Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems.

Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

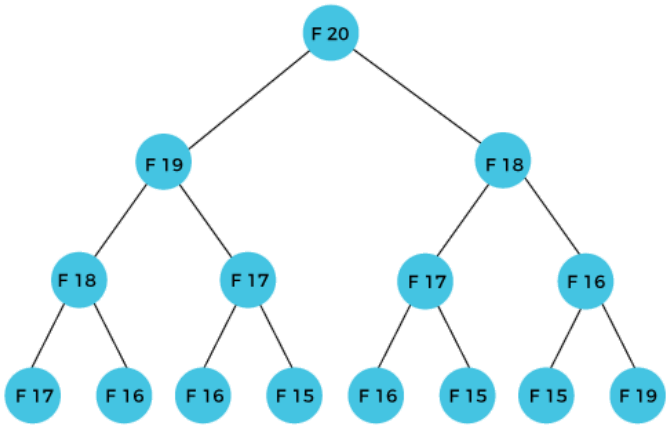
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…

The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?
The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18). In the dynamic programming approach, we try to divide the problem into the similar subproblems.

We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice.

In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

Multistage Graph

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where $k > 1$) number of disjoint subsets $S = \{s_1, s_2, \ldots, s_k\}$ such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{i+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

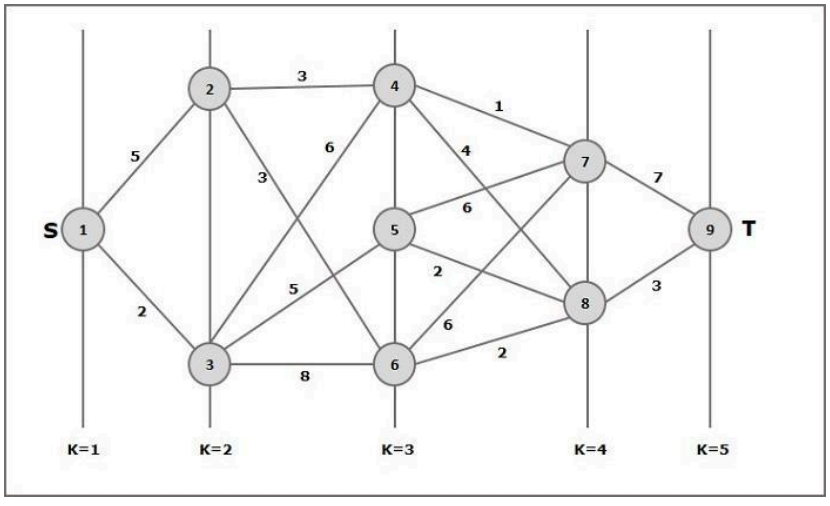
The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step 1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step 2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

Step 3: Cost (K-4, j)

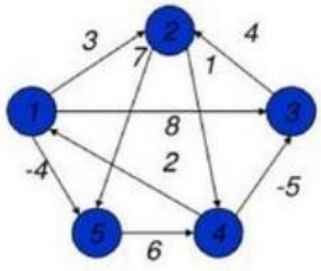
Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

All-Pairs Shortest Paths

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.

Input − The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

Output − Matrix of all pair shortest path.

0 3 4 5 6 7 7

3 0 2 1 3 4 4

4 2 0 1 3 2 3

5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

Algorithm
floydWarshal(cost)

Input − The cost matrix of given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

Optimal Binary Search Tree

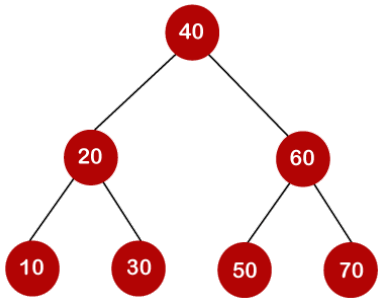
As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

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



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

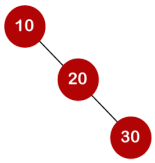
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

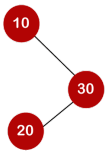
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



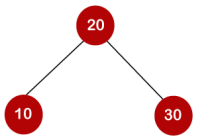
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



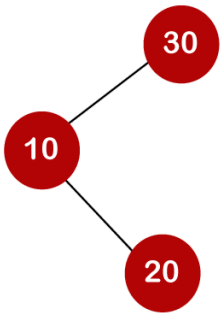
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



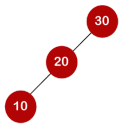
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

String Edit distance

The **Edit distance** is a problem to measure how much two strings are different from one another by counting the minimum number of operations required to convert one string into the other.

Edit distance problem can be solved by many different approaches.But the most efficient approach to solve the **Edit distance problem** is **Dynamic programming approach** which takes the **O(N * M)** time complexity, where N and M are sizes of the strings.
Edit distance has different definitions which uses different sets of string operations.

Levenshtein distance operations is the basic set of operations which is used in Edit distance Problem.
Operation allowed are:

1. Delete any character from the string.
2. Replace any character with any other
3. Add any character into any part of the string.

Problem Statement

Given two strings **str1** and **str2**, and the task is to find minimum number operations required to convert string **str1** into **str2**.

Edit Distance Problem Example

Below is the example of Edit Distance Problem with input- output constraint and the solution for the example using the Dynamic programming approach.

Input – Output Data for the Algorithm

- **Str_1 :** This contains the first string.
- **Str_2 :** This contains the second string.
- **N :** This contains the size of the first string.
- **M :** This contains the size of the second string.
- **Solution :** This is used to store the number of operations required.

Input and Output of the Example

Given two strings **str1 = "Big"** and **str2 = "Bang"** of size of **N = 3** and **N = 3** respectively, and the task is to find minimum number operations required to convert string **str1** into **str2**.
Answer: 2

Solution of the Edit Distance Problem Example

Solution of the above example using the Dynamic programming approach.
Given data are

String 1 :	**B**	**i**	**g**

String 1 :	**B**	**a**	**n**	**g**

1.Create a empty table where First column represents the String 1 and First Row represents the String 2 with additional Value(empty value) in both.

String 1 \ String 2	**Φ**	**B**	**a**	**n**	**g**
Φ					
B					
i					
g					

2.Let us start filling the table untill one of the string is empty. We will compare "Big" to Φ and then "Bang" to Φ.

- To convert Φ to Φ, we need no operation so value is 0.
- To convert B to Φ, we need 1 operation of modify, so value is 1.
- To convert i to Φ, we need 2 operation of modify and insert, so value is 2.
- for g to Φ, value is 3.
- Similarly for Bang, values will be 0, 1, 2, 3, 4.

Updated table will be

String 1 \ String 2	**Φ**	**B**	**a**	**n**	**g**
Φ	0	1	2	3	4
B	1				
i	2				
g	3				

3.Now check the each character of String 1 with String 2. And update the table according to approach.

```
If Str_1[i-1] == Str_2[j-1]:
    Table[i,j] = Table[i-1,j-1]
else:
    Table[i,j] = 1 + min(Table[i-1][j-1], Table[i-1][j], Table[i][j-1])
```

String 1 \ String 2	**Φ**	**B**	**a**	**n**	**g**
Φ	0	1	2	3	4
B	1	0	1	2	3
i	2	1	1	2	3
g	3	2	2	2	2

6.Return Table[-1][-1] for the answer

0/1 Knapsack problem

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

We will discuss both the problems one by one. First, we will learn about the 0/1 knapsack problem.

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

What is the fractional knapsack problem?

The fractional knapsack problem means that we can divide the item. For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

Example of 0/1 knapsack problem.

Consider the problem having weights and profits are:

Weights: {3, 4, 6, 5}

Profits: {2, 3, 1, 4}

The weight of the knapsack is 8 kg

The number of items is 4

The above problem can be solved by using the following method:

x_i = {1, 0, 0, 1}

= {0, 0, 0, 1}

= {0, 1, 0, 1}

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach. In dynamic programming approach, the complicated problem is divided into sub-problems, then we find the solution of a sub-problem and the solution of the sub-problem will be used to find the solution of a complex problem.

x = { 1, 0, 0}

The profit corresponding to the weight is 3. Therefore, the remaining profit is (5 - 3) equals to 2. Now we will compare this value 2 with the row i = 2. Since the row (i = 1) contains the value 2; therefore, the pointer shifted upwards shown below:

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

Again we compare the value 2 with a above row, i.e., i = 1. Since the row i =0 does not contain the value 2, so row i = 1 will be selected and the weight corresponding to the i = 1 is 3 shown below:

X = {1, 1, 0, 0}

The profit corresponding to the weight is 2. Therefore, the remaining profit is 0. We compare 0 value with the above row. Since the above row contains a 0 value but the profit corresponding to this row is 0. In this problem, two weights are selected, i.e., 3 and 4 to maximize the profit.

Reliability design problem

In **reliability design**, the problem is to design a system that is composed of several devices connected in series.

If we imagine that r1 is the reliability of the device.

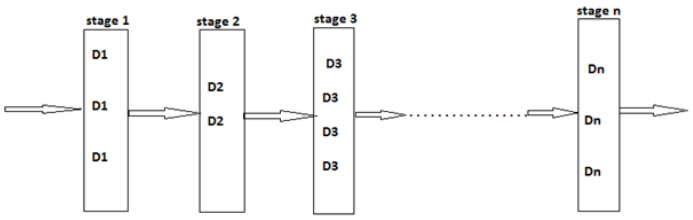
Then the reliability of the function can be given by **πr1**.

If **r1 = 0.99** and **n = 10** that **n** devices are set in a series, **1 <= i <= 10**, then reliability of the whole system **πri** can be given as: **Πri = 0.904**

So, if we duplicate the devices at each stage then the reliability of the system can be increased.

It can be said that multiple copies of the same device type are connected in parallel through the use of switching circuits. Here, switching circuit determines which devices in any given group are functioning properly. Then they make use of such devices at each stage, that result is increase in reliability at each stage. If at each stage, there are **mi** similar types of devices **Di**, then the probability that all **mi** have a malfunction is **(1 - ri)^mi**, which is very less.

And the reliability of the stage **I** becomes **(1 – (1 - ri) ^mi)**. Thus, if **ri = 0.99** and **mi = 2**, then the stage reliability becomes **0.9999** which is almost equal to **1**. Which is much better than that of the previous case or we can say the reliability is little less than **1 - (1 - ri) ^mi** because of less reliability of switching circuits.



Multiple Devices Connected in Parallel in Each Stage

In reliability design, we try to use device duplication to maximize reliability. But this maximization should be considered along with the cost.

Let **c** is the maximum allowable cost and **ci** be the cost of each unit of device **i**. Then the maximization problem can be given as follows:

Maximize π Øi (mi) for 1 <= I <= n

Subject to:

$$\sum_{i=1}^{n} ci^{\square} mi^{\square} <= c$$

mi>= 1 and integer 1 <= i <= n

Here, **Øi (mi)** denotes the reliability of the stage **i**.

The reliability of the system can be given as follows:

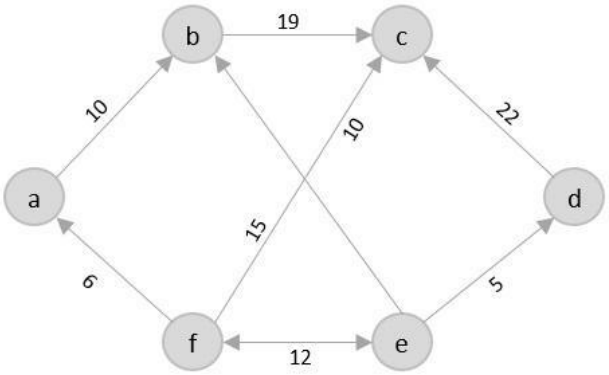
Π Øi (mi) for 1 <= i <= n

If we increase the number of devices at any stage beyond the certain limit, then also only the cost will increase but the reliability could not increase.

Travelling Salesman Problem

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

If you look at the graph below, considering that the salesman starts from the vertex ‘a’, they need to travel through all the remaining vertices b, c, d, e, f and get back to ‘a’ while making sure that the cost taken is minimum.



There are various approaches to find the solution to the travelling salesman problem: naïve approach, greedy approach, dynamic programming approach, etc. In this tutorial we will be learning about solving travelling salesman problem using greedy approach.

Travelling Salesperson Algorithm

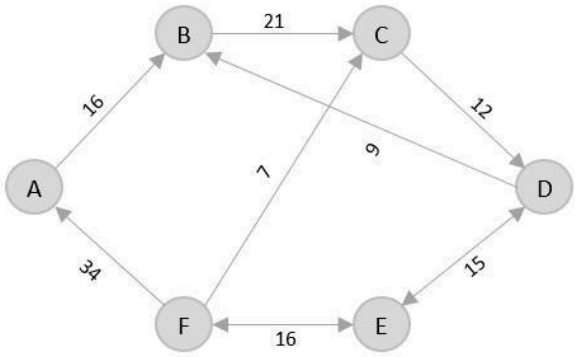
As the definition for greedy approach states, we need to find the best optimal solution locally to figure out the global optimal solution. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

Algorithm

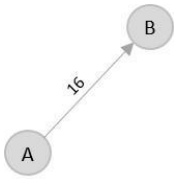
- Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

Examples

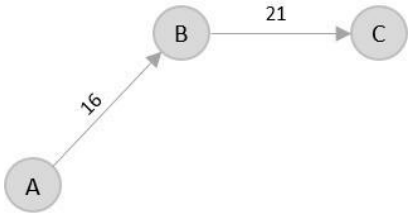
Consider the following graph with six cities and the distances between them −



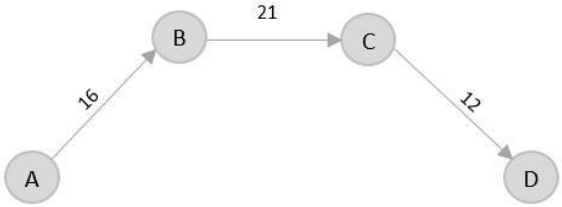
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



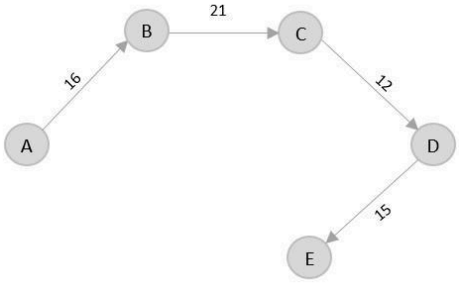
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



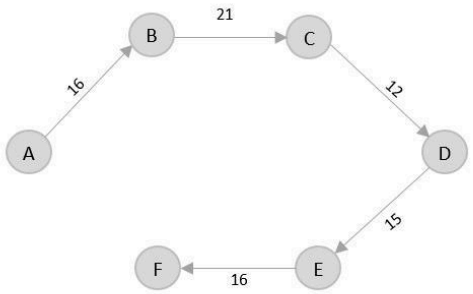
There's only one edge between C → D, therefore it is added to the output graph.



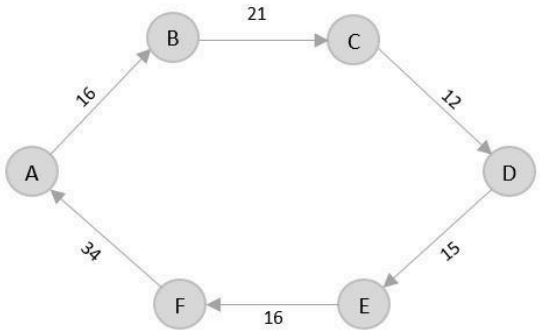
There's two outward edges from D. Even though, D → B has lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.



There's only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

Even though, the cost of path could be decreased if it originates from other nodes but the question is not raised with respect to that.

Traversal technique for Binary Tree

Binary Tree

A binary tree is a finite collection of elements or it can be said it is made up of nodes. Where each node contains the left pointer, right pointer, and a data element. The root pointer points to the topmost node in the tree. When the binary tree is not empty, so it will have a root element and the remaining elements are partitioned into two binary trees which are called the left pointer and right pointer of a tree.

Traversing in the Binary Tree

Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

1) Preorder traversal

To **traverse a binary tree in preorder**, following operations are carried out:

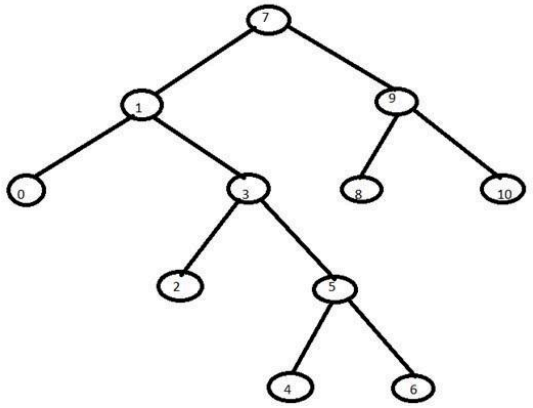
1. Visit the root.
2. Traverse the left sub tree of root.
3. Traverse the right sub tree of root.

Note: Preorder traversal is also known as NLR traversal.

Algorithm:

```
Algorithm preorder(t)
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Visit(t);
                Preorder(t->lchild);
                Preorder(t->rchild);
        }
}
```

Example: Let us consider the given binary tree,



Therefore, the preorder traversal of the above tree will be: **7,1,0,3,2,5,4,6,9,8,10**

2) Inorder traversal

To traverse a binary tree in inorder traversal, following operations are carried out:

1. Traverse the left most sub tree.
2. Visit the root.
3. Traverse the right most sub tree.

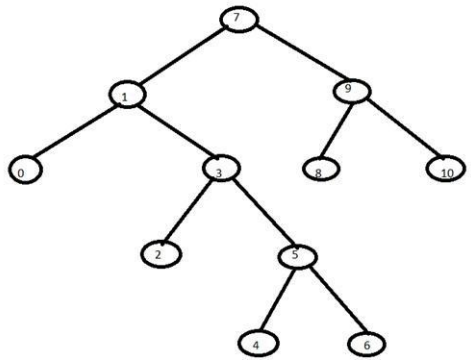
Note: Inorder traversal is also known as LNR traversal.

Algorithm:

```
Algorithm inorder(t)
```

```
/*t is a binary tree. Each node of t has three fields:
lchild, data, and rchild.*/
{
        If t! =0 then
        {
                Inorder(t->lchild);
                Visit(t);
                Inorder(t->rchild);
        }
}
```

Example: Let us consider a given binary tree.



Therefore the inorder traversal of above tree will be: **0,1,2,3,4,5,6,7,8,9,10**

3) Postorder traversal

To traverse a binary tree in postorder traversal, following operations are carried out:

1. Traverse the left sub tree of root.
2. Traverse the right sub tree of root.
3. Visit the root.

Note: Postorder traversal is also known as LRN traversal.

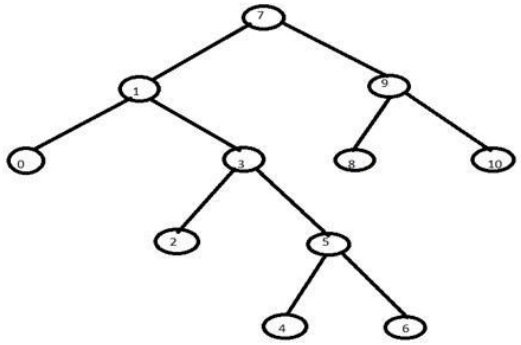
Algorithm:

```
Algorithm postorder(t)

/*t is a binary tree .Each node of t has three fields:
lchild, data, and rchild.*/
{
```

```
        If t! =0 then
        {
                Postorder(t->lchild);
                Postorder(t->rchild);
                Visit(t);
        }
}
```

Example: Let us consider a given binary tree.



Therefore the postorder traversal of the above tree will be: **0,2,4,6,5,3,1,8,10,9,7**

Techniques of graph

Definition

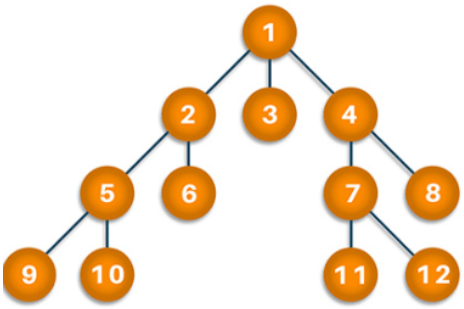
A graph G(V, E) is a non-linear data structure that consists of node and edge pairs of objects connected by links.

There are 2 types of graphs:

- Directed
- Undirected

Breadth-First Search

Traversing or searching is one of the most used operations that are undertaken while working on graphs. Therefore, in **breadth-first-search** (BFS), you start at a particular vertex, and the algorithm tries to visit all the neighbors at the given depth before moving on to the next level of traversal of vertices. Unlike trees, graphs may contain cyclic paths where the first and last vertices are remarkably the same always. Thus, in BFS, you need to keep note of all the track of the vertices you are visiting. To implement such an order, you use a queue data structure which First-in, First-out approach. To understand this, see the image given below.



BREADTH FIRST SEARCH

Algorithm

1. Start putting anyone vertices from the graph at the back of the queue.
2. First, move the front queue item and add it to the list of the visited node.
3. Next, create nodes of the adjacent vertex of that list and add them which have not been visited yet.
4. Keep repeating steps two and three until the queue is found to be empty.

Pseudocode

```
1.  Set all nodes to "not visited";
2.    q = new Queue();
3.    q.enqueue(initial node);
4.    while ( q ? empty ) do
5.    {
6.      x = q.dequeue();
7.      if ( x has not been visited )
8.      {
9.        visited[x] = true;        // Visit node x !
10.
11.       for ( every edge (x, y)  /* we are using all edges ! */ )
12.         if ( y has not been visited )
13.         q.enqueue(y);       // Use the edge (x,y) !!!
14.     }
15.   }
```

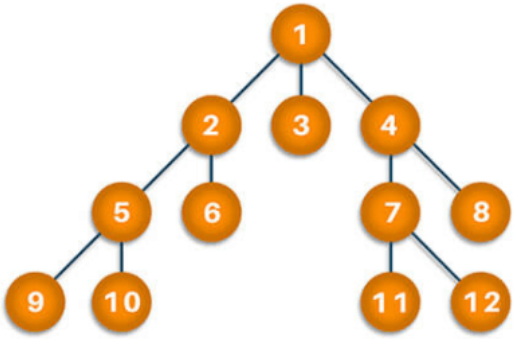
Complexity: 0(V+E) where V is vertices and E is edges.

Applications

BFS algorithm has various applications. For example, it is used to determine the **shortest path** and **minimum spanning tree.** It is also used in web crawlers to creates web page indexes. It is also used as powering search engines on social media networks and helps to find out peer-to-peer networks in BitTorrent.

Depth-first search

In depth-first-search (DFS), you start by particularly from the vertex and explore as much as you along all the branches before backtracking. In DFS, it is essential to keep note of the tracks of visited nodes, and for this, you use stack data structure.



DEPTH FIRST SEARCH

Algorithm

1. Start by putting one of the vertexes of the graph on the stack's top.
2. Put the top item of the stack and add it to the visited vertex list.
3. Create a list of all the adjacent nodes of the vertex and then add those nodes to the unvisited at the top of the stack.
4. Keep repeating steps 2 and 3, and the stack becomes empty.

Pseudocode

```
1.  DFS(G,v)   ( v is the vertex where the search starts )
2.        Stack S := {};   ( start with an empty stack )
3.        for each vertex u, set visited[u] := false;
4.        push S, v;
5.        while (S is not empty) do
6.          u := pop S;
7.          if (not visited[u]) then
8.            visited[u] := true;
9.            for each unvisited neighbour w of uu
10.             push S, w;
11.         end if
12.       end while
13.    END DFS()
14.
```