) = (0, 1, 1/2) with profit = 31.5 This greedy approach always results ***optimal solution***.

3. Explain in detail job sequencing with deadlines problem with an example

Answer;
When we are given a set of 'n' jobs. Associated with each Job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], 1 ≤ r ≤ k are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, 1 ≤ r ≤ k+1.

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

Algorithm GreedyJob (d, J, n)

```
// J is a set of jobs that can be completed by their deadlines.
{
        J := {1};
        for i := 2 to n do
        {
                if (all jobs in J U {i} can be completed by their dead lines)
                then J := J U {i};
        }
}
```

Example:

Let n = 4, (P_1, P_2, P_3, P_4,) = (100, 10, 15, 27) and (d_1 d_2 d_3 d_4) = (2, 1, 2, 1). The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	**OPTIMAL**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

4.	Find solution generated by job sequencing problem with deadlines for 7 jobs given profits 3, 5, 20, 18, 1, 6, 30 and deadlines 1, 3, 4, 3, 2, 1, 2 respectively.	7

Solution: Given

	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

Sort the jobs as per the decreasing order of profit

	J_7	J_3	J_4	J_6	J_2	J_1	J_5
Profit	30	20	18	6	5	3	1
Deadline	2	4	3	1	3	1	2

Maximum deadline is 4. Therefore create 4 slots. Now allocate jobs to highest slot, starting from the job of highest profit

Select Job 7 – Allocate to slot-2

Select Job 3 – Allocate to slot-4

Select Job 4 – Allocate to slot-3

Select Job 6 – Allocate to slot-1 Total profit earned is = 30+20+18+6=74

Slot	1	2	3	4
Job	J_6	J_7	J_4	J_3

5. Find the solution generated by job sequencing when n = 5, (P1, P2, P3, P4, P5) = (20, 15, 10, 5, 1), (d1, d2, d3, d4, d5) = (2, 2, 1, 3, 3) 7

Solution

The Jobs are already sorted according to decreasing order of profit.

Maximum deadline is 3. Therefore create 4 slots. Allocate jobs to highest slot, starting from the job of highest profit

Select Job 1 – Allocate to slot-2

Select Job 2 – Allocate to slot-1 as 2 is already filled

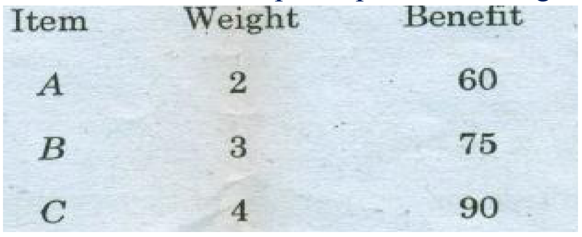
Select Job 3 –Slot-2 &1 are already filled. Cannot be allocated.

Select Job 4 – Allocate to slot-3

Slot	1	2	3
Job	J_2	J_1	J_4

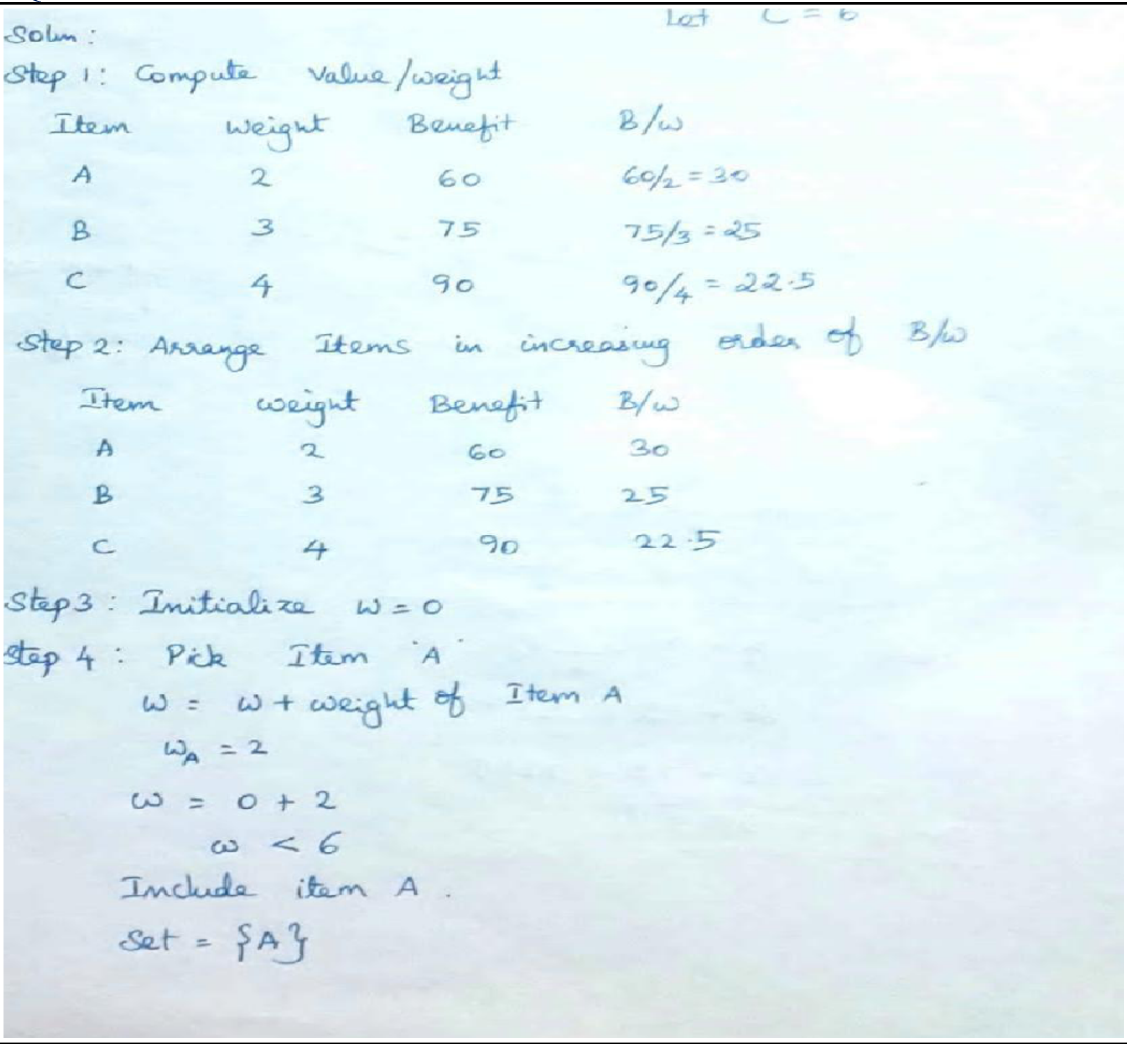
Total profit earned is = 20+15+5=40

6. Find the optimal solution to the fractional knapsack problem with given data; 4M



Item	Weight	Benefit
A	2	60
B	3	75
C	4	90

Capacity C=6



Let $C = 6$

Soln :

Step 1: Compute Value/weight

Item	Weight	Benefit	B/w
A	2	60	$60/2 = 30$
B	3	75	$75/3 = 25$
C	4	90	$90/4 = 22.5$

Step 2: Arrange Items in increasing order of B/w

Item	weight	Benefit	B/w
A	2	60	30
B	3	75	25
C	4	90	22.5

Step 3 : Initialize $w = 0$

Step 4 : Pick Item 'A'

$w = w +$ weight of Item A

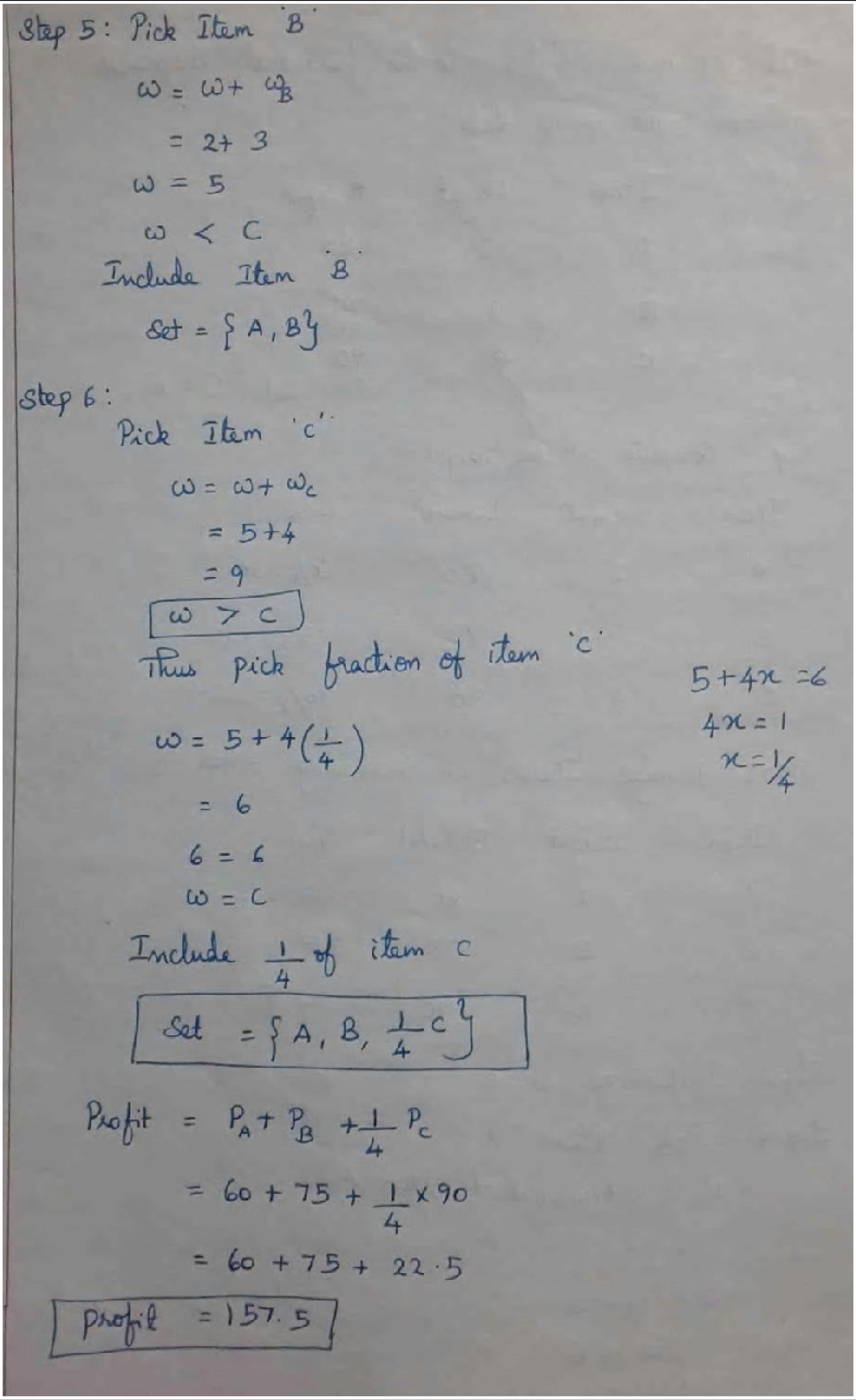
$w_A = 2$

$w = 0 + 2$

$w < 6$

Include item A.

Set = $\{A\}$

Step 5: Pick Item 'B'

$\omega = \omega + \omega_B$

$= 2 + 3$

$\omega = 5$

$\omega < C$

Include Item 'B'

Set $= \{A, B\}$

Step 6:

Pick Item 'c'

$\omega = \omega + \omega_c$

$= 5 + 4$

$= 9$

$\boxed{\omega > c}$

Thus pick fraction of item 'c'

$\omega = 5 + 4\left(\frac{1}{4}\right)$

$= 6$

$6 = 6$

$\omega = C$

$5 + 4x = 6$

$4x = 1$

$x = 1/4$

Include $\frac{1}{4}$ of item c

$\boxed{\text{Set} = \{A, B, \tfrac{1}{4}c\}}$

Profit $= P_A + P_B + \frac{1}{4} P_c$

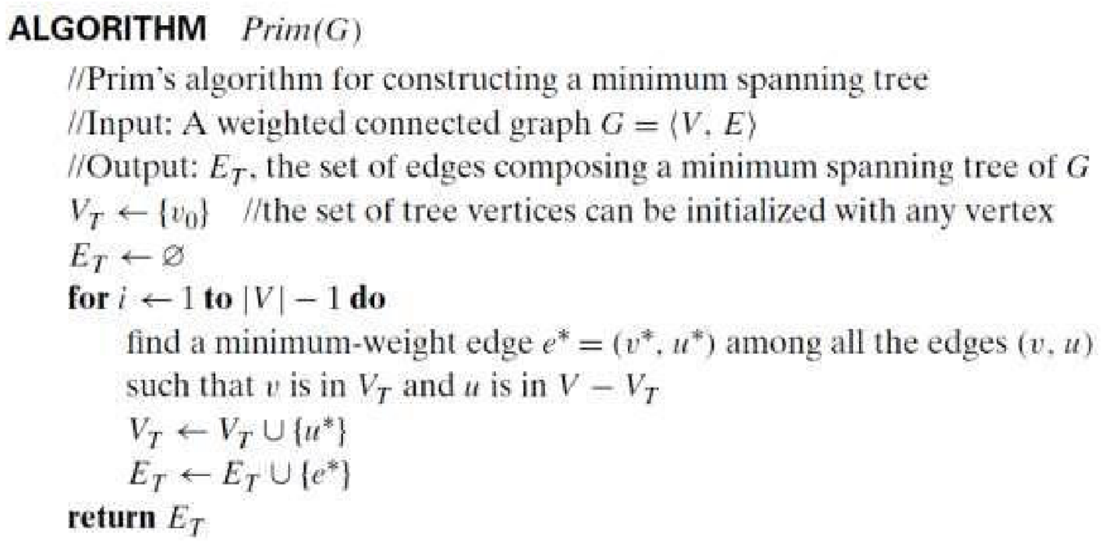
$= 60 + 75 + \frac{1}{4} \times 90$

$= 60 + 75 + 22.5$

$\boxed{\text{profit} = 157.5}$

7. Write and Analyse Prim’s Algorithm 6

Answer:



```
ALGORITHM Prim(G)
    //Prim’s algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    $V_T \leftarrow \{v_0\}$    //the set of tree vertices can be initialized with any vertex
    $E_T \leftarrow \varnothing$
    for $i \leftarrow 1$ to $|V| - 1$ do
        find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges $(v, u)$
        such that $v$ is in $V_T$ and $u$ is in $V - V_T$
        $V_T \leftarrow V_T \cup \{u^*\}$
        $E_T \leftarrow E_T \cup \{e^*\}$
    return $E_T$
```

Analysis of Efficiency;

The efficiency of Prim’s algorithm depends on the data structures chosen for the **graph** itself and for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

1. If a graph is represented by its **weight matrix** and the priority queue is implemented as an **unordered array**, the algorithm’s running time will be in **$\Theta(|V|^2)$**. Indeed, on each

   of the $|V| - 1$iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

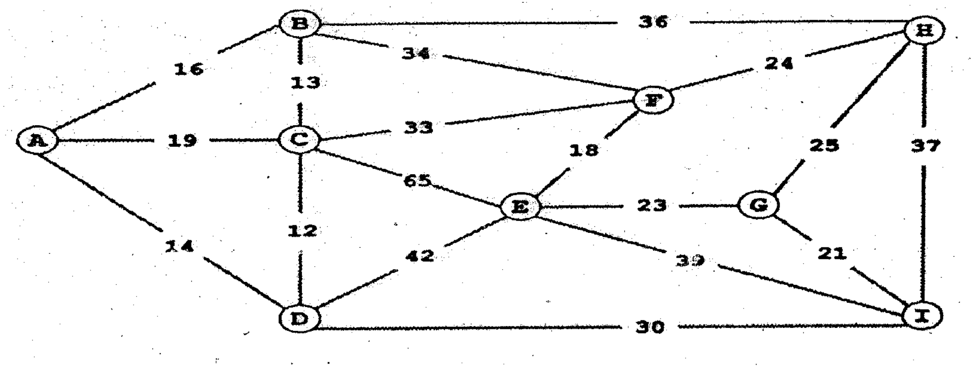
We can implement the priority queue as a **min-heap**. (A min-heap is a complete binary tree in which every element is less than or equal to its children.) Deletion of the smallest element from and insertion of a new element into a min-heap of size n are **O(log n)** operations.

2. If a graph is represented by its **adjacency lists** and the priority queue is implemented as a **min-heap**, the running time of the algorithm is in **$O(|E| \log |V|)$.**

This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibly, changes of an element’s priority in a min-heap of size not exceeding $|V|$. Each of these operations, as noted earlier, is a $O(\log |V|)$ operation. Hence, the running time of this implementation of Prim’s algorithm is in

$(|V| - 1 + |E|)\, O(\log |V|) = O(|E| \log |V|)$ because, in a connected graph, $|V| - 1 \le |E|$.

8. Consider the following weighted Graph:



Give the list of edges in the MST in the order that Prim's Algorithm inserts them. Start Prim's algorithm from Vertex A.

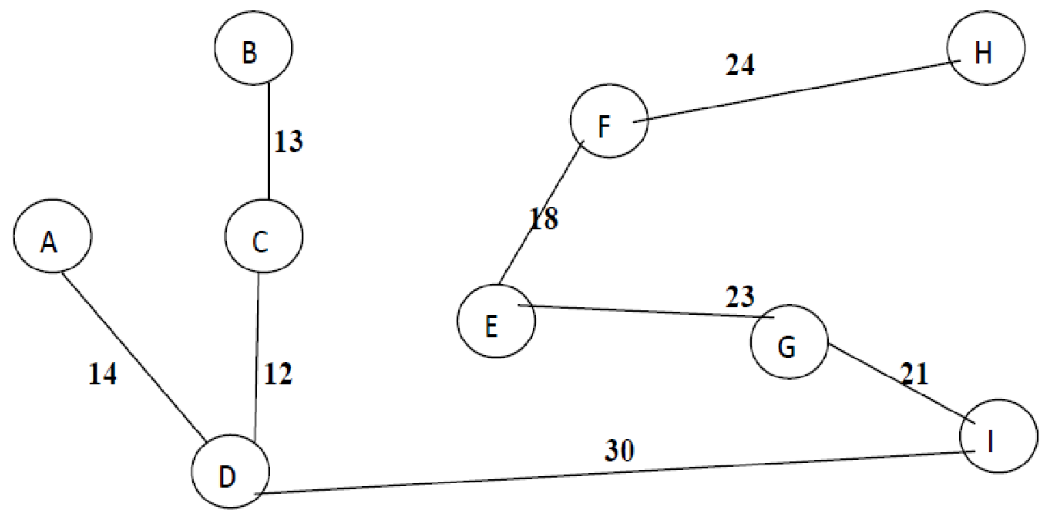
Answer;

Tree vertices	Remaining Vertices
A(-,-)	B(A,16),D(A,14),C(A,19),E(-∞,0), F(-∞,0),G(-∞,0),H(-∞,0),I(-∞,0)
D(A,14)	B(A,16),C(D,12),E(D,42),F(-∞,0), G(-∞,0),H(-∞,0),I(D,30)
C(D,12)	B(C,13),E(D,42),F(C,33),G(-∞,0), H(-∞,0),I(D,30)
B(C,13)	E(D,42),F(C,33),G(-∞,0),H(B,36),I(D,30)
I(D,30)	E(I,39),F(C,33),G(I,21),H(B,36)

8

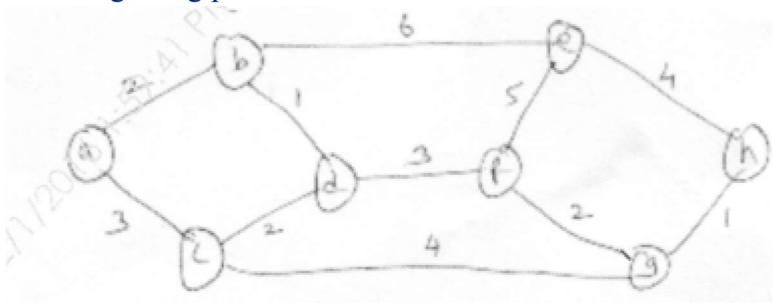
G(I,21)	E(G,23),F(C,33),H(G,25)
E(G,23)	F(E,18),H(G,25)
F(E,18)	H(F,24)
H(F,24)	--------

The minimum spanning tree for the given graph is:



Total cost of the minimum spanning tree=155

9. Compare Prims and Kruskals method for finding Minimum spanning Tree find MST for following using prims method — 14



Answer:

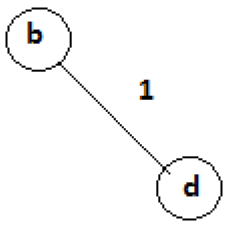
Prims	**Kruskal's**
This algorithm is for obtaining minimum spanning tree by	This algorithm is for obtaining minimum spanning tree but it is not necessary to

selecting the adjacent vertices of already selected vertices.	choose adjacent vertices of already selected vertices.
Prim's algorithm initializes with a node	Kruskal's algorithm initiates with an edge
Prim's algorithms span from one node to another	Kruskal's algorithm select the edges in a way that the position of the edge is not based on the last step
In prim's algorithm, graph must be a connected graph	Kruskal's can function on disconnected graphs too.
Prim's algorithm has a time complexity of O(V2)	Kruskal's time complexity is O(logV).

Problem:

Step1:

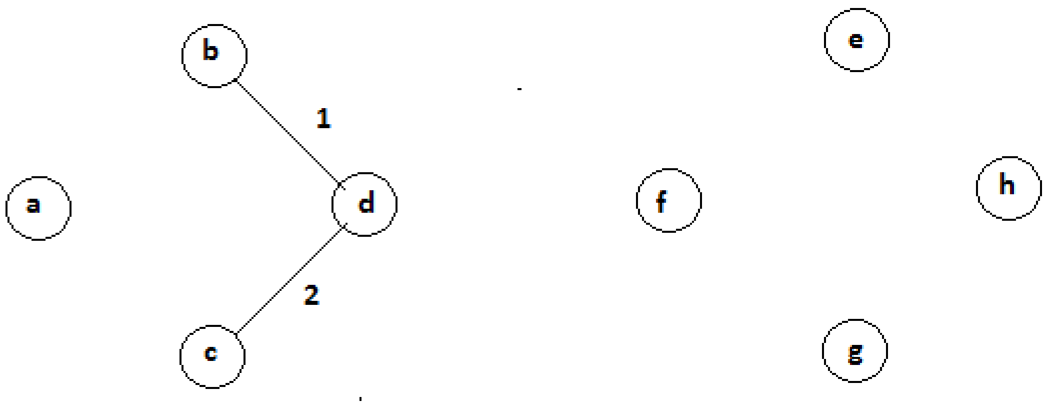
We will first select a minimum distance edge from given graph.



V={b,d} Cost=1

Step 2:

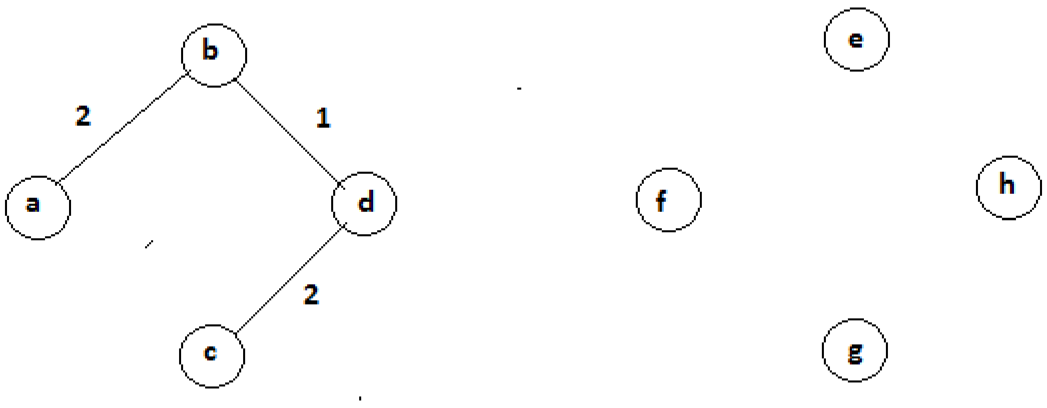
The next minimum distance edge is d-c. This edge is adjacent to previously selected vertex d.



V={b,d,c} Cost=3

Step 3:

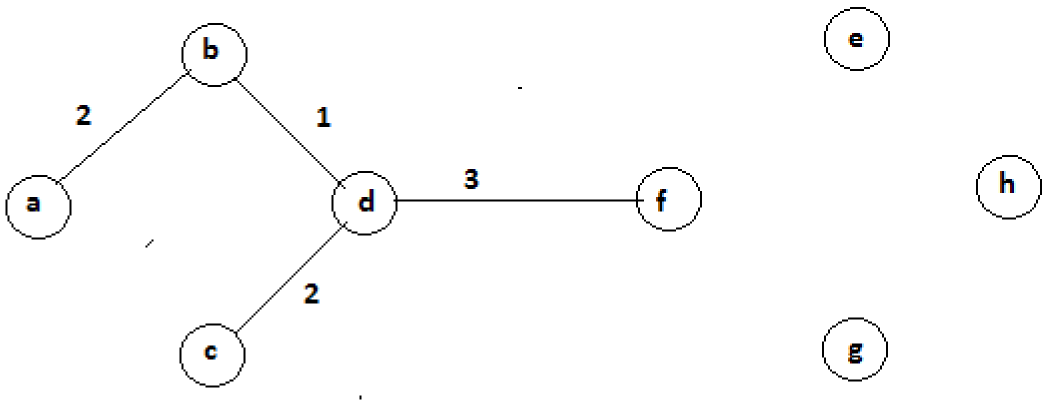
The next minimum distance edge is a-b. This edge is adjacent to previously selected vertex b.



V={b,d,c,a} Cost=5

Step 4:

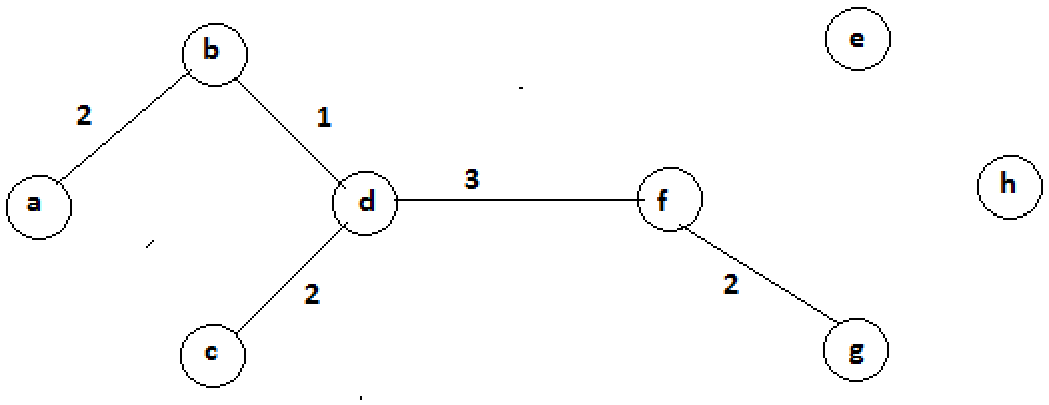
The next minimum distance edge is d-f. This edge is adjacent to previously selected vertex d.



V={ b,d,c,a ,f} Cost=8

Step 5:

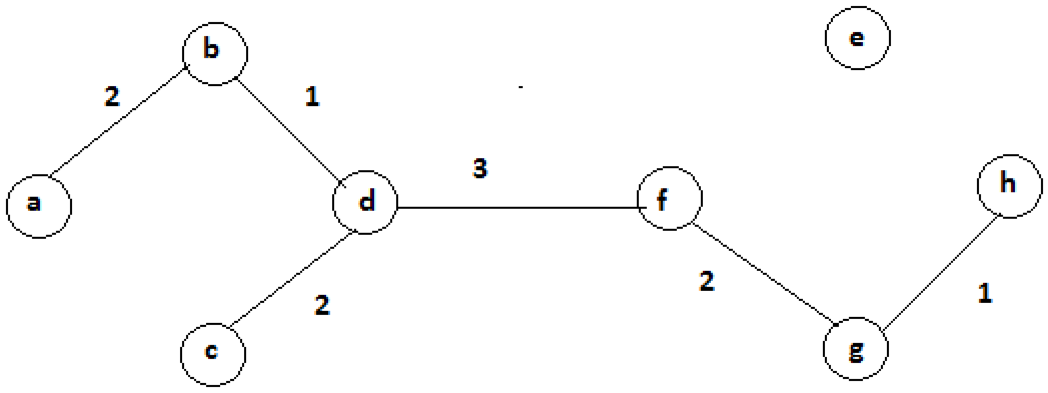
The next minimum distance edge is f-g. This edge is adjacent to previously selected vertex f.



V={ b,d,c,a ,f ,g} Cost=10

Step 6:

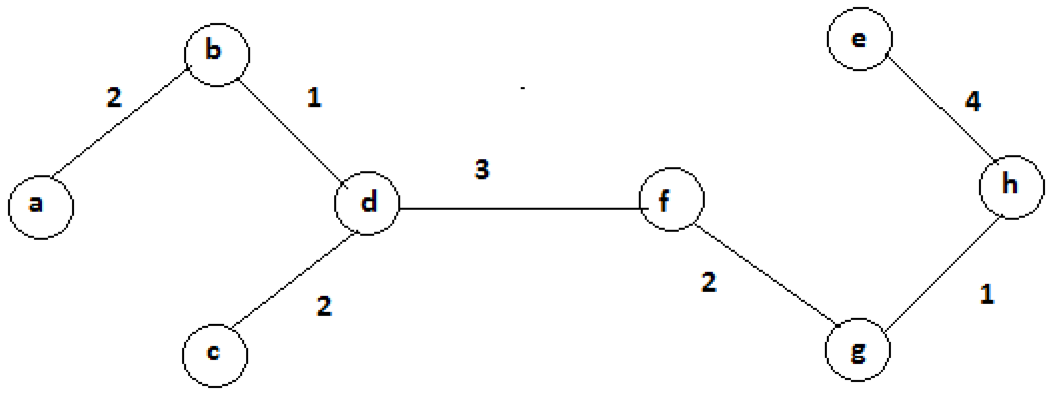
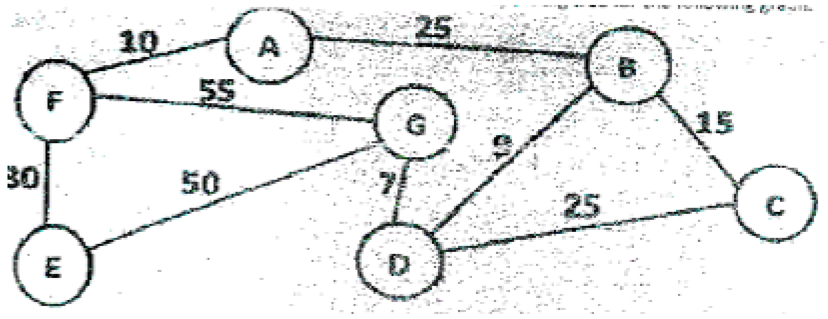
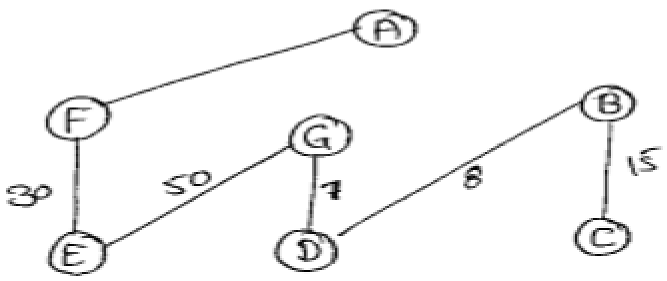
The next minimum distance edge is g-h. This edge is adjacent to previously selected vertex g.



V={ b,d,c,a ,f ,g ,h} Cost=11

Step 7:

The next minimum distance edge is e-h. This edge is adjacent to previously selected vertex h.

	 b 2 1 e 4 a d 3 f h c 2 2 g 1 V={ b,d,c,a ,f ,g ,h} Cost=15 Hence, the above is the minimum spanning tree with total cost 15.
10.	Using Prim's and Kruskal's Algorithm, find minimum spanning tree for following graph:  10 A 25 B F 55 G 15 30 50 8 7 25 C E D Answer; **solving using Prim's Algorithm:**  Step 1: draw the given graph.

Step2: remove all loops. Any edge that starts and ends at the same vertex is called a loop. In this case, there is no loop.

Step 3: remove all parallel edges two vertex except one with the least weight. In this case, there are no parallel edges.

Step 4: Create a table, where number of rows = number of columns = number of vertex in the graph.

Step 5: now, put 0 in cells having same row and column name.

Step 6: Now, Start filling other columns. Start with Vertex A. Find the edge that directly connects Vertex A and B. In this case, we have edge of weight 25 that directly connects A and B.

Step 7: put 25 in AB and BA.

Step 8: Repeat these steps for all the vertex that are directly connected.

Step 9: If any vertex is not connected directly, for eg: Vertex A and D; then put ∞(infinity symbol).

Here AD and DA will have ∞.

-	**A**	**B**	**C**	**D**	**E**	**F**	**G**
A	0	25	∞	∞∞	∞	10	∞
B	25	0	15	8	∞	∞	∞
C	∞	15	0	25	∞	∞	∞
D	∞	8	25	0	∞	∞	7
E	∞	∞	∞	∞	0	30	50
F	10	∞	∞	∞	30	0	55
G	∞	∞	∞	7	50	55	0

Table is now completely filled. Next Task is to find Minimum spanning Tree.

Step 10: Start from vertex A. Find the smallest value in row A

Smallest Value in row A is 10. Mark AF and FA and draw the graph.

Smallest Value in row B is 8. Mark BD and DB and draw the graph.

Smallest Value in row C is 15. Mark CB and BC and draw the graph.

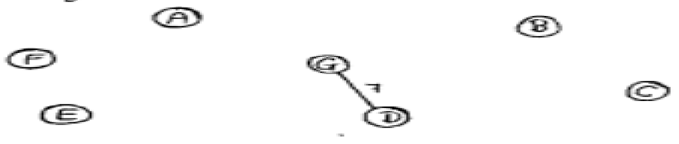
Similarly, do for the all the rows and you will get the following table.

-	**A**	**B**	**C**	**D**	**E**	**F**	**G**
A	0	25	∞	∞∞	∞	10	∞
B	25	0	15	8	∞	∞	∞
C	∞	15	0	25	∞	∞	∞
D	∞	8	25	0	∞	∞	7
E	∞	∞	∞	∞	0	30	50
F	10	∞	∞	∞	30	0	55
G	∞	∞	∞	7	50	55	0

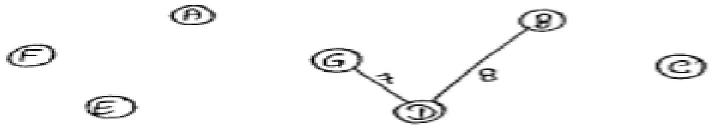
(Note: we will not consider 0 as it will correspond to the same vertex.)

So minimum spanning tree using prim’s algorithm is below:

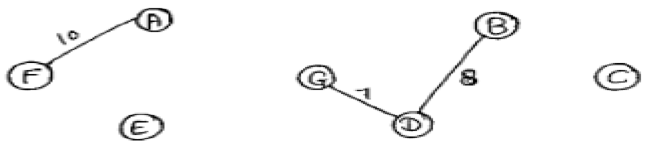
// Drawing GD



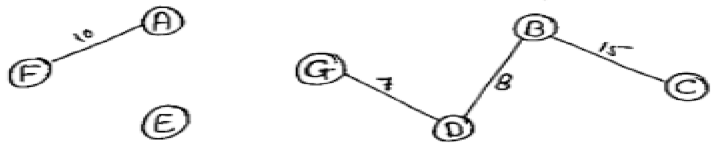
// Connecting BD



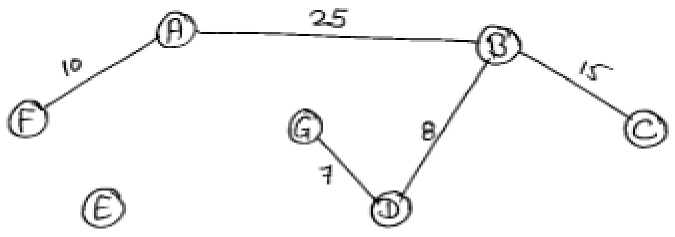
// Connecting AF



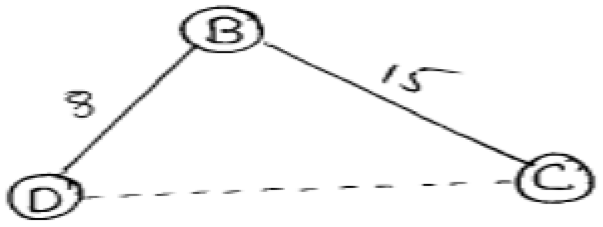
// Connecting BC



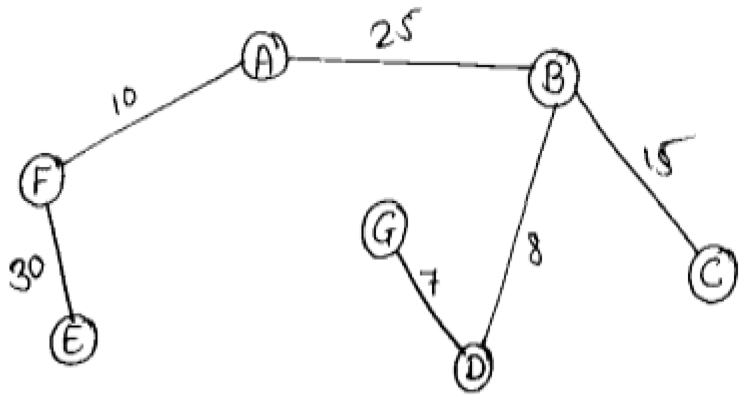
// Connecting AB



Now, if connect DC, so it will form a ckt like below:



Therefore, we will skip this edges and select next edge FE.



Since our minimum spanning tree should be of 6 edges, so we will stop here and this is our MST.

solving using kruskal's algorithm:

Step 1: Remove all the loops. Any edge that starts and ends at the same vertex is a loop. In our case, it does not exist.

Step 2: remove all parallel edges two vertex except one with the least weight. In this case, we don't have any parallel edges.

Step 3: Create the edge table. An edge table will have name of all the edges along with their weight in ascending order.

If you look at the graph, you will notice there are 9 edges in total, so our edge table will have 9 (nine) columns.

Edge	**GD**	**BD**	**AF**	**BC**	**AB**	**DC**	**FE**	**EG**	**FG**
Weight	7	8	10	15	25	25	30	50	55

In our case, AB and DC both edges have weight 25 so we will consider both. And you can write them in any order i.e AB first then DC or vice versa.

Step 4: To find minimum spanning tree, number of edges will be

Number of edges=No. of vertices- 1

In our case, no. of vertices are 8, so our minimum spanning tree will have 7 edges.

Step 5: To find the MST, we will start with the smallest weight edge and keep selecting edges that does not form any circuit with the previously selected edges.

Since 7 is the smallest weight so we will select GD.

Drawing this edge GD

11. **Write and analyse Kruskal's Algorithm** 7

Background: Kruskal's algorithm is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named Kruskal's algorithm, after Joseph Kruskal. Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G

= (V, E) as an acyclic sub graph with |V | - 1 edges for which the **sum of the edge weights is the smallest**. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of sub graphs, which are always **acyclic** but are not necessarily connected on the intermediate stages of the algorithm.

Working: The algorithm begins by **sorting** the graph's edges in **non-decreasing** order of their **weights**. Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current sub graph if such an inclusion **does not create a cycle** and simply **skipping the edge otherwise.**

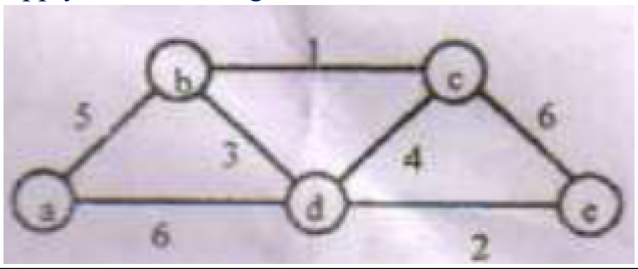
```
ALGORITHM  Kruskal(G)
    //Kruskal's algorithm for constructing a minimum spanning tree
    //Input: A weighted connected graph $G = \langle V, E \rangle$
    //Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
    sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
    $E_T \leftarrow \varnothing$;   $ecounter \leftarrow 0$     //initialize the set of tree edges and its size
    $k \leftarrow 0$                          //initialize the number of processed edges
    while $ecounter < |V| - 1$ do
        $k \leftarrow k + 1$
        if $E_T \cup \{e_{i_k}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{i_k}\}$;   $ecounter \leftarrow ecounter + 1$
    return $E_T$
```

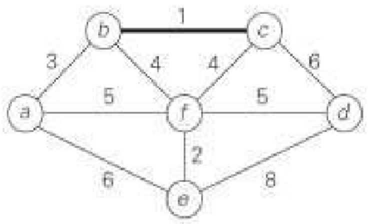
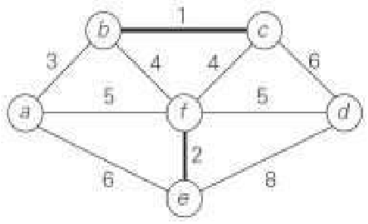
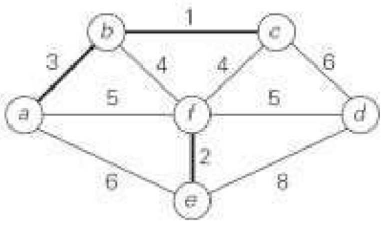
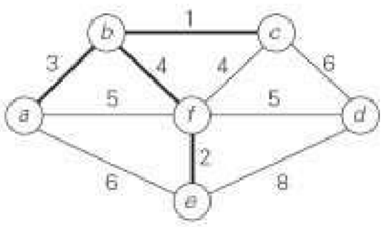
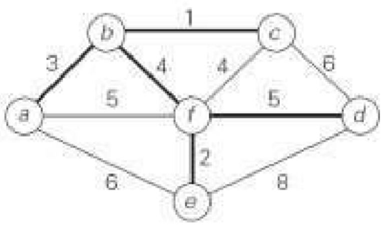
Analysis of Efficiency

The crucial check whether two vertices belong to the same tree can be found out using **union-find algorithms.**

Efficiency of Kruskal's algorithm is based on the time needed for **sorting the edge weights** of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in **O (|E| log |E|).**

12. Apply Kruskal's Algorithm to find the minimum spanning tree of the following graph: 7



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

13. Explain about single source shortest path problem using Dijksta's Algorithm with example 14

Answer;

Single-source shortest-paths problem is defined as follows. For a given vertex called the *source* in a weighted connected graph, the problem is to find shortest paths to all its other vertices. The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

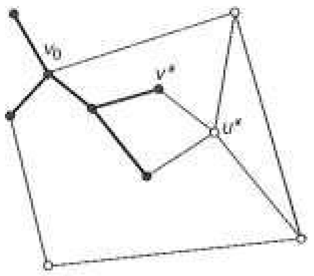
Dijkstra's Algorithm

Dijkstra's Algorithm is the best-known algorithm for the single-source shortest-paths problem. This algorithm is applicable to undirected and directed graphs with nonnegative weights only.
Working - Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source.

▪ First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.

▪ In general, before its ith iteration commences, the algorithm has already identified the shortest paths to *i*-1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph shown in the figure.

▪ Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. The set of vertices adjacent to the vertices in T_i can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source.



To identify the ith nearest vertex, the algorithm computes, for every fringe vertex *u,* the sum of the distance to the nearest tree vertex *v* (given by the weight of the edge (*v,* u)) and the length *d.,* of the shortest path from the source to *v* (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

▪ To facilitate the algorithm's operations, we label each vertex with two labels.

○ The numeric label **d** indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex.

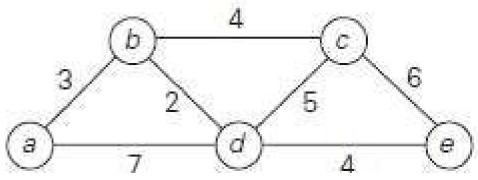
○ The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the sources and vertices that are adjacent to none of the current tree vertices.)
With such labeling, finding the next nearest vertex u* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.

▪ After we have identified a vertex u* to be added to the tree, we need to perform two operations:

○ Move u^* from the fringe to the set of tree vertices.

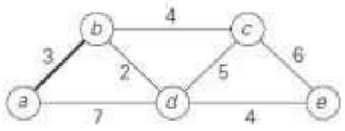
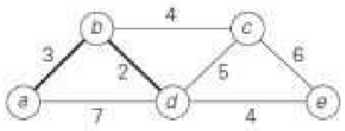
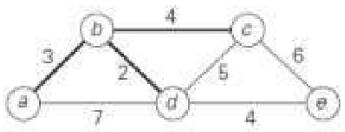
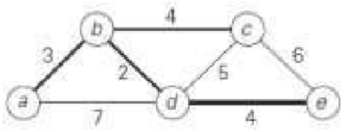
○ For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

Example;



The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

```
ALGORITHM Dijkstra(G, s)
    //Dijkstra's algorithm for single-source shortest paths
    //Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
    //         and its vertex $s$
    //Output: The length $d_v$ of a shortest path from $s$ to $v$
    //           and its penultimate vertex $p_v$ for every vertex $v$ in $V$
    Initialize(Q)    //initialize priority queue to empty
    for every vertex $v$ in $V$
        $d_v \leftarrow \infty$;   $p_v \leftarrow$ null
        Insert(Q, v, $d_v$)   //initialize vertex priority in the priority queue
    $d_s \leftarrow 0$;   Decrease(Q, s, $d_s$)   //update priority of $s$ with $d_s$
    $V_T \leftarrow \varnothing$
    for $i \leftarrow 0$ to $|V| - 1$ do
        $u^* \leftarrow$ DeleteMin(Q)   //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) < d_u$
                $d_u \leftarrow d_{u^*} + w(u^*, u)$;   $p_u \leftarrow u^*$
                Decrease(Q, u, $d_u$)
```

Analysis:
The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.
Efficiency is **$\Theta(|V|^2)$** for graphs represented by their ***weight matrix*** and the priority queue implemented as an ***unordered array***.
For graphs represented by their ***adjacency lists*** and the priority queue implemented as a ***min-heap***, it is in **O (|E| log |V|)**