GREEDY ALGORITHMS

Definition

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

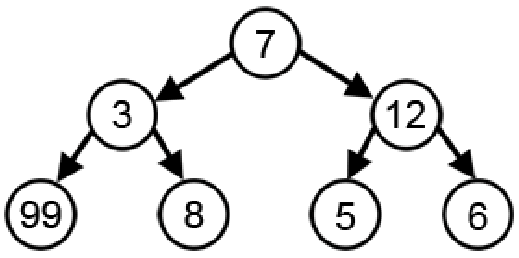
Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Drawback of Greedy Approach

As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm.

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.



To reach the largest sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

Therefore, greedy algorithms do not always give an optimal/feasible solution.

Explanation

Greedy Algorithm

1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

Let's now use this algorithm to solve a problem.

Example – Greedy Approach

Problem: You must make a change of an amount using the smallest possible number of coins.

Amount: ₹18

Available coins are:

- ₹5 coin
- ₹2 coin
- ₹1 coin

There is no limit to the number of each coin you can use.

Solution:

1. Create an empty solution-set = { }. Available coins are {5, 2, 1}.
2. We are supposed to find the sum = 18. Let's start with sum = 0.
3. Always select the coin with the largest value (i.e. 5) until the sum > 18. (When we select the largest value at each step, we hope to reach the destination faster. This concept is called **greedy choice property**.)
4. In the first iteration, solution-set = {5} and sum = 5.
5. In the second iteration, solution-set = {5, 5} and sum = 10.
6. In the third iteration, solution-set = {5, 5, 5} and sum = 15.
7. In the fourth iteration, solution-set = {5, 5, 5, 2} and sum = 17. (We cannot select 5 here because if we do so, sum = 20 which is greater than 18. So, we select the 2nd largest item which is 2.)
8. Similarly, in the fifth iteration, select 1. Now sum = 18 and solution-set = {5, 5, 5, 2, 1}.

Applications

Greedy algorithms typically (but not always) fail to find the globally optimal solution because they usually do not operate exhaustively on all the data. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees and the algorithm for finding optimum Huffman trees.

Greedy algorithms appear in the network routing as well. Using greedy routing, a message is forwarded to the neighbouring node which is "closest" to the destination.

Examples

Fractional Knapsack Problem

Here, we will look at one form of the knapsack problem. The knapsack problem involves deciding which subset of items you should take from a set of items if you want to optimize some value: perhaps the worth of the items, the size of the items, or the ratio of worth to size.

In this problem, we will assume that we can take a fractional part of an item. Our knapsack has a fixed size, and we want to optimize the worth of the items we take, so we must choose the items we take with care.

(In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.)

Problem Statement: Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Here is the list of items and their worth.

Item	Price	Weight
Laptop	60	10
PlayStation	100	20
Basketball	120	30

Our knapsack can hold at most 50 units of space.

Which items do we choose to optimize for price?

Input:
Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;

Output:
Maximum possible value = 240
by taking items of weight 10 and 20 kg and 2/3 fraction
*of 30 kg. Hence total price will be 60 + 100 + (2/3) * (120) = 240*

Solution:

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.

An **efficient solution** is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can, which will always be the optimal solution to this problem.

Implementation:

```
import java.util.Arrays;
import java.util.Comparator;

// Greedy approach
public class FractionalKnapSack {
   // function to get maximum value
   private static double getMaxValue(int[] wt, int[] val,
                                     int capacity)
   {
      ItemValue[] iVal = new ItemValue[wt.length];

      for (int i = 0; i < wt.length; i++) {
         iVal[i] = new ItemValue(wt[i], val[i], i);
      }

      // sorting items by value;
      Arrays.sort(iVal, new Comparator<ItemValue>() {
         @Override
         public int compare(ItemValue o1, ItemValue o2)
         {
            return o2.cost.compareTo(o1.cost);
         }
      });

      double totalValue = 0d;

      for (ItemValue i : iVal) {

         int curWt = (int)i.wt;
         int curVal = (int)i.val;

         if (capacity - curWt >= 0) {
            // this weight can be picked while
            capacity = capacity - curWt;
            totalValue += curVal;
         }
```

```
        else {
            // item cant be picked whole
            double fraction
                = ((double)capacity / (double)curWt);
            totalValue += (curVal * fraction);
            capacity
                = (int)(capacity - (curWt * fraction));
            break;
        }
    }

    return totalValue;
}

// item value class
static class ItemValue {
    Double cost;
    double wt, val, ind;

    // item value function
    public ItemValue(int wt, int val, int ind)
    {
        this.wt = wt;
        this.val = val;
        this.ind = ind;
        cost = new Double((double)val / (double)wt);
    }
}

// Driver code
public static void main(String[] args)
{
    int[] wt = { 10, 40, 20, 30 };
    int[] val = { 60, 40, 100, 120 };
    int capacity = 50;

    double maxValue = getMaxValue(wt, val, capacity);

    // Function call
    System.out.println("Maximum value we can obtain = "
                    + maxValue);
    }
}
```

As main time taking step is sorting, the problem can be solved in **O(n log n)**.

Activity Selection Problem

Activity selection problem is a problem in which a person has a list of works to do. Each of the activities has a starting time and ending time. We need to schedule the activities in such a way the person can complete a maximum number of activities. Since the timing of the activities can collapse, so it might not be possible to complete all the activities and thus we need to schedule the activities in such a way that the maximum number of activities can be finished.

Problem Statement:

Given **N** activities with their **start** time and **end** time. The task is to find the solution set having **a maximum number of non-conflicting** activities that can be executed within the given time, assuming only a single activity can be performed at a given time.

Examples:

Input: start[] = [10, 12, 20}]
          end[] = [20, 25, 30]
Output: [0, 2]

Input: start[] = [1, 3, 0, 5, 8, 5]
          finish[] = [2, 4, 6, 7, 9, 9]
Output: [0, 1, 3, 4]

Algorithm:

Two activities **A1** and **A2** are said to be non-conflicting if **S1 >= F2** or **S2 >= F1,** where **S** and **F** denote the start and end time respectively.

Since we need to maximize the maximum number of activities. The idea would be to choose the activity with the least finish time. Finishing the smallest activities would help adjust the remaining tasks.

- Sort the arrays according to their finish time, in case they are not sorted.
- Choose the first activity from the array and insert it into the **sol** array.

- If the start time of **i**th activity is greater than or equal to the finish time of the **(i – 1)**th activity, print the **i**th activity, since it is ready for execution.
- Repeat the above steps till the end of the array.

Implementation:

```
public static void printMaxActivities(int s[], int f[], int n)
{
   int i, j;

   System.out.print("Following activities are selected : n");
   i = 0;
   System.out.print(i+" ");
   for (j = 1; j < n; j++)
   {
       if (s[j] >= f[i])
       {
           System.out.print(j+" ");
           i = j;
        }
    }
}
```

Time Complexity:

- **Case 1:** O(N), in case, the given array is sorted according to their finish times, where N is total steps.
- **Case 2:** O(N logN), in case, the given array is not sorted according to their finish times, where N is total steps.

Space Complexity: O(1), since no extra space is used.

Policemen Catch Thieves

Problem Statement:

Given an array of size n that has the following specifications:

1. Each element in the array contains either a policeman or a thief.
2. Each policeman can catch only one thief.
3. A policeman cannot catch a thief who is more than K units away from the policeman.

We need to find the maximum number of thieves that can be caught.

Examples:

Input: arr[] = {'P', 'T', 'T', 'P', 'T'},

k = 1.

Output: 2.

(Here maximum 2 thieves can be caught, first policeman catches first thief and second police-man can catch either second or third thief.)

Input: arr[] = {'T', 'T', 'P', 'P', 'T', 'P'},

k = 2.

Output: 3.

Input: arr[] = {'P', 'T', 'P', 'T', 'T', 'P'},

k = 3.

Output: 3.

Approach:

A **brute force** approach would be to check all feasible sets of combinations of police and thief and return the maximum size set among them. Its time complexity is exponential, and it can be optimized if we observe an important property.
An **efficient** solution is to use a greedy algorithm.

Algorithm:

We can observe that thinking irrespective of the policeman and focusing on just the allotment works:

- Get the lowest index of policeman p and thief t. Make an allotment if |p-t| <= k and increment to the next p and t found.
- Otherwise increment min(p, t) to the next p or t found.
- Repeat above two steps until next p and t are found.
- Return the number of allotments made.

Implementation:

```
import java.util.*;
import java.io.*;

class PolicemenCatchThieves
{
    // Returns maximum number of thieves
    // that can be caught.
    static int policeThief(char arr[], int n, int k)
    {
        int res = 0;
        ArrayList<Integer> thi = new ArrayList<Integer>();
        ArrayList<Integer> pol = new ArrayList<Integer>();

        // store indices in the ArrayList
        for (int i = 0; i < n; i++) {
            if (arr[i] == 'P')
            pol.add(i);
            else if (arr[i] == 'T')
            thi.add(i);
        }

        // track lowest current indices of
        // thief: thi[l], police: pol[r]
        int l = 0, r = 0;
        while (l < thi.size() && r < pol.size()) {

            // can be caught
            if (Math.abs(thi.get(l) - pol.get(r)) <= k) {
            res++;
            l++;
            r++;
            }

            // increment the minimum index
            else if (thi.get(l) < pol.get(r))
                l++;
            else
                r++;
        }
        return res;
    }

    // Driver program
    public static void main(String args[])
    {
        int k, n;
        char arr1[] =new char[] { 'P', 'T', 'T',
                                 'P', 'T' };
        k = 2;
        n = arr1.length;
        System.out.println("Maximum thieves caught: "
                          +policeThief(arr1, n, k));

        char arr2[] =new char[] { 'T', 'T', 'P', 'P',
                                 'T', 'P' };
        k = 2;
        n = arr2.length;
        System.out.println("Maximum thieves caught: "
                          +policeThief(arr2, n, k));

        char arr3[] = new char[]{ 'P', 'T', 'P', 'T',
                                 'T', 'P' };
        k = 3;
        n = arr3.length;
        System.out.println("Maximum thieves caught: "
                          +policeThief(arr3, n, k));
    }
}
```

Maximum Product Subset of an Array

Problem Statement:

Given an array a, we must find maximum product possible with the subset of elements present in the array. The maximum product can be single element also.

Examples:

Input: a[] = { -1, -1, -2, 4, 3 }

Output: 24

Explanation: Maximum product will be (-2 * -1 * 4 * 3) = 24

Input: a[] = { -1, 0 }

Output: 0

Explanation: 0(single element) is maximum product possible

Algorithm:

A **simple solution** is to generate all subsets, find product of every subset and return maximum product.
A **better solution** is to use the below facts.

1. If there are even number of negative numbers and no zeros, result is simply product of all
2. If there are odd number of negative numbers and no zeros, result is product of all except the negative integer with the least absolute value.
3. If there are zeros, result is product of all except these zeros with one exceptional case. The exceptional case is when there is one negative number, and all other elements are 0. In this case, result is 0.

Implementation:

```
class MaximuProductSubarray {
  static int maxProductSubset(int a[], int n) {
    if (n == 1) {
      return a[0];
    }

    // Find count of negative numbers, count
    // of zeros, negative number
    // with least absolute value
    // and product of non-zero numbers
    int max_neg = Integer.MIN_VALUE;
    int count_neg = 0, count_zero = 0;
    int prod = 1;
    for (int i = 0; i < n; i++) {

      // If number is 0, we don't
      // multiply it with product.
      if (a[i] == 0) {
          count_zero++;
          continue;
      }

      // Count negatives and keep
      // track of negative number
      // with least absolute value.
      if (a[i] < 0) {
          count_neg++;
          max_neg = Math.max(max_neg, a[i]);
      }

      prod = prod * a[i];
    }

    // If there are all zeros
    if (count_zero == n) {
      return 0;
    }

    // If there are odd number of
    // negative numbers
    if (count_neg % 2 == 1) {

      // Exceptional case: There is only
      // negative and all other are zeros
      if (count_neg == 1
              && count_zero > 0
              && count_zero + count_neg == n) {
          return 0;
      }

      // Otherwise result is product of
      // all non-zeros divided by
      //negative number with
      // least absolute value.
      prod = prod / max_neg;
    }
```

```
        return prod;
    }

    // Driver Code
    public static void main(String[] args) {
        int a[] = {-1, -1, -2, 4, 3};
        int n = a.length;
        System.out.println(maxProductSubset(a, n));

    }
}
```

Time Complexity: O(n)

Space Complexity: O(1)

Problem Statements

Huffman Encoding and Decoding

Explanation:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

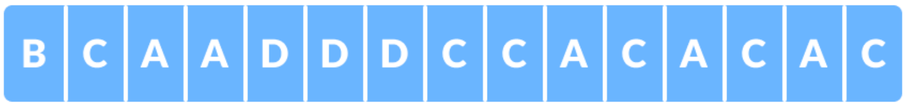
There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Example:

Encoding

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

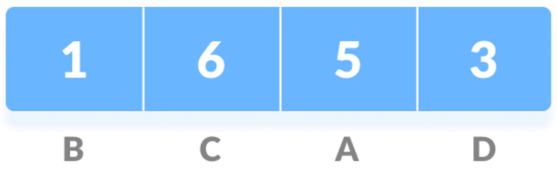


Initial string

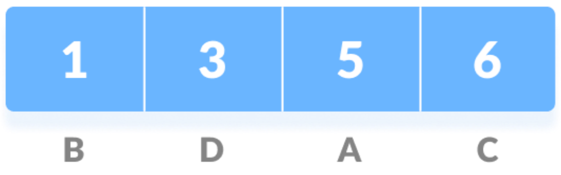
Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits is required to send this string.

Using the Huffman Coding technique, we can compress the string to a smaller size.

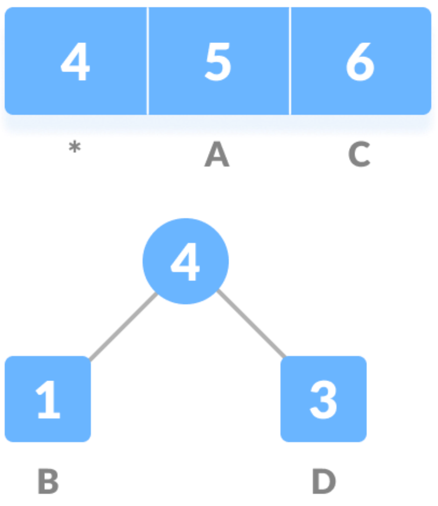
Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.



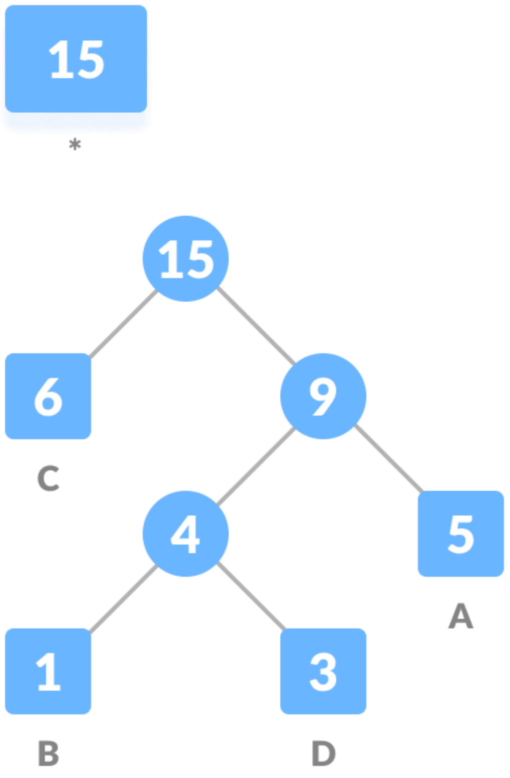
Frequency of string



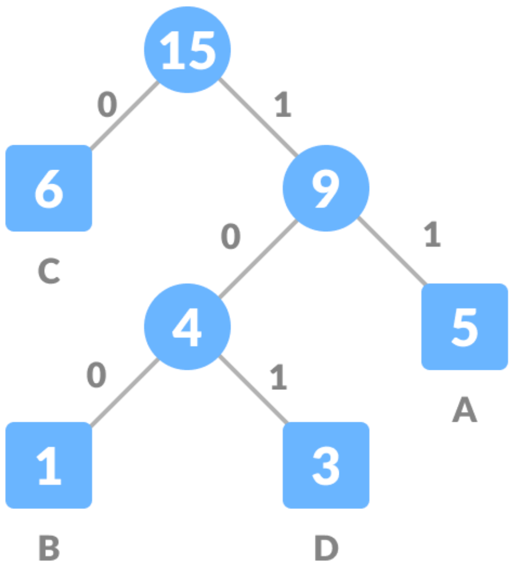
Characters sorted according to frequency



Getting the sum of the least numbers



Repeat the steps for all the characters



Assign 0 to the left edge and 1 to the right edge

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

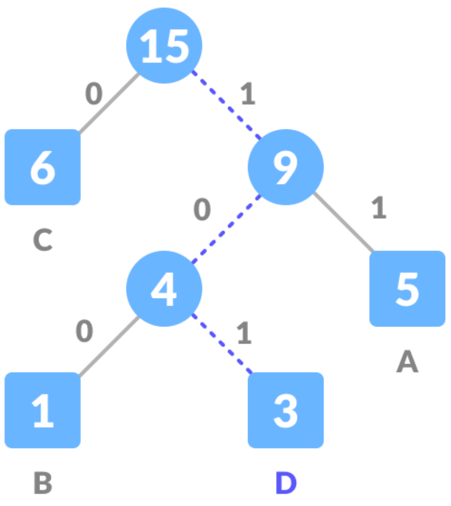
Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

Decoding

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



Decoding

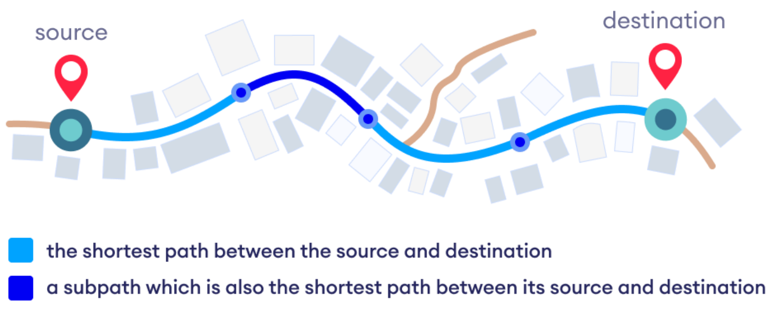
Dijkstra’s Algorithm

Explanation:

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

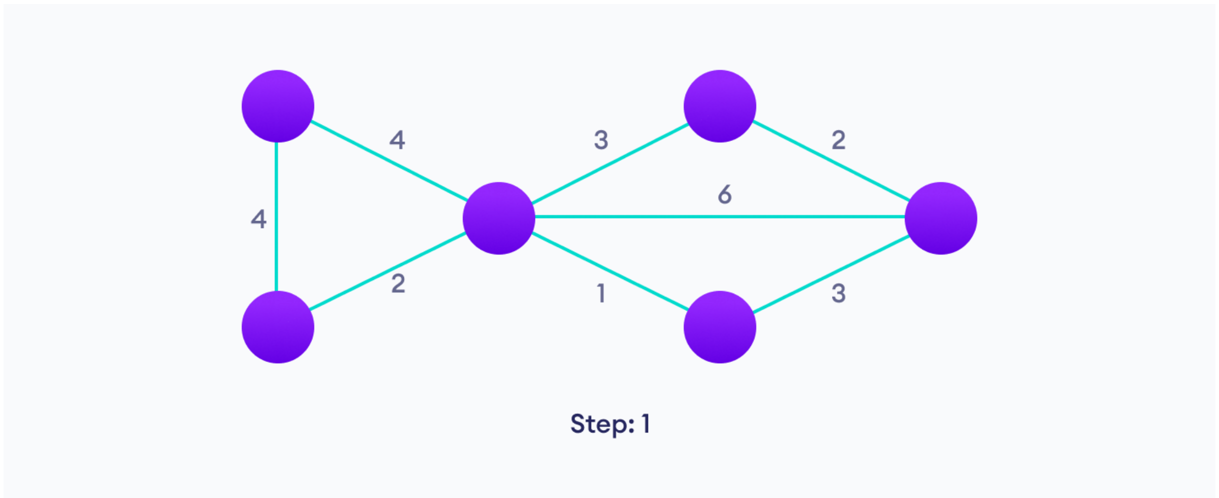
Dijkstra's Algorithm works on the basis that any sub path B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.


source
destination
the shortest path between the source and destination
a subpath which is also the shortest path between its source and destination

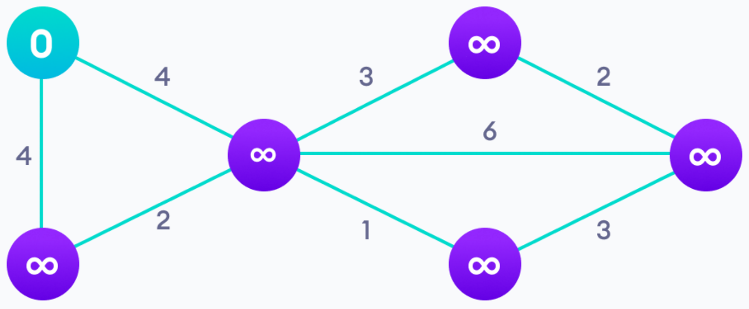
Each sub path is the shortest path

Dijkstra used this property in the opposite direction i.e., we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest sub path to those neighbours.

Example:

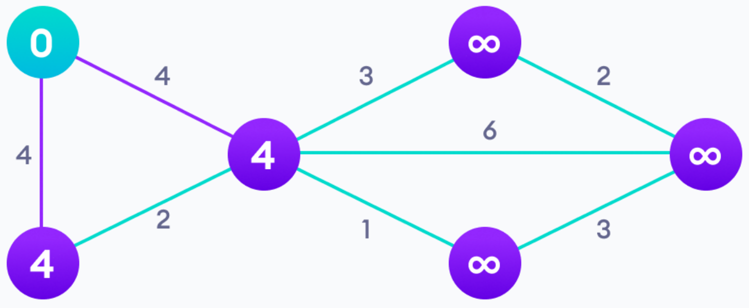


Start with a weighted graph



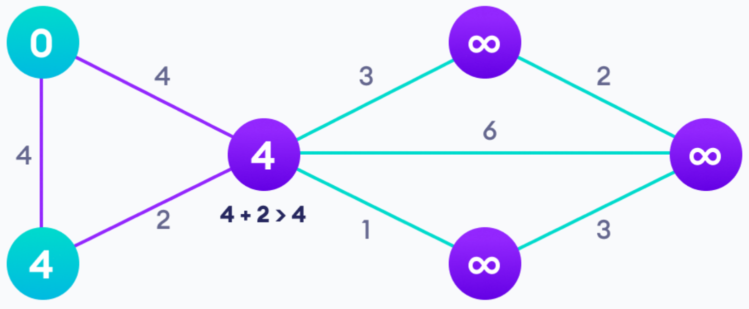
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



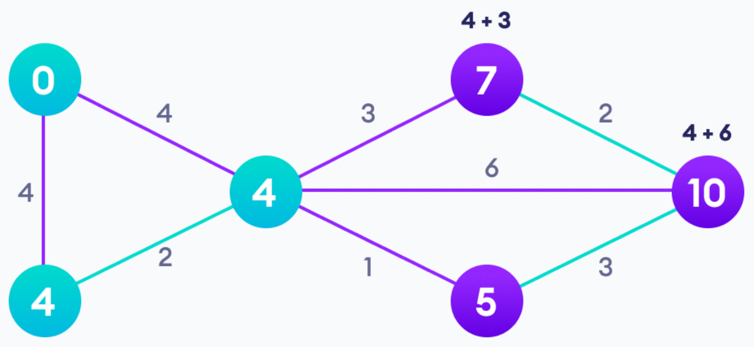
Step: 3

Go to each vertex and update its path length



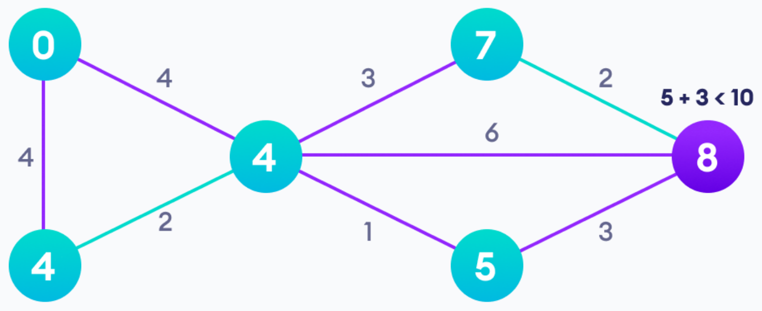
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



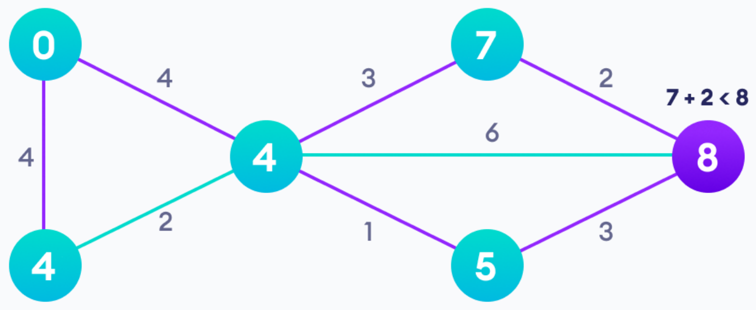
Step: 5

Avoid updating path lengths of already visited vertices



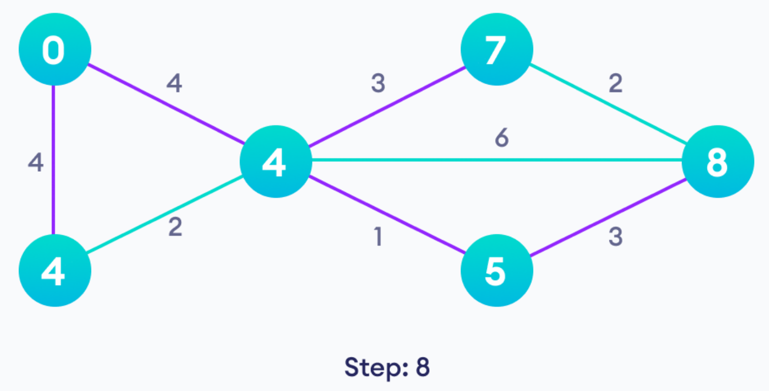
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Repeat until all the vertices have been visited

Kruskal's Minimum Spanning Tree

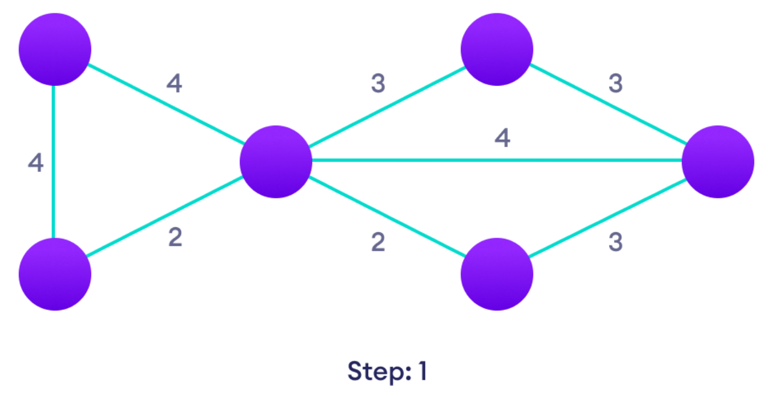
Explanation:

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

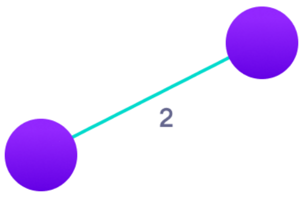
- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

Example:

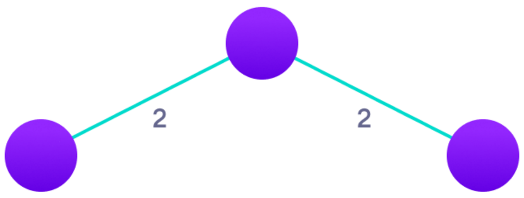


Start with a weighted graph



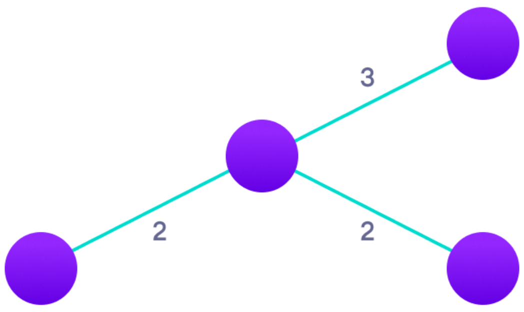
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



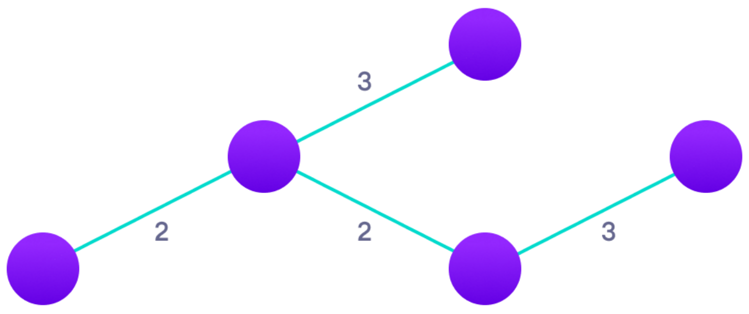
Step: 3

Choose the next shortest edge and add it



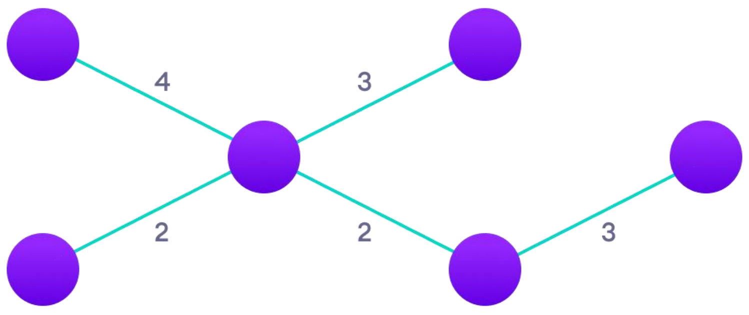
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Prim’s Minimum Spanning Tree

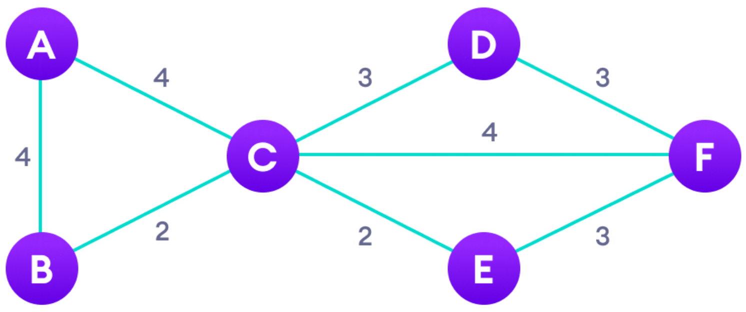
Explanation:

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

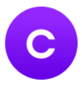
We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

Example:



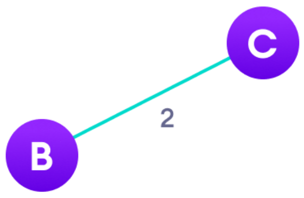
Step: 1

Start with a weighted graph



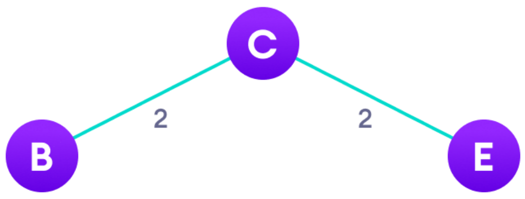
Step: 2

Choose a vertex



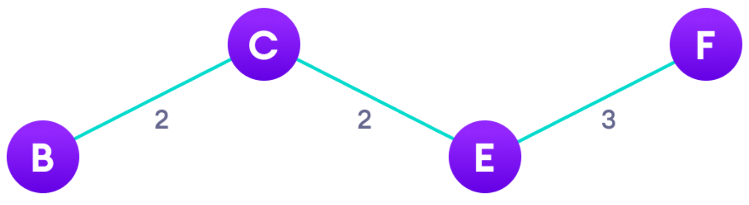
Step: 3

Choose the shortest edge from this vertex and add it



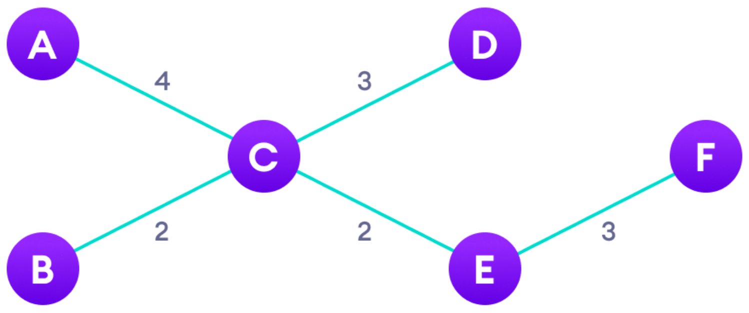
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Assessment

1) Fractional knapsack problem is also known as __________
   a) 0/1 knapsack problem
   b) Continuous knapsack problem
   c) Divisible knapsack problem
   d) Non continuous knapsack problem

2) What is the objective of the knapsack problem?
   a) To get maximum total value in the knapsack
   b) To get minimum total value in the knapsack
   c) To get maximum weight in the knapsack
   d) To get minimum weight in the knapsack

3) Which of the following statement about 0/1 knapsack and fractional knapsack problem is correct?
   a) In 0/1 knapsack problem items are divisible and in fractional knapsack items are indivisible
   b) Both are the same
   c) 0/1 knapsack is solved using a greedy algorithm and fractional knapsack is solved using dynamic programming
   d) In 0/1 knapsack problem items are indivisible and in fractional knapsack items are divisible

4) Time complexity of fractional knapsack problem is ____________
   a) O(n log n)
   b) O(n)
   c) $O(n^2)$
   d) O(nW)

5) Given items as {value,weight} pairs {{40,20},{30,10},{20,5}}. The capacity of knapsack=20. Find the maximum value output assuming items to be divisible.
   a) 60
   b) 80
   c) 100
   d) 40

6) The main time taking step in fractional knapsack problem is ___________

   a) Breaking items into fraction
   b) Adding items into knapsack
   c) Sorting
   d) Looping through sorted items

7) How many bits are needed for standard encoding if the size of the character set is X?
   a) log X
   b) X+1
   c) 2X
   d) X^2

8) In Huffman coding, data in a tree always occur?
   a) roots
   b) leaves
   c) left sub trees
   d) right sub trees

9) The type of encoding where no character code is the prefix of another character code is called?
   a) optimal encoding
   b) prefix encoding
   c) frequency encoding
   d) trie encoding

10) What is the running time of the Huffman encoding algorithm?
   a) O(C)
   b) O(log C)
   c) O(C log C)
   d) O(N log C)