UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Course Name : Design Analysis & Algorithm Laboratory

Algorithm Design Techniques

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:
Divide the original problem into a set of sub problems.
Solve every sub problem individually, recursively.
Combine the solution of the sub problems (top level) into a solution of the whole original problem.
2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.
Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

Algorithm Design Techniques

3. Dynamic Programming: Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to Optimization Problems.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

Algorithm Design Techniques

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

7. Randomized Algorithm: A randomized algorithm uses a random number at least once during the computation make a decision.

Complexities of an Algorithm

Constant Complexity:

It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.

Logarithmic Complexity:

It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.
For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.

Complexities of an Algorithm

Linear Complexity: It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements. For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.

Cubic Complexity: It imposes a complexity of **$O(n^3)$**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.

For example, if there exist 100 elements, it is going to execute 1,000,000 steps.

Exponential Complexity: It imposes a complexity of **$O(2^n)$, O(N!), $O(n^k)$, ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.

For example, if N = 10, then the exponential function 2^N will result in 1024.

Quadratic Complexity: It imposes a complexity of **$O(n^2)$**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem. If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.

- The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique.
- The main function of this approach is that the decision is taken on the basis of the currently available information.

Characteristics of Greedy method :

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.

- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

The components that can be used in the greedy algorithm are:

Candidate set: A solution that is created from the set is known as a candidate set.

Selection function: This function is used to choose the candidate or subset which can be added in the solution.

Feasibility function: A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.

Objective function: A function is used to assign the value to the solution or the partial solution.

Solution function: This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm

It is used in finding the shortest path.

It is used to find the minimum spanning tree using the prim's algorithm or the

Kruskal's algorithm.

It is used in a job sequencing with a deadline.

This algorithm is also used to solve the fractional knapsack problem.

Disadvantages of using Greedy algorithm:

Greedy algorithm makes decisions based on the information available at each phase without considering the broader problem.

So, there might be a possibility that the greedy solution does not give the best solution for every problem.

- **Greedy Approach**

- Simple, straightforward and easy to implement.
- Always looking for immediate gain by overlooking the long term effect.
- Doesn’t always produce the optimal solution.

- Algorithms use the greedy approach. Here is a list of few of them −
- Knapsack Problem
- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover
- Job Scheduling Problem

The activity selection problem is a mathematical optimization problem.

Our first illustration is the problem of scheduling a resource among several challenge activities.

We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc.

Each Activity "i" has **start time** s_i and a **finish time** f_i, where $s_i \leq f_i$. If selected activity "i" take place meanwhile the half-open time interval $[s_i, f_i)$. Activities i and j are **compatible** if the intervals (s_i, f_i) and $[s_i, f_i)$ do not overlap (i.e. i and j are compatible if $s_i \geq f_i$ or $s_i \geq f_i$).

Activity Selection Problem

•Greedy approach
•Perform sorting on the activities according to their finish time
•Pick the first activity from the sorted list and print
•For the remaining activities, if the start time of the activity is greater than or equal to the finish time of the previously selected activity then select the activity and print it.

Activity Selection Problem

```
i = 0;                                   // first activity always gets selected
printf("%d ", i);

for (j = 1; j < n; j++) {                         //n=total no. of activities

        if (start_time[j] >= finish_time[i])
 {

                printf ("%d ", j);
                i = j;
}
}
```

Activity Selection Problem

GREEDY- ACTIVITY SELECTOR (s, f)

```
1.  n ← length [s]
2.  A ← {1}
3.  j ← 1.
4.  for i ← 2 to n
5.  do if $s_i \geq f_i$
6.  then A ← A ∪ {i}
7.  j ← i
8.  return A
```

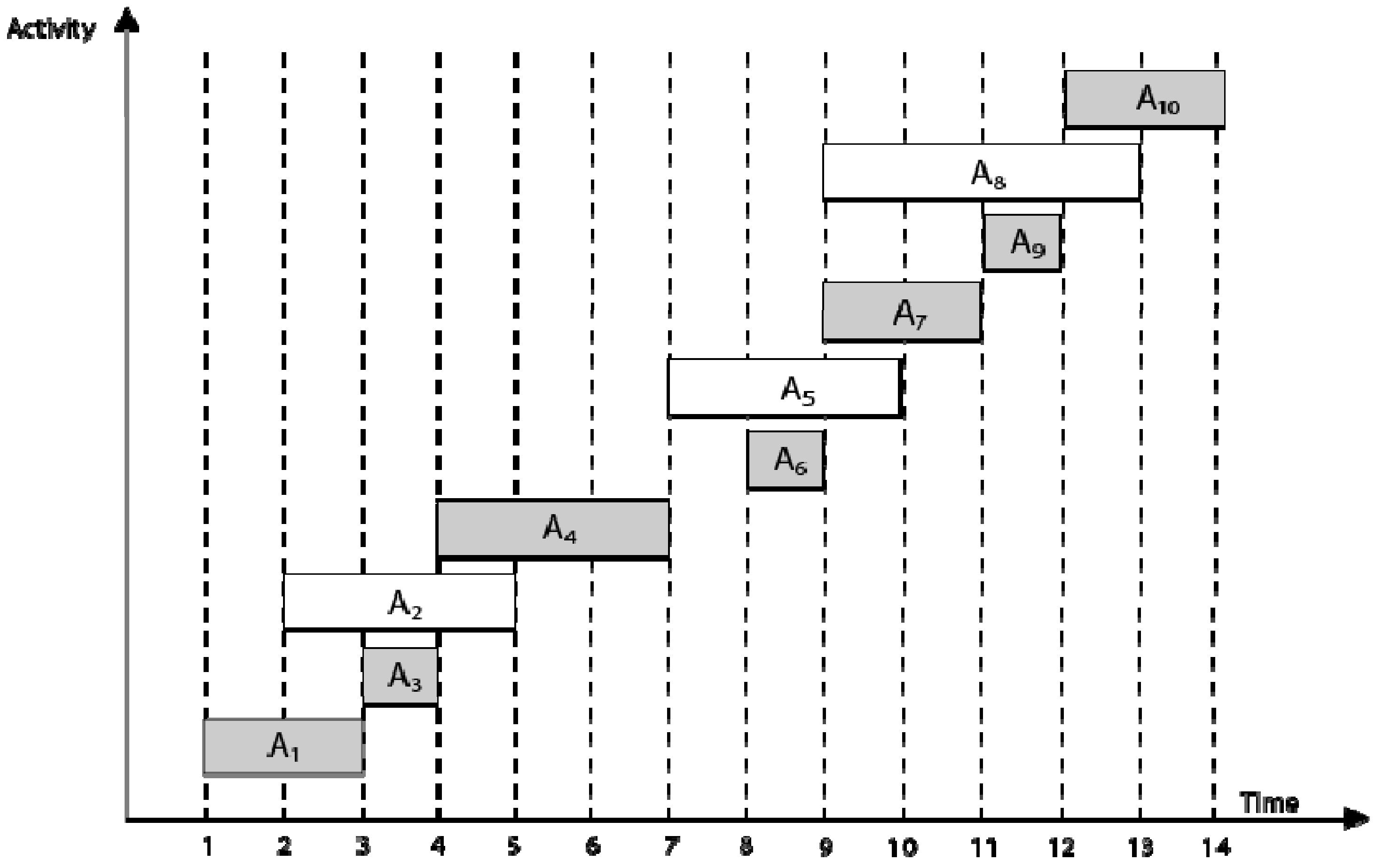
Example: Given 10 activities along with their start and end time as

$S = (A_1\ A_2\ A_3\ A_4\ A_5\ A_6\ A_7\ A_8\ A9\ A10)$

Si = (1,2,3,4,7,8,9,9,11,12) fi = (3,5,4,7,10,9,11,13,12,14)

➢ **The complexity of this problem is O(n log n) when the list is not sorted. When the sorted list is provided the complexity will be O(n).**

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Activity Selection Problem

Now, schedule A1

Next schedule A3 as A1 and A3 are non-interfering.

Next skip A2 as it is interfering.

Next, schedule A4 as A1 A3 and A4 are non-interfering, then next, schedule A6 as A1 A3 A4 and A6 are non-interfering.

Skip A5 as it is interfering.

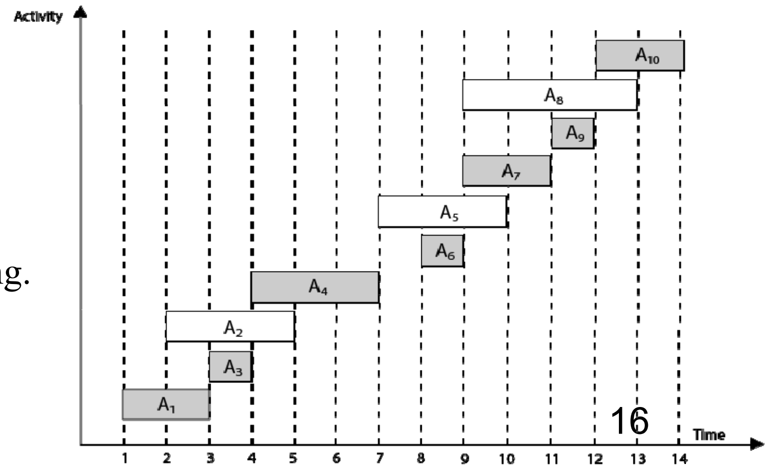
Next, schedule A7 as A1 A3 A4 A6 and A7 are non-interfering.

Next, schedule A9 as A1 A3 A4 A6 A7 and A9 are non-interfering.

Skip A8 as it is interfering.

Next, schedule A10 as A1 A3 A4 A6 A7 A9 and A10 are non-interfering.

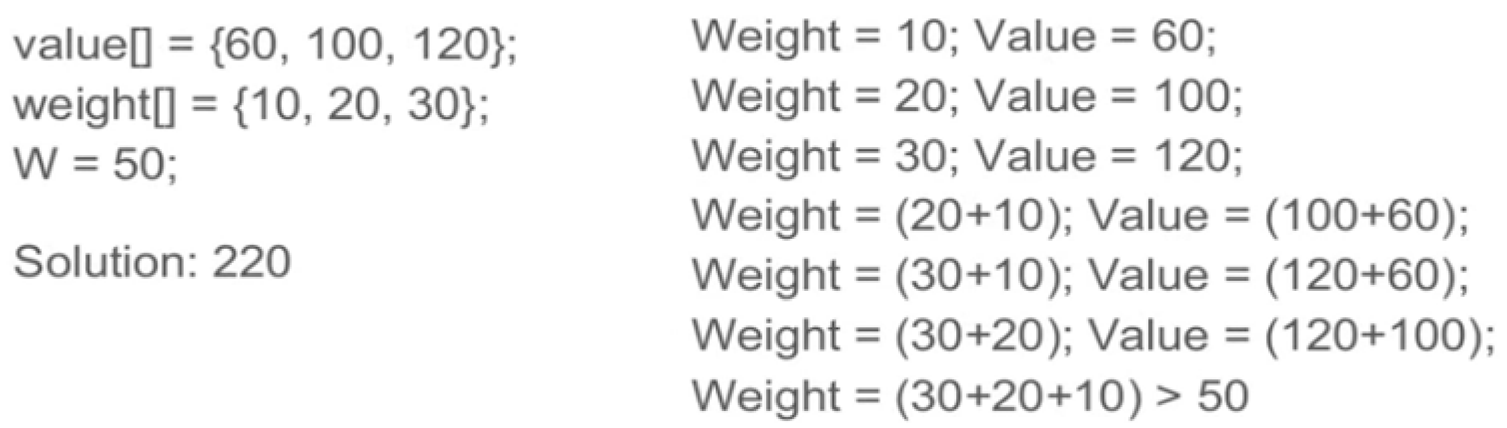
Thus the final Activity schedule is:



Knapsack Problem

- Knapsack is similar to a container.
- Items with weights and profits.
- Some items need to be put in container in such a way total value produces a maximum profit.
- Two types of knapsack:
  - 0/1 knapsack
  - fractional knapsack
- 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack.
- Items can be broken into smaller pieces in fractional knapsack.

0/1 Knapsack Problem



value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

0/1 Knapsack Problem

Method 1: **Recursion by Brute-Force algorithm**

Approach: A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.
***Optimal Sub-structure*:** To consider all subsets of items, there can be two cases for every item.
Case 1: The item is included in the optimal subset.
Case 2: The item is not included in the optimal set.
Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values.
Maximum value obtained by n-1 items and W weight (excluding nth item).
Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).
If the weight of 'nth' item is greater than 'W', then the nth item cannot be included and **Case 1** is the only possibility.

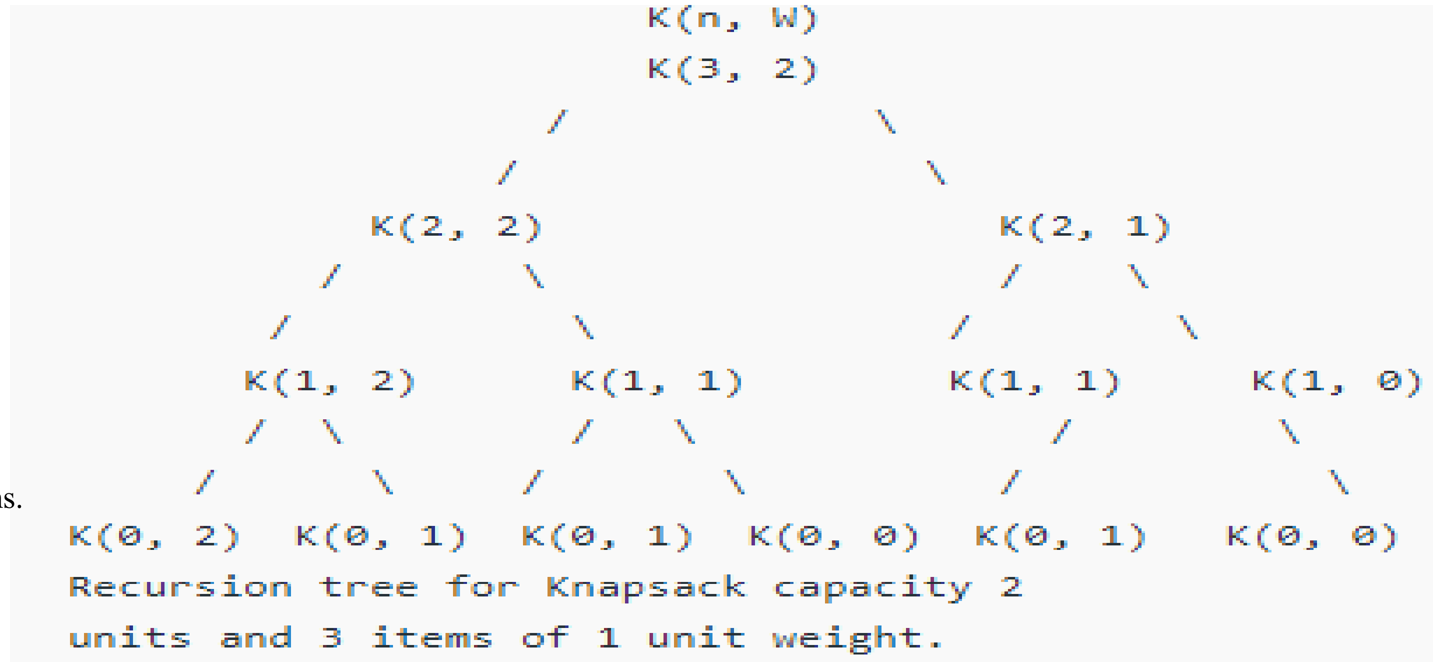
It should be noted that the above function computes the same sub-problems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. The time complexity of this naive recursive solution is exponential (2^n).
In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W. The recursion tree is for following sample inputs. wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

KN01[N-1][W_max- Wi]+P_i

KN01[N-1][W_max]

If(Wi>W_max)
{
KN01[N-1][W_max]
}

Complexity Analysis:
Time Complexity: O(2^n).
As there are redundant sub problems.
Auxiliary Space O(1).
As no extra data structure has been used for storing values.



```
                       K(n, W)
                       K(3, 2)
                   /            \
                 /                \
            K(2, 2)                  K(2, 1)
          /       \                  /    \
        /           \              /        \
       K(1, 2)      K(1, 1)        K(1, 1)     K(1, 0)
       /  \         /   \              /         \
     /      \     /       \          /             \
K(0, 2)  K(0, 1)  K(0, 1)  K(0, 0)  K(0, 1)   K(0, 0)
Recursion tree for Knapsack capacity 2
units and 3 items of 1 unit weight.
```

0/1 Knapsack Problem

<u>Method 2</u>: Like other typical Dynamic Programming(DP) problems, re-computation of same subproblems can be avoided by constructing a temporary array K[][] in bottom-up manner. Following is Dynamic Programming based implementation.

Approach: In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.

The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values > wi'. Now two possibilities can take place:

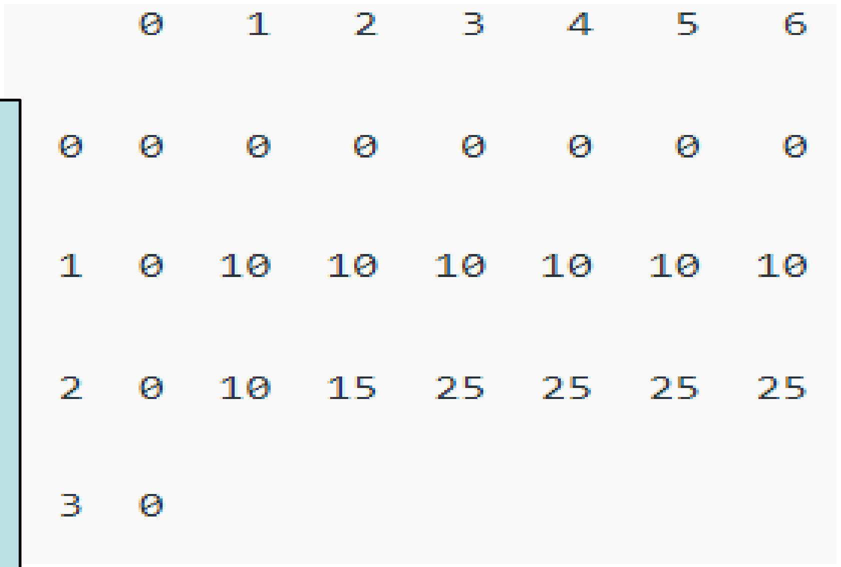
Fill 'wi' in the given column.
Do not fill 'wi' in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' weight in 'jth' column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of 'wi'+ value of the column weighing 'j-wi' in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualization will make the concept clear:

Let weight elements = {1, 2, 3} Let weight values = {10, 15, 40} Capacity=6.

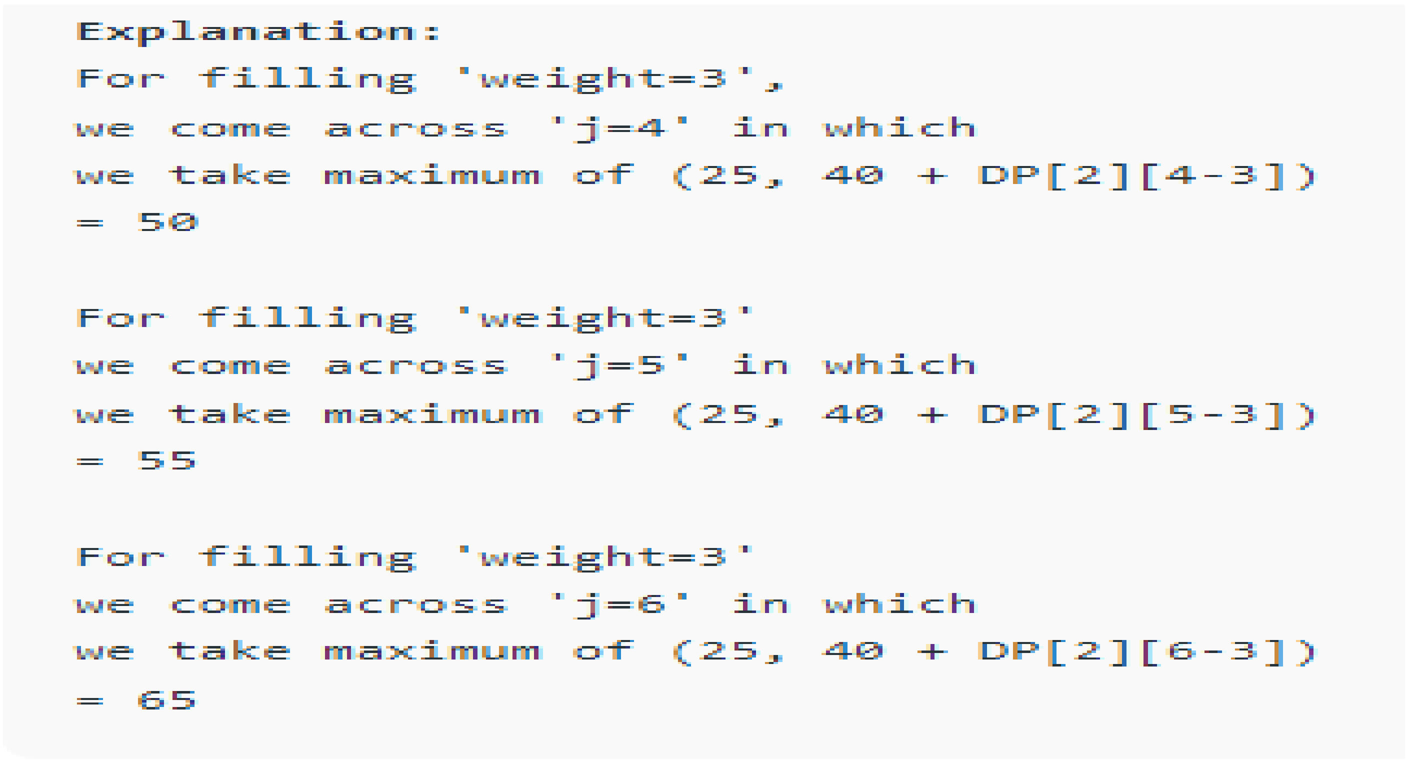
DP[i][M]=0 =0 if i=0, M=0
DP[i-1][M] if Wi>M
Max(Vi+DP[i-1][M-Wi],DP[i-1][M]) if i>0 and Wi<M

i-> Number of item
M-> maximum weight of Knapsack



	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0						

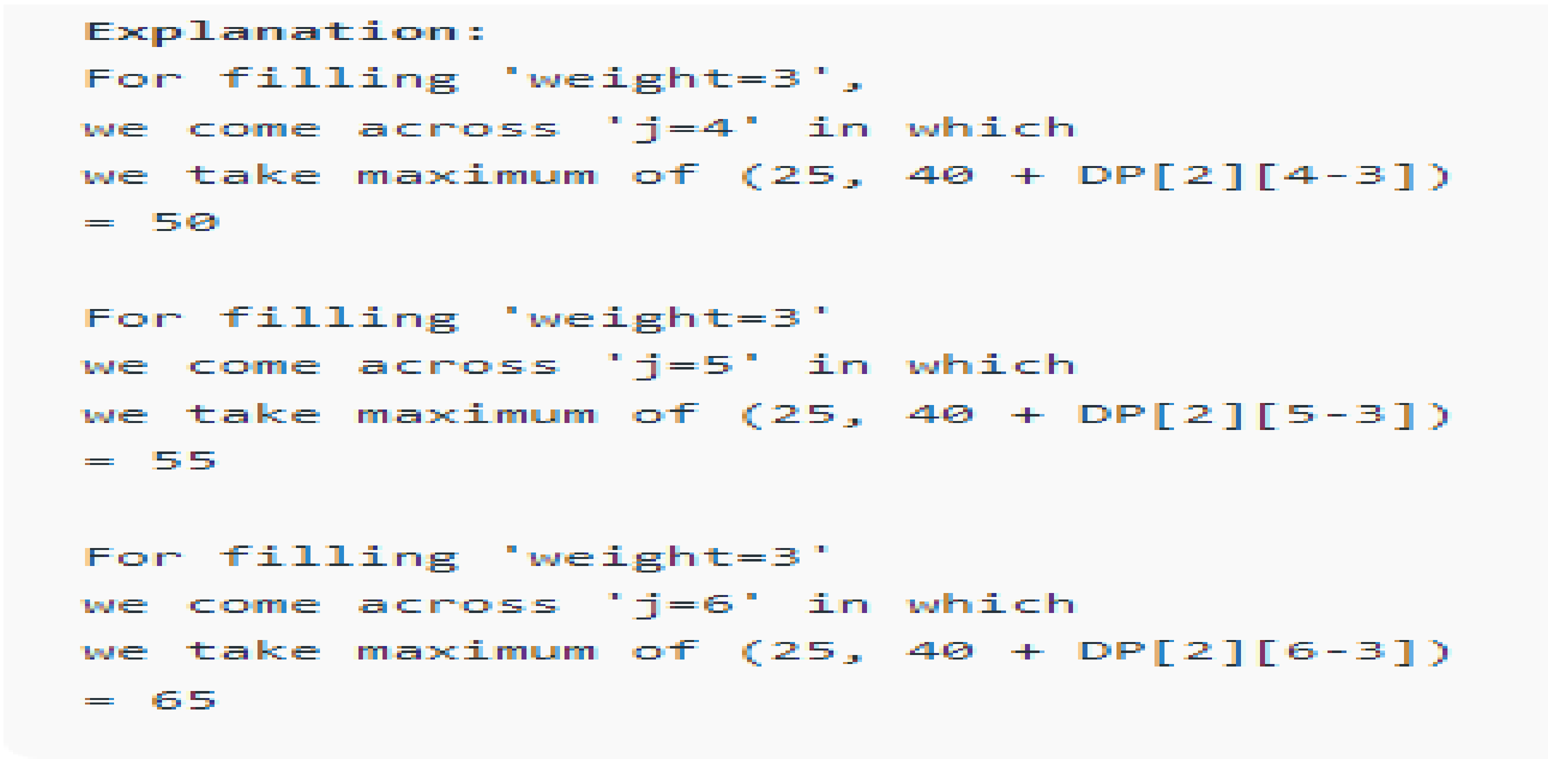
0/1 Knapsack Problem



```
Explanation:
For filling 'weight=3',
we come across 'j=4' in which
we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3'
we come across 'j=5' in which
we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3'
we come across 'j=6' in which
we take maximum of (25, 40 + DP[2][6-3])
= 65
```

Complexity Analysis:
Time Complexity: O(N*W).
where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
Auxiliary Space: O(N*W).
The use of 2-D array of size ‘N*W

0/1 Knapsack Problem



```
Explanation:
For filling 'weight=3',
we come across 'j=4' in which
we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3'
we come across 'j=5' in which
we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3'
we come across 'j=6' in which
we take maximum of (25, 40 + DP[2][6-3])
= 65
```

Complexity Analysis:
Time Complexity: O(N*W).
where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
Auxiliary Space: O(N*W).
The use of 2-D array of size ‘N*W

Example Dynamic programming approach

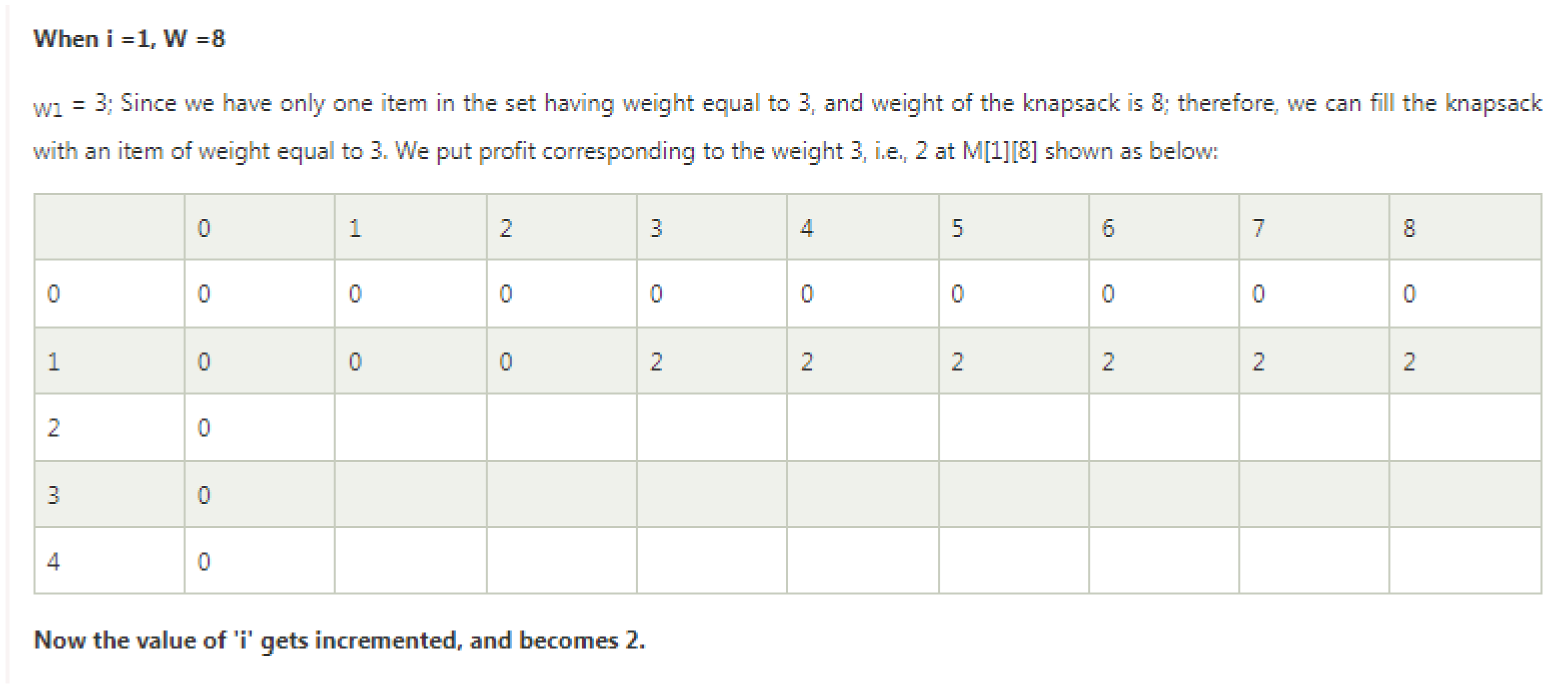
	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

First, we write the weights in the ascending order and profits according to their weights shown as below:

w_i = {3, 4, 5, 6}

p_i = {2, 3, 4, 1}

Example Dynamic programming approach



When i =1, W =8

w_1 = 3; Since we have only one item in the set having weight equal to 3, and weight of the knapsack is 8; therefore, we can fill the knapsack with an item of weight equal to 3. We put profit corresponding to the weight 3, i.e., 2 at M[1][8] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0								
3	0								
4	0								

Now the value of 'i' gets incremented, and becomes 2.

First, we write the weights in the ascending order and profits according to their weights shown as below:

w_i = {3, 4, 5, 6}

p_i = {2, 3, 4, 1}

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	1	3	3	3	5	5
4	0	0	0	2	3	3	4		

When i = 4, W = 7

The weight corresponding to the value 4 is 6, i.e., $w_4 = 6$. Since we have four items in the set of weights 3, 4, 5, and 6 respectively, and the weight of the knapsack is 7. Here, if we add two items of weights 3 and 4 then it will produce the maximum profit, i.e., (2 + 3) equals to 5, so we add 5 at M[4][7] shown as below:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2
2	0	0	0	2	3	3	3	5	5
3	0	0	0	2	3	3	3	5	5
4	0	0	0	2	3	3	4	5	

When i = 4, W = 8

Fractional Knapsack Problem:

Fractional Knapsack Problem
The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

This problem can be solved with the help of using two techniques:
Brute-force approach: The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
Greedy approach: In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.

There are basically three approaches to solve the problem:

- The first approach is to select the item based on the maximum profit.
- The second approach is to select the item based on the minimum weight.
- **The third approach is to calculate the ratio of profit/weight.**

Fractional Knapsack Problem:

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], m)

```
for   i = 1 to n
Compute Pi/Wi,
Sort object in decreasing order of Pi/Wi
for i =1 to n,          // n -> number of items ; m -> Size of the knapsack Wi->Weight of the items

If(m>0 && Wi<= m)
{
m= m-Wi;
P=P+Pi;
}
Else break;
}
If(m>0)
P=P+Pi(m/Wi)
```

Analysis

If the provided items are already sorted into a decreasing order of pi/wi, then the whileloop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Fractional Knapsack Problem:

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for   i = 1 to n
do x[i] = 0
weight = 0
for  i = 1 to n
if weight + w[i] ≤ W then
 x[i] = 1
weight = weight + w[i]
else
x[i] = (W - weight) / w[i]
weight = W
Break
 return x
```

Analysis

If the provided items are already sorted into a decreasing order of pi/wi, then the whileloop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Fractional Knapsack Problem:

Example
Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

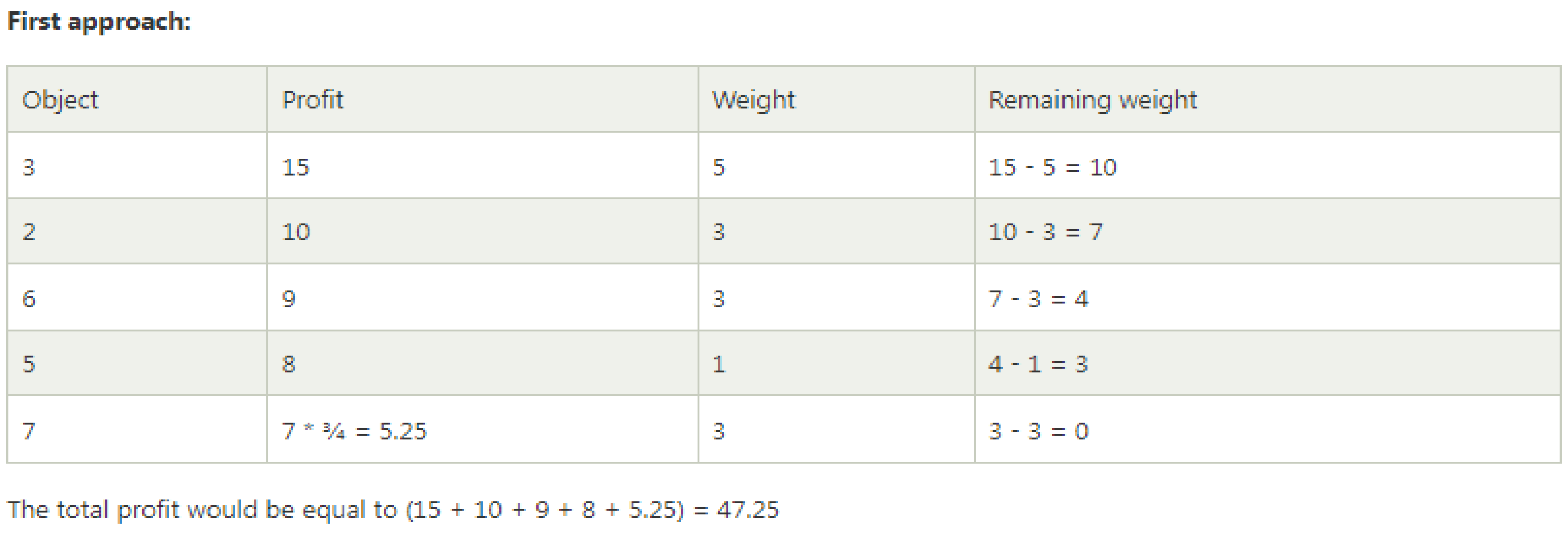
First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item.

However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.
Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.
Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.
The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**
And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

Fractional Knapsack Problem:

Objects:	1	2	3	4	5	6	7
Profit (P):	10	15	7	8	9	4	7
Weight(w):	1	3	5	4	1	3	2

W (Weight of the knapsack): 15
n (no of items): 7
First approach: Based o highest Profit



First approach:

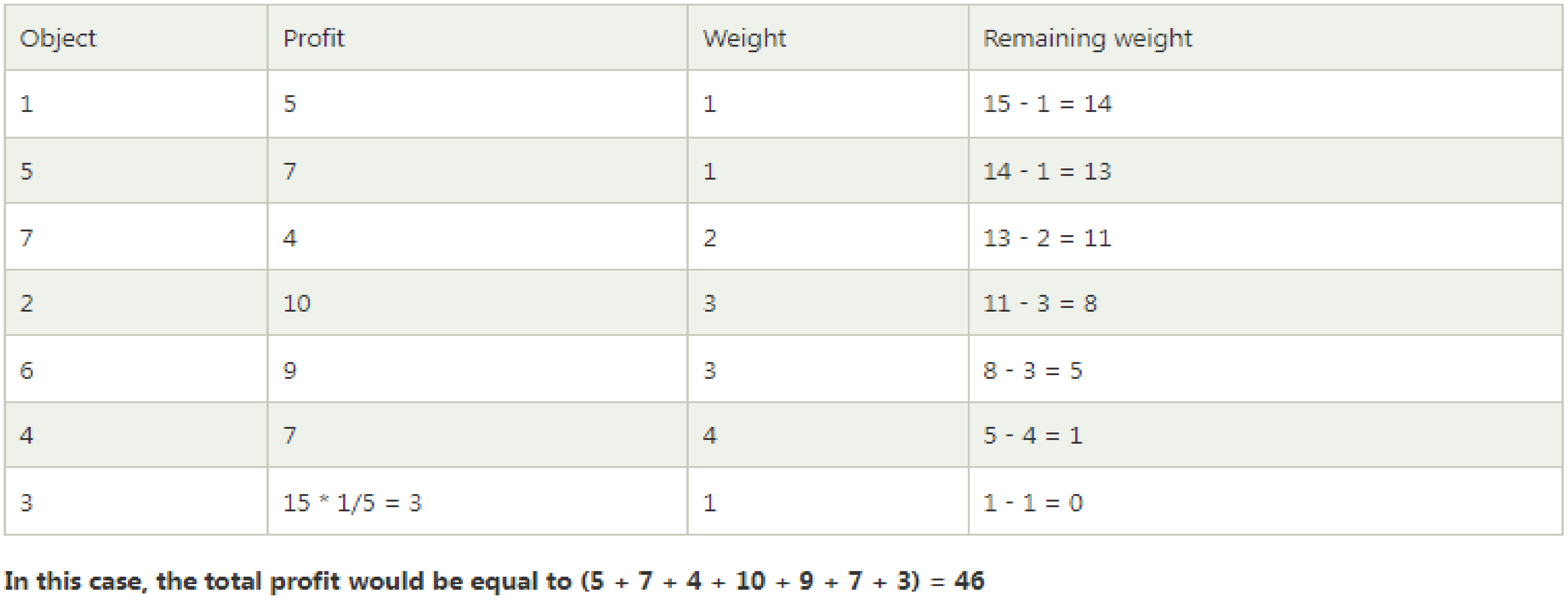
Object	Profit	Weight	Remaining weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
7	7 * ¾ = 5.25	3	3 - 3 = 0

The total profit would be equal to (15 + 10 + 9 + 8 + 5.25) = 47.25

Fractional Knapsack Problem:

Objects:	1	2	3	4	5	6	7
Profit (P):	10	15	7	8	9	4	7
Weight(w):	1	3	5	4	1	3	2

W (Weight of the knapsack): 15
n (no of items): 7
The second approach is to select the item based on the minimum weight.

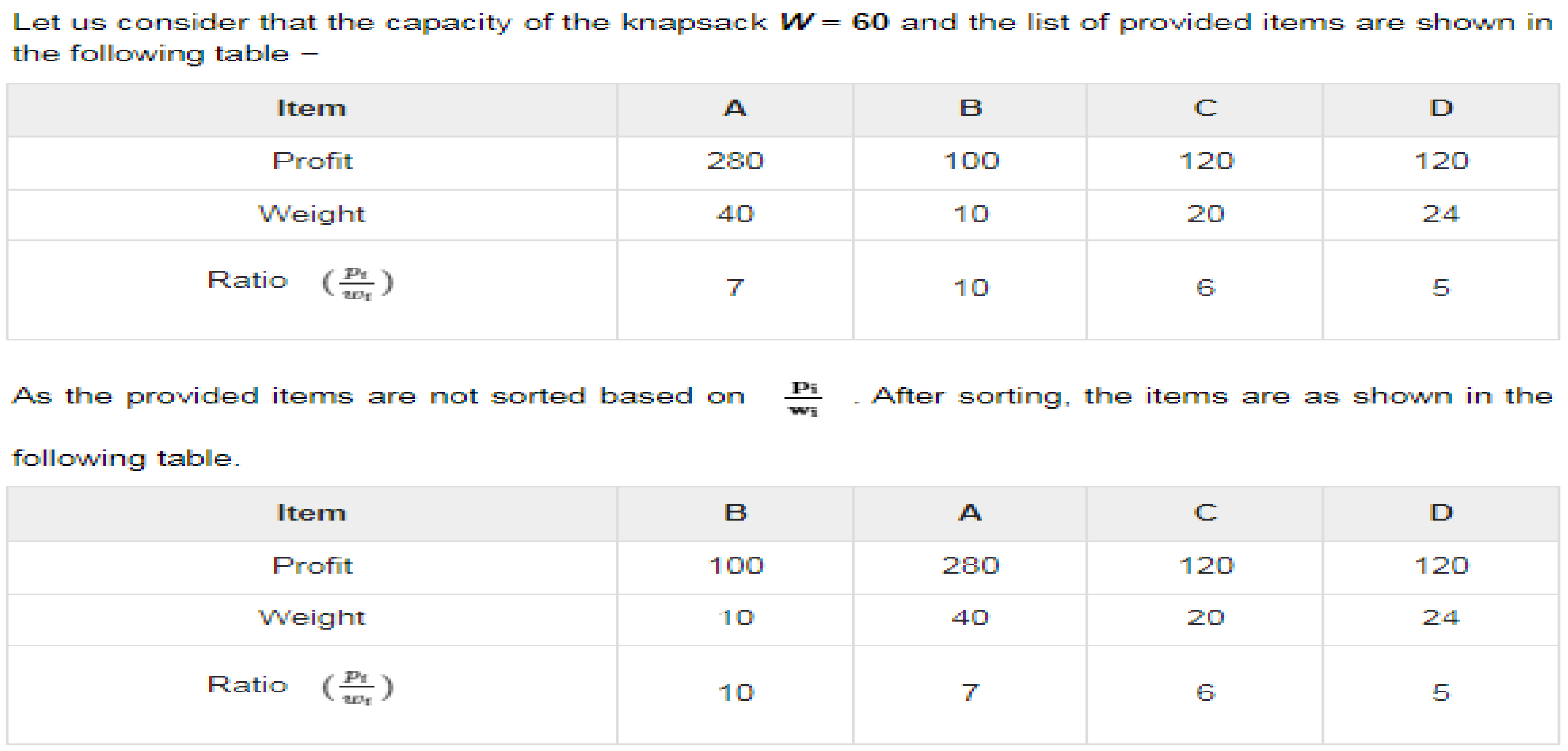


Object	Profit	Weight	Remaining weight
1	5	1	15 - 1 = 14
5	7	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	15 * 1/5 = 3	1	1 - 1 = 0

In this case, the total profit would be equal to (5 + 7 + 4 + 10 + 9 + 7 + 3) = 46

Fractional Knapsack Problem:

Example
Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −



Let us consider that the capacity of the knapsack *W* = 60 and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(\frac{p_i}{w_i})$	7	10	6	5

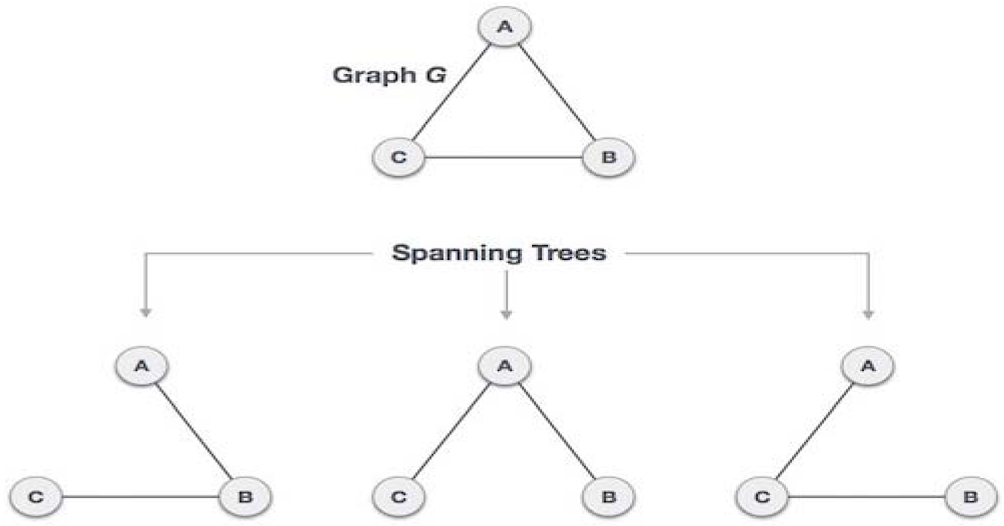
As the provided items are not sorted based on $\frac{p_i}{w_i}$. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(\frac{p_i}{w_i})$	10	7	6	5

Spanning tree

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

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



Spanning tree

We found three spanning trees off one complete graph. A complete undirected graph can have maximum **n^{n-2}** number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence **$3^{3-2} = 3$** spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

A connected graph G can have more than one spanning tree.

All possible spanning trees of graph G, have the same number of edges and vertices.

The spanning tree does not have any cycle (loops).

Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.

Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).

- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

- Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

Civil Network Planning
Computer Network Routing Protocol
Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST)

Minimum Spanning Tree (MST):

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

- In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

- Minimum Spanning-Tree Algorithm

- We shall learn about two most important spanning tree algorithms here −

Kruskal's Algorithm
Prim's Algorithm

Kruskal’s Minimum Spanning Tree Algorithm

What is Minimum Spanning Tree?

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees.

A ***minimum spanning tree (MST)*** or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

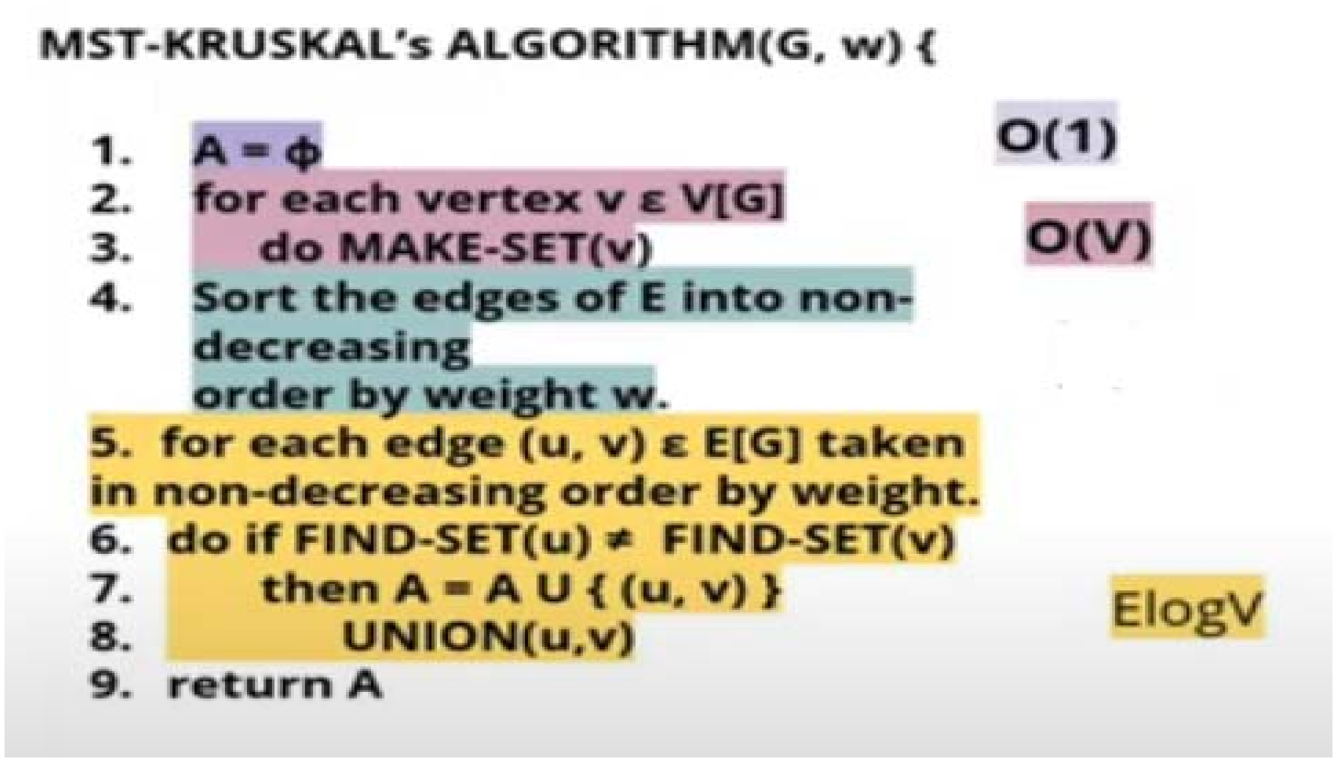
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Kruskal’s Minimum Spanning Tree Algorithm

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
Step 2: Create a set E that contains all the edges of the graph.
Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
Step 4: Remove an edge from E with minimum weight
Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(**for** combining two trees into one tree).
ELSE
Discard the edge
Step 6: END

Time complexity Analysis Kruskal’s Minimum Spanning Tree Algorithm



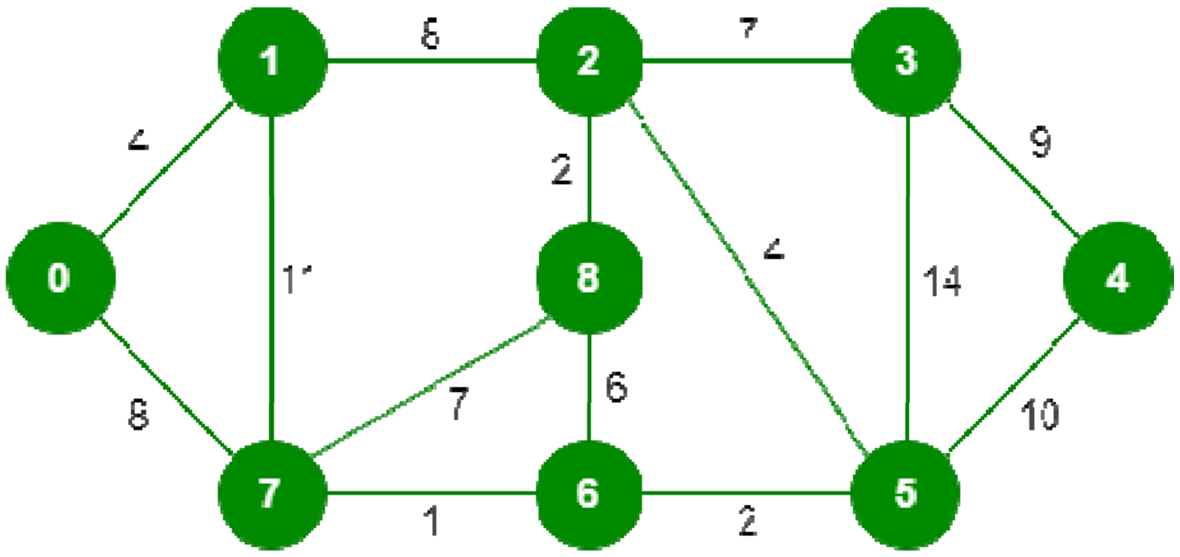
```
MST-KRUSKAL's ALGORITHM(G, w) {

1.   A = ϕ                                              O(1)
2.   for each vertex v ε V[G]
3.       do MAKE-SET(v)                                 O(V)
4.   Sort the edges of E into non-
     decreasing
     order by weight w.
5.  for each edge (u, v) ε E[G] taken
in non-decreasing order by weight.
6.  do if FIND-SET(u) ≠ FIND-SET(v)
7.        then A = A U { (u, v) }                       ElogV
8.             UNION(u,v)
9.  return A
```

Time Complexity

The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

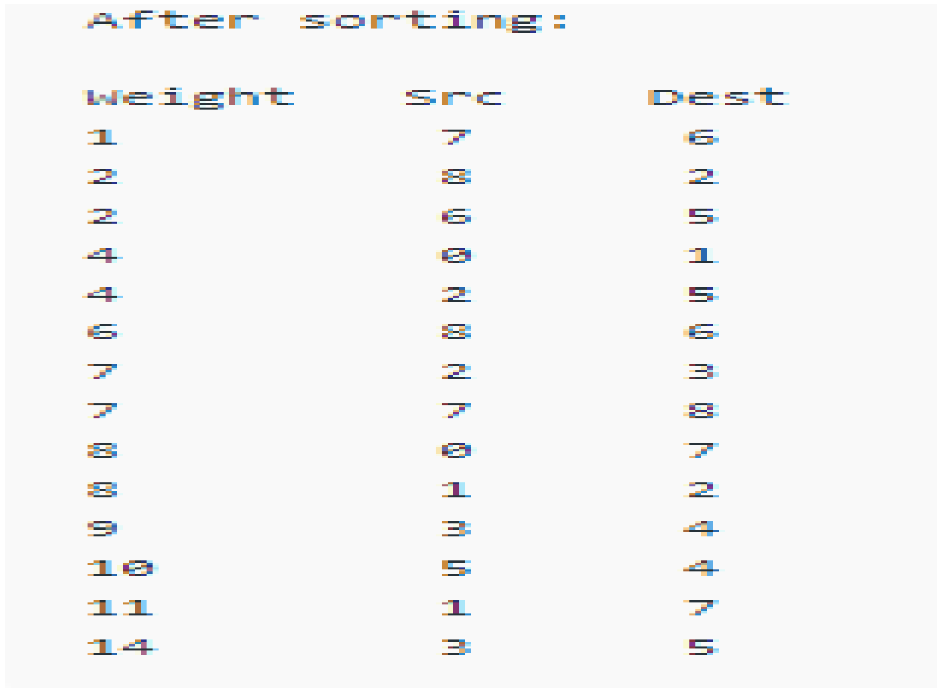
The steps for finding MST using Kruskal’s algorithm

1. *Sort all the edges in non-decreasing order of their weight.*
2. *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*
3. *Repeat step#2 until there are (V-1) edges in the spanning tree.*

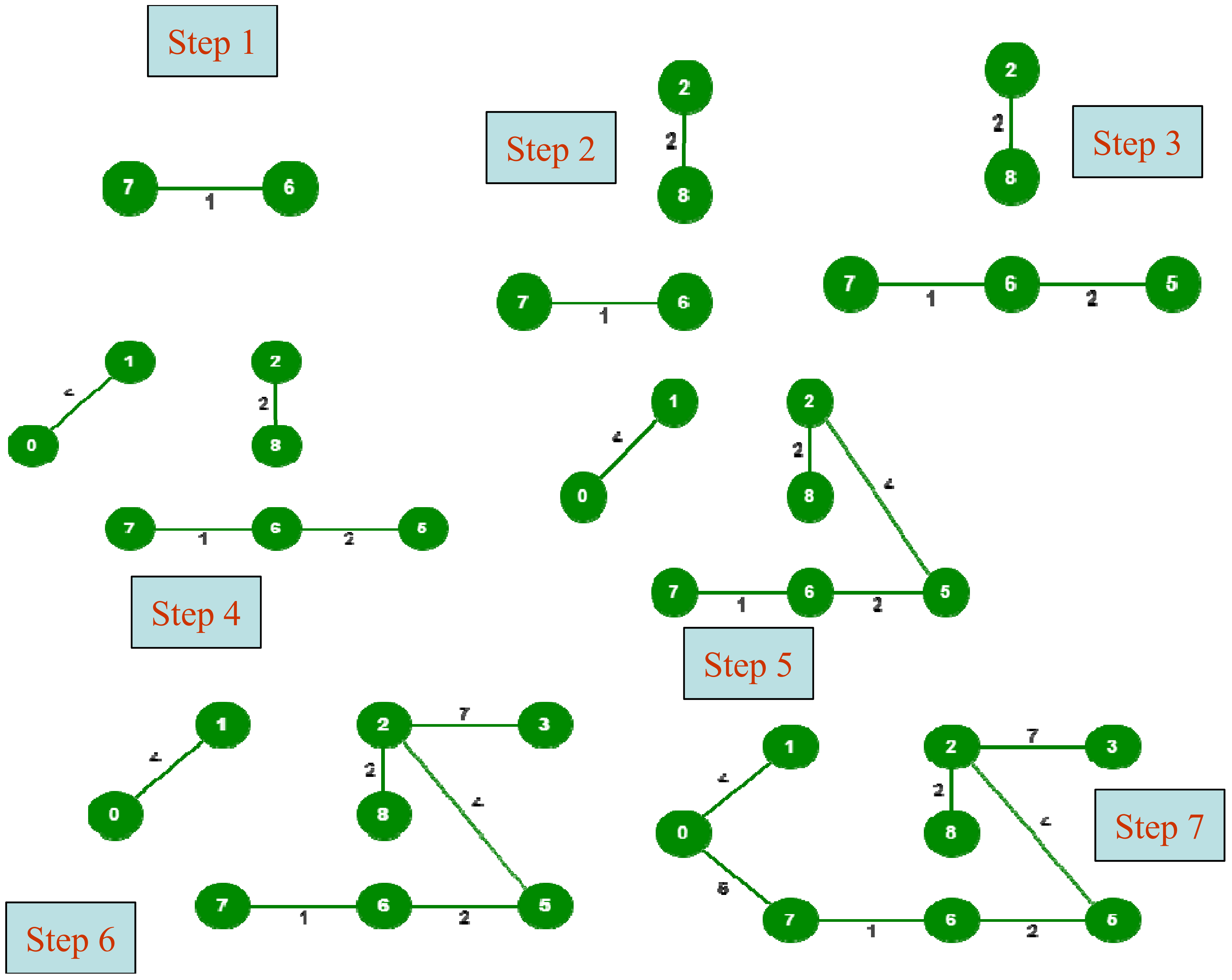


The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9–1) = 8 edges.



```
After sorting:

Weight      Src       Dest
1            7          6
2            8          2
2            6          5
4            0          1
4            2          5
6            8          6
7            2          3
7            7          8
8            0          7
8            1          2
9            3          4
10           5          4
11           1          7
14           3          5
```


Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7

How does Prim’s Algorithm Work?

➢Prim’s algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices.

➢The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.

➢The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected.

➢So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Algorithm

1) Create a set *mstSet* that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn't include all vertices
a) Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
b) Include *u* to mstSet.
c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*
The idea of using key values is to pick the minimum weight edge.

The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

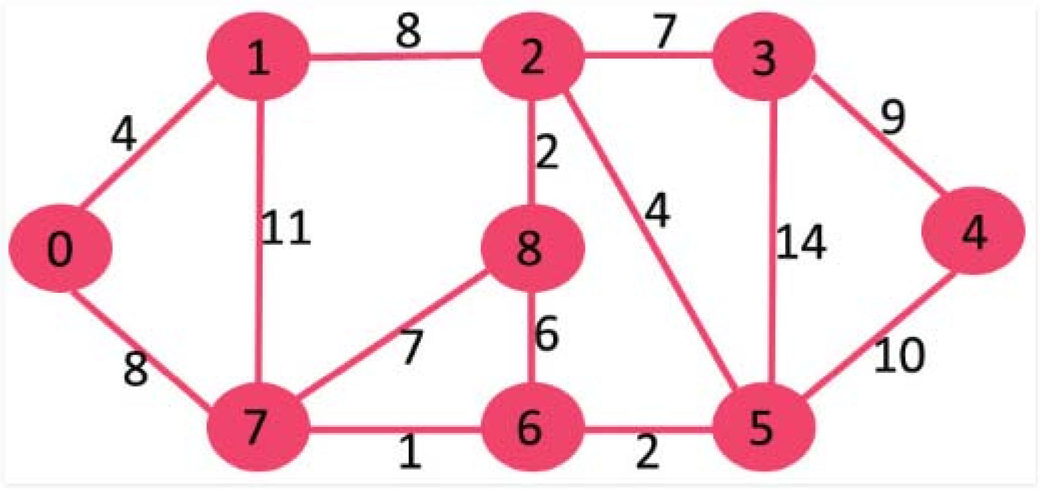
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF,INF} where INF indicates infinite.

Now pick the vertex with the minimum key value.

The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}.

After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7.

The key values of 1 and 7 are updated as 4 and 8

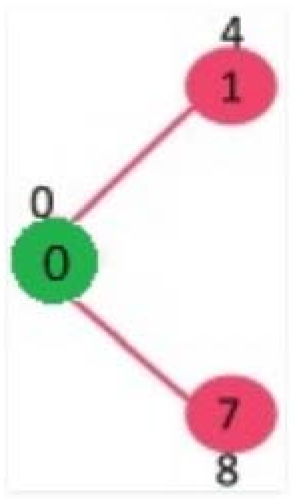


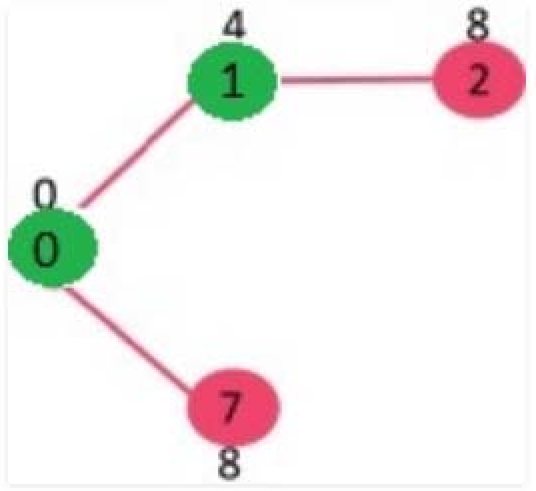
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet.

So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked.

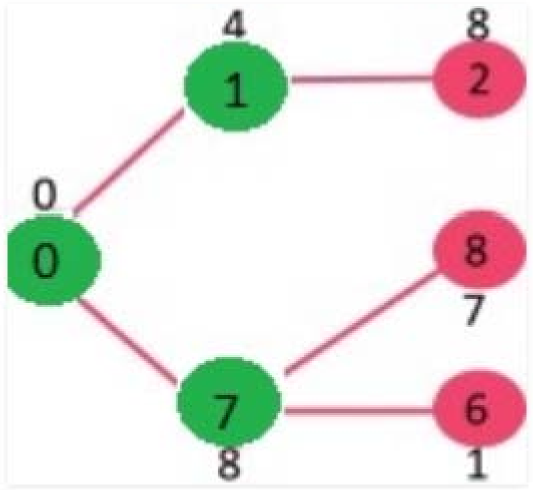
So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).

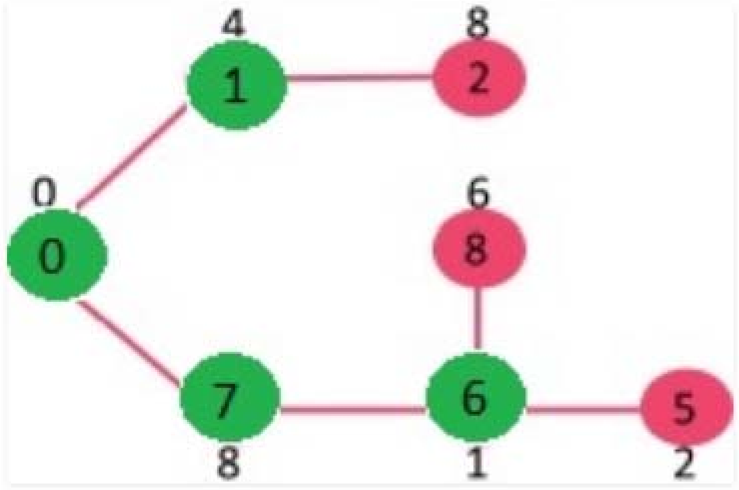


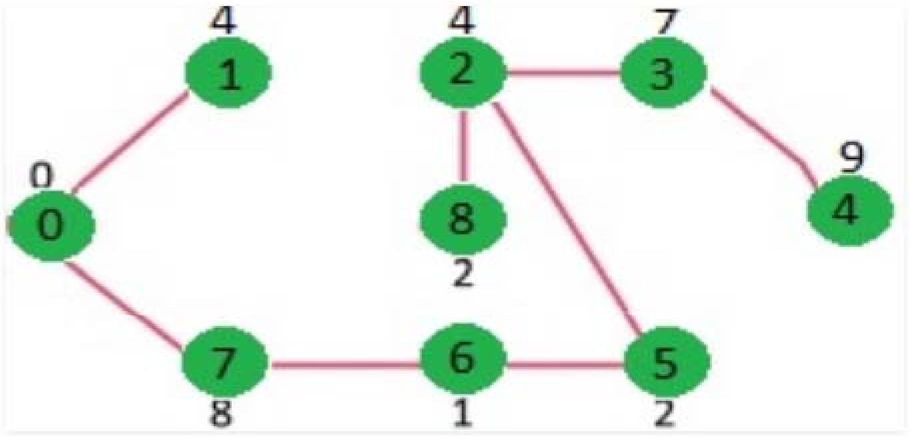


The vertices included in MST are shown in green color.

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.





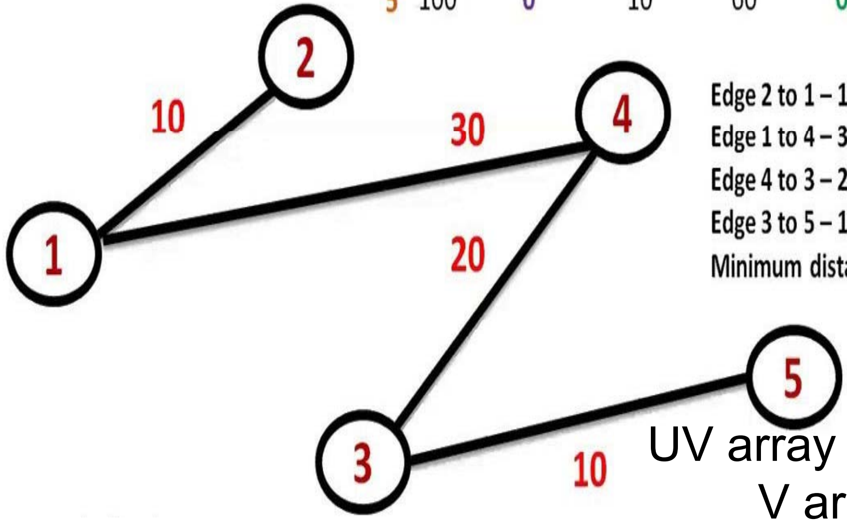


Prims Algorithm

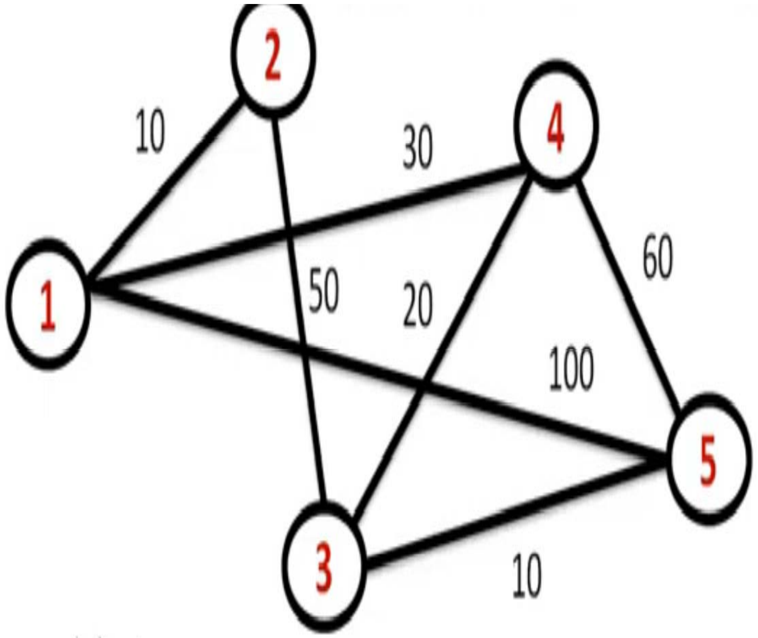
Minimum Spanning Tree with
Starting vertex - 2

Adjacency matrix

	1	2	3	4	5
1	0	10	0	30	100
2	10	0	50	0	0
3	0	50	0	20	10
4	30	0	20	0	60
5	100	0	10	60	0



Edge 2 to 1 – 10
Edge 1 to 4 – 30
Edge 4 to 3 – 20
Edge 3 to 5 – 10
Minimum distance – 70



UV array ={1,2,3,4,5}
V array= {}

Time Complexity using Adjacency list and Array O(v^2)
Time Complexity using Adjacency list and MinHeap O(ElogV)

Huffman Codes

(i) Data can be encoded efficiently using Huffman Codes.

(ii) It is a widely used and beneficial technique for compressing data.

(iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string. Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

	a	b	c	d	e	f	Total
Frequency	45	13	12	16	9	5	100

Huffman Codes

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For example:

a 000 ; b 001 ; c 010 ; d 011 ; e 100 ; f 101

For a file with 10^5 characters, we need 3×10^5 bits.

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

For example:

a 0 b 101 c 100 d 111 e 1101 f 1100 Number of bits = $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000$ **= 2.24×10^5bits**

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

.

Huffman Codes

Prefix Codes:

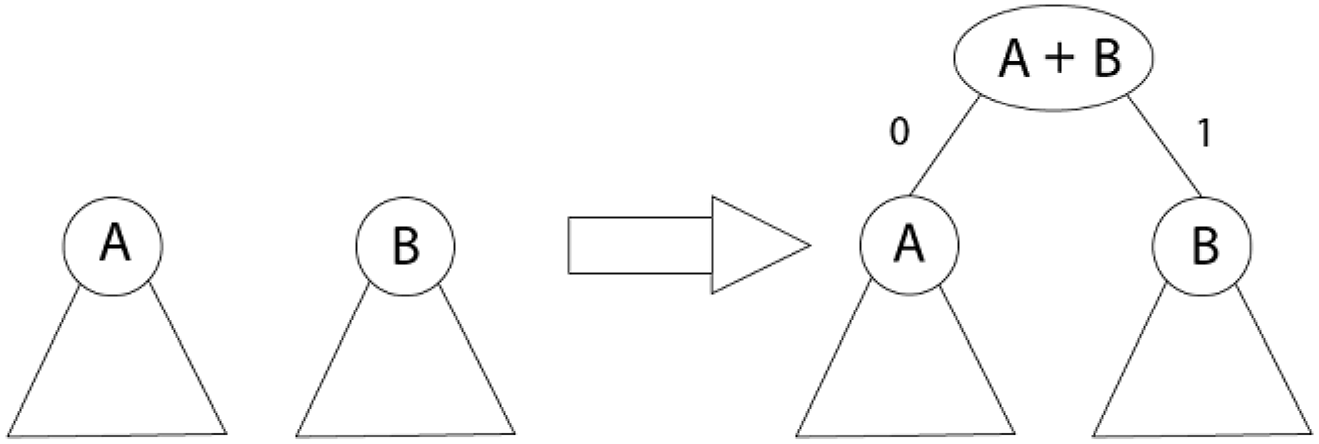
The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.
Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous

Huffman Codes

Greedy Algorithm for constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.

The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of |C| leaves (C is the number of characters) and performs |C| - 1 'merging' operations to create the final tree. In the Huffman algorithm 'n' denotes the quantity of a set of characters, z indicates the parent node, and x & y are the left & right child of z respectively.



Huffman Codes

Algorithm of Huffman Code

```
Huffman (C)
1. n=|C|
 2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x= left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)
```

Huffman Codes

Given that: $C = \{a, b, c, d, e\}$

$f(C) = \{50, 25, 15, 40, 75\}$

$n = 5$

$Q \leftarrow c$

i.e.

c	15	b	25	d	40	a	50	e	75

for $i \leftarrow 1$ to 4

$i = 1$ $Z \leftarrow$ Allocate node

$x \leftarrow$ Extract-Min (Q)

$y \leftarrow$ Extract-Min (Q)

c	15	b	25	d	40	a	50	e	75

Left [z] $\leftarrow x$

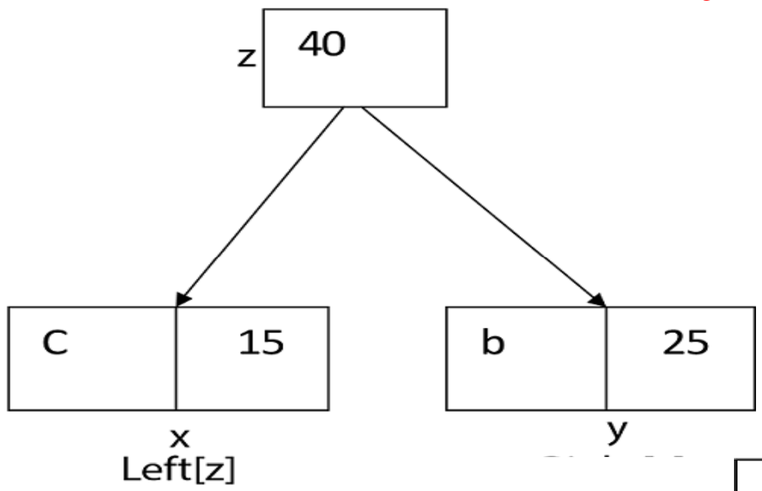
Right [z] $\leftarrow y$

$f(z) \leftarrow f(x) + f(y) = 15 + 25$

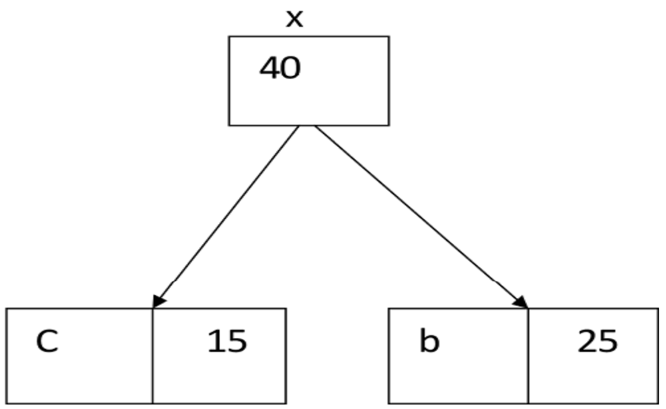
$f(z) = 40$

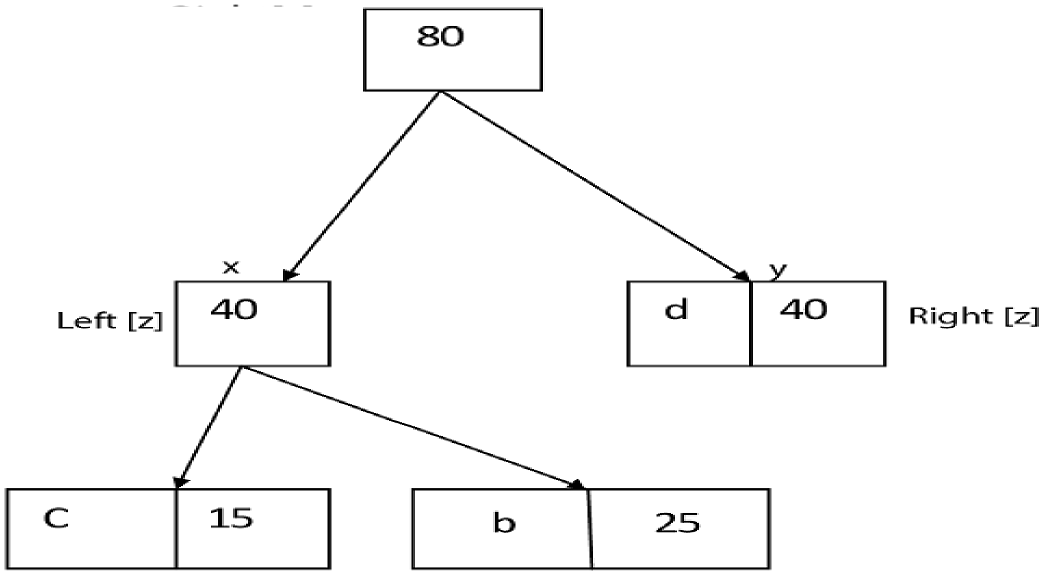
d	40	a	50	e	75

Huffman Codes

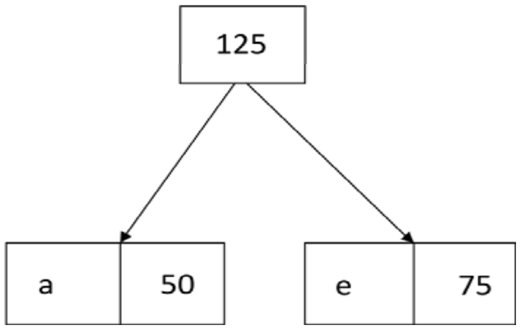


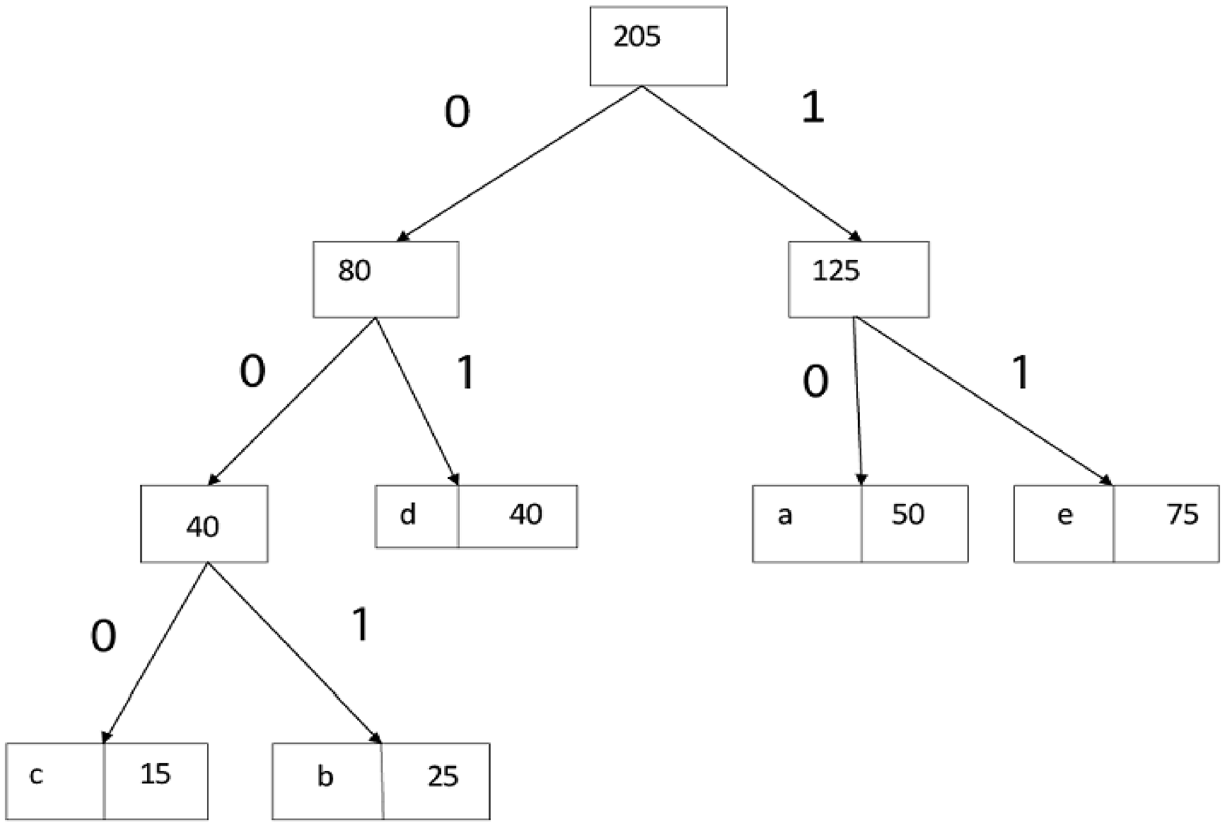
Again for i=2





Huffman Codes





Huffman Codes

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

There are mainly two major parts in Huffman Coding
Build a Huffman Tree from input characters.
Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree
Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

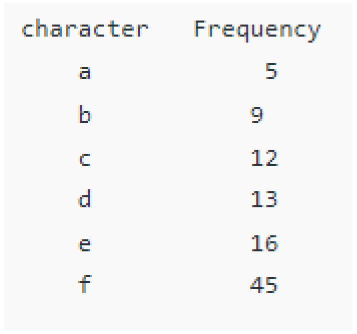
Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue.

The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

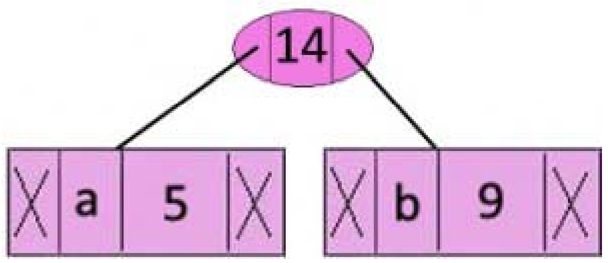
Extract two nodes with the minimum frequency from the min heap.

Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Huffman Codes



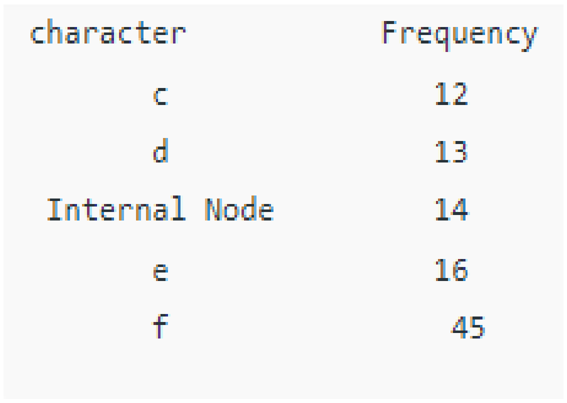
character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45



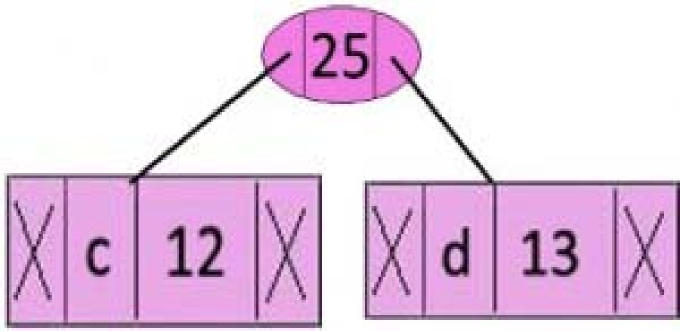
Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

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

Huffman Codes

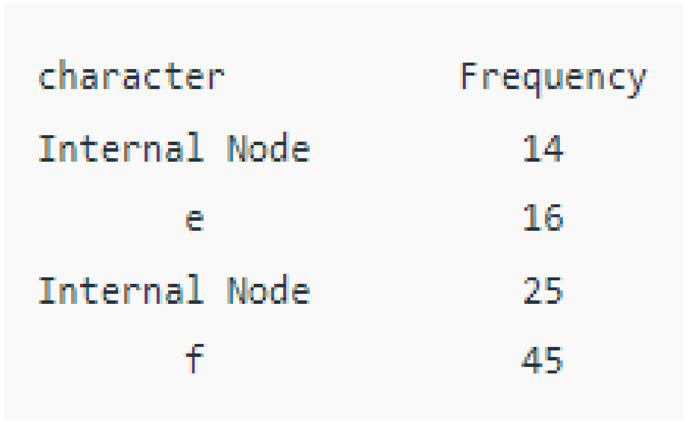


character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

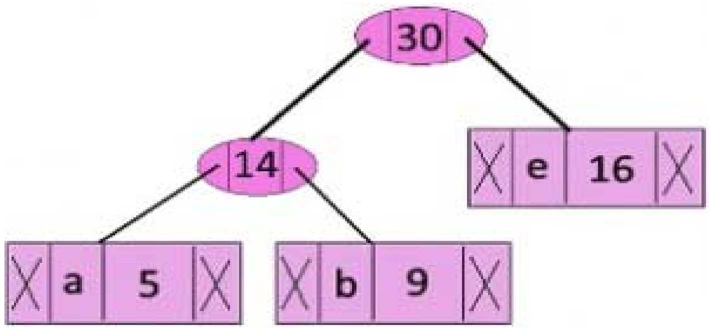


Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

Huffman Codes



character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

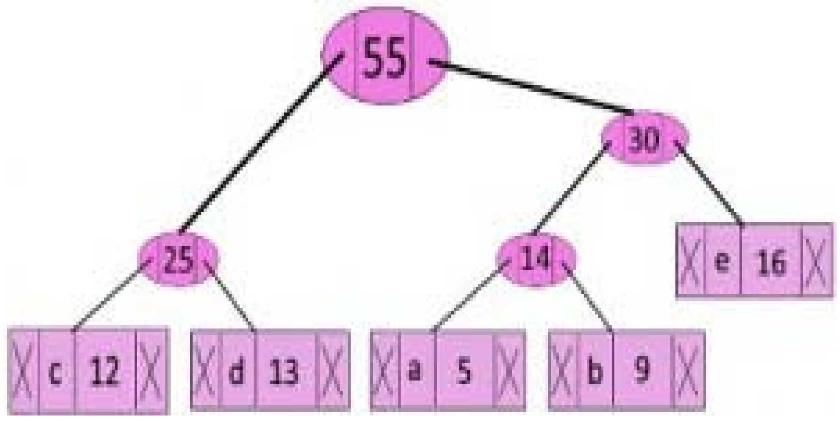


Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

Huffman Codes

Now min heap contains 3 nodes.

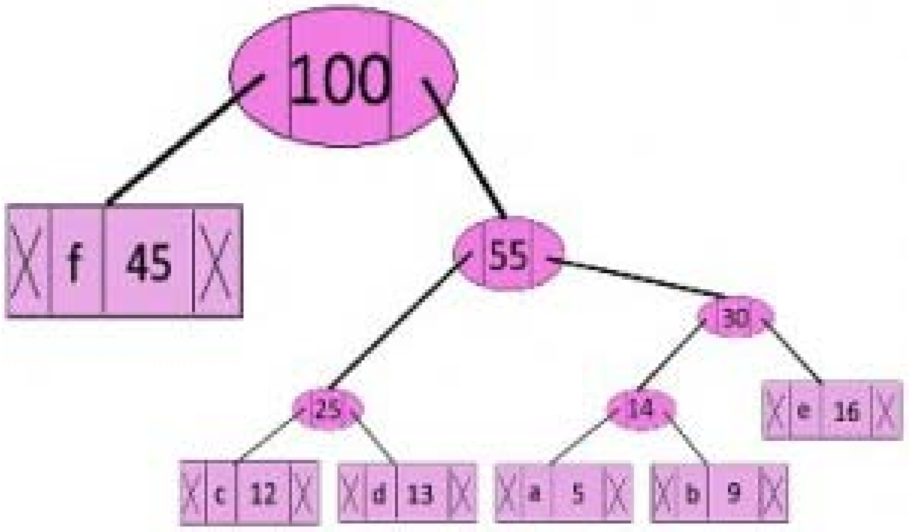
character	Frequency
Internal Node	25
Internal Node	30
f	45



Huffman Codes

Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55



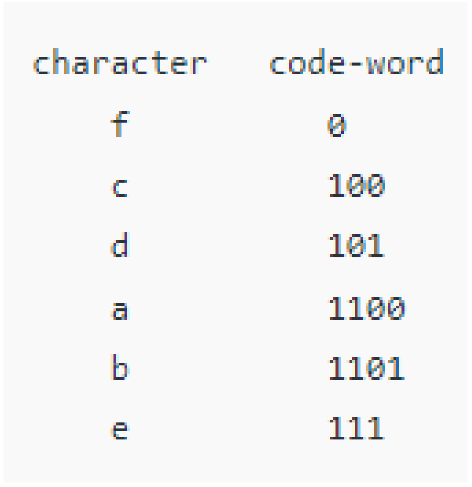
Huffman Codes

min heap contains only one node.

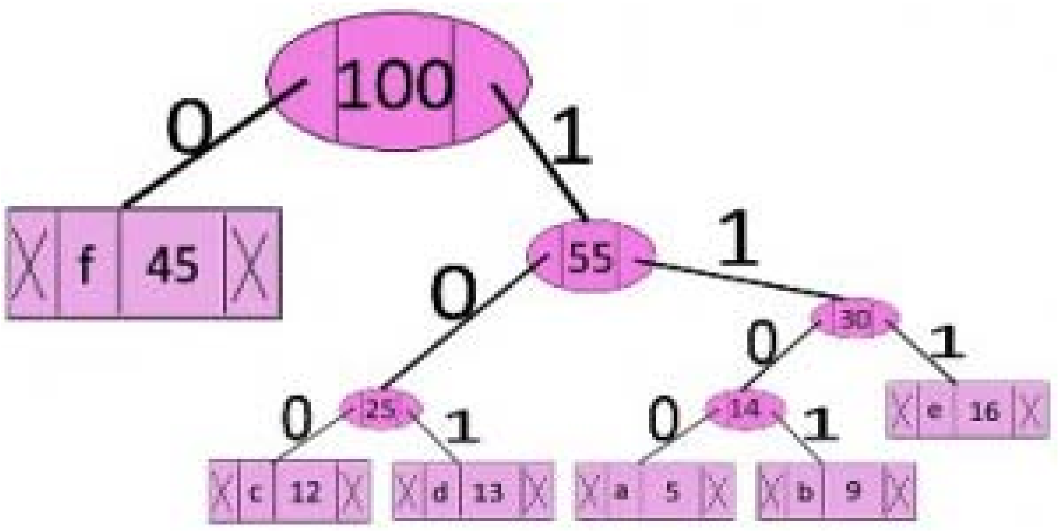
Character	Frequency
Internal Node	100

Steps to 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. Print the array when a leaf node is encountered.



character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111



Prefix Code: No Code is prefix of Another code

Huffman algorithm Time complexity

- The time complexity of the Huffman algorithm is **O(nlogn)**.
- Using a heap to store the weight of each tree, each iteration requires **O(logn)** time to determine the cheapest weight and insert the new weight.
- There are **O(n)** iterations, one for each item.

Hackerrank QR Code for 30/03/2022

UNIVERSITY OF ENGINEERING & MANAGEMENT, KOLKATA

Attendance QR Code for 30/03/2022

Attendance Based on IEMCRP

The link for students attendance : https://www.iemcrp.com/iemEn/ct304.jsp

Examination code is 2022000570 Section A
Examination code is 2022000574 Section B
Examination code is 2022000576 Section C
Examination code is 2022000582 Section D
Examination code is 2022000582 Section D
Examination code is 2022000588 Section IoT
Examination code is 2022000591 Section AIML
Examination code is 2022000593 Section CSBS

Thank You