DSA Practice 7 (19-11-2024)

Name: Atchayaa T

Department: B.Tech CSBS

Register No.: 22CB004

1. **Minimum path sum**
   Given a cost matrix cost[][] and a position (M, N) in cost[][], write a function that returns cost of minimum cost path to reach (M, N) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (M, N) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1), and (i+1, j+1) can be traversed.

   Note: You may assume that all costs are positive integers.

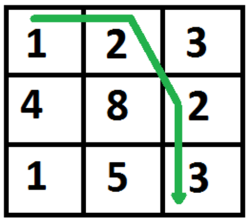
   Example:

   Input:

1	2	3
4	8	2
1	5	3

The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).

Output:



1	2	3
4	8	2
1	5	3

Code:

```java
public class Minimum_Path_Sum {
    public static int solution(int[][]cost,int m,int n,int[][]dp){
        int i,j;
        dp[0][0] = cost[0][0];
        for (i=1;i<m;i++){
            dp[i][0] = cost[i][0] + dp[i-1][0];
        }

        for (j=1;j<n;j++){
            dp[0][j] = cost[0][j] + dp[0][j-1];
        }

        for (i=1;i<m;i++){
            for (j=1;j<n;j++){
                dp[i][j] = cost[i][j] + Math.min(dp[i-1][j-1],Math.min(dp[i-
1][j],dp[i][j-1]));
            }
        }
        return dp[m-1][n-1];
    }
    public static void main(String[] args) {
        int cost[][] = { { 1, 2, 3 }, { 4, 8, 2 }, { 1, 5, 3 } };
        int m = cost.length;
        int n = cost[0].length;
        int dp[][] = new int[m][n];

        System.out.println(solution(cost,m,n,dp));
    }
}
```

Output: 8
Time Complexity: O(M*N)

2. **Validate binary search tree**
   *Given the root of a binary tree. Check whether it is a **Binary Search Tree** or **not**. A Binary Search Tree (BST) is a node-based binary tree data structure with the following properties.*

- *All keys in the left subtree are **smaller** than the root and all keys in the right subtree are **greater**.*
- *Both the left and right subtrees must also be binary search trees.*
- *Each key must be **distinct**.*

Code:

```
class Node {
    int data;
    Node left, right;

    Node(int value) {
        data = value;
        left = right = null;
    }
}

public class Validate_BST {
    public static boolean solution(Node root, int min, int max) {
        if (root == null) return true;
        if (root.data < min || root.data > max) return false;
        return solution(root.left, min, root.data - 1) && solution(root.right,
root.data + 1, max);

    }
    public static void main(String[] args) {
        Node root = new Node(4);
        root.left = new Node(2);
        root.right = new Node(5);
        root.left.left = new Node(1);
        root.left.right = new Node(3);

        if (solution(root,Integer.MIN_VALUE,Integer.MAX_VALUE)) {
            System.out.println("True");
        }
        else {
            System.out.println("False");
        }
    }

}
```

Output: True
Time Complexity: O(n)

3. **Word ladder**
Given a dictionary, and two words 'start' and 'target' (both of same length). Find length of the smallest chain from 'start' to 'target' if it exists, such that adjacent words in the chain only differ by one character and each word in the chain is a valid word i.e., it exists in the dictionary. It may be assumed that the 'target' word exists in dictionary and length of all dictionary words is same.

Input: *Dictionary = {POON, PLEE, SAME, POIE, PLEA, PLIE, POIN}, start = TOON, target = PLEA*
Output: *7*
Explanation: *TOON – POON – POIN – POIE – PLIE – PLEE – PLEA*
Input: *Dictionary = {ABCD, EBAD, EBCD, XYZA}, start = ABCV, target = EBAD*
Output: *4*
Explanation: *ABCV – ABCD – EBCD – EBAD*

Code:

```
import java.util.*;

class WordLadder {

    static int solution(String start, String target, Set<String> D) {

        if (start.equals(target))
            return 0;

        if (!D.contains(target))
            return 0;

        int level = 0, wordLength = start.length();

        Queue<String> Q = new LinkedList<>();
        Q.add(start);

        while (!Q.isEmpty()) {
            ++level;
            int sizeOfQ = Q.size();

            for (int i = 0; i < sizeOfQ; ++i) {

                char[] word = Q.peek().toCharArray();
                Q.remove();

                for (int pos = 0; pos < wordLength; ++pos) {
                    char origChar = word[pos];
                    for (char c = 'a'; c <= 'z'; ++c) {
                        word[pos] = c;

                        if (String.valueOf(word).equals(target))
                            return level + 1;

                        if (!D.contains(String.valueOf(word)))
                            continue;

                        D.remove(String.valueOf(word));
                        Q.add(String.valueOf(word));
                    }

                    word[pos] = origChar;
                }
            }
        }

        return 0;
    }

    public static void main(String[] args) {

        Set<String> D = new HashSet<>();
        D.add("poon");
        D.add("plee");
        D.add("same");
        D.add("poie");
        D.add("plie");
        D.add("poin");
        D.add("plea");
        String start = "toon";
        String target = "plea";
        System.out.print(solution(start, target, D));
    }
}
```

Output: 7
Time Complexity: O(N*M)

4. **Word ladder -II**
   Given a dictionary, and two words ***start*** *and* ***target*** *(both of the same length). Find length of the smallest chain from* ***start*** *to* ***target*** *if it exists, such that adjacent words in the chain only differ by one character and each word in the chain is a valid word i.e., it exists in the dictionary. It may be assumed that the* ***target*** *word exists in the dictionary and the lengths of all the dictionary words are equal.*

   Input: *Dictionary = {POON, PLEE, SAME, POIE, PLEA, PLIE, POIN}*
   start = "TOON"
   target = "PLEA"
   Output: *7*
   TOON -> POON –> POIN –> POIE –> PLIE –> PLEE –> PLEA

   Code:

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

```java
public class WordTransformation {

    static List<List<String>> solution(String beginWord, String endWord, List<String>
wordList) {
        List<List<String>> ans = new ArrayList<>();
        Set<String> vis = new HashSet<>(wordList);
        List<String> usedOnLevel = new ArrayList<>();

        Queue<List<String>> qu = new LinkedList<>();
        List<String> temp = new ArrayList<>();
        temp.add(beginWord);
        qu.add(temp);
        int level = 0;

        while (!qu.isEmpty()) {
            List<String> vec = qu.poll();

            if (vec.size() > level) {
                level++;

                for (String str : usedOnLevel) {
                    vis.remove(str);
                }
                usedOnLevel.clear();
            }

            String last = vec.get(vec.size() - 1);

            if (last.equals(endWord)) {
                if (ans.size() == 0) {
                    ans.add(vec);
                } else if (ans.get(0).size() == vec.size()) {
                    ans.add(vec);
                }
            }

            for (int i = 0; i < last.length(); i++) {
                char[] arr = last.toCharArray();
                char org = arr[i];
                for (char ch = 'a'; ch <= 'z'; ch++) {
                    arr[i] = ch;
                    String newStr = new String(arr);
                    if (vis.contains(newStr)) {
                        List<String> tempVec = new ArrayList<>(vec);
                        tempVec.add(newStr);
                        qu.add(tempVec);
                        usedOnLevel.add(newStr);
                    }
                }

                arr[i] = org;
                last = new String(arr);
            }
        }
        return ans;
    }

    public static void main(String[] args) {
        List<String> wordList = new ArrayList<>();
        wordList.add("poon");
        wordList.add("plee");
        wordList.add("same");
        wordList.add("poie");
        wordList.add("plie");
        wordList.add("poin");
        wordList.add("plea");

        String start = "toon";
        String target = "plea";

        List<List<String>> ans = solution(start, target, wordList);
        for (List<String> a : ans) {
            for (String s : a) {
                System.out.print(s + " ");
            }
            System.out.println("\nLength of sequence is " + a.size());
        }
    }
}
```

Output: 7
Time Complexity: O(N*M*26)

5. **Course schedule**
*There are a total of **N** tasks, labeled from **0** to **N-1.** Some tasks may have prerequisites, for example to do task 0 you have to first complete task 1, which is expressed as a pair: **[0, 1]**. Given the total number of **tasks N** and a list of **prerequisite pairs P**, find if it is possible to finish all tasks.*

***Input:** N = 4, P = 3, prerequisites = {{1,0},{2,1},{3,2}}*
***Output:** Yes*
***Explanation:** To do task 1 you should have completed task 0, and to do task 2 you should have finished task 1, and to do task 3 you should have finished task 2. So it is possible.*
***Input:** N = 2, P = 2, prerequisites = {{1,0},{0,1}}*
***Output:** No*
***Explanation:** To do task 1 you should have completed task 0, and to do task 0 you should have finished task 1. So it is impossible.*

Code:

```java
import java.util.*;

public class TaskScheduler {

    static class Pair {
        int first, second;

        Pair(int first, int second) {
            this.first = first;
            this.second = second;
        }
    }

    static ArrayList<ArrayList<Integer>> makeGraph(int numTasks, Vector<Pair>
prerequisites) {
        ArrayList<ArrayList<Integer>> graph = new ArrayList<>(numTasks);

        for (int i = 0; i < numTasks; i++) {
            graph.add(new ArrayList<>());
        }

        for (Pair pre : prerequisites) {
            graph.get(pre.second).add(pre.first);
        }

        return graph;
    }

    static boolean dfsCycle(ArrayList<ArrayList<Integer>> graph, int node, boolean[]
onPath, boolean[] visited) {
        if (visited[node]) {
            return false;
        }
        onPath[node] = visited[node] = true;

        for (int neighbor : graph.get(node)) {
            if (onPath[neighbor] || dfsCycle(graph, neighbor, onPath, visited)) {
                return true;
            }
        }

        onPath[node] = false;
        return false;
    }

    static boolean solution(int numTasks, Vector<Pair> prerequisites) {
        ArrayList<ArrayList<Integer>> graph = makeGraph(numTasks, prerequisites);

        boolean[] onPath = new boolean[numTasks];
        boolean[] visited = new boolean[numTasks];

        for (int i = 0; i < numTasks; i++) {
            if (!visited[i] && dfsCycle(graph, i, onPath, visited)) {
                return false;
            }
        }

        return true;
    }
    public static void main(String[] args) {
        int numTasks = 4;

        Vector<Pair> prerequisites = new Vector<>();
        prerequisites.add(new Pair(1, 0));
        prerequisites.add(new Pair(2, 1));
        prerequisites.add(new Pair(3, 2));

        if (solution(numTasks, prerequisites)) {
            System.out.println("Possible to finish all tasks");
        } else {
            System.out.println("Impossible to finish all tasks");
        }
    }
}
```

Output: Yes
Time Complexity: O(V+E)