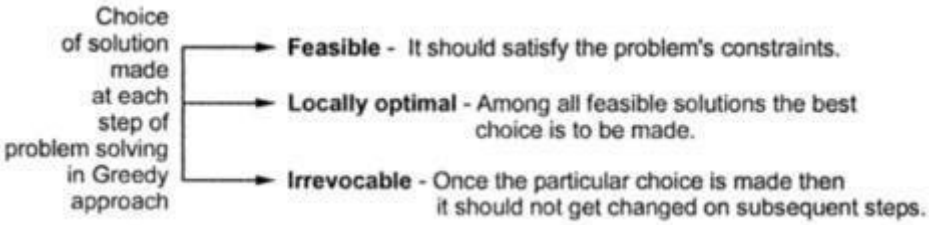
GREEDY METHOD

A Greedy algorithm is an approach to solving a problem that selects the most appropriate option based on the current situation. This algorithm ignores the fact that the current best result may not bring about the overall optimal result. Even if the initial decision was incorrect, the algorithm never reverses it.

This simple, intuitive algorithm can be applied to solve any optimization problem which requires the maximum or minimum optimum result. The best thing about this algorithm is that it is easy to understand and implement.

General method:

- In this method, the decision of solution is taken based on the information available.
- It is a straight forward method.
- It is used for obtaining optimizing solution.
- In greedy technique, the solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained, until a complete solution to the problem is reached,
- At each step the choice made should be



These activities are have to be performed

- First, select some solution from input domain.
- Then, check whether the solution is feasible or not.
- From the set of feasible solution, particular solution that satisfies or nearly satisfies the objective of the function. Such a solution is called optimal solution.
- As greedy method works in stages. At each stage, only one input is considered at each time. Based on this input, it is decided whether particular input gives the optimal solution or not.

Algorithm:

```
Greedy(D,n)

// in Greedy approach D is a domain

//from which solution is to be obtained of size n

//Initially assume

Solution=0;

For i=0 to n do

{

S= select(D)  //select solution from D
```

```
If( feasible(solution, s)) then // check if the selected solution is feasible or not

Solution=union (solution, S) ;//make feasible choices and select optimum solution

}
Return solution;
}
```

Some properties in greedy method:

- ✓ **Optimal substructure property:** The globally optimal solution to a problem includes the optimal sub solutions within it.
- ✓ **Greedy choice property:** This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

Note: Making locally optimal choices does not always work. Hence, Greedy algorithms will not always give the best solutions.

Applications of greedy method:

- Job Sequencing with deadlines
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in finding the shortest path.
- This algorithm is also used to solve the fractional knapsack problem.

Job Sequencing

Consider that there are n jobs that are to be executed. At any time t=1,2, 3,… only exactly one job is to be executed. The profits p_i are given. These profits are gained by corresponding jobs. For obtaining feasible solution we should take care that the jobs get completed within their given deadlines.

Let n=4

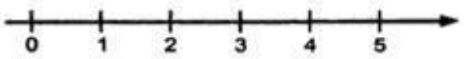
n	p_i	d_i
1	70	2
2	12	1
3	18	2
4	35	1

We will follow following rules to obtain feasible solution

- Each job takes one unit of time.
- If job starts before or at its deadline, profit is obtained, otherwise no profit.

- Goal is to schedule jobs to maximize the total profit.
- Consider all possible schedules and compute the minimum total time in the system

Consider the Time line as



The feasible solutions are obtained by various permutations and combinations of jobs.

n	p_i
1	70
2	12
3	18
4	35
1, 3	88
2, 1	82
2, 3	30
3, 1	88
4, 1	105
4, 3	53

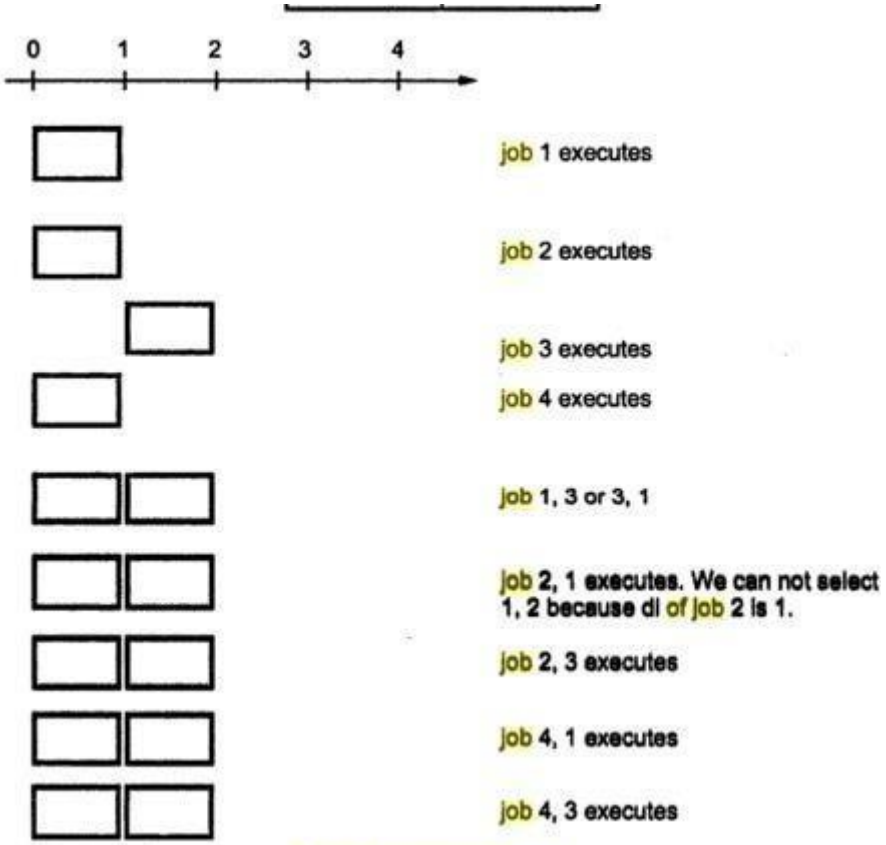


Fig. 3.1 Job sequencing with deadline

Each job takes only one unit of time. Deadline of job means a time on which or before which the job has to be executed. The sequence {2,4} is not allowed because both have deadline 1.

If job 2 is started at 0 it will be completed on 1 but we cannot start job 4 on 1 since deadline

of job 4 is 1.the feasible sequence is a sequence that allows all jobs in a sequence to be executed within their deadlines and highest profit can be gained.

The optimal solution is a feasible solution with maximum profit.

In above example sequence 3,2 is not considered as $d_3 > d_2$ but we have considered the sequence 2,3 as feasible solution because $d_2 < d_3$

We have chosen job 1 first then we have chosen job 4. The solution 4, 1 is feasible as the order of execution is 4 then 1. Also, $d_4 < d_1$. If we try {1,3,4} then it is not a feasible solution, hence reject 3 from the set. Similarly,if we add job 2 in the sequence then the sequence becomes {1,2,4}.This is also not a feasible solution hence reject it. Finally the feasible sequence is 4,1 . This sequence is optimum solution as well.

Algorithm:

```
Algorithm Job_seq(D,J,n)

{

//Problem description: This algorithm is for job sequencing using Greedy method.

//D[i] denotes i^th deadline where  1 <= I <= n

// J[i] denotes  i^th job

// D[J[i] <= D[J[i+1]]

Initially D[0]= 0; J[0] = 0; J[1] = 1;

Count =1 ;

For 2 to n do

{t = count;

While D[J[t] >D[1] AND D[J[t] != t)) do t= t-1;

If ((D(J[t] <= D(i) AND  (D(i) > t)) then

{

// insertion of i^th feasible sequence into J array

For  s←count to(t+1) step -1 do

J[s+1] ← J[s];

J[t=1] ← I;

Count ← count +1 ;

}// end of if;
```

} // end of while

Return count;

}

The sequence of J will be inserted if and only if D[J[t]] != t .This also means that the job J will be processede if it is in within the deadline.

The computing time taken by above Job_seq algorithm is $O(n^2)$,because the basic operation of computing sequence in array J is within two nested for loops.

Optimal Merge Pattern:

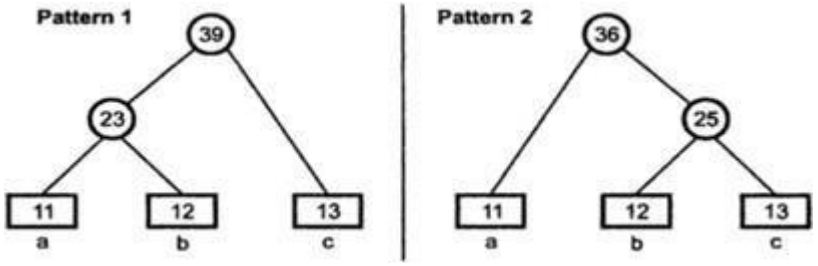
The merge sort is a technique in which two or more sorted files can be merged together in one sorted file. When two or more files need to be merged in a single file, then the merging of two sorted files must be performed repeatedly. That means merging of sorted files must be done pair-wise.

For instance :if there are a, b,c and d files which are already sorted and has to be merged then-

- We may merge a,b to produce x_1 then x_1 can be merged with file c to produce X_2 ,then X_2 can be merged with file d to produce X_3.
- We may merge a,b to produce x_1,then merge c and d to produce X_2 finally merge x_1 and X_2 to produce x_3

Thus, different pairings can be possible to produce final merged file. Clearly these pairing may require different amount of computing time. The pairing of files which require optimum (minimum) computing time to get a single merged file denotes the optimal merge pattern.

For example: consider that there are 3 files a,b,c of lengths 11,12 and 13 respectively. Then following are two sample merge patterns that can be produced.

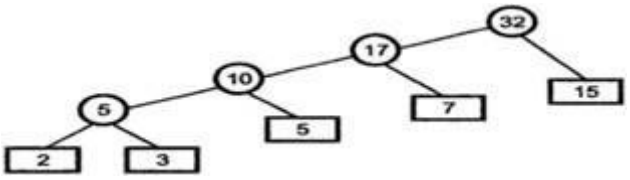


Pattern 2 is faster than first pattern. A Greedy method is used to obtain optimal merge pattern

Representation of Optimal Merge pattern:

The merge pattern is typically represented by a binary tree, it is also called as two-way merge pattern.

For example, 2,3,5,7,15 are six files with given lengths then we can build a merge pattern as



The leaf nodes denoted by squares called external nodes and remaining nodes are called internal nodes. The path length from root to corresponding external node is called external path length. An application of optimal merge pattern is Huffman code

Huffman Code:
Huffman coding is a data compression algorithm that is used to reduce the size of files or data streams without losing any information. It is used for lossless data compression, and was developed by David A. Huffman in 1952 while he was a graduate student at MIT.

Lossless compression is a method of data compression where the original data can be perfectly reconstructed from the compressed data. The primary goal of lossless compression is to reduce the size of the data without any loss of information. In lossless compression, the compressed data retains all the original data's details and quality.

The basic idea behind Huffman coding is to assign variable-length codes to different symbols in a given dataset, such as characters in a text document, based on their frequency of occurrence. The more frequently a symbol appears, the shorter its corresponding code will be. This allows for more efficient representation of commonly occurring symbols, resulting in overall compression of the data.
In Huffman‟s coding method, the data is inputted as a sequence of characters.Then a table of frequency of occurrence of each character in the data is built.

From the table of frequencies Huffman‟s tree is constructed using Greedy approach.

The Huffman‟s tree is further used for encoding each character, so that binary encoding is obtained for given data.

In Huffman coding there is a specific method of representing each symbol. This method produces a code in such a manner that no code word is prefix of some other code word. such codes are called prefix code or prefix-free codes. thus, this method is useful for obtaining optimal data compression.

Example: Obtain the Huffman‟s encoding for following data

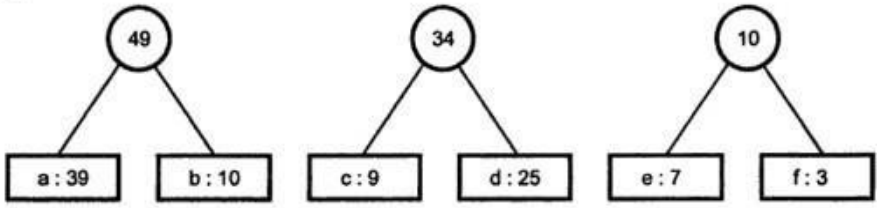
A:39 b:10 c:9 d:25 e:7 f:3

Solution: Basically, there are two types of coding-1. **Fixed length encoding** and 2. **Variable length coding**. If we use fixed length coding, we need 3 bits to represent any character from a to h.

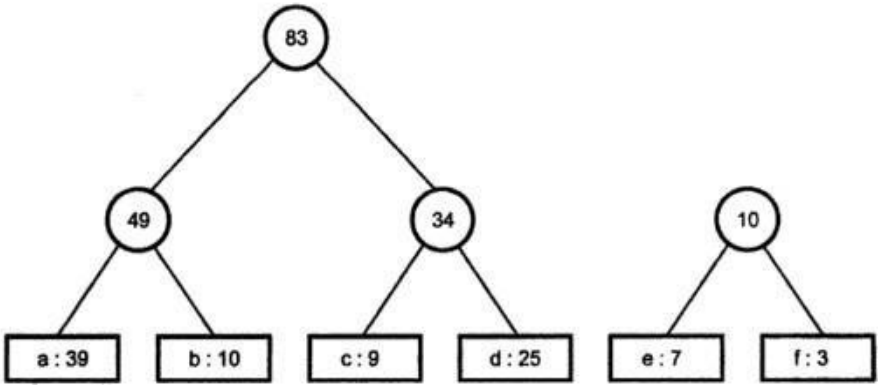
Let us arrange the symbols along with their frequencies as follows:

Step 1 :

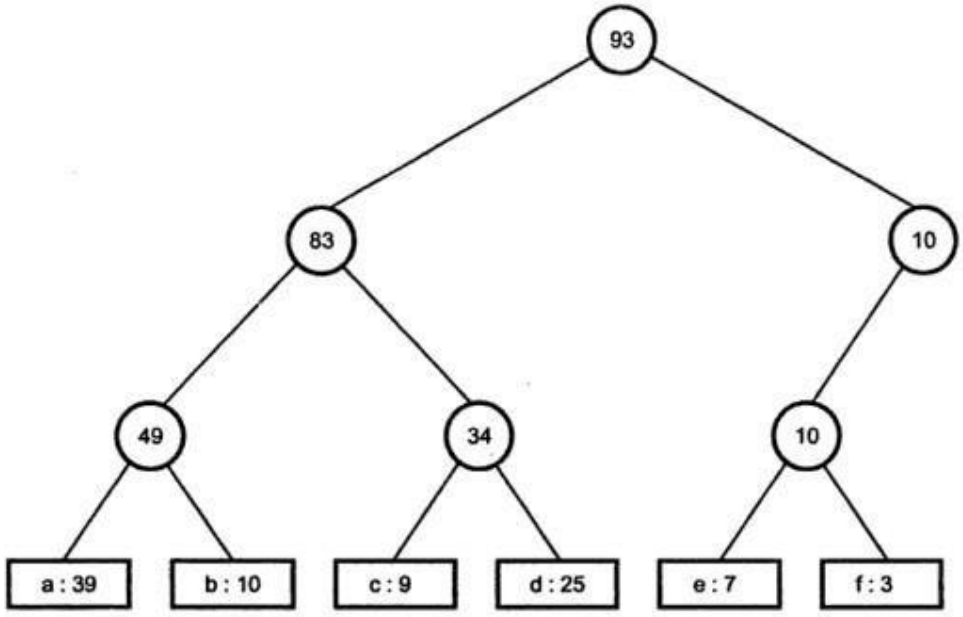
Step 2 :



Step 3 :

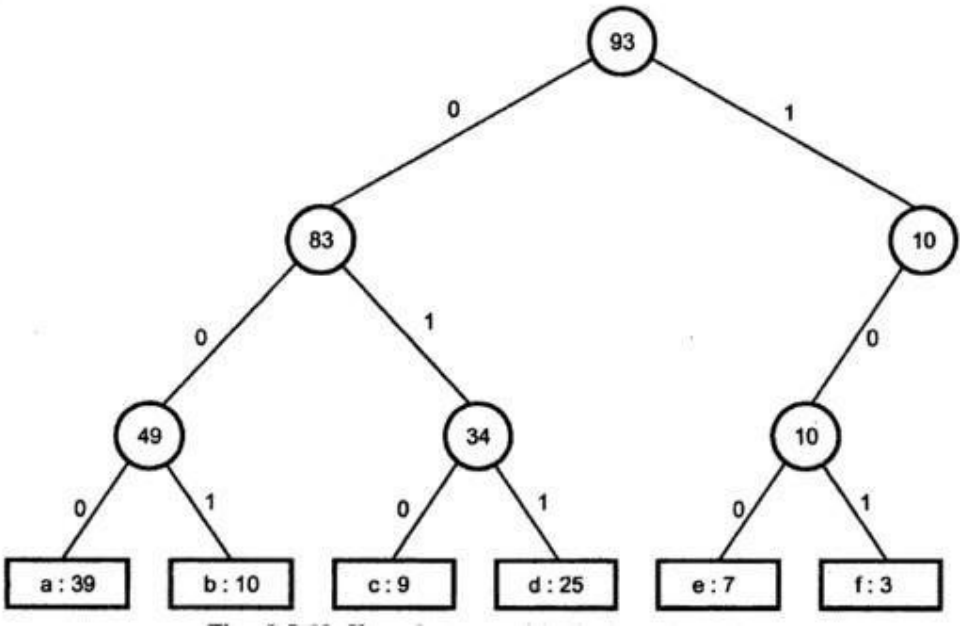


Step 4 :



Now we start encoding this tree. For that purpose we start coding the tree from top to down. If we follow the **left branch** then we should encode it as **"0"** and if we follow the **right branch** then we should encode it as **"1"**. Hence we get

Step 5 :



Step 6 :

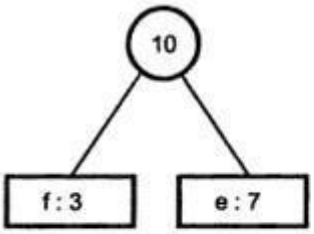
Symbol	Code word
a	000
b	001
c	010
d	011
e	100
f	101

Step 7:

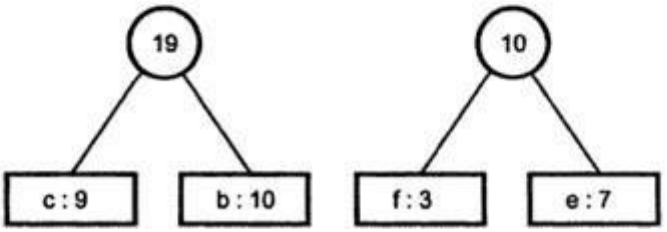
Hence if the string is “aeed” then we get 000100100011 as a code word. But this method is not that much efficient.

Hence variable length encoding has come up to bring efficiency in Huffman‟s algorithm. By this method, we assign short code word for frequently used characters and long code word for infrequent characters.

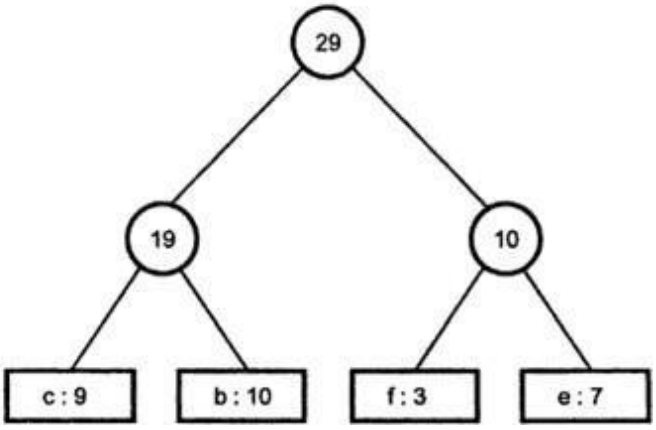
Step 1 :



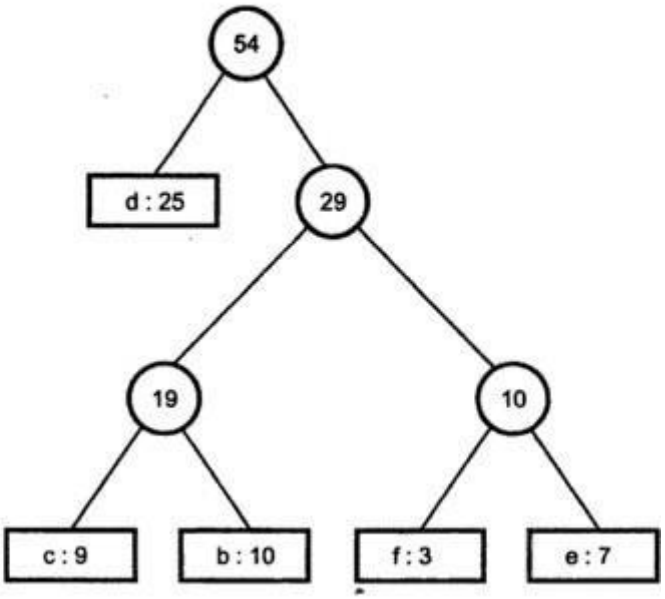
Step 2 :



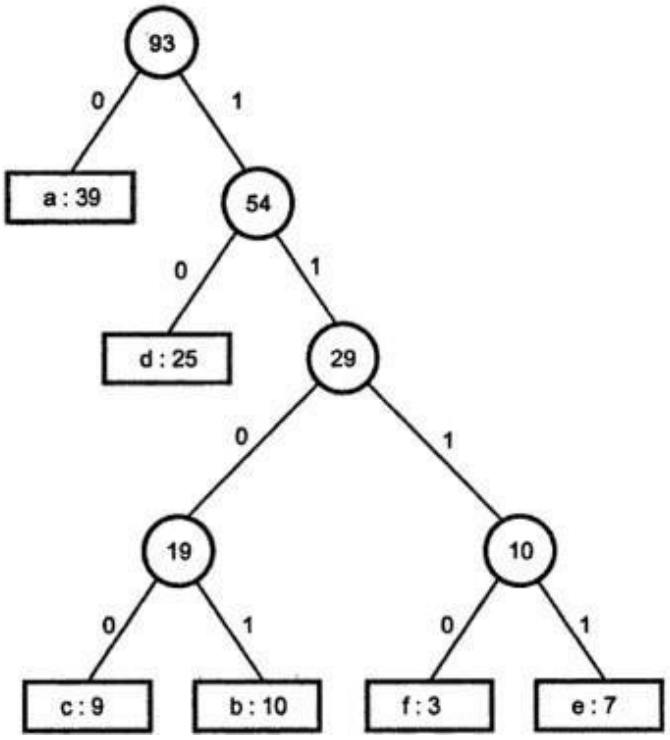
Step 3 :



Step 4 :


54
d : 25
29
19
10
c : 9
b : 10
f : 3
e : 7

Step 5 :


93
0
1
a : 39
54
0
1
d : 25
29
0
1
19
10
0
1
0
1
c : 9
b : 10
f : 3
e : 7

Step 6 :

Symbol	Code word
a	0
b	1101
c	1100
d	10
e	1111
f	1110

Step 7:

The encoding for the word "aeed" will be 01111111110.

The method of using variable length encoding is really efficient than fixed length encoding.

Consider,for each character number of bits required = frequency * number of bits used for representation.

i.e Total bits required in fixed length encoding

= 39 * 3 +10 * 3+ 9*3 +25 *3 +7 *3 + 3*3 = 279 bits

Similarly, Total bits required in Variable length encoding

= 39 * 1 +25 *2 +9 *4 +10*4 +7 *4 + 3*4 = 205 bits

Clearly Variable length encoding requires less number of bits than fixed length encoding.

Huffman's Algorithm: The Greedy method is used to construct optimal prefix code called Huffman code. This algorithm builds a tree in bottom up manner. We can denote this tree byT.

Let |c| be number of leaves.
| c | -1 are number of operations required to merge the nodes.
Q be the priority queue which can be used while constructing binary heap
Algorithm Huffman (c)
{
n = | c |
Q = c
For i ← 1 to n-1
Do
{
Temp ← get_node ()
Left(temp) = Get_min (Q)
Right (temp) = Get_min (Q)

a = left(temp)
b=right(temp)
f(temp) ← f(a) + f(b)
insert (Q,temp)
}
Return Get_min (Q)

Analysis:
The Priority queue Q can be implemented as binary Heap. The construction of binary heap can be done in O(n) time. The for loop will be executed in O(log n) time. Hence for (n-1) nodes O(n log n) time will be required. Therefore total running time of Huffman‟s algorithmwill be O(log n)

Graphs:

Definition of Graph:

A graph is a collection of two sets V and E where V is a finite non empty set of vertices and E is a finite non empty set of edges.

Vertices are nothing but eh nodes in the graph and the two adjacent vertices are joined by edges. The graph is thus a set of two sets.

Any graph G is denoted by G= {V,E}

Type of Graphs:

Basically there are two graphs:

Directed Graphs

Undirected Graphs

In the directed graph the directions are shown on the edges.The edges between the vertices are ordered.in this type of graph,the edge E1 is in between the vertices V1 and V2.the V1 is called head and the V2 is called the tail. similarly for V1 head , the tail is V3 and so on.

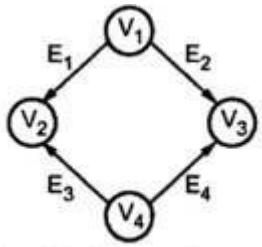


Fig. 3.3 Directed graph

We can say E1 is the set of (V1,V2)and not of (V2,V1).similarly,in an undirected graph,the edges are not ordered.In this type ,the edge E1 is set of (V1,V2) or (V2,V1).

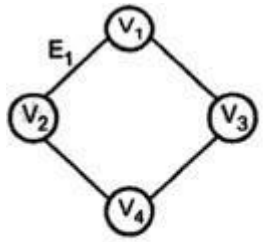


Fig. 3.4 Undirected graph

Minimum Spanning tree:

Spanning tree:

A spanning tree of a graph G is a subgraph which is basically a tree and it contains all the vertices of G containing no circuit.

Minimum Spanning tree:

A minimum spanning tree of a weighted connected graph G is a spanning tree with minimum or smallest weight.

Weight of the tree:

A weight of the tree is defined as the sum of weights of all its edges.

For example: consider a weighted connected graph G because some weights are given along every edge and the graph is a connected graph.

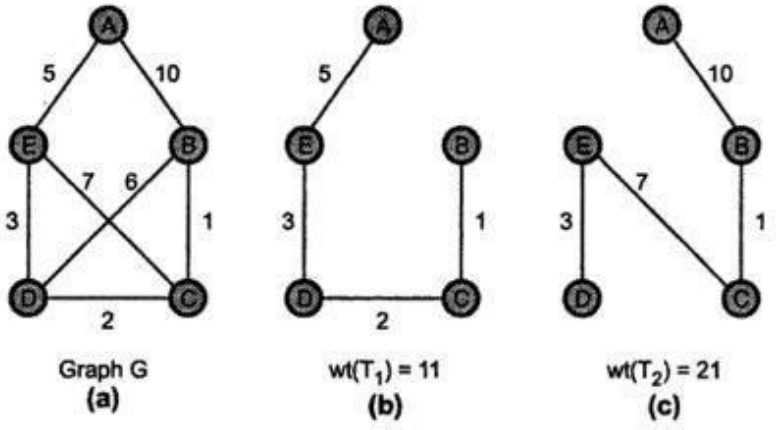


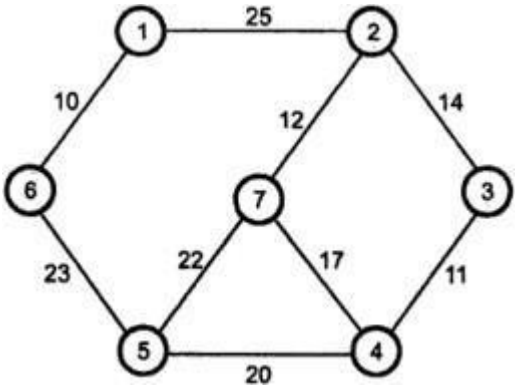
Fig. 3.12 Graph and two spanning trees out of which (b) is a minimum spanning tree

Application of Spanning trees:

1.spanning trees are very important in designing efficient routing algorithms

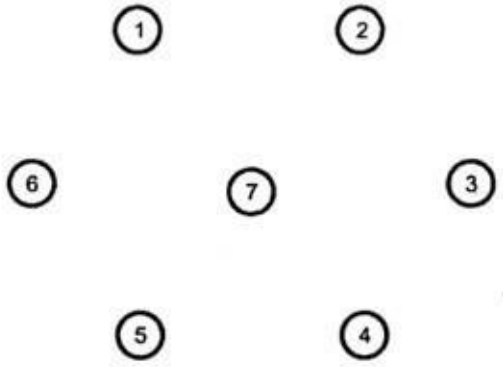
2.Spanning trees have wide applications in many areas such as network design.

Example of Prim"s algorithm: consider the below graph:



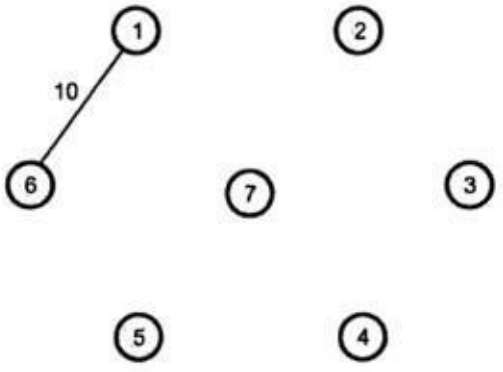
Now, we will consider all the vertices first. Then we will select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight. Care should be taken for not forming circuit

Step 1 :



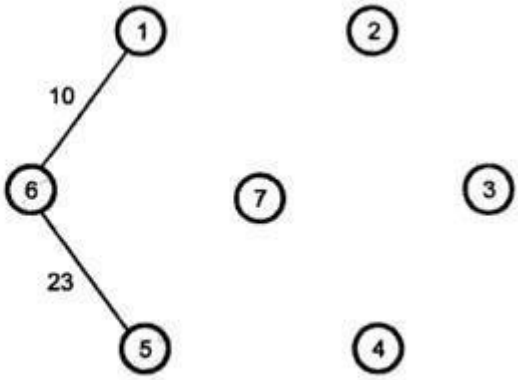
Total weight = 0

Step 2 :



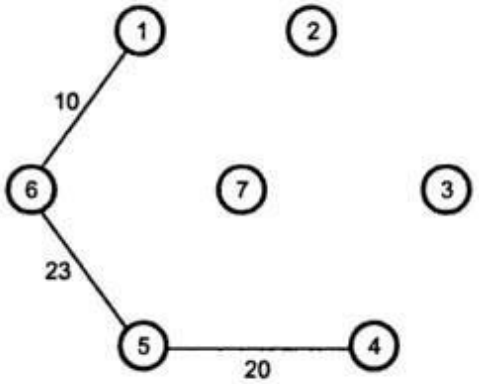
Total weight = 10

Step 3 :



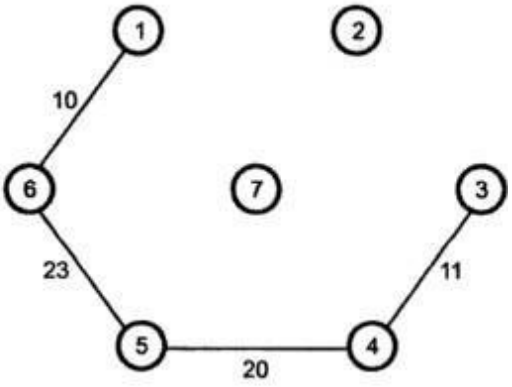
Total weight = 33

Step 4 :



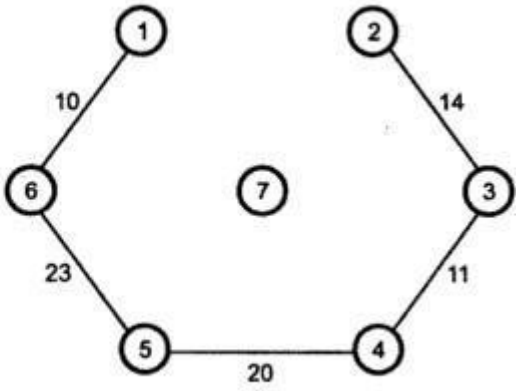
Total weight = 53

Step 5 :



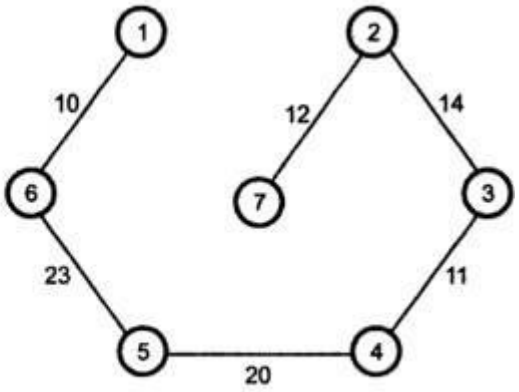
Total weight = 64

Step 6 :



Total weight = 78

Step 7 :



Total weight = 90

Algorithm:

Prim (G[0,….size-1,0….size-1],nodes)

//Problem Description: This algorithm is for implementing Prim‟s algorithm for finding the spanning tree

//Input: Weighted Graph G and total number of nodes

//Output: spanning tree gets printed with total path length

Total = 0;

//Initialize the selected vertices list

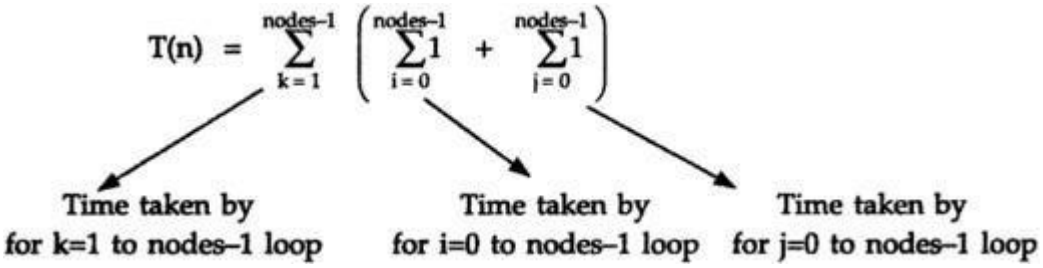
For i ←o to nodes -1 do

Tree[i] ← 0

Tree[0] =1 //take initial vertex

```
For K← 1 to nodes  do

{

Min_dist ← ∞

// initially assign minimum dist as infinity

For i← 0 to nodes-1 do

{

For j← 0 to nodes-1 do (

If(G[i,j] AND ((tree[i] AND !tree[j]) OR

(! Tree[i] AND tree[j])))) then

{

If (G[i,j] < Min_dist then

//select an edge such that one vertex is selected and the other is not and the edge has the least weight

{

Min_dist ← G[i,j] // Obtained edge with minimum wt.

V1← i;

V2 ← j; // Picking up those vertices yielding minimum edge

}}}}

Write (V1,V2,Min_dist);

Tree[V1] ← tree{V2] ← 1

Total ← total +min_dist

}

Write(“Total path length is “,Total)
```

Analysis: The algorithm spends most of the time in selecting the edge with minimum length. Hence the basic operation of this algorithm is to find the edge with minimum path length. This can be given by following formula



We take variable n for „nodes" for the sake of simplicity of solving the equation then

$T(n) = \sum_{k=1}^{n-1}(\sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} 1$

$=\sum_{k=1}^{n-1}[((n-1)+0+1)+((n-1)+0+1)]$

$=\sum_{k=1}^{n-1} 2n$

$=2n \sum_{k=1}^{n-1} 1$

$= 2n ((n-1) -1 +1) = 2n(n-1)$

$= 2n^2 – 2n$

$T(n) = n^2$

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

N= no.of. vertices or nodes of the tree

The time complexity of the Prim"s algorithm $T(n) = \Theta (n^2)$

Kruskal's Algorithm:

It is an another algorithm for obtaining minimum spanning tree.n this algorithm always the minimum cost edge has to be selected.But it is not necessary that selected optimum edge is adjacent.

Difference between Prim's and Kruskal's Algorithm:

Prim"s Algorithm	Kruskal"s Algorithm
This algorithm is for obtaining the minimum spanning tree by selecting the adjacent vertices of already selected vertices	This algorithm is for obtaining the minimum spanning tree but it is not necessary to choose adjacent vertices of already selected vertices

Example:

Consider the graph given below:

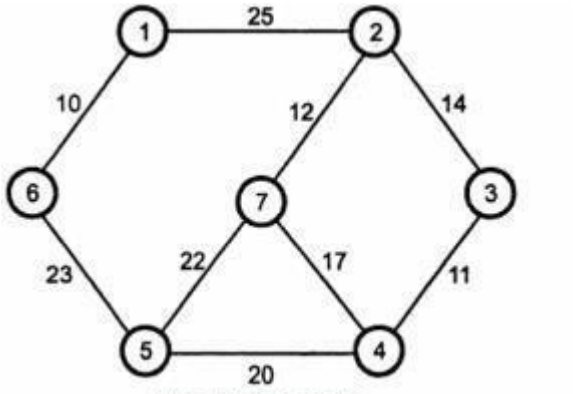
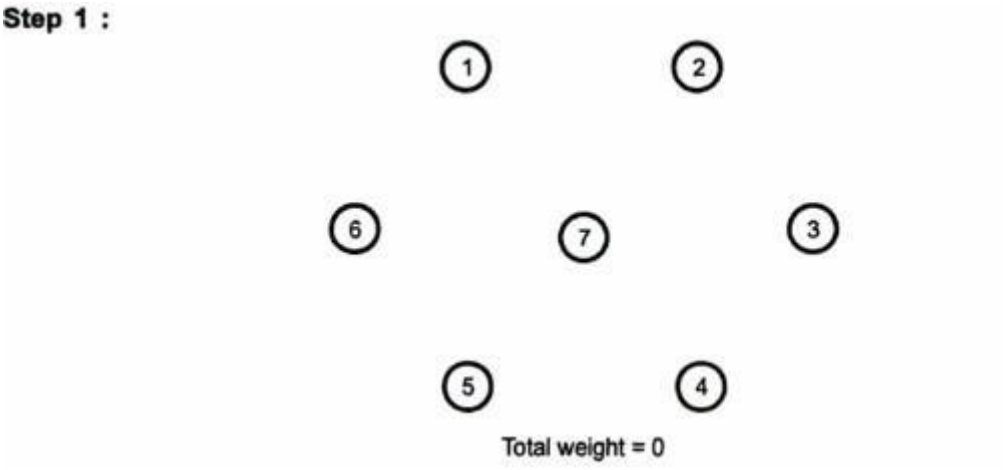


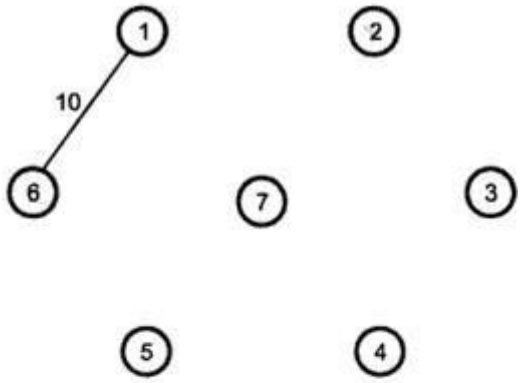
Fig. 3.15 Graph

First , we will select all the vertices .Then an edge with optimum weight is selected from heap,even though it is not adjacent to previously seelcted edge.Care should be taken for not forming circuit.

Step 1 :

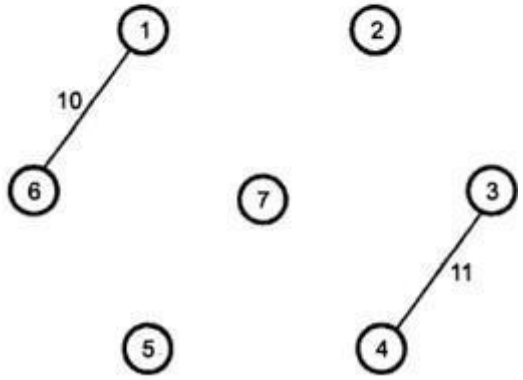


Step 2 :



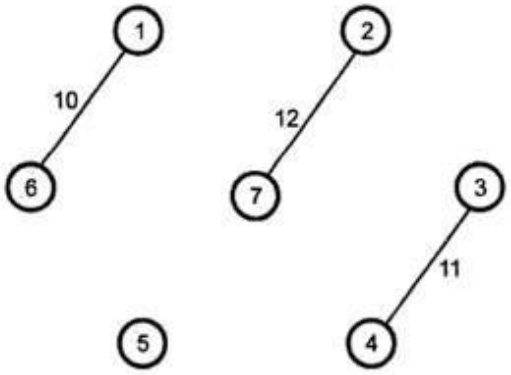
Total weight = 10

Step 3 :



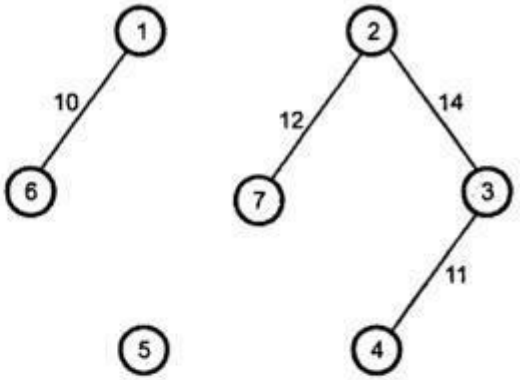
Total weight = 21

Step 4 :



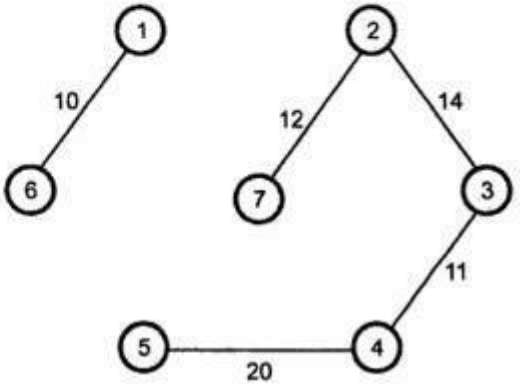
Total weight = 33

Step 5 :



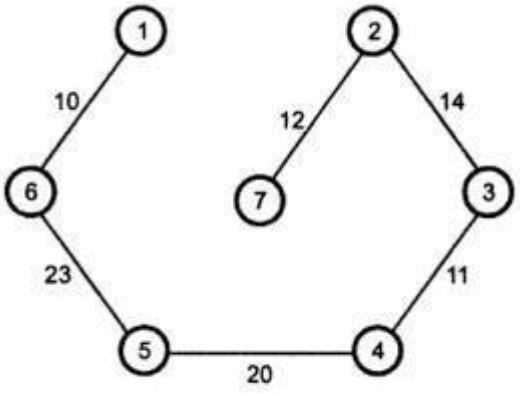
Total weight = 47

Step 6 :



Total weight = 67

Step 7 :



Total weight = 90

Algorithm: spanning _tree()
//Problem Description:This algorithm finds the minimum spanning tree using Kruskal‟s Algorithm
// Input:The adjacency matrix graph G containing cost
//Output:Prints the spanning tree with the total cost of spanning tree\
Count← 0
K ←0

```
Sum ← 0
For i← 0 to tot_nodes do
parents[i] ← i
while ( count != tot_nodes -1 ) do
{
Post _Minimum (tot_edges); // finding the minimum cost edge
If(pos = -1 then  //Perhaps no node in the graph
Break
V1 ← G[pos].V1
V2 ← G[pos].V2
i  ←find(v1,parent)
j ←Find(v2,parent)
if( i !=j) then
{
Tree[k][0] ← V1 //Tree[] is an array in which the spanning tree edges are stored.
Tree[k] [1] ← v2
K++
Count ++;
Sum + ← G[pos].cost //computing the total cost of all the minimum distances
Union( I,j,parent);
}
G[pos].cost INFINITY
}
If(count = tot_nodes -1 ) then
{
For i ← 0 to tot_nodes -1
{
Write (tree[i][0],tree[i] [1] //for each node of I, the minimum distnace edges are collected in
the array tree[][].The spanning treeis printed here
}
Write (“cost of spanning tree is”,sum)
}
```

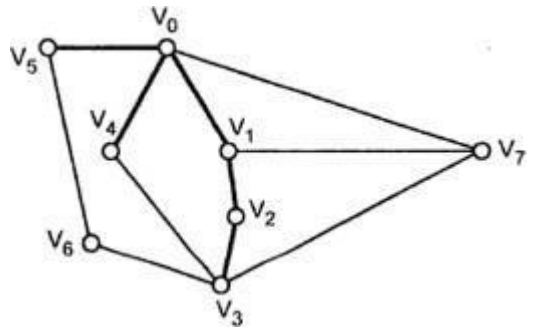
Analysis: The efficiency of Kruskal‟s algorithm is Θ(|E| log |E|) where E is the edges in thegraph.

Dijkstra’s Algorithm:

It is a popular algorithm for finding the shortest path. This is also called the single source shortest path algorithm. In this algorithm, for a given vertex called source the shortest path to all other vertices is obtained and the main focus is not to find only one single path but to find the shortest paths from any vertex to all other remaining vertices.

This algorithm applicable to graphs with non-negative weight only.

This algorithm finds a shortest path to graph‟s vertices in order of their distance from a given source. In this process of finding shortest path, first it finds the shortest path from the source toa vertex nearest to it, then second nearest and so on.



The shortest path from V0 is obtained. First, we find shortest path from V0-V1 then V1-V2 then from V2-V3 the shortest distance is obtained. Consider a weighted connected graph below:

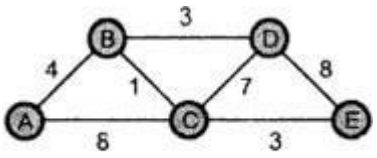
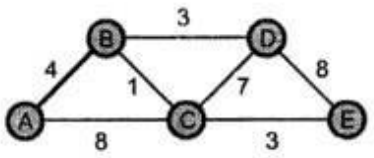
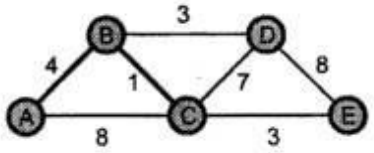
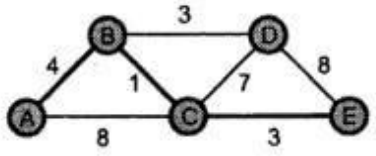


Fig. 10.11

Now, we will consider each vertex as a source and will find the shortest distance from this vertex to every other remaining vertex. Let us start with vertex A

Source vertex	Distance with other vertices	Path shown in graph
A	A-B, path = 4 A-C, path = 8 A-D, path = ∞ A-E path = ∞	
B	B-C, path = 4 + 1 B-D, path = 4 + 3 B-E path = ∞	
C	C-D, path = 5 + 7 = 12 C-E, path = 5 + 3 = 8	
D	D-E, path = 7 + 8 = 15	

But we have one shortest distance obtained from A to E and that is A-B-C-E with path length =4+1+3 =8. Similarly other shortest paths can be obtained by choosing appropriate source and destination.

Path[v2] ← v1

}}}}} // V1 is next selected destination vertex with shortest distance .All such vertices are accumulated in array path []

Knapsack Algorithm

The weights (Wi) and profit values (Pi) of the items to be added in the knapsack are taken as an input for the fractional knapsack algorithm and the subset of the items added in the knapsack without exceeding the limit and with maximum profit is achieved as the output.

Algorithm

- Consider all the items with their weights and profits mentioned respectively.
- Calculate P_i/W_i of all the items and sort the items in descending order based on their P_i/W_i values.
- Without exceeding the limit, add the items into the knapsack.
- If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
- Hence, giving it the name fractional knapsack problem.

Examples

- For the given set of items and the knapsack capacity of 10 kg, find the subset of the items to be added in the knapsack such that the profit is maximum.

Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	12	8

Solution

Step 1

Given, n = 5

W_i = {3, 3, 2, 5, 1}
P_i = {10, 15, 10, 12, 8}

Calculate P_i/W_i for all the items

Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	20	8
P_i/W_i	3.3	5	5	4	8

Step 2

Arrange all the items in descending order based on P_i/W_i

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
P_i/W_i	8	5	5	4	3.3

Step 3

Without exceeding the knapsack capacity, insert the items in the knapsack with maximum profit.

Knapsack = {5, 2, 3}

However, the knapsack can still hold 4 kg weight, but the next item having 5 kg weight will exceed the capacity. Therefore, only 4 kg weight of the 5 kg will be added in the knapsack.

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
Knapsack	1	1	1	4/5	0

Hence, the knapsack holds the weights = [(1 * 1) + (1 * 3) + (1 * 2) + (4/5 * 5)] = 10, with maximum profit of [(1 * 8) + (1 * 15) + (1 * 10) + (4/5 * 20)] = 37.