UNIT III
GREEDY AND DYNAMIC PROGRAMMING

Syllabus

Introduction – Greedy

Huffman Coding

Knapsack Problem

Minimum Spanning Tree (Kruskals Algorithm)

Minimum Spanning Tree (Prim's Algorithm)

Dynamic Programming

Introduction

0/1 knapsack problem

Matrix chain multiplication using dynamic programming

Longest common subsequence using dynamic programming

Optimal binary search tree (OBST)using dynamic programming

Greedy Method

- Greedy algorithm obtains an optimal solution by making a sequence of decisions.
- Decisions are made one by one in some order.
- Each decision is made using a greedy-choice property or greedy criterion.
- A decision, once made, is (usually) not changed later.

Introduction

- Greedy is a strategy that works well on optimization problems.
- Optimization - the action of making the best or most effective use of a situation or resource.
- **Characteristics:**

1. **Greedy-choice property:** 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.
2. **Optimal substructure:** A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

- **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

- Huffman coding Algorithm for finding optimum Huffman trees.
- 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.

Pseudo code of Greedy Algorithm

- Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

```
Algorithm Greedy (a, n)
{
  Solution : = 0;
 for i = 0 to n do
 {
    x: = select(a);
   if feasible(solution, x)
   {
      Solution: = union(solution , x)
   }
     return solution;
 } }
```

Let's understand through an example.

- Suppose there is a problem 'P'. I want to travel from A to B shown as below:
- **P : A → B**
- The problem is that we have to travel this journey from A to B. There are various solutions to go from A to B.
- We can go from A to B by **walk, car, bike, train, aeroplane**, etc.
- There is a constraint in the journey that we have to travel this journey within 12 hrs.
- If I go by train or aeroplane then only, I can cover this distance within 12 hrs.
- There are many solutions to this problem but there are only two solutions that satisfy the constraint.

Let's understand through an example.

- If we say that we have to cover the journey at the minimum cost.
- This means that we have to travel this distance as minimum as possible, so this problem is known as a minimization problem.
- Till now, we have two feasible solutions, i.e., one by train and another one by air. Since travelling by train will lead to the minimum cost so it is an optimal solution.
- An optimal solution is also the feasible solution, but providing the best result so that solution is the optimal solution with the minimum cost.
- There would be only one optimal solution

Huffman Coding

Huffman's algorithm

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman.
- Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Steps to build Huffman Tree

- **Input:** Array of unique characters along with their frequency of occurrences
- **Output:** Huffman Tree.

Huffman Tree-

The steps involved in the construction of Huffman Tree are as follows-

- **Step-01:**

  Create a leaf node for each character of the text.

  Leaf node of a character contains the occurring frequency of that character.

- **Step-02:**

  Arrange all the nodes in increasing order of their frequency value.

- **Step-03:**

  Considering the first two nodes having minimum frequency,

  Create a new internal node.

  The frequency of this new node is the sum of frequency of those two nodes.

  Make the first node as a left child and the other node as a right child of the newly created node.

- **Step-04:**

  Keep repeating Step-02 and Step-03 until all the nodes form a single tree.

  The tree finally obtained is the desired Huffman Tree.

Huffman's algorithm

How Huffman Coding works?

- Suppose the string below is to be sent over a network.

- Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits are required to send this string.
- Using the Huffman Coding technique, we can compress the string to a smaller size.
- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.
- Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

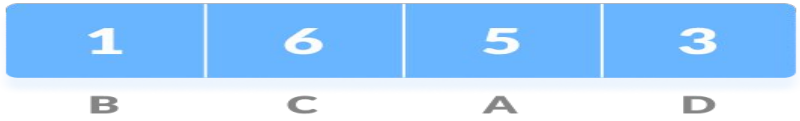
Huffman's code

- Huffman Coding prevents any ambiguity in the decoding process using the concept of **prefix code** ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property.
- Huffman coding is done with the help of the following steps.
  1. Calculate the frequency of each character in the string.
  2. Sort the characters in increasing order of the frequency.
  3. Make each unique character as a leaf node.
  4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.
  5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies
  6. Repeat this step until you end to a single node.

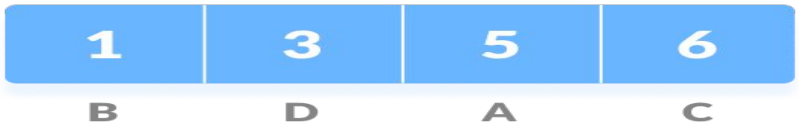
Huffman's Logic

Question:

Calculate the frequency of each character in the string.

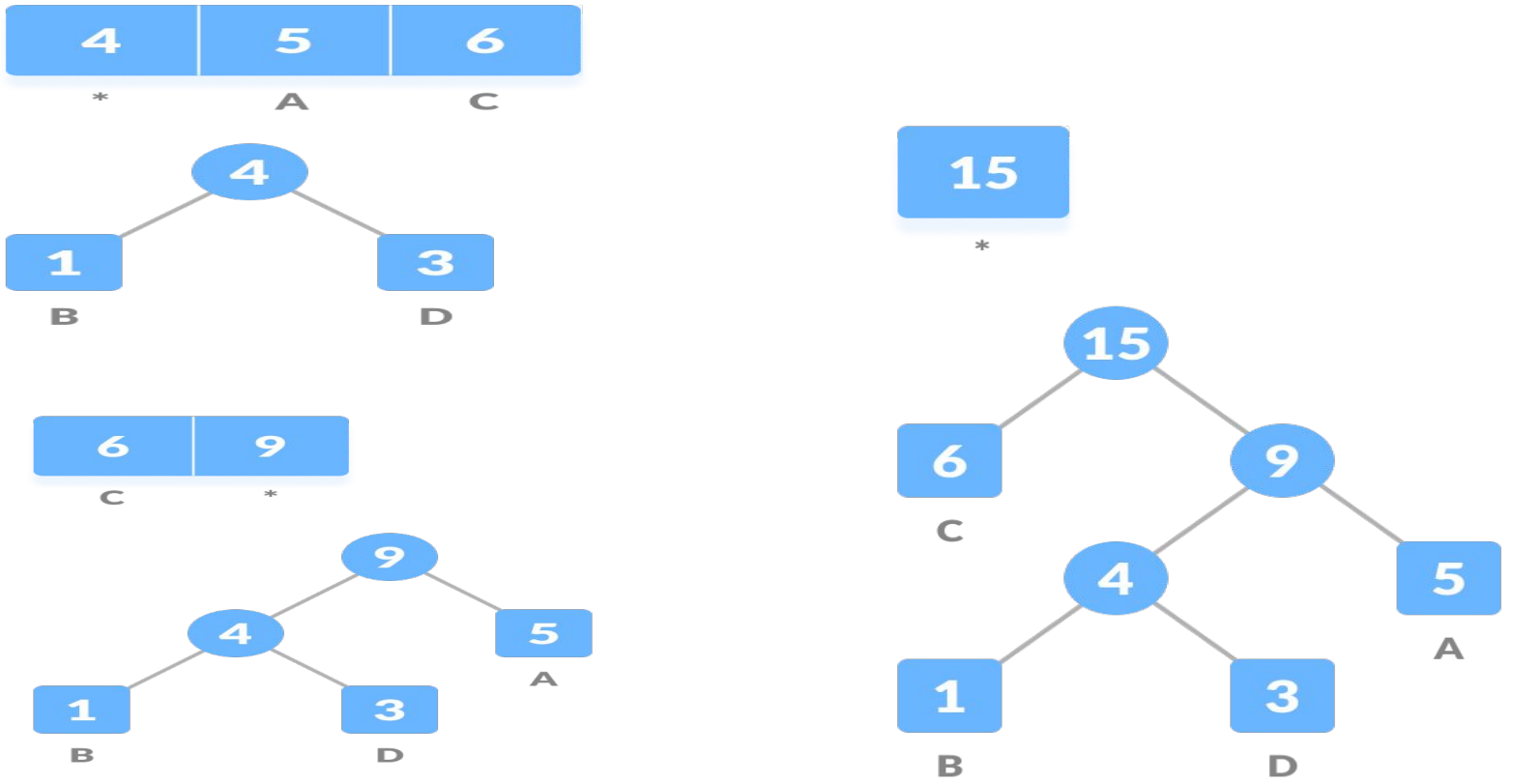


Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.



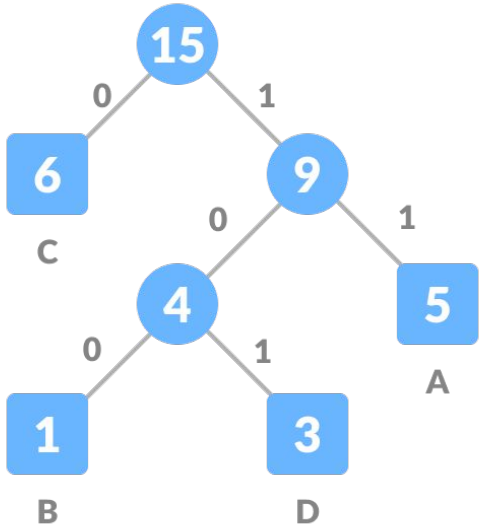
Huffman's Logic

Make each unique character as a leaf node.



Huffman's Logic

For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Huffman's Logic

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

Huffman Coding Algorithm

- create a priority queue Q consisting of each unique character.
- sort then in ascending order of their frequencies.
- for all the unique characters:
  - create a newNode
  - extract minimum value from Q and assign it to leftChild of newNode
  - extract minimum value from Q and assign it to rightChild of newNode
  - calculate the sum of these two minimum values and assign it to the value of newNode
  - insert this newNode into the tree
- return rootNode

Algorithm for Huffman Coding

Data Structure used: Priority queue = Q

Huffman (c)

n = |c|

Q = c

for i =1 to n-1

do z = Allocate-Node ()

x = left[z] = EXTRACT_MIN(Q)

y = right[z] = EXTRACT_MIN(Q)

f[z] = f[x] + f[y]

INSERT (Q, z)

return EXTRACT_MIN(Q)

HUFFMAN ENCODING

- Problem: Finding the minimum length bit string which can be used to encode a string of symbols.
- Used for compressing data.
- Uses a simple heap based priority queue.
- Each leaf is labeled with a character and its frequency of occurrence.
- Each internal node is labeled with the sum of the weights of the leaves in its subtree.
- Huffman coding is a lossless data compression algorithm.

Example

- Suppose we have a data consists of 100,000 characters that we want to compress. The characters in the data occur with following frequencies.

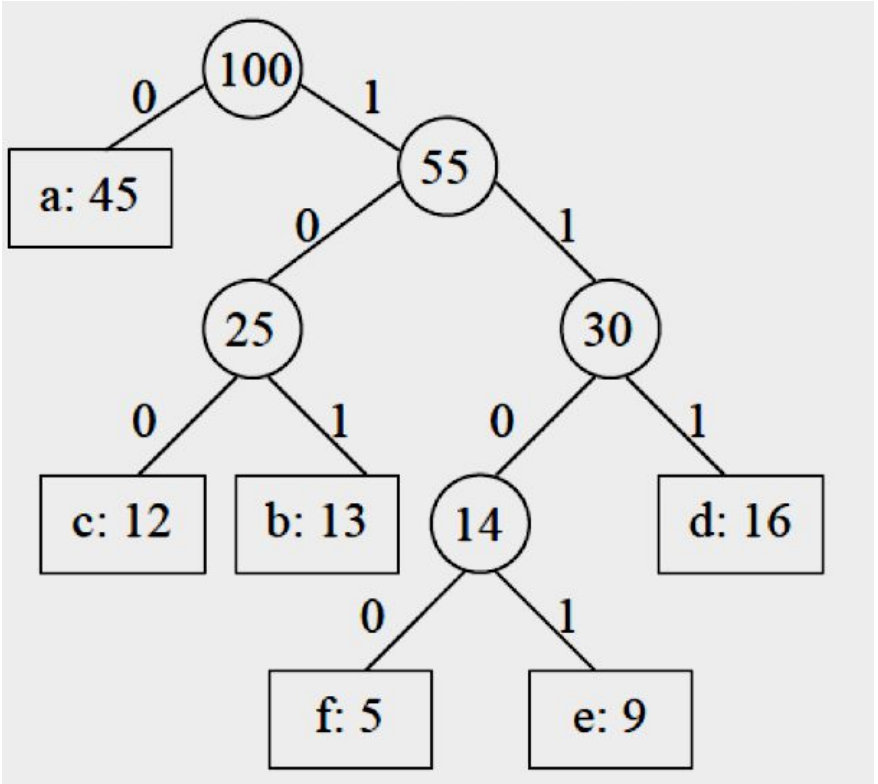
	A	B	C	D	E	F
Frequency	45,000	13,000	12,000	16,000	9,000	5,000

Methods to solve

- **Solve**

Variable Length coding

	A	B	C	D	E	F
Frequency	**45000**	**13000**	**12000**	**16000**	**9000**	**5000**
Fixed Length code	**000**	**001**	**010**	**011**	**100**	**101**
Variable length code	**0**	**101**	**100**	**111**	**1101**	**1100**



Huffman Coding - *Time complexity*

- The time complexity for encoding each unique character based on its frequency is O(nlog n).
- Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).

- Thus, Overall time complexity of Huffman Coding becomes **O(nlogn)**.
- Here, n is the number of unique characters in the given text.

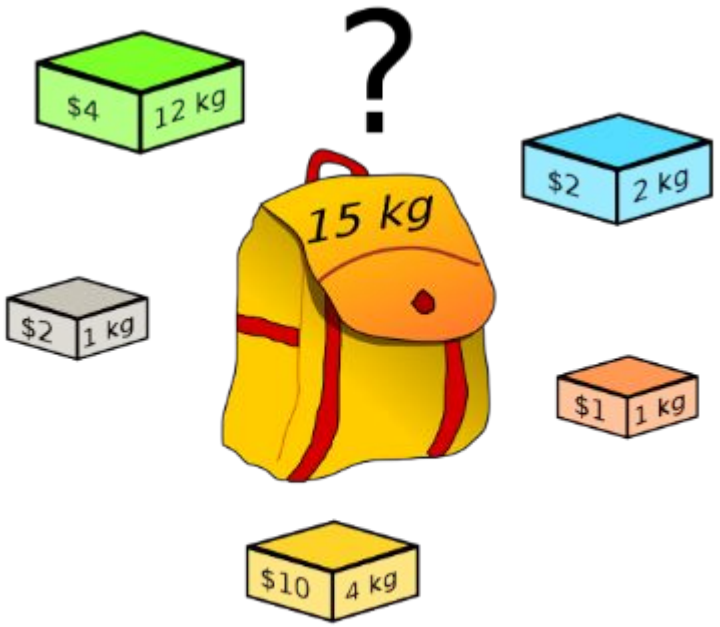
Knapsack Problem

Knapsack Problem

- Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

- You are given the following-
  - A knapsack (kind of shoulder bag) with limited weight capacity.
  - Items each having some weight and value.

- **The problem states-**
- Which items should be placed into the knapsack such that-
- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



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.

- **Knapsack Problem Variants-**
- Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

- **Fractional Knapsack Problem-**
- In Fractional Knapsack Problem,
- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

Algorithm

Greedy-fractional-knapsack (*w, P, M*)

```
M= total weight;
For i =1 to n
{
Compute Pi/wi;
   Sort object in non increasing order of p/w;
   }
For i=1 to n  (number of object)
{
If(M>0 && wi<=M)
{
M=M-wi;
P=P+Pi;
   else
   if(M>0)
        P = P+ Pi (M/wi);
}
}
```

Fractional Knapsack Problem Using Greedy Method-

- Fractional knapsack problem is solved using greedy method in the following steps-
- **Step-01:**
- For each item, compute its (value / weight) ratio.
- **Step-02:**
- Arrange all the items in decreasing order of their value / weight ratio.
- **Step-03:**
- Start putting the items into the knapsack beginning from the item with the highest ratio.
- Put as many items as you can into the knapsack.

Example Problem

- Input: 5 objects, C = 100

O	1	2	3	4	5
W	10	20	30	40	50
V	20	30	66	40	60

Solve the knapsack problem **using greedy approach**

Example Problem

- Input: 5 objects, C = 100

O	1	2	3	4	5
W	10	20	30	40	50
V	20	30	66	40	60

Solution:

- Given Total no of items = 5, sack capacity = 100

Step 1 : Find the Value/weight ratio

Ratio = v_i/w_i

Object	1	2	3	4	5
Ratio = v_i/w_i	2	1.5	2.2	1	1.2

- **Step 2 :** Sort the items according to the ratio and Select the item according to its highest ratio. First item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = vi/wi	2.2	2	1.5	1.2	1
Selected item	1				

- **Sack Weight = 30 <100**
- **Sack value = 2.2 * 30 = 66**

- **Step 3: Second item is selected**

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = Vi/Wi	2.2	2	1.5	1.2	1
Selected item	1	1			

- **Sack Weight = 30 +10 = 40 <100**
- **Sack value = 66 + (2*10) =86**

- Step 4: **Third item is selected**

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = Vi/Wi	2.2	2	1.5	1.2	1
Selected item	1	1	1		

- **Sack Weight = 40 +20 = 60 <100**
- **Sack value = 86 + (1.5*20) = 116**

•Step 5: Fourth item is selected

Object	3	1	2	5	4
W	30	10	20	50	40
Ratio = Vi/Wi	2.2	2	1.5	1.2	1
Selected item (x_i)	1	1	1	4/5	0

Sack Weight = 60 +50 = **110 >100 Hence item 4 is selected partially.**
Sack Weight = 60+(100-60) = **100 <=100**
Sack value = 116 + (1.2*40) = 116+48= 164

- **Now the sack is FULL. Hence we stop**
- Total selected weight 100 and

  Total Weight **(xi * w) = 1*30 + 1* 10 + 1* 20 + 4/5 *50 + 0*40 = 100**

  Total value (xi * v) = 1* 66 + 1* 20 + 1* 30 + 4/5 * 60 + 0 * 40
- **Total value** = 66+20+30+ 48 **= 164**

Problem-
Knapsack Problem Using Greedy Method-

- For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach.

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Or

- Find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

- n = 5
- w = 60 kg
- (w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)
- (b1, b2, b3, b4, b5) = (30, 40, 45, 77, 90)

Problem-
Knapsack Problem Using Greedy Method-

- A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Solve

KNAPSACK PROBLEM

- We are given n objects and a knapsack(bag). Object i has a weight w_i and capacity m.
- If a fraction $x_i, 0 \leq x_i \leq 1,$ of object i is placed into the knapsack, then the profit $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned.
- We require the total weight of the chosen objects to be at most m.
- Hence, the objective of this algorithm is to

Maximize $\sum_{1 \leq i \leq n} p_i x_i$

Subject to $\sum_{1 \leq i \leq n} w_i x_i$ <= m

and $0 \leq x_i \leq 1, 1 \leq i \leq n$

The profits and weights are positive numbers.

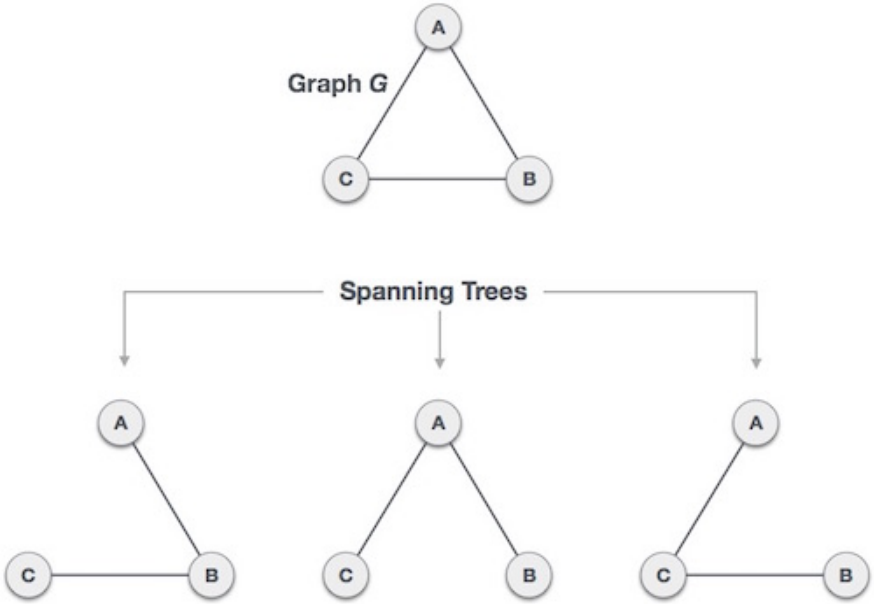
Time Complexity-
Knapsack Problem Using Greedy Method-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of **Quick Sort** is O(nlogn).
- Therefore, total time taken including the sort is O(nlogn).

Minimum Spanning Tree

Minimum Spanning Tree

- **A spanning tree** is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- Graph have G = (V,E). V Vertices or nodes and E Edges.



Mathematical properties of spanning tree

- Spanning tree has n-1 edges, where n is number of nodes (vertices)
- So we can conclude here that spanning trees are subset of a connected Graph G.

Application of Spanning Tree

- Spanning tree is basically used to find minimum paths to connect all nodes in a graph.
  - **Civil Network Planning**
  - **Computer Network Routing Protocol**
  - **Cluster Analysis**

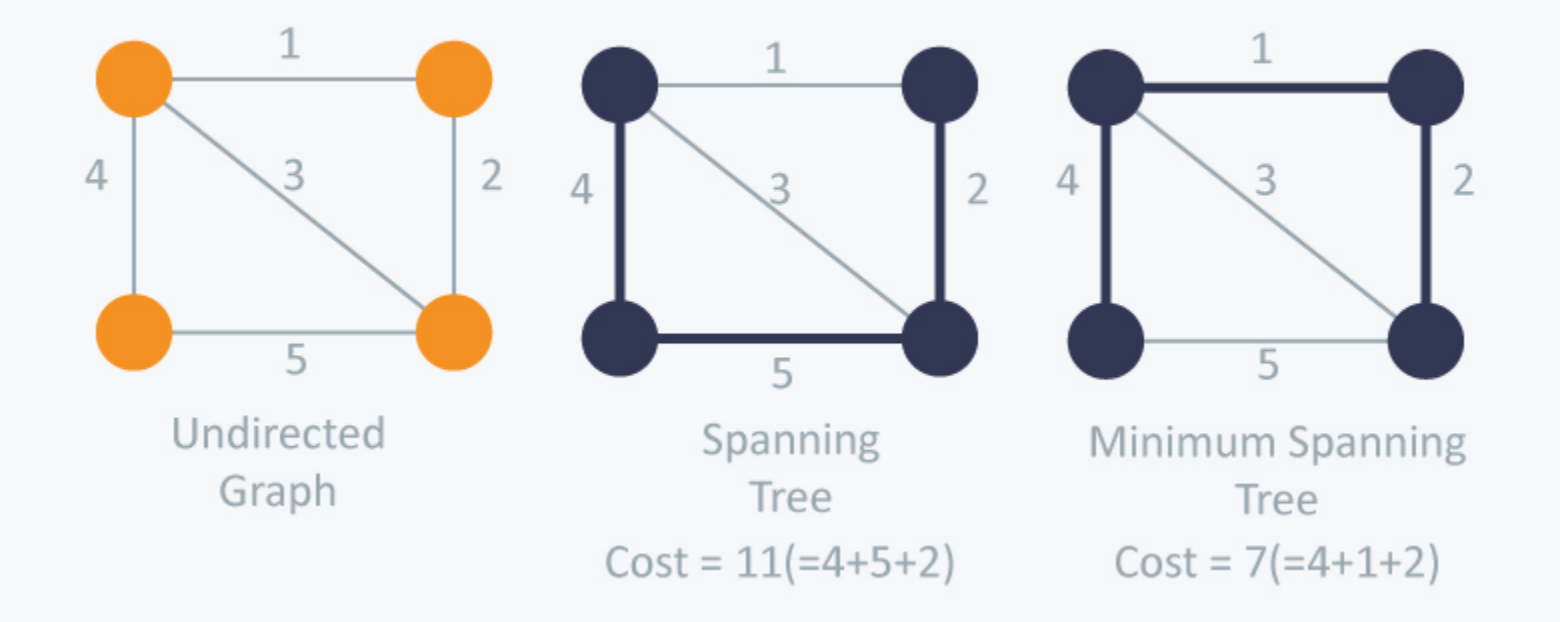
Minimum Spanning Tree (MST)

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

MST Algorithm

- Kruskal's Algorithm
- Prim's Algorithm

Both are greedy algorithms.



Kruskal's Algorithm

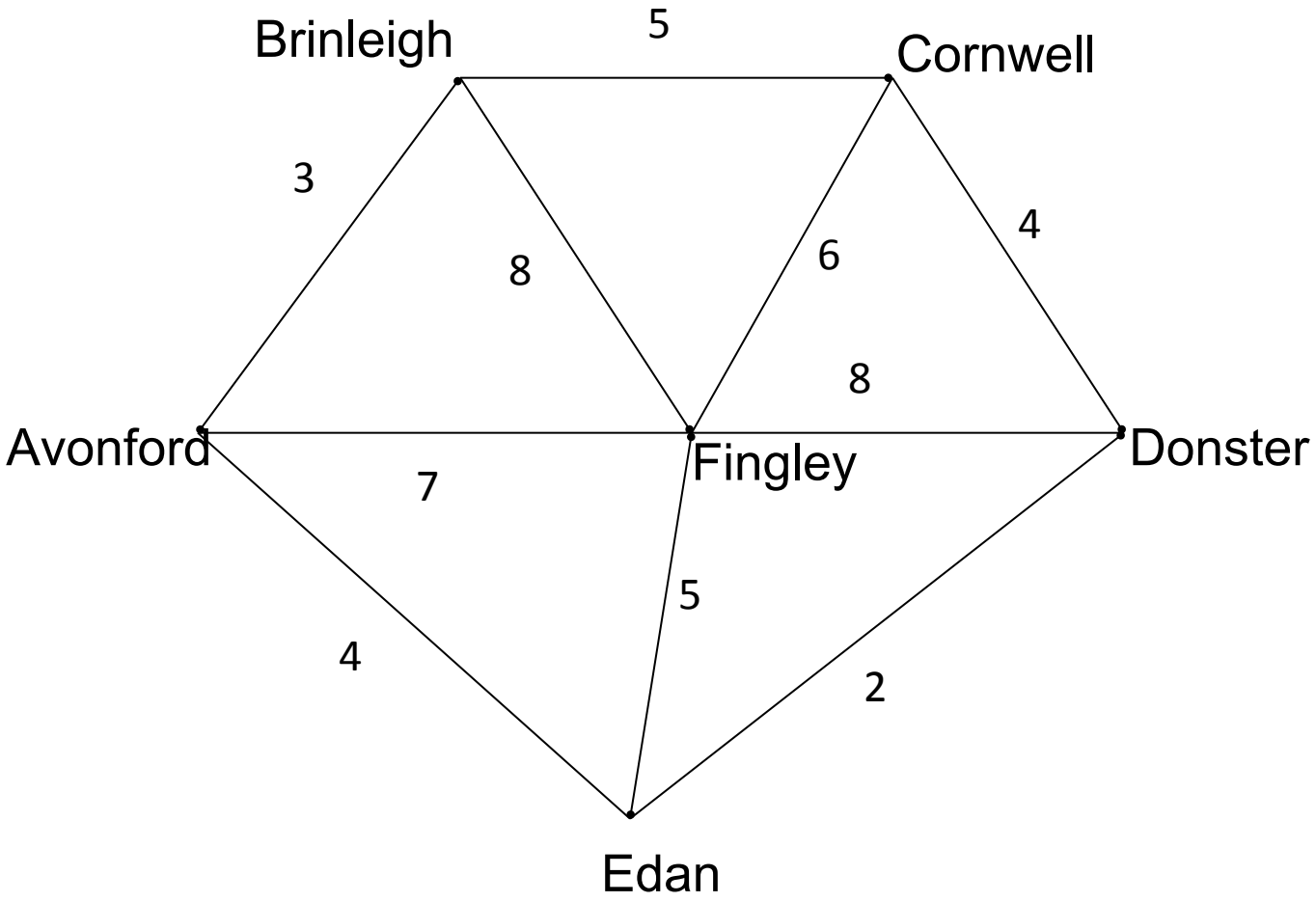
Kruskal Algorithm

Algorithm

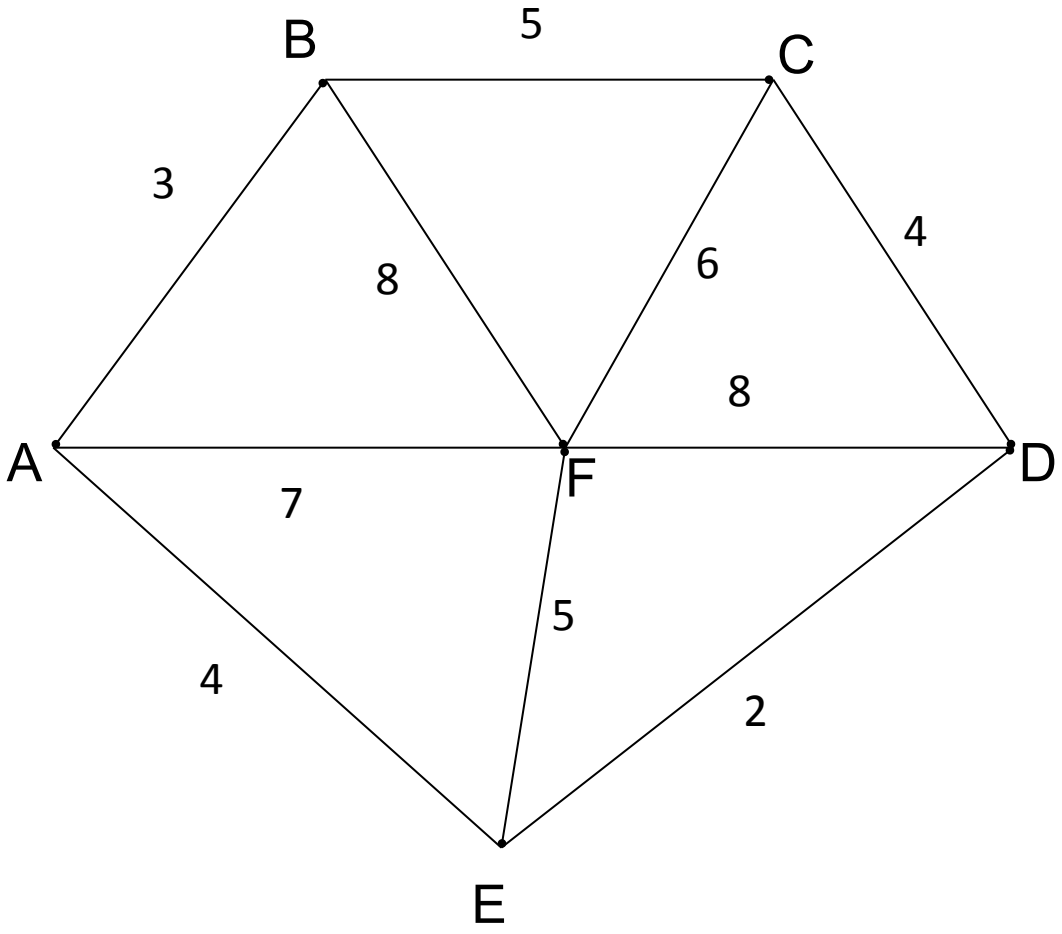
- Remove all loops & Parallel Edges from the given graph
- Sort all the edges in non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
- Repeat step#2 until there are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

Example

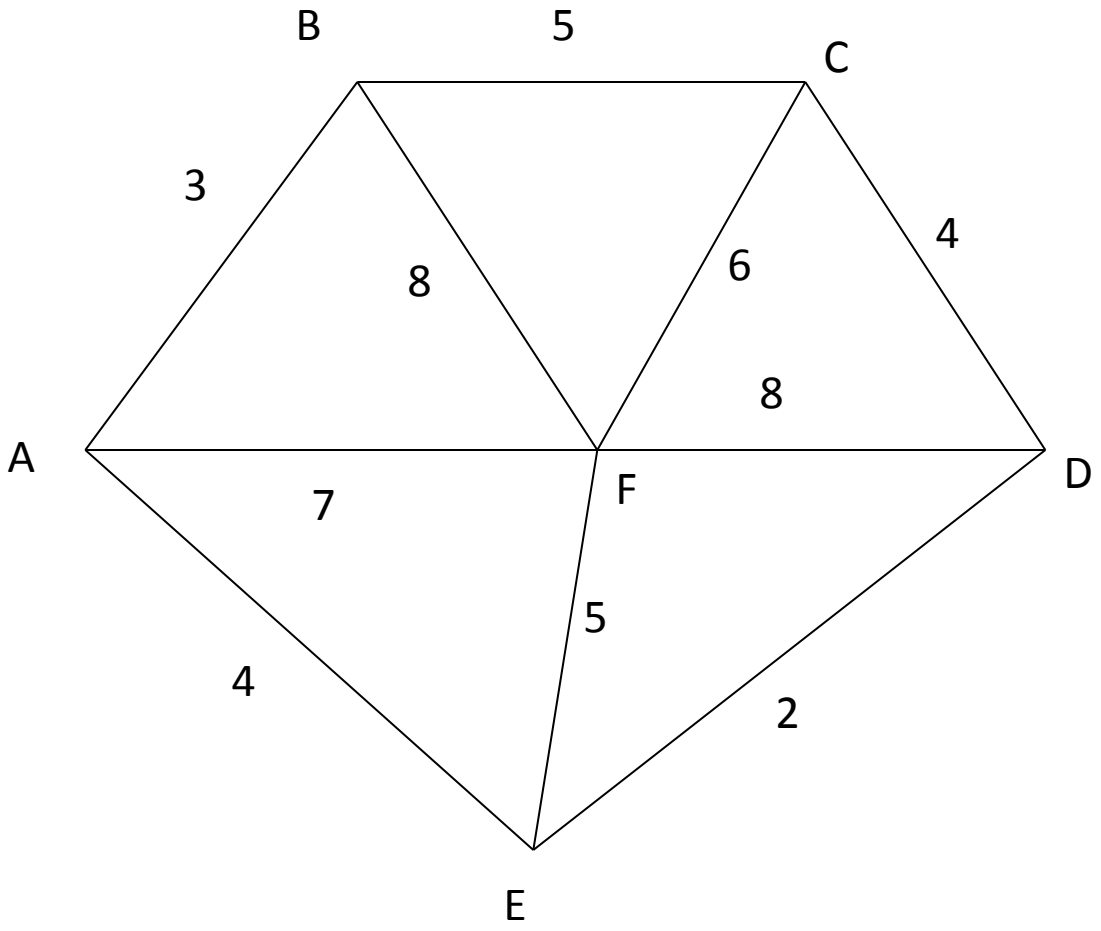
A cable company want to connect five villages to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



We model the situation as a network, then the problem is to find the minimum connector for the network



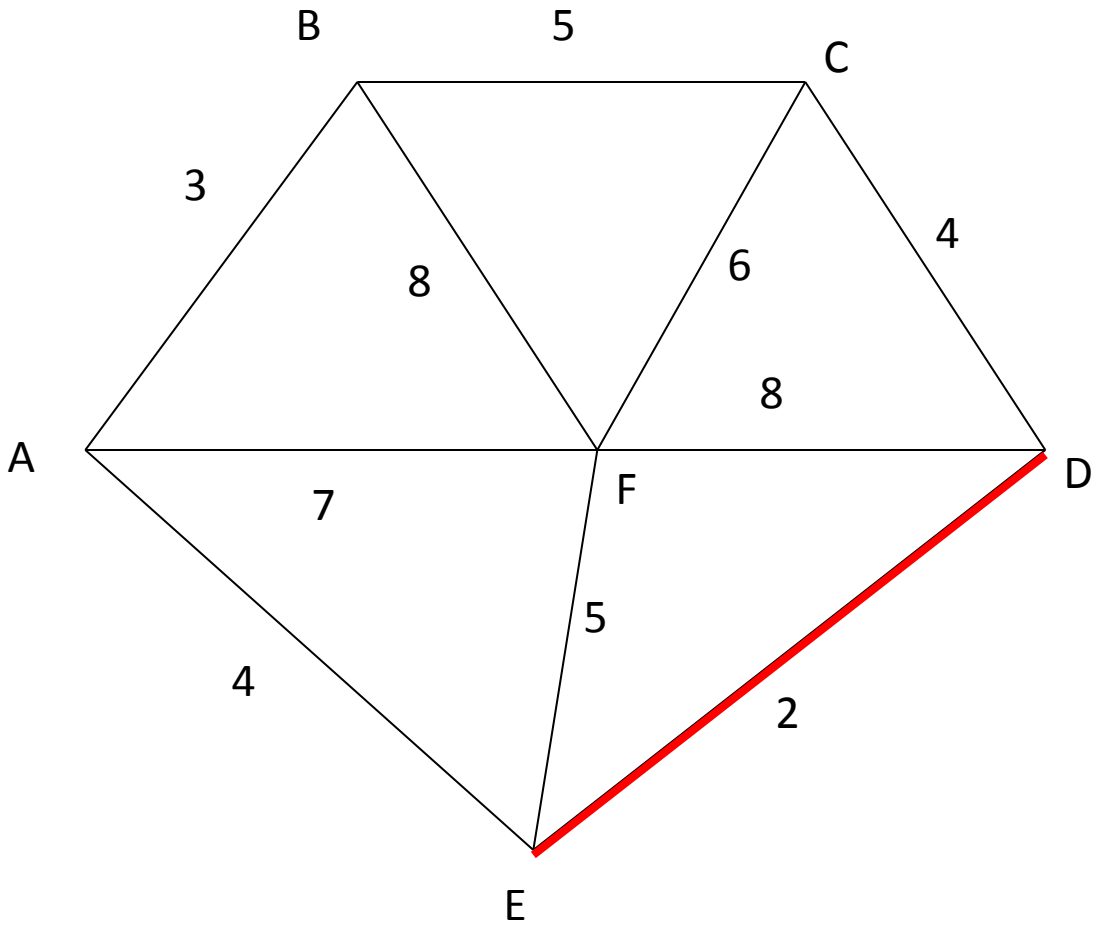
Kruskal's Algorithm



List the edges in order of size:

ED 2
AB 3
AE 4
CD 4
BC 5
EF 5
CF 6
AF 7
BF 8
DF 8

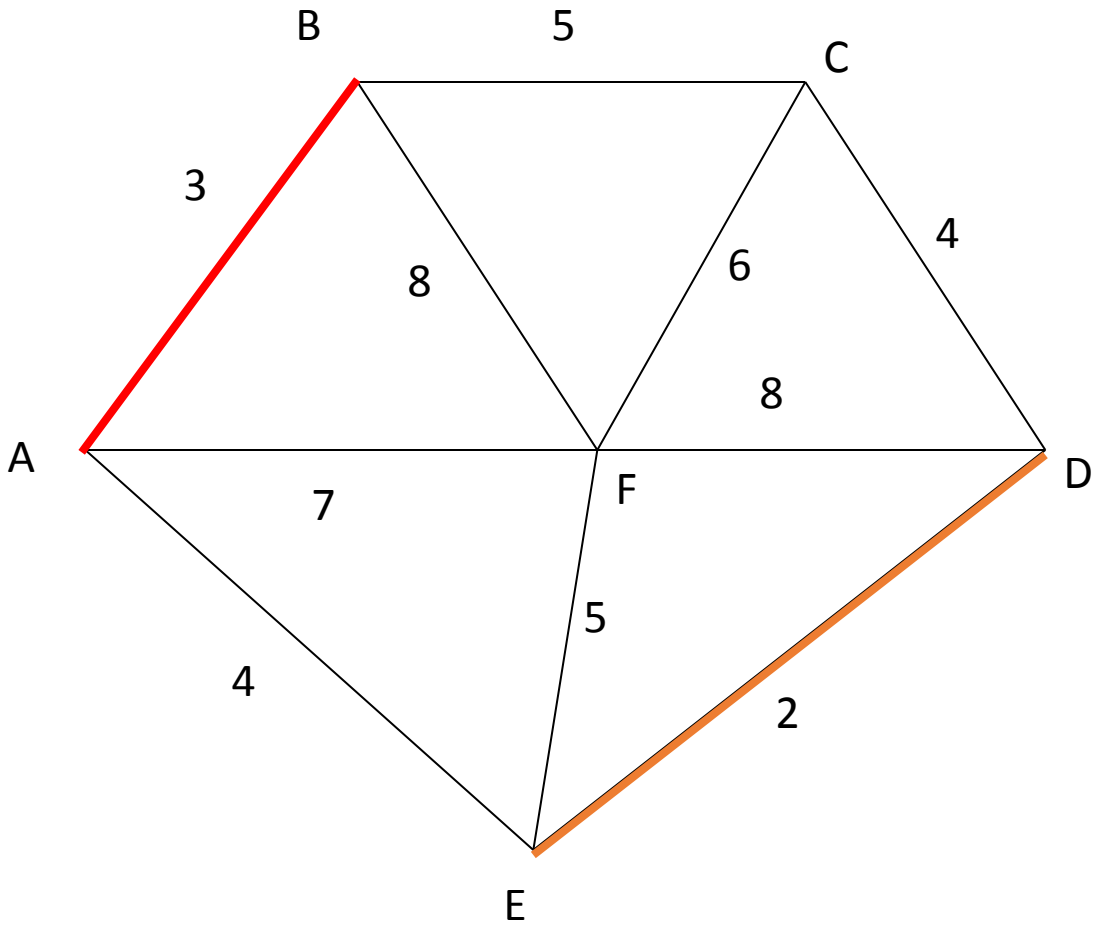
Kruskal's Algorithm



Select the shortest edge in the network

ED 2

Kruskal's Algorithm

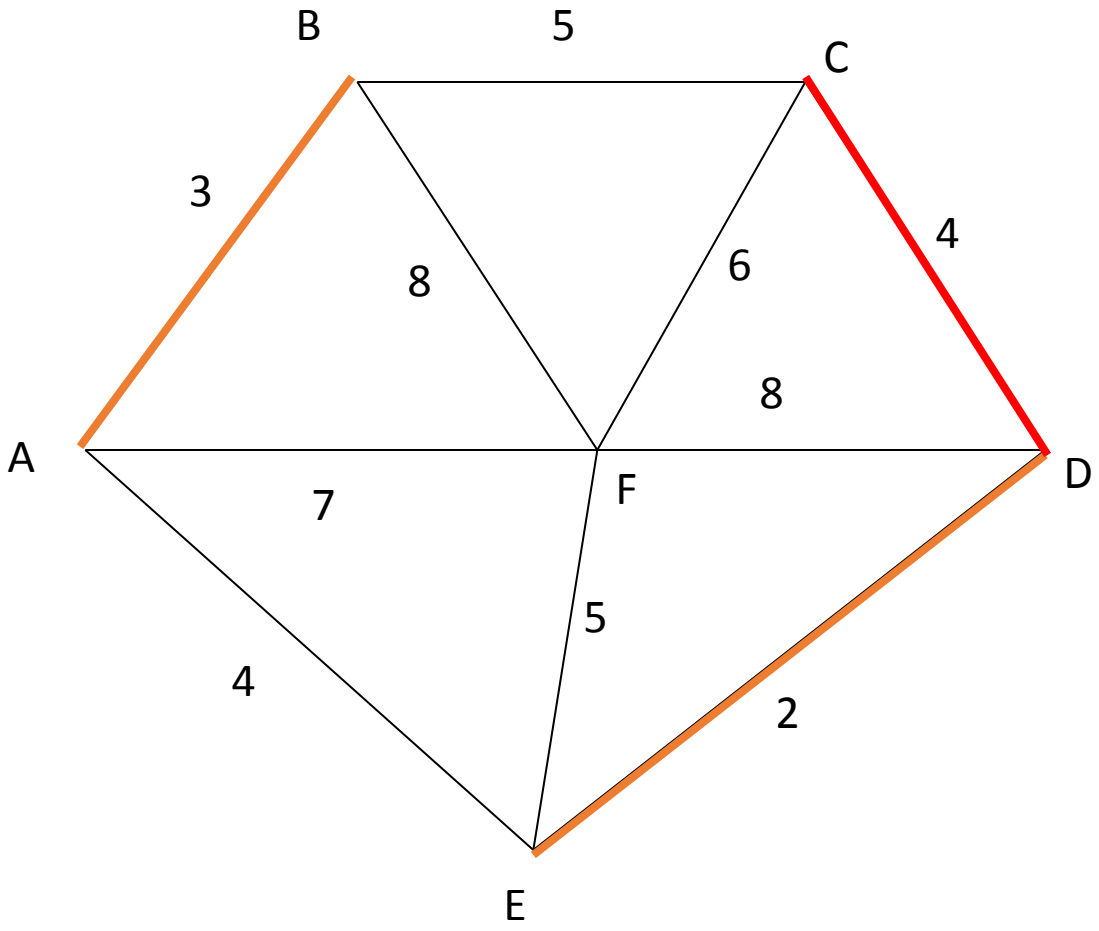


Select the next shortest edge which does not create a cycle

ED 2

AB 3

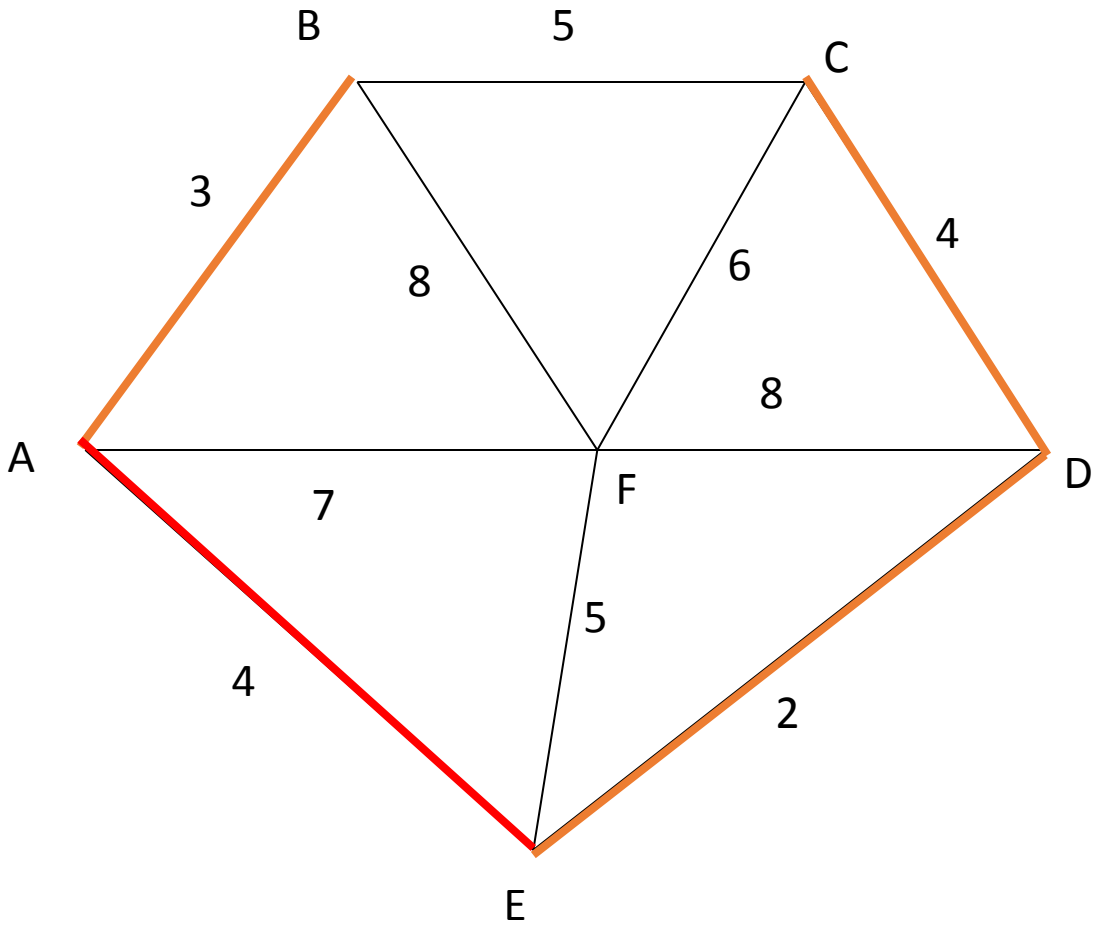
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4 (or AE 4)

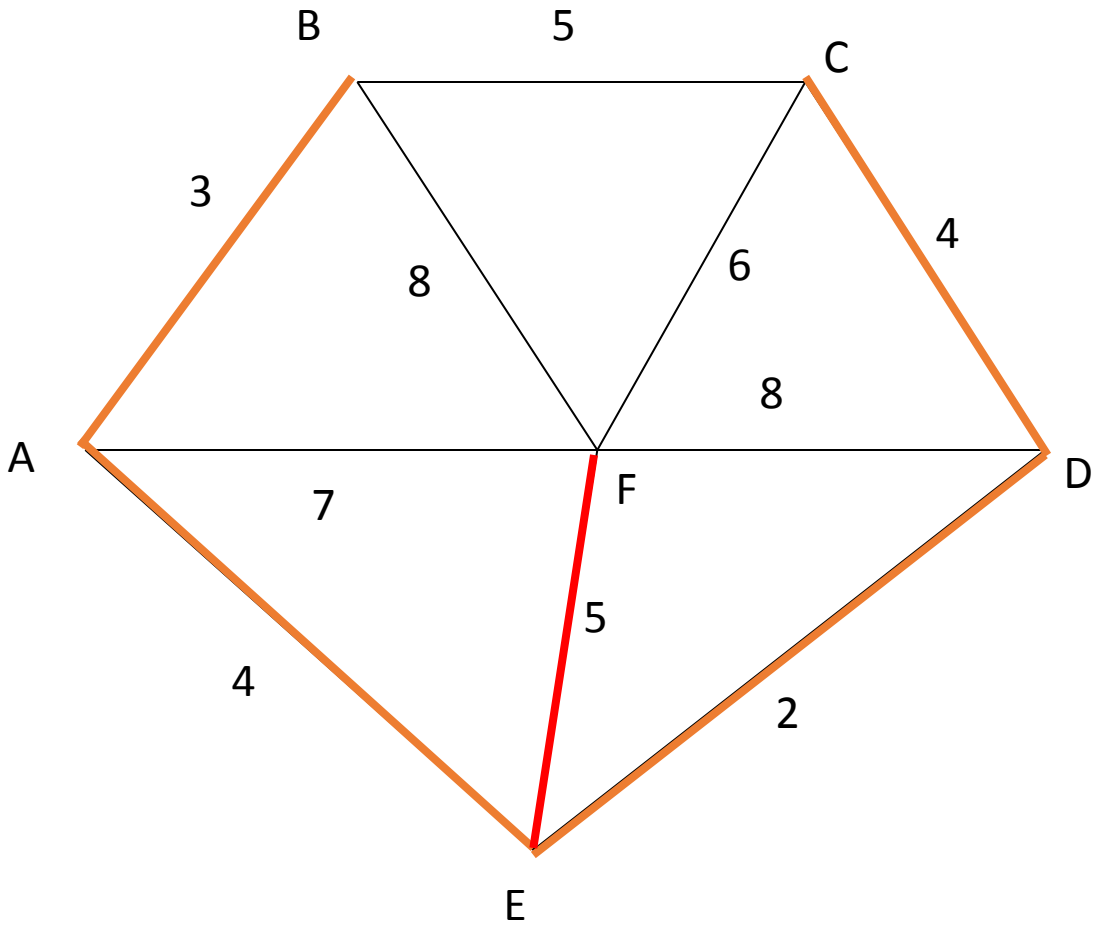
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4

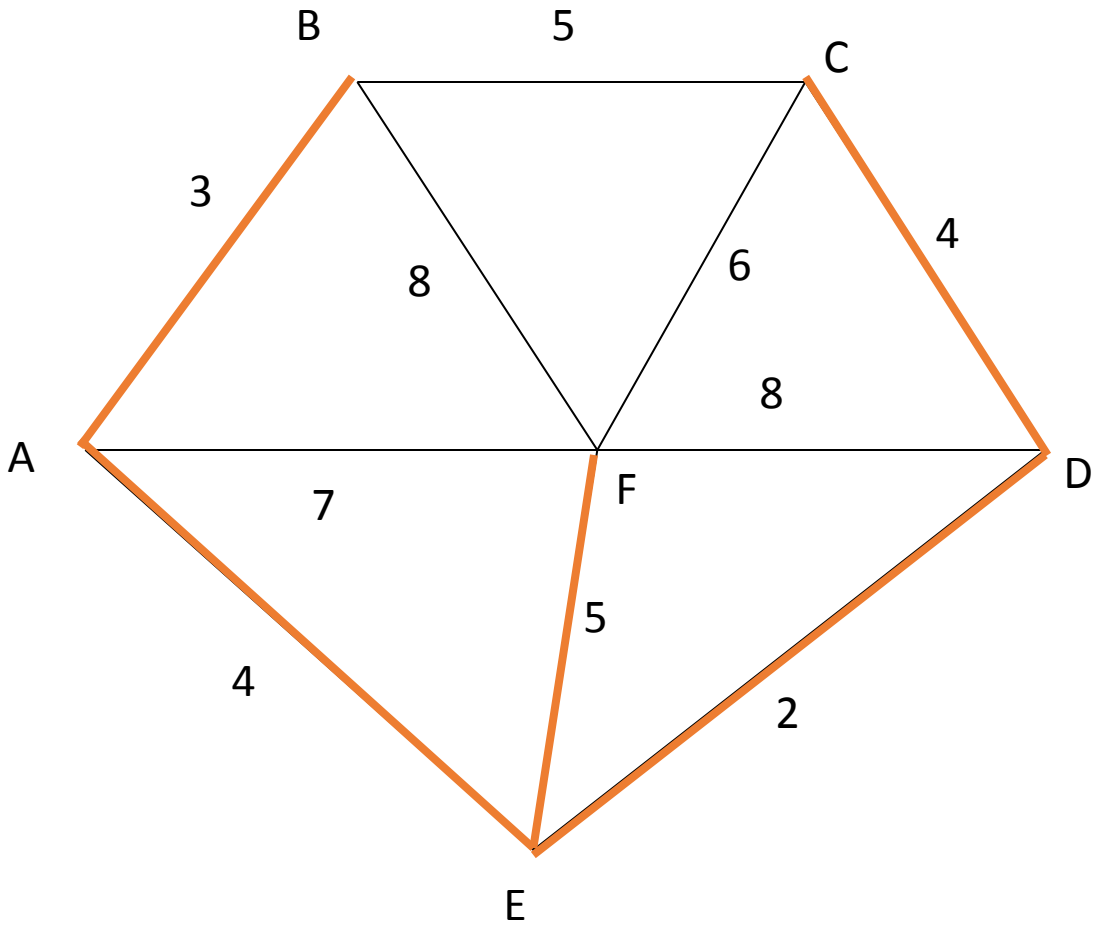
Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2
AB 3
CD 4
AE 4
BC 5 – forms a cycle
EF 5

Kruskal's Algorithm



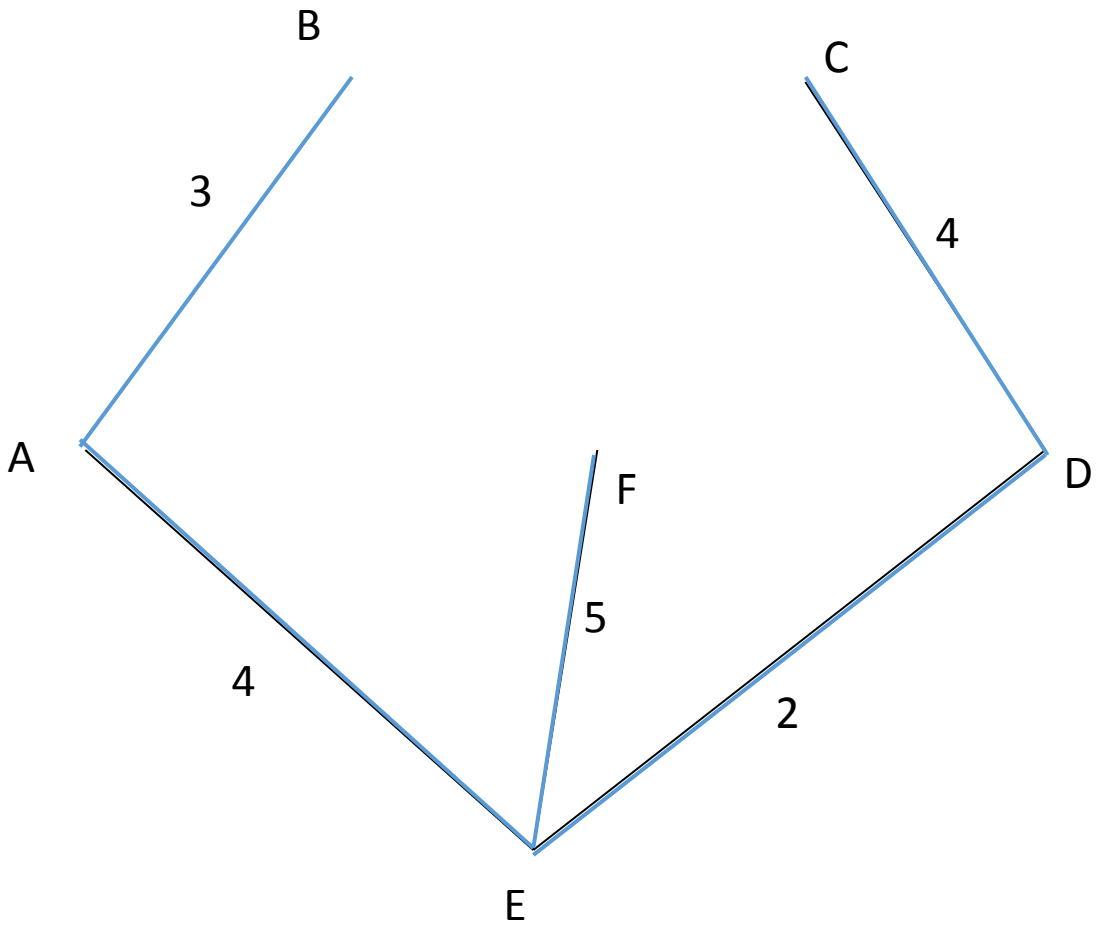
All vertices have been connected.

The solution is

ED 2
AB 3
CD 4
AE 4
EF 5

Total weight of tree: 18

Kruskal's Algorithm



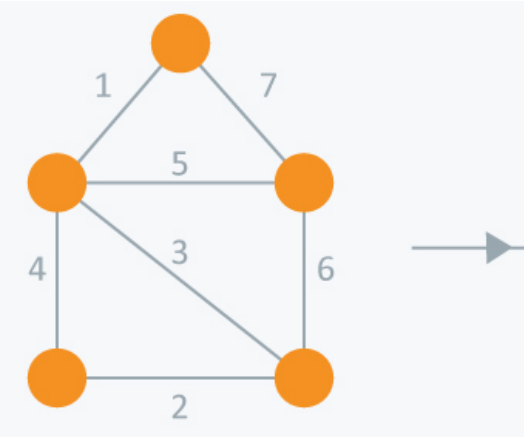
All vertices have been connected.

The solution is

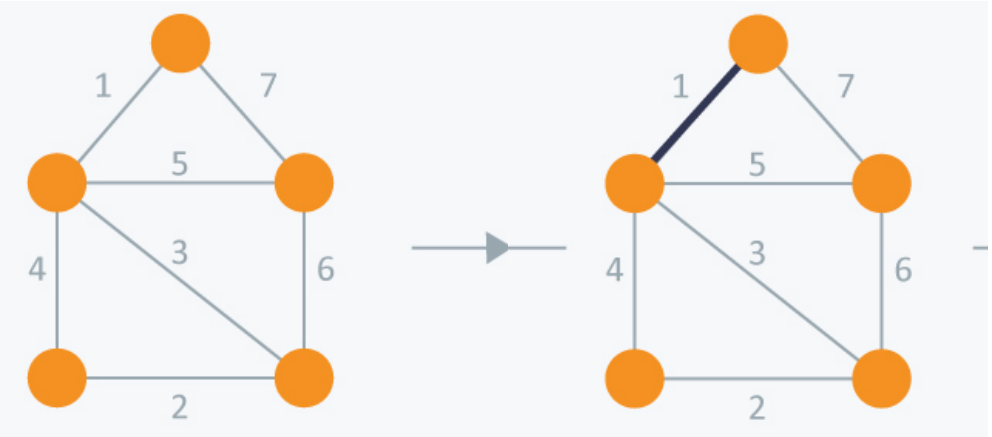
ED 2
AB 3
CD 4
AE 4
EF 5

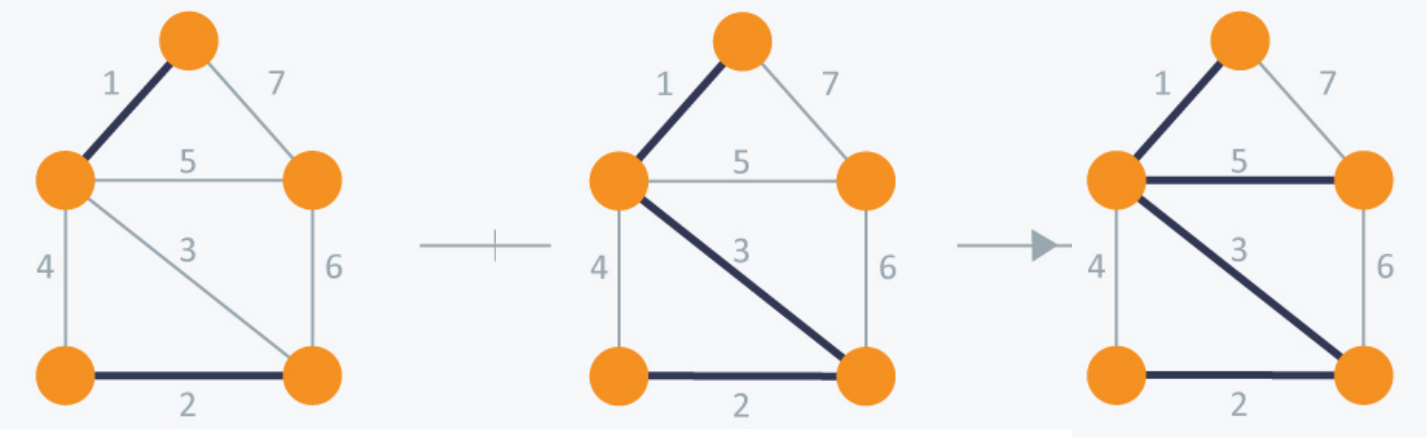
Total weight of tree: 18

Kruskal Algorithm

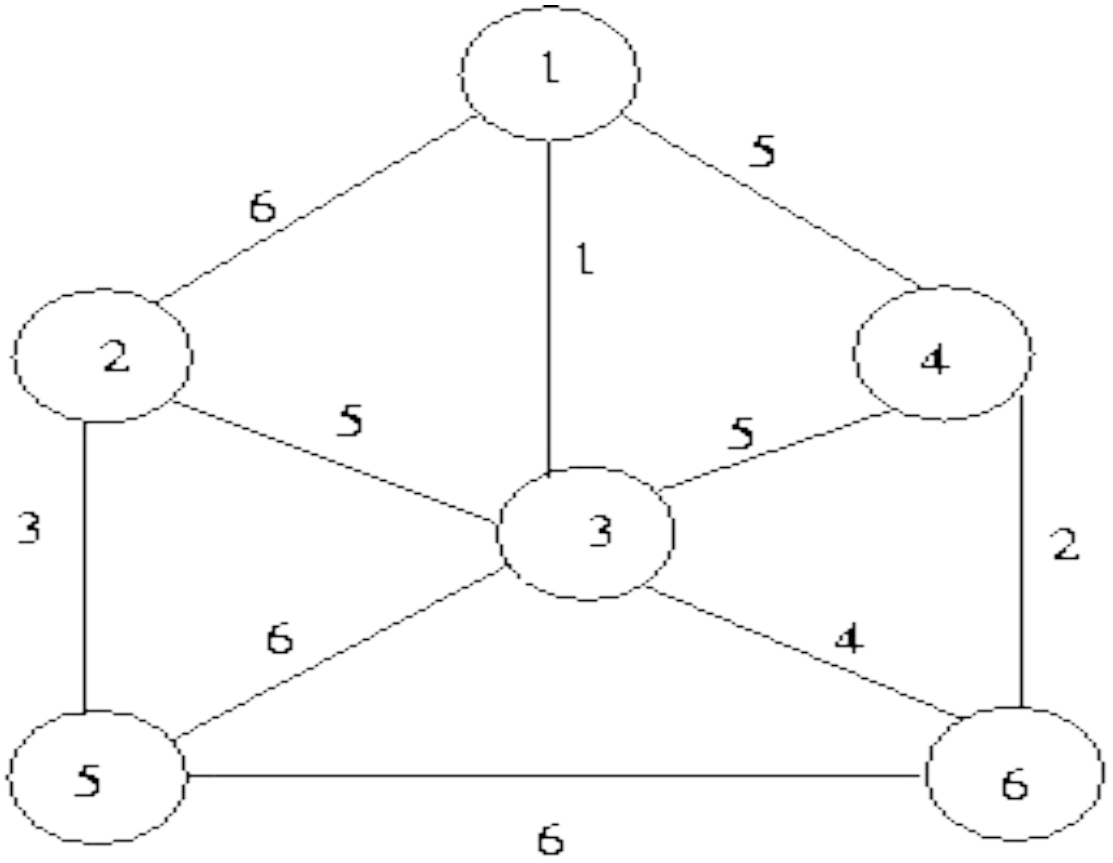


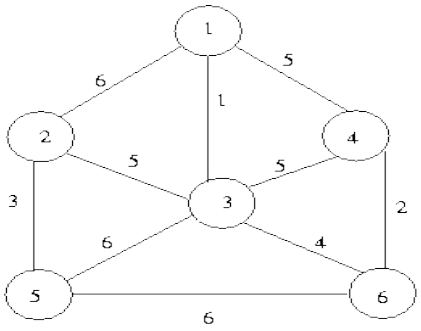
Kruskal Algorithm



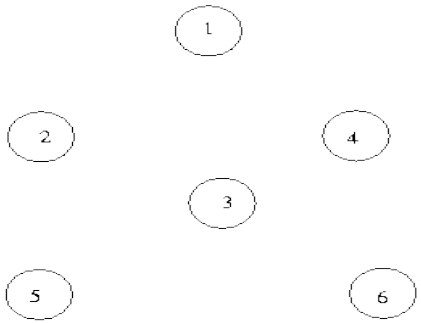


Find a MST using Kruskal Algorithm

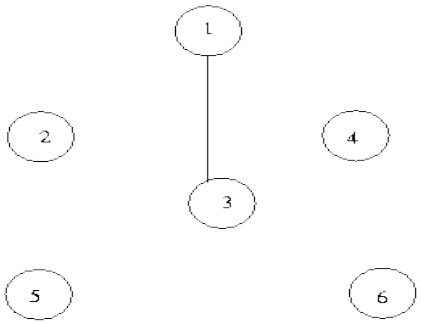




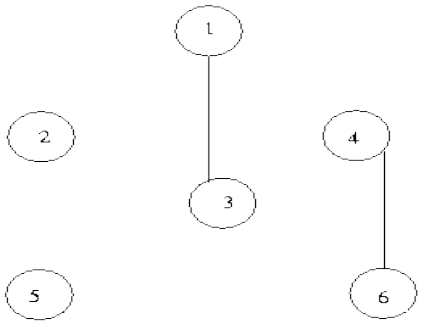
Kruskal algorithm to create MST



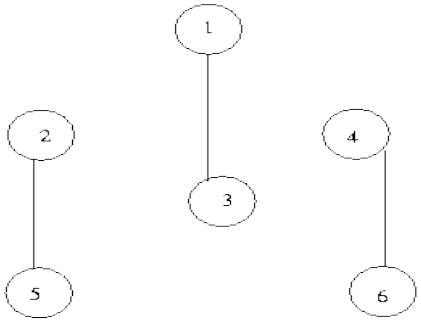
Initial Configuration



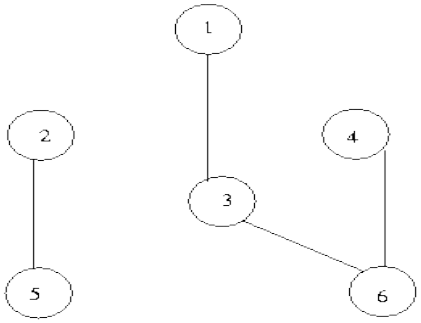
step1. choose (1,3)



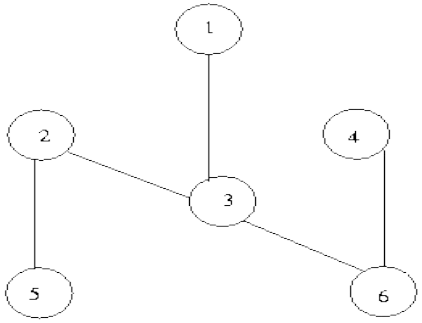
step2. choose (4,6)



step3. choose (2,5)



step4. choose (3,6)



step5. choose (2,3)

Algorithm

```
MST-KRUSKAL(G, w)
A ← Ø
 for each vertex v  V[G]
 do MAKE-SET(v)
       sort the edges of E into non-decreasing order by weight w
       for each edge (u, v)  E, taken in non-decreasing order by weight
       do if (u,v) not forming any cycle
             then A ← A  {(u, v)}
       UNION(u, v)
 return A
```

Algorithm Analysis

- **Time complexity**
- **To sort the vertices it takes nlogn**
- O(nlogn) where n is the number of unique characters.

Prim's Algorithm

Prim's Spanning Tree Algorithm

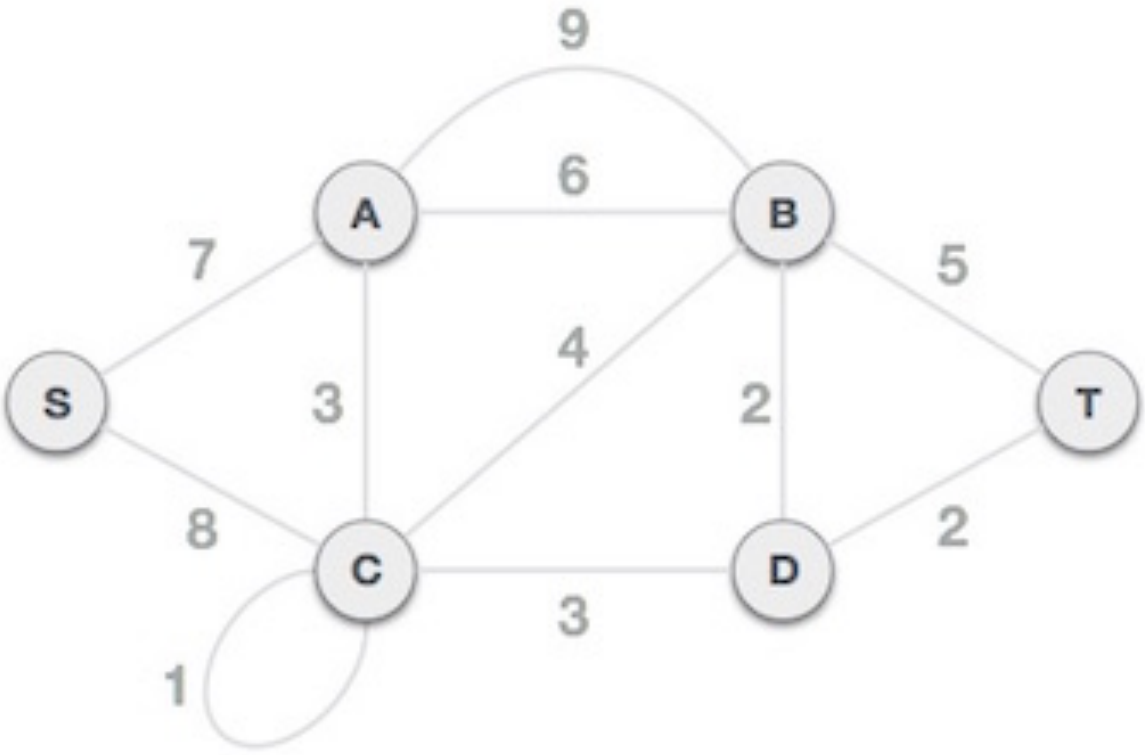
- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

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

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

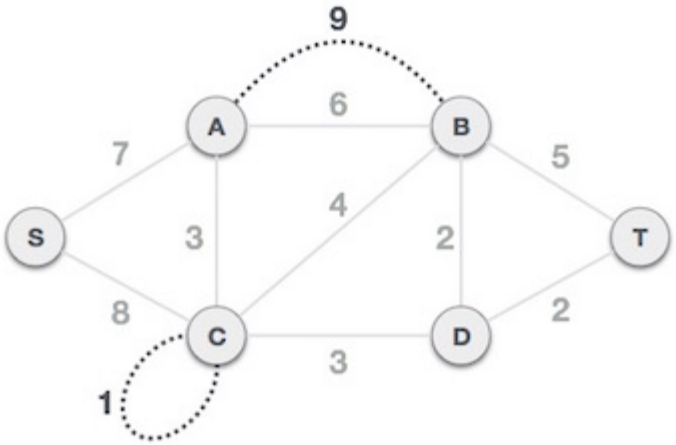
Prim's Spanning Tree Algorithm

- Example

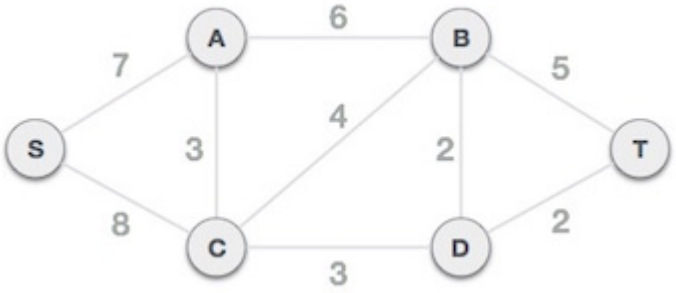


Prim's Spanning Tree Algorithm

- Step 1 - Remove all loops and parallel edges

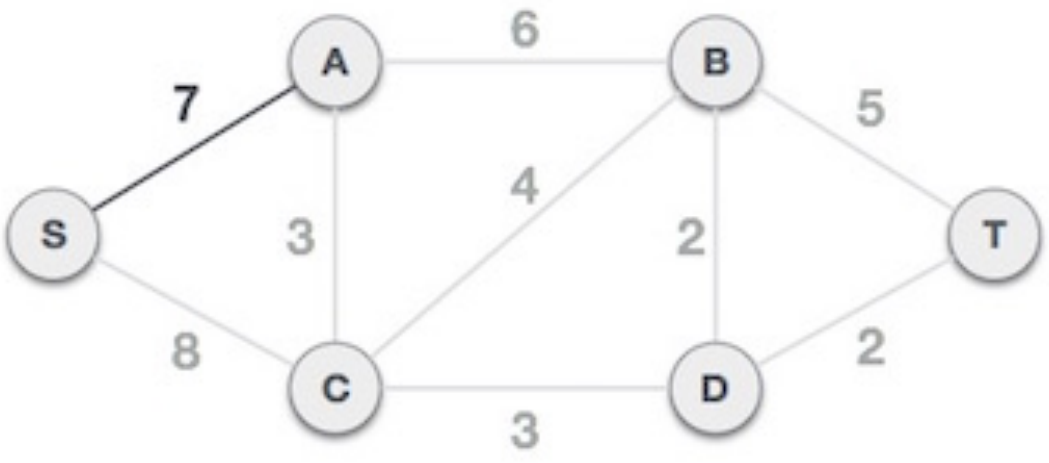


- Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.



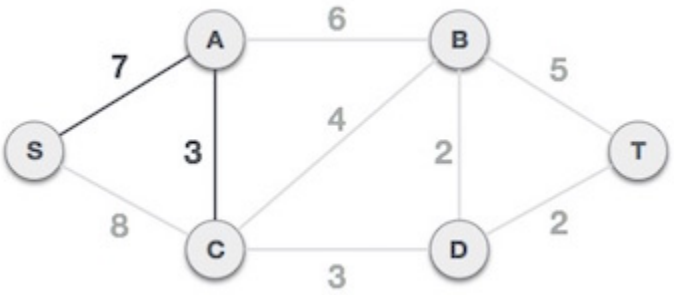
Prim's Spanning Tree Algorithm

- **Step 2 - Choose any arbitrary node as root node**
- In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node.
- **Step 3 - Check outgoing edges and select the one with less cost**
- After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

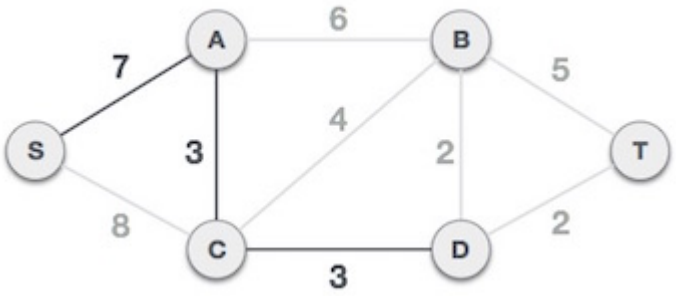


Prim's Spanning Tree Algorithm

- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

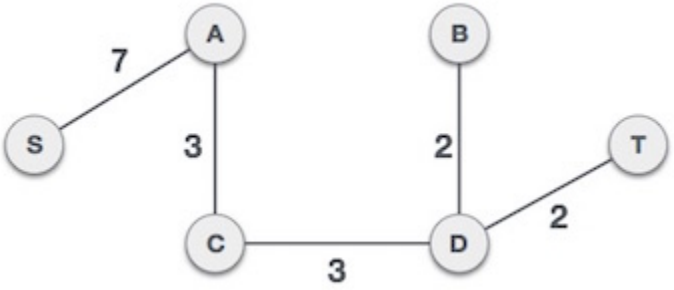


- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



Prim's Spanning Tree Algorithm

- After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



Algorithm

MST-Prims(G, w)

A ← Ø

for each vertex v V[G]

do MAKE-SET(v)

Select the source vertices u

Choose the minimal out degree edge from u to v

do if (u,v) not forming any cycle

then A ← A {(u, v)}

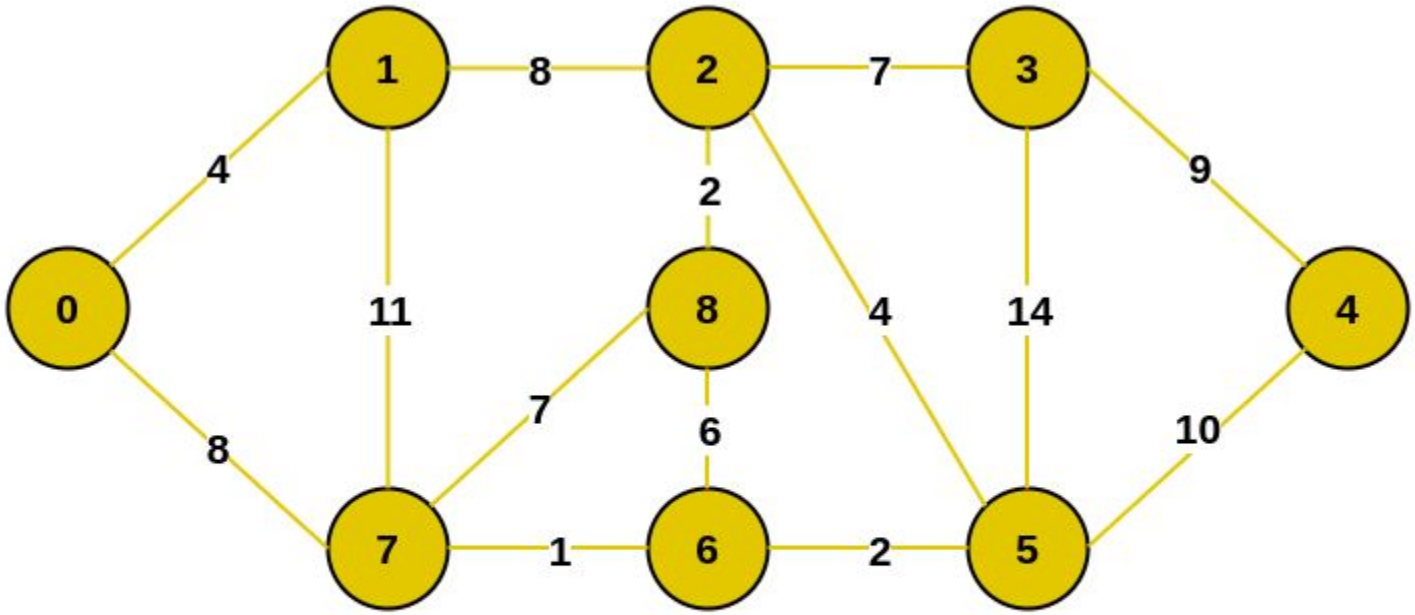
UNION(u, v)

return A

Algorithm Analysis

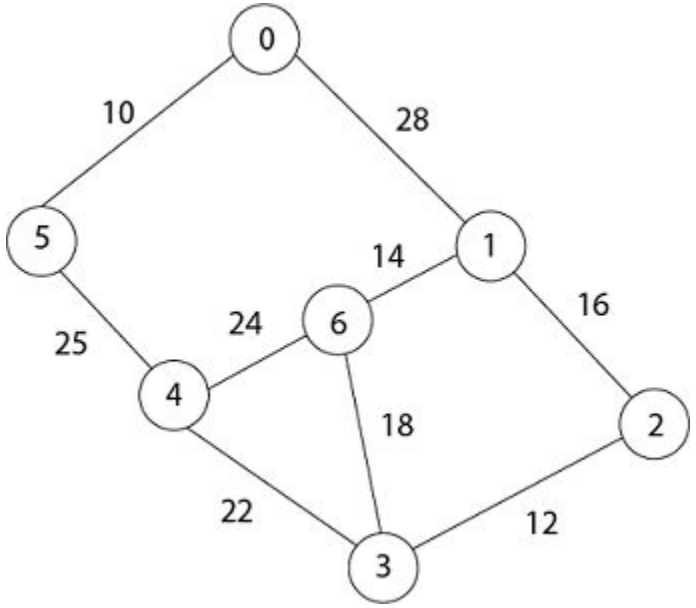
- **Time complexity**
- $O(E * \log V)$where E – edges and V - Vertices.
- Each edges connect two vertices so logV.

Solve using Prim's Spanning Tree Algorithm



Example of a Graph

Solve using Prim's Spanning Tree Algorithm



Dynamic Programming

Introduction

0/1 knapsack problem

Matrix chain multiplication using dynamic programming

Longest common subsequence using dynamic programming

Optimal binary search tree (OBST)using dynamic programming

Introduction to dynamic programming

- Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems.
- Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are
  - **overlapping sub-problems and**
  - **optimal substructure.**

Introduction to dynamic programming

- **Overlapping Sub-Problems**
  - Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.
- **Optimal Sub-Structure**
  - A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

Greedy Vs Dynamic Programming

Greedy method	**Dynamic Programming**
make an optimal choice (without knowing solutions to subproblems) and then solve remaining subproblems	solve subproblems first, then use those solutions to make an optimal choice
solutions are top down	solutions are bottom up
Best choice does not depend on solutions to subproblems.	Choice at each step depends on solutions to subproblems
Make best choice at current time, then work on subproblems. Best choice does depend on choices so far	Many subproblems are repeated in solving larger problems. This repetition results in great savings when the computation is bottom up
Optimal Substructure: solution to problem contains within it optimal solutions to subproblems	**Optimal Substructure**: solution to problem contains within it optimal solutions to subproblems
Fractional knapsack: at each step, choose item with highest ratio	0-1 Knapsack: to determine whether to include item i for a given size, must consider best solution, at that size, with and without item i

Divide & Conquer Method vs Dynamic Programming

Divide & Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	•**1.**It involves the sequence of four steps:Characterize the structure of optimal solutions. •Recursively defines the values of optimal solutions. •Compute the value of optimal solutions in a Bottom-up minimum. •Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

0/1 KnapSack Problem

Ex :1- n=4,m=8 P={1,2,5,6} and W={2,3,4,5}

Using Tabular Method:

0/1 KnapSack Problem

Ex :1- n=4,m=8 P={1,2,5,6} and W={2,3,4,5}

Tabular Method:

		Capacity / Item	0	1	2	3	4	5	6	7	8
P	W	0									
1	2	1									
2	3	2									
5	4	3									
6	5	4									

0/1 KnapSack Problem

Ex :1- n=4,m=8 P={1,2,5,6} and W={2,3,4,5}

Tabular Method:

		Capacity / Item	0	1	2	3	4	5	6	7	8
P	W	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

0/1 KnapSack Problem

- For the zero row, no item is selected and so no weight is included into knapsack. So fill first row with all 0's and first column with all 0's
- For the first row consider first element. First element weight is 2. so fill second row second column with profit of the first item i.e 1. Only first element can be selected. So all the remaining columns values are 1 only, and left side column with previous value.
- For the second row select second element. Second element weight is 3. so fill 3^{rd} column in the 3 row with profit of the second item i.e 2. fill all the left side colmns with previous values. Here we must select both the items. So first two items weight is 5. total profit of first two item is 3. so fill column 5 with 3. fillleft side columns with previous valus. And right side columns with 3, because we can select first two items only.
- For the third row, select 3^{rd} item. Its weight is 4. so fill 4^{th} column with its profit i.e 5. here we select first 3 items. But we may not select all three. But we must identify the combinations with 3^{rd} item.

If we select 3^{rd} and 1^{st} items, total weight is 6, so fill 6^{th} column with total profit of 1^{st} and 3^{rd} items i.e 6 , in the same way select 3^{rd} and 2^{nd} items, total weight is 7 , so fill 7^{th} column with total profit of 2^{nd} and 3^{rd} items, i.e 7.

0/1 Knapsack Problem

- Fill the last column with 7 only, because we cannot select remaining items.
- For the 4^{th} row, select 4^{th} item , the weight of the item is 5, so fill 5^{th} column in 4^{th} row with the profit of the 4^{th} item, i.e 6. all the previous columns with the old values. Now select remaining items alongwith 4^{th} itam.

  - select 4^{th} item with 1^{st} item , the total weight of these two items is 7, so fill 7^{th} column with the total profit of these two items i.e 7.

  - select item 4 with second item, the total weight of these two items is 8, so fillthe 8^{th} column with these two items profit, i.e 8.

  - 6^{th} column with 5^{th} column value.

- After filling the entire row, now construct the solutionx1, x2, x3 and x4.

0/1 Knapsack Problem

- Formula for filling all the rows:

  V[i, w] =max { V[i-1, w], V[i-1,w-w[i]]+p[i] }

- V[4, 1] = max{ V[3, 1], V[3, 1-5] +6 }

  = max{ 0, v[3, -4] + 6}

   undefined..

So upto w=4 take the same values as previous row.

- V[4, 5] = max{ V[3, 5], V[3, 5-5] +6 }

  = max{ 5, v[3, 0] + 6}

  = max{ 5, 0 + 6} = max{ 5, 6} = 6

- V[4, 6] = max{ V[3, 6], V[3, 6-5] +6 }

  = max{ 6, v[3, 1] + 6}

  = max{ 6, 0 + 6} = max{ 6, 6} = 6

0/1 Knapsack Problem

- V[4, 7] = max{ V[3, 7], V[3, 7-5] +6 }
  = max{ 7, v[3, 2] + 6}
  = max{ 7, 1 + 6} = max{ 7, 7} = 7
- V[4, 8] = max{ V[3, 8], V[3, 8-5] +6 }
  = max{ 7, v[3, 3] + 6}
  = max{ 6, 2 + 6} = max{ 6, 8} = 8

Algorithm

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if wi ≤ w then
      if vi + c[i-1, w-wi] then
        c[i, w] = vi + c[i-1, w-wi]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

0/1 Knapsack Problem

- Select the maximum profit value , i.e. 8, which is there in 4th row, check whether 8 is there in 3rd row or not. Value is not there, means 4th row is included, **so x4=1**.
  - 4th row profit is 6. remaining profit is 8-6=2.
- 2 is there in row 3, check whether 2 is there in row 2 or not. Value is there in 2nd row also , means 3rd item is not included. **X3=0**.
- So 2 is there in row 2, check whether 2 is there in row 1 or not. No, value 2 is not there in row 1. so second item is included, **x2=1**.
  - so the remaining profit is 2-2=0
- 0 is there in row 1, check whether 0 is there in row 0 or not, yes row zero contain 0, so item 1 is not included, **x1=0.**
- So the solution is **: x1=0, x2=1, x3=0, x4=1.**
- Total profit obtained is = **p1 * x1+ p2 * x2 + p3 * x3 + p4 * x4**

= 1 *0 + 2 * 1 + 5 * 0 + 6 * 1= 8

Knapsack filled with weight = 2*0 + 3*1 + 4*0 + 5*1 = 8

0/1 KnapSack Problem

Ex :2- n=4,m=7 P={1,4,5,7} and W={1,3,4,5}

Using Tabular Method:

Solve

0/1 KnapSack Problem

Ex :3- n=4, m=8, P={2,3,1,4} and W={3,4,6,5}

Using Tabular Method:

Solve

Matrix Chain Multiplication

Matrix Chain Multiplication

Matrix Chain Multiplication is the optimization problem. It can be solved using dynamic programming. The problem is defined below:

Problem: In what order, n matrices A_1, A_2, A_3, …. A_n should be multiplied so that it would take a minimum number of computations to derive the result.

Cost of matrix multiplication: Two matrices are called compatible only if the number of columns in the first matrix and the number of rows in the second matrix are the same. Matrix multiplication is possible only if they are compatible. Let A and B be two compatible matrices of dimensions p x q and q x r

Matrix Chain Multiplication

- To compute m[i, j] when i < j, we take advantage of the structure of an optimal solution of the first step. Let us assume that the optimal parenthesizations split the product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then m[i, j] is equal to the minimum cost for computing the subproducts $A_{j \ldots k}$ and $A_{k+1 \ldots j}$ plus the cost of multiplying these two matrices together.
- The optimal substructure is defined as

$$m[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + d_{i-1} \times d_k \times d_j\} & , \text{if } i < j \end{cases}$$

- Where $d = \{d_0, d_1, d_2, \ldots, d_n\}$ is the vector of matrix dimensions.

Matrix Chain Multiplication

- Example: Find a minimum number of multiplications required to multiply: A [1 × 5], B [5 × 4], C [4 × 3], D [3 × 2], and E [2 × 1]. Also, give optimal parenthesization.

Or

- We are given the sequence {1, 5, 4, 3, 2, and 1}. Find a minimum number of multiplications required to multiply

Matrix Chain Multiplication

- **Example: Find a minimum number of multiplications required to multiply:**

A [1 × 5], B [5 × 4], C [4 × 3], D [3 × 2], and E [2 × 1]. Also, give optimal parenthesization.

- **Solution:**
- Here d = {d_0, d_1, d_2, d_3, d_4, d_5} = {1, 5, 4, 3, 2, 1}
- Optimal substructure of matrix chain multiplication is

$$m[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + d_{i-1} \times d_k \times d_j\} & , \text{if } i < j \end{cases}$$

- Single matrix
- m[i, j] = 0, for i = 1 to 5
- m[1, 1] = m[2 , 2] = m[3, 3] = m[4, 4] = m[5, 5] = 0

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

- **Two Matrix**
- **m[1, 2]** = {m[1, 1] + m[2, 2] + $d_0 \times d_1 \times d_2$}
  = {0 + 0 + 1 × 5 × 4}
  = 20
- **m[2, 3]** = {m[2, 2] + m[3, 3] + $d_1 \times d_2 \times d_3$}
  = {0 + 0 + 5 × 4 × 3}
  = 60
- **m[3, 4]** = {m[3, 3] + m[4, 4] + $d_2 \times d_3 \times d_4$}
  = {0 + 0 + 4 × 3 × 2}
  = 24
- **m[4, 5]** = {m[4, 4] + m[5, 5] + $d_3 \times d_4 \times d_5$}
  = {0 + 0 + 3 × 2 × 1}
  = 6

	1	2	3	4	5
1	0	20			
2		0	60		
3			0	24	
4				0	6
5					0

- **Three Matrix**

$$m[1,3] = \min\begin{Bmatrix} m[1,1] + m[2,3] + d_0 \times d_1 \times d_3 \\ m[1,2] + m[3,3] + d_0 \times d_2 \times d_3 \end{Bmatrix}$$

$$= \min\begin{Bmatrix} (0 + 60 + 1 \times 5 \times 3) \\ (20 + 0 + 1 \times 4 \times 3) \end{Bmatrix} = \min\begin{Bmatrix} 75 \\ 32 \end{Bmatrix} = 32$$

$$m[2,4] = \min\begin{Bmatrix} m[2,2] + m[3,4] + d_1 \times d_2 \times d_4 \\ m[2,3] + m[4,4] + d_1 \times d_3 \times d_4 \end{Bmatrix}$$

$$= \min\begin{Bmatrix} (0 + 24 + (5 \times 4 \times 2)) \\ (60 + 0 + (5 \times 3 \times 2)) \end{Bmatrix} = \min\begin{Bmatrix} 64 \\ 90 \end{Bmatrix} = 64$$

$$m[3,5] = \min\begin{Bmatrix} m[3,3] + m[4,5] + d_2 \times d_3 \times d_5 \\ m[3,4] + m[5,5] + d_2 \times d_4 \times d_5 \end{Bmatrix}$$

$$= \min\begin{Bmatrix} (0 + 6 + (4 \times 3 \times 1)) \\ (24 + 0 + (4 \times 2 \times 1)) \end{Bmatrix} = \min\begin{Bmatrix} 18 \\ 32 \end{Bmatrix} = 18$$

	1	2	3	4	5
1	0	20	32		
2		0	60	64	
3			0	24	18
4				0	6
5					0

- **Four Matrix**

$$m[1,4] = \min\begin{Bmatrix} m[1,1] + m[2,4] + d_0 \times d_1 \times d_4 \\ m[1,2] + m[3,4] + d_0 \times d_2 \times d_4 \\ m[1,3] + m[4,4] + d_0 \times d_3 \times d_4 \end{Bmatrix}$$

$$= \min\begin{Bmatrix} (0 + 64 + (1 \times 5 \times 2)) \\ (20 + 24 + (1 \times 4 \times 2)) \\ (32 + 0 + (1 \times 3 \times 2)) \end{Bmatrix} = \min\begin{Bmatrix} 74 \\ 52 \\ 38 \end{Bmatrix} = 38$$

$$m[2,5] = \min\begin{Bmatrix} m[2,2] + m[3,5] + d_1 \times d_2 \times d_5 \\ m[2,3] + m[4,5] + d_1 \times d_3 \times d_5 \\ m[2,4] + m[5,5] + d_1 \times d_4 \times d_5 \end{Bmatrix}$$

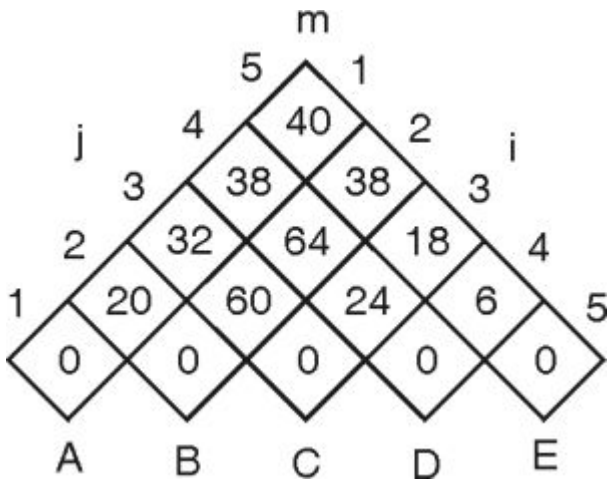
$$= \min\begin{Bmatrix} (0 + 18 + (5 \times 4 \times 1)) \\ (60 + 6 + (5 \times 3 \times 1)) \\ (64 + 0 + (5 \times 2 \times 1)) \end{Bmatrix} = \min\begin{Bmatrix} 38 \\ 81 \\ 74 \end{Bmatrix} = 38$$

	1	2	3	4	5
1	0	20	32	38	
2		0	60	64	38
3			0	24	18
4				0	6
5					0

- **Five Matrix**

$$m[1,5] = \min \begin{Bmatrix} m[1,1] + m[2,5] + d_0 \times d_1 \times d_5 \\ m[1,2] + m[3,5] + d_0 \times d_2 \times d_5 \\ m[1,3] + m[4,5] + d_0 \times d_3 \times d_5 \\ m[1,4] + m[5,5] + d_0 \times d_4 \times d_5 \end{Bmatrix}$$

$$= \min \begin{Bmatrix} (0 + 38 + (1 \times 5 \times 1)) \\ (20 + 18 + (1 \times 4 \times 1)) \\ (32 + 6 + (1 \times 3 \times 1)) \\ (38 + 0 + (1 \times 2 \times 1)) \end{Bmatrix} = \min \begin{Bmatrix} 43 \\ 42 \\ 41 \\ 40 \end{Bmatrix} = 40$$



	1	2	3	4	5
1	0	20	32	38	40
2		0	60	64	38
3			0	24	18
4				0	6
5					0

- <u>The optimal sequence is</u>
- _ 5th matrix (A*B*C*D) E
- 4th matrix (A*B*C) (D) E
- 3rd matrix (A*B) * (C) * (D) * E

- The optimal answer is (A*B) * (C) * (D) * E

  Or

  (1*2) * (3) * (4) * 5

Algorithm for MATRIX-CHAIN-ORDER

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n     // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

Complexity analysis for MATRIX-CHAIN-ORDER

- **Analysis:**
  - There are three nested loops.
  - Each loop executes a maximum n times.

1. l, length, O (n) iterations.
2. i, start, O (n) iterations.
3. k, split point, O (n) iterations

- Body of loop constant complexity
- **Total Complexity is: O (n^3)**

Matrix Chain Multiplication - Question

- **Solve : We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], $0 \leq i, j \leq 5$. We know M [i, i] = 0 for all i.**

- **Solve it**

Longest Common Subsequence

- The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.
- If $S1$ and $S2$ are the two given sequences then, Z is the common subsequence of $S1$ and $S2$ if Z is a subsequence of both $S1$ and $S2$. Furthermore, Z must be a **strictly increasing sequence** of the indices of both $S1$ and $S2$.
- In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

Longest Common Subsequence

Problem

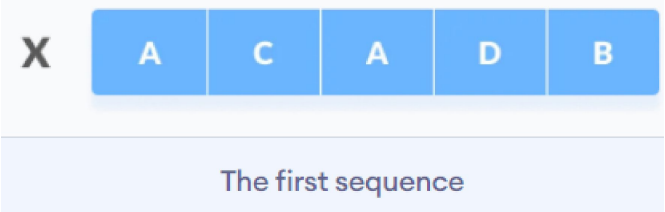
- Let us understand LCS with an example.
- If

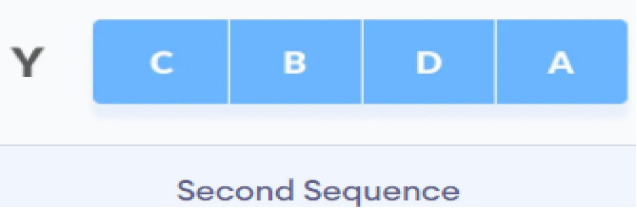
  S1 = {B, C, D, A, A, C, D}

  S2 = {A, C, D, B, A, C}
- Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ...

- Among these subsequences, {C, D, A, C} is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

- **Problem Using Dynamic Programming to find the LCS**
- Let us take two sequences:

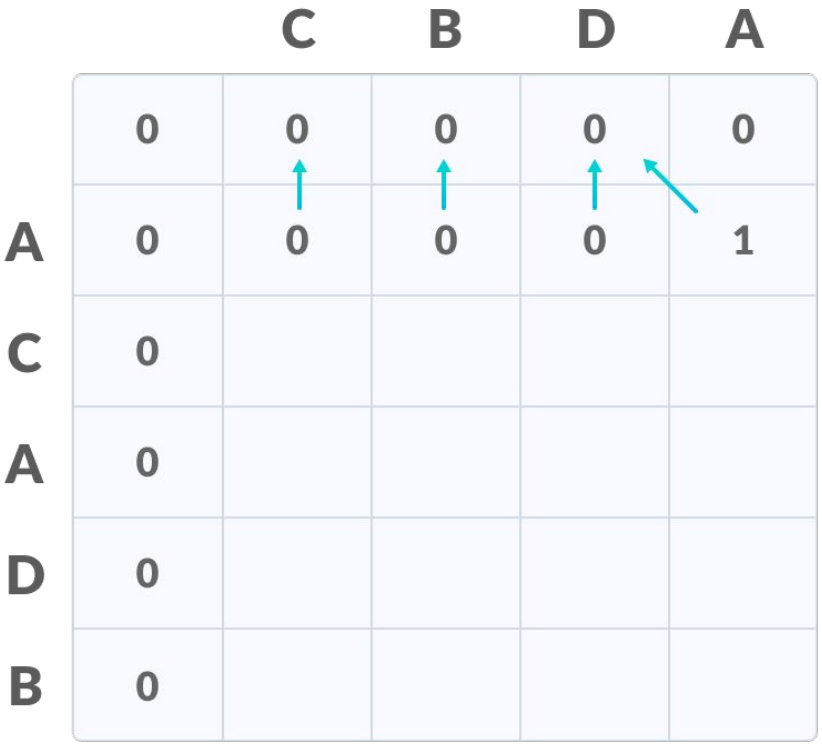


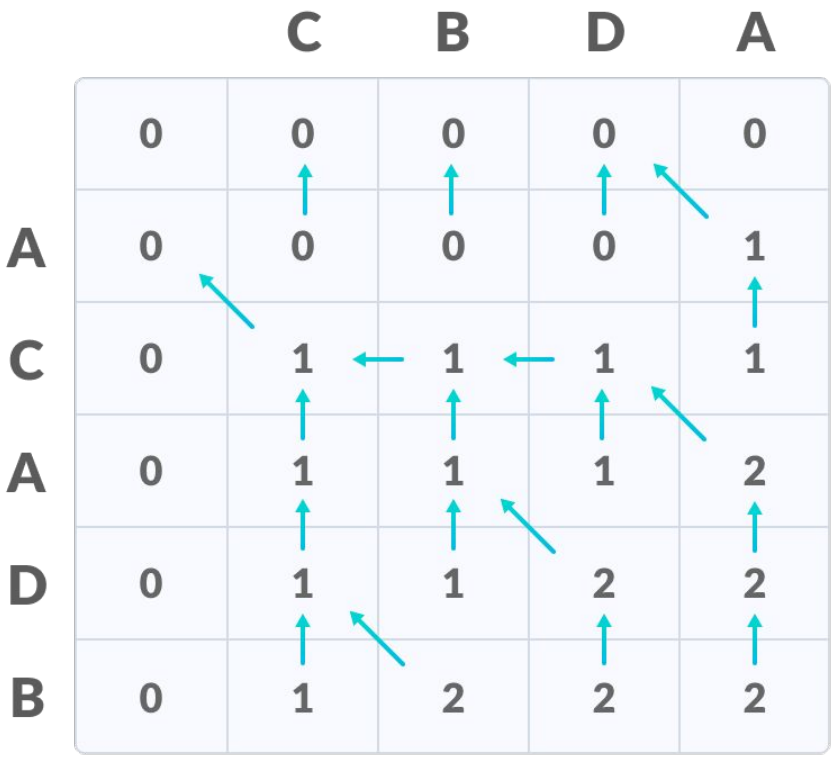


- **Solution**
- The following steps are followed for finding the longest common subsequence.
- Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

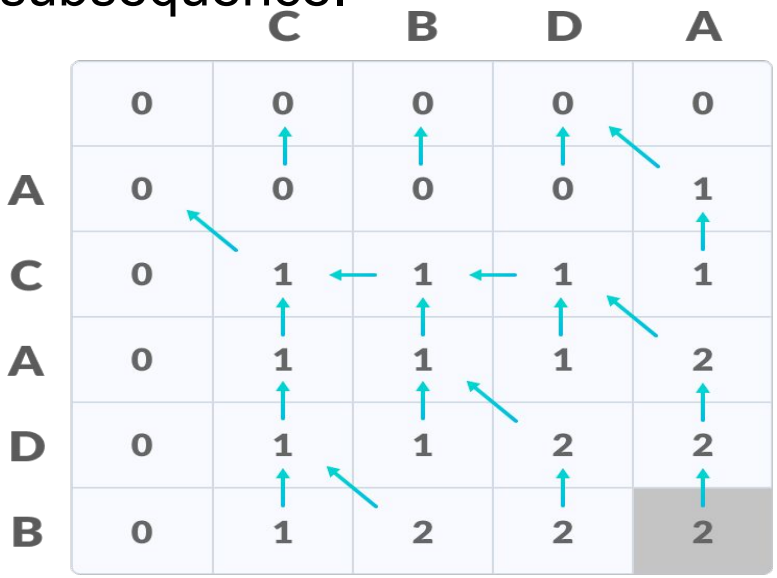
		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

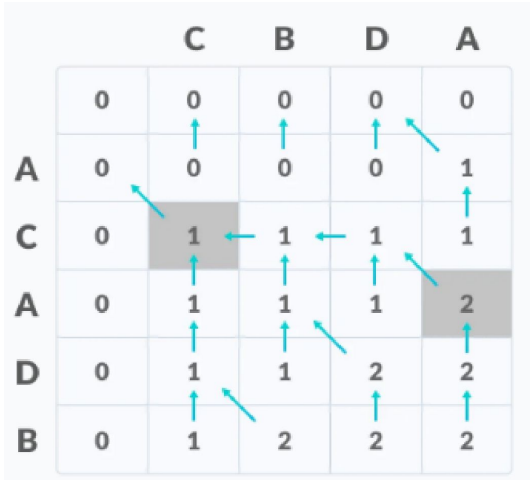
1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.





- The value in the last row and the last column is the length of the longest common subsequence.





- Thus, the longest common subsequence is **CA.**

Algorithm of Longest Common Subsequence

Suppose X and Y are the two given sequences

Initialize a table of LCS having a dimension of X.length * Y.length

XX.label = X

YY.label = Y

LCS[0][] = 0

LCS[][0] = 0

Loop starts from the LCS[1][1]

Now we will compare X[i] and Y[j]

if X[i] is equal to Y[j] then

LCS[i][j] = 1 + LCS[i-1][j-1]

Point an arrow LCS[i][j]

Else

LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])

Analysis of Longest Common Subsequence

- If we use the dynamic programming approach, then the number of function calls are reduced. The dynamic programming approach stores the result of each function call so that the result of function calls can be used in the future function calls without the need of calling the functions again.
- In the above dynamic algorithm, the results obtained from the comparison between the elements of x and the elements of y are stored in the table so that the results can be stored for the future computations.
- The time taken by the dynamic programming approach to complete a table is O(m*n) and the time taken by the recursive algorithm is $2^{max(m, n)}$.

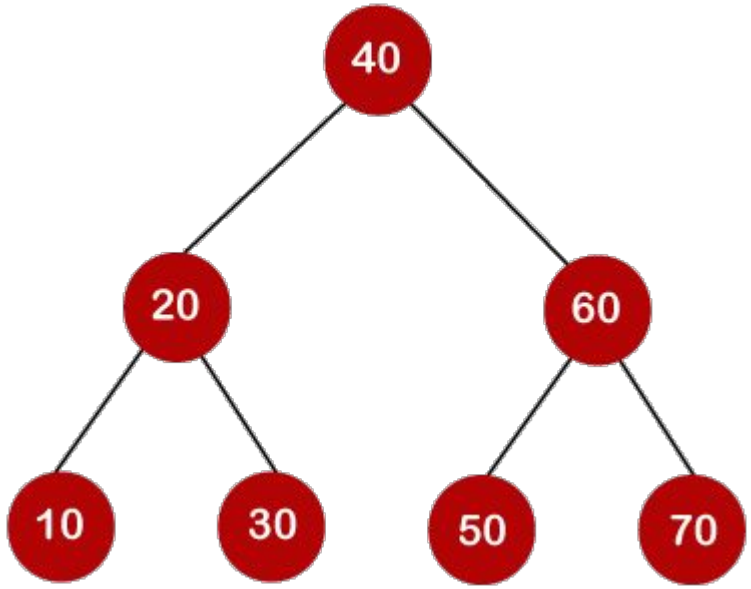
Optimal binary search tree (OBST)using dynamic programming

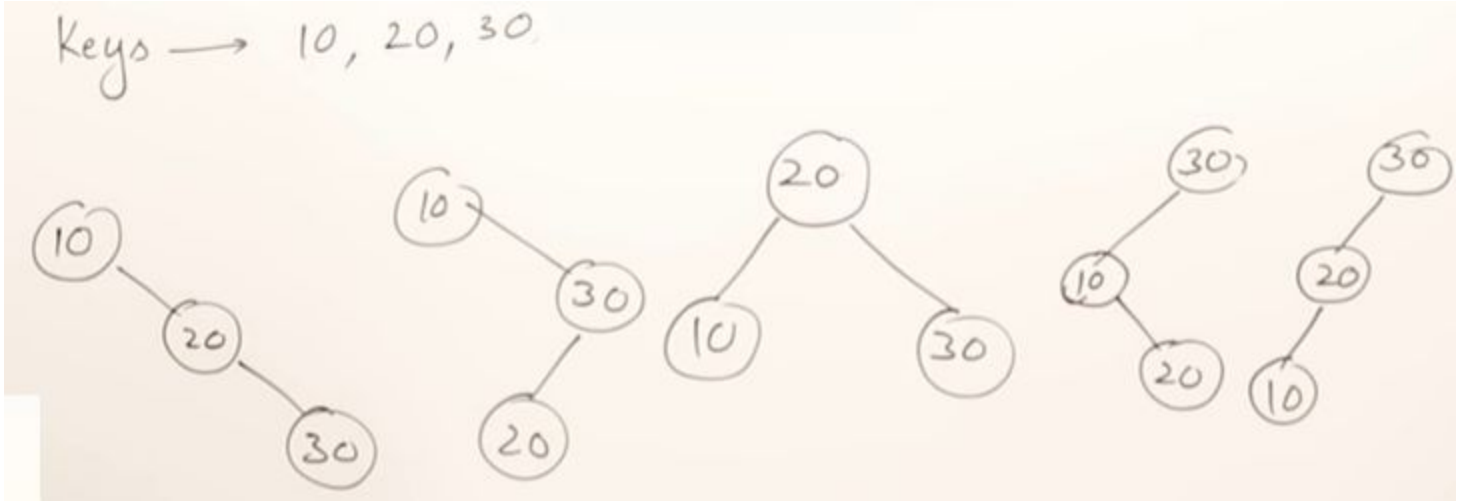
Optimal binary search tree (OBST)

- *A binary search tree T is a binary tree, either it is empty or each node in the tree contains an identifier and,*
  - All identifiers in the left subtree of T are *less than* the identifier in the *root* node T.
  - All identifiers in the right subtree are *greater than* the identifier in the *root* node T.
  - The *left and right* subtree of T are also *binary search* trees.

- We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node.
- The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications.
- The overall cost of searching a node should be less.
- The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST.
- There is one way that can reduce the cost of a binary search tree is known as an **optimal binary search tree**.

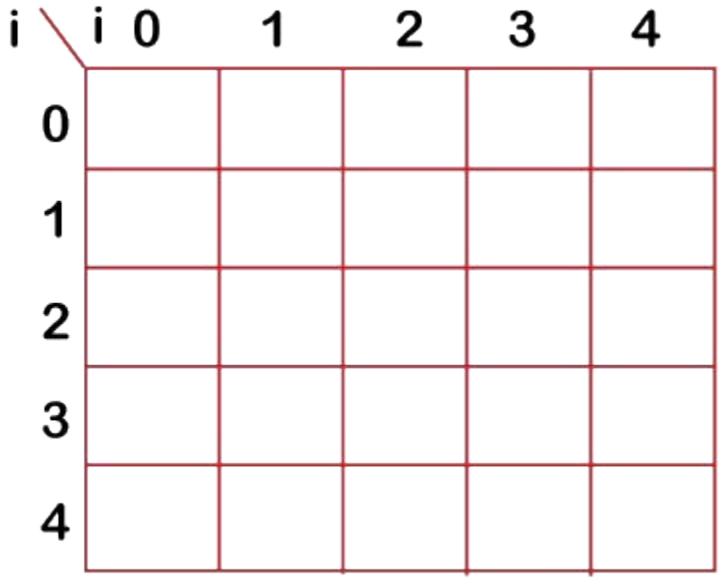
- **Let's understand through an example.**
- If the keys are 10, 20, 30, 40, 50, 60, 70




Keys → 10, 20, 30
10
20
30
10
30
20
20
10
30
30
10
20
30
20
10

- Dynamic Approach
- Consider the below table, which contains the keys and frequencies.

	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3



- **First, we will calculate the values where j-i is equal to zero.**
- When i=0, j=0, then j-i = 0
- When i = 1, j=1, then j-i = 0
- When i = 2, j=2, then j-i = 0
- When i = 3, j=3, then j-i = 0
- When i = 4, j=4, then j-i = 0
- Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0
- **Now we will calculate the values where j-i equal to 1.**
- When j=1, i=0 then j-i = 1
- When j=2, i=1 then j-i = 1
- When j=3, i=2 then j-i = 1
- When j=4, i=3 then j-i = 1
- Now to calculate the cost, we will consider only the jth value.
- The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).
- The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).
- The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)
- The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

- **Now we will calculate the values where j-i = 2**
- When j=2, i=0 then j-i = 2
- When j=3, i=1 then j-i = 2
- When j=4, i=2 then j-i = 2
- In this case, we will consider two keys.
- When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:
- In the first binary tree, cost would be: 4*1 + 2*2 = 8
- In the second binary tree, cost would be: 4*2 + 2*1 = 10
- The minimum cost is 8; therefore, c[0,2] = 8
- When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:
- In the first binary tree, cost would be: 1*2 + 2*6 = 14
- In the second binary tree, cost would be: 1*6 + 2*2 = 10
- The minimum cost is 10; therefore, c[1,3] = 10
- When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:
- In the first binary tree, cost would be: 1*6 + 2*3 = 12
- In the second binary tree, cost would be: 1*3 + 2*6 = 15
- The minimum cost is 12, therefore, c[2,4] = 12

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

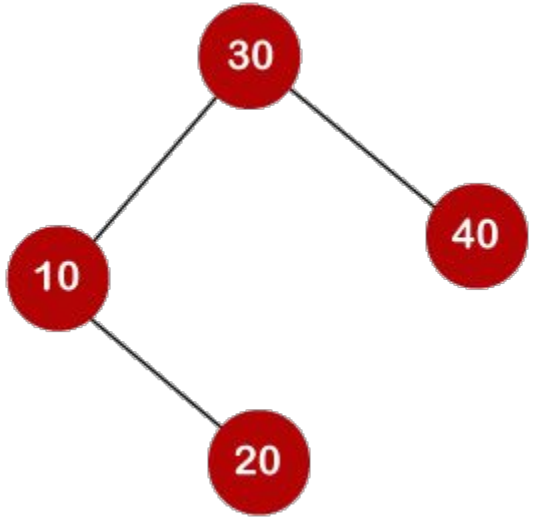
i \ 1	0	1	2	3	4
0	0	4	8^{1}		
1		0	2	10^{3}	
2			0	6	12^{3}
3				0	3
4					0

•Similarly

i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Final answer is



Algorithm search(x)

```
{
    found:=false;
    t:=tree;
    while( (t≠0) and not found ) do
    {
        if( x=t->data )  then found:=true;
        else if( x<t->data ) then t:=t->lchild;
            else  t:=t->rchild;
    }
     if( not found ) then return 0;
     else  return 1;
}
```

Time complexity for over all problem.

Ploynomial Time

Linear Search — n

Binary Search — $\log n$

Insertion Sort — n^2

MergeSort — $n \log n$

Matrix Multiplication — n^3

Expontial Time

0/1 Knapsack — 2^n

Traveling SP — 2^n

Sum of Subsets — 2^n

Graph Coloring — 2^n

Hamiltonian Cycle — 2^n