ISE-2

Name: Atharva Dharmendra Jagtap **Dept:** CSE **Roll no. 10937**

Subject: Analysis of Algorithms

Activity Type: Practical Coding & Conceptual Analysis

Total Marks: 20

Target Group: Students who have not appeared for the NPTEL exam

Objective:

To apply algorithmic thinking to solve problems beyond the classroom syllabus, encouraging independent exploration and deeper understanding.

Topics for Exploration

Students must solve four problems, one from each of the following categories:

1. Dynamic Programming
2. Backtracking
3. Branch and Bound
4. String Matching Algorithms

1. String Matching

5. Longest Palindromic Substring

Level: **Medium**

Given a string s, return *the longest palindromic substring* in s.

Example 1:
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
Example 2:
Input: s = "cbbd"
Output: "bb"

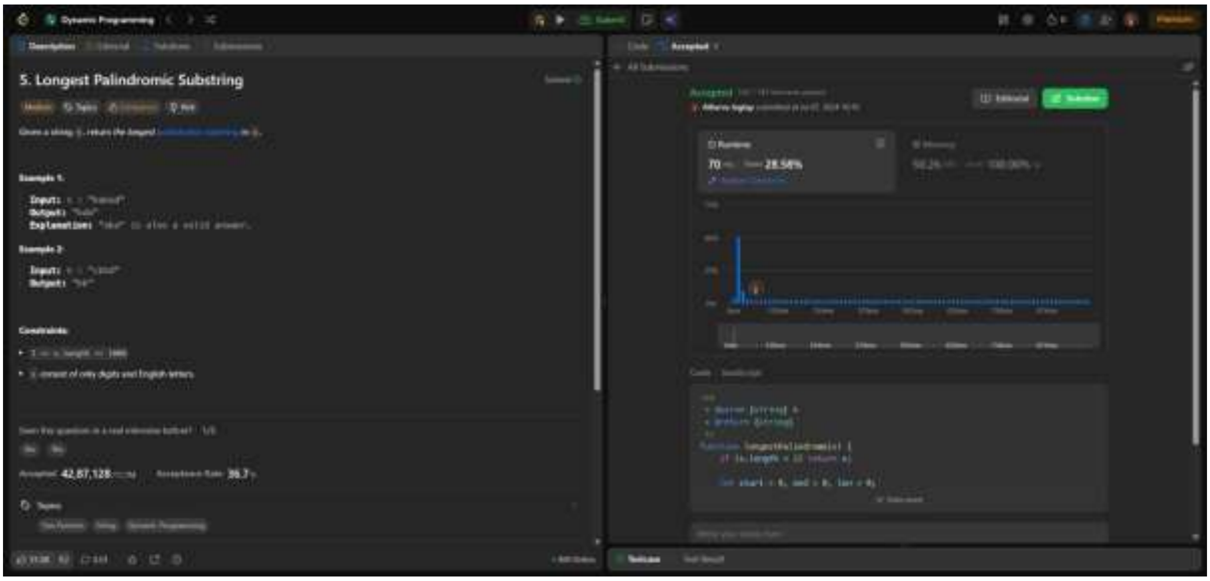
Constraints:

- 1 <= s.length <= 1000
- s consist of only digits and English letters.

Code:

```
Time Complexity - O(N^2), Space Complexity - O(N^2) (caching all substring)
*/

class Solution
{
public:
    string longestPalindrome(string s)
    {
        int n = s.size();
        if (n == 0)
            return "";

        // dp[i][j] will be 'true' if the string from index i to j is a palindrome.
        bool dp[n][n];

        //Initialize with false

        memset(dp, 0, sizeof(dp));

        //Every Single character is palindrome
        for (int i = 0; i < n; i++)
            dp[i][i] = true;

        string ans = "";
        ans += s[0];

        for (int i = n - 1; i >= 0; i--)
        {
            for (int j = i + 1; j < n; j++)
            {
                if (s[i] == s[j])
                {
                    //If it is of two character OR if its substring is palindrome.
                    if (j - i == 1 || dp[i + 1][j - 1])
                    {
                        //Then it will also a palindrome substring
                        dp[i][j] = true;

                        //Check for Longest Palindrome substring
                        if (ans.size() < j - i + 1)
                            ans = s.substr(i, j - i + 1);
                    }
                }
            }
        }
        return ans;
    }
};
```

Submission:



2. Dynamic Programming

[124. Binary Tree Maximum Path Sum]

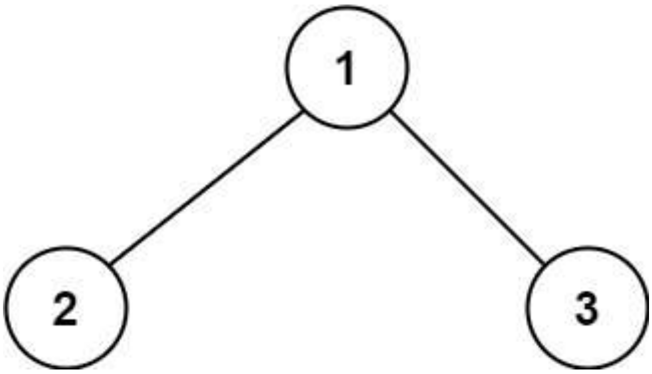
Level: **Hard**

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

Example 1:

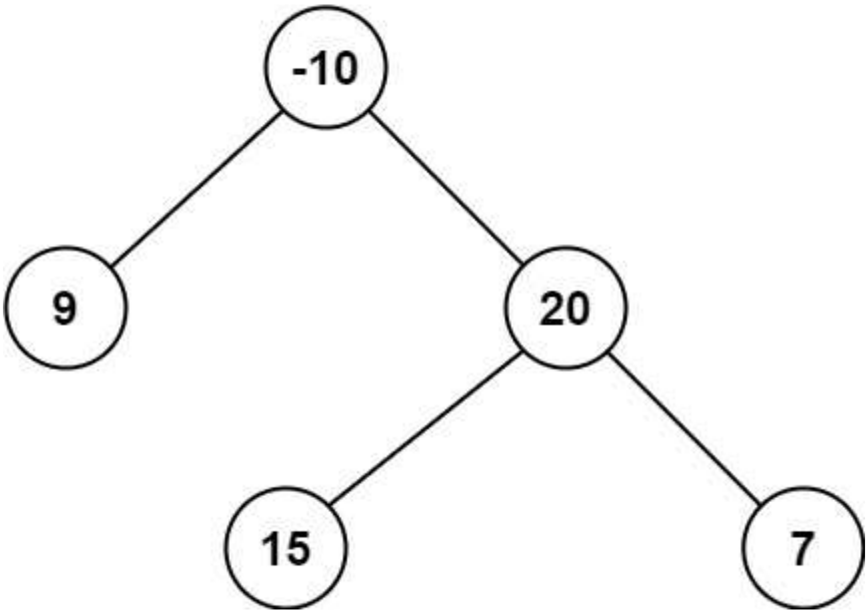


Input: root = [1,2,3]

Output: 6

Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

Example 2:



Input: root = [-10,9,20,null,null,15,7]

Output: 42

Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.

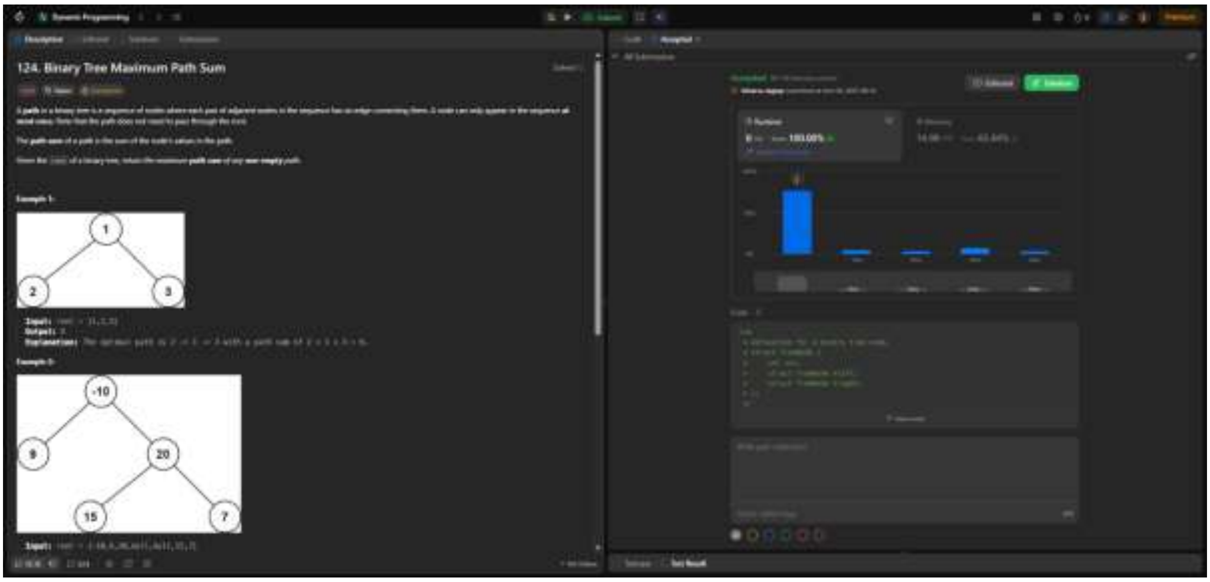
Constraints:

- The number of nodes in the tree is in the range [1, 3 * 10^4].
- -1000 <= Node.val <= 1000

Code:

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
int max(int a, int b) { return a > b ? a : b; }

int maxPathSumUtil(struct TreeNode* root, int* maxSum) {
    if (!root) return 0;
    int left = max(0, maxPathSumUtil(root->left, maxSum));
    int right = max(0, maxPathSumUtil(root->right, maxSum));
    *maxSum = max(*maxSum, root->val + left + right);
    return root->val + max(left, right);
}


int maxPathSum(struct TreeNode* root) {
    int maxSum = -10000;
    maxPathSumUtil(root, &maxSum);
    return maxSum;
}
```

Submission:



3. Backtracking

[491. Non-decreasing Subsequences]

Level: Medium

Given an integer array nums, return *all the different possible non-decreasing subsequences of the given array with at least two elements*. You may return the answer in **any order**.

Example 1:
Input: nums = [4,6,7,7]
Output: [[4,6],[4,6,7],[4,6,7,7],[4,7],[4,7,7],[6,7],[6,7,7],[7,7]]
Example 2:
Input: nums = [4,4,3,2,1]
Output: [[4,4]]

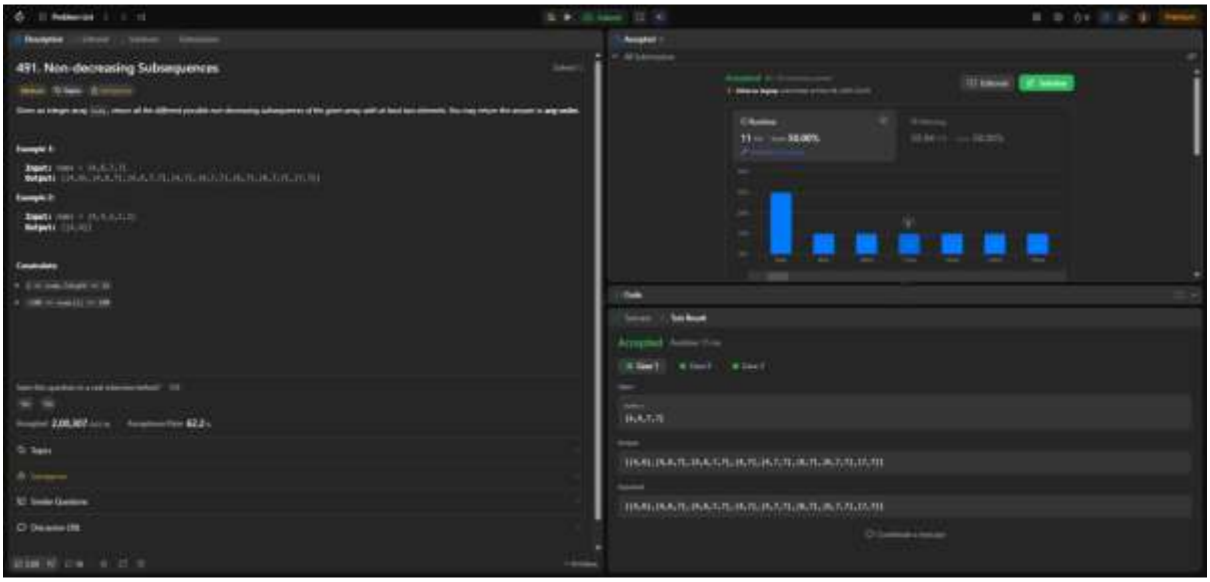
Constraints:

- 1 <= nums.length <= 15
- -100 <= nums[i] <= 100

Topics:

- Array
- Hash Table
- Backtracking
- Bit Manipulation

Submission:



Code:

```c
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller calls free().
 */
#define INITIAL_CAPACITY 1024
#define MAX_LEN 15

int** res;
int resSize;
int resCapacity;
int* colSizes;

int path[MAX_LEN];
int pathSize;

void ensureCapacity() {
    if (resSize >= resCapacity) {
        resCapacity *= 2;
        res = (int**)realloc(res, resCapacity * sizeof(int*));
        colSizes = (int*)realloc(colSizes, resCapacity * sizeof(int));
    }
}

void backtrack(int* nums, int numsSize, int startIndex) {
    if (pathSize >= 2) {
        ensureCapacity();
        res[resSize] = (int*)malloc(pathSize * sizeof(int));
        memcpy(res[resSize], path, pathSize * sizeof(int));
        colSizes[resSize++] = pathSize;
    }

    int used[201] = {0}; // For duplicates in current recursion level

    for (int i = startIndex; i < numsSize; i++) {
        if ((pathSize > 0 && nums[i] < path[pathSize - 1]) || used[nums[i] + 100])
            continue;

        used[nums[i] + 100] = 1;
        path[pathSize++] = nums[i];

        backtrack(nums, numsSize, i + 1);

        pathSize--;
    }
}

int** findSubsequences(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
    resCapacity = INITIAL_CAPACITY;
    res = (int**)malloc(resCapacity * sizeof(int*));
    colSizes = (int*)malloc(resCapacity * sizeof(int));
    resSize = 0;
    pathSize = 0;

    backtrack(nums, numsSize, 0);

    *returnSize = resSize;
    *returnColumnSizes = (int*)malloc(resSize * sizeof(int));
    memcpy(*returnColumnSizes, colSizes, resSize * sizeof(int));

    return res;
}
```

4. Branch and Bound

[2959. Number of Possible Sets of Closing Branches]

Level: **Hard**

There is a company with n branches across the country, some of which are connected by roads. Initially, all branches are reachable from each other by traveling some roads.
The company has realized that they are spending an excessive amount of time traveling between their branches. As a result, they have decided to close down some of these branches (**possibly none**). However, they want to ensure that the remaining branches have a distance of at most maxDistance from each other.

The **distance** between two branches is the **minimum** total traveled length needed to reach one branch from another.

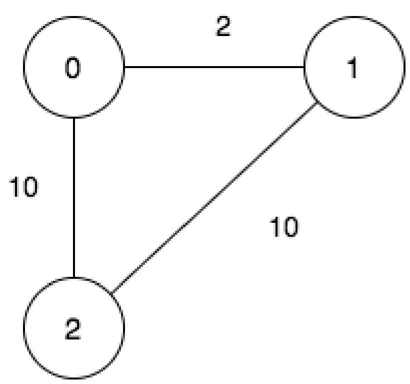
You are given integers n, maxDistance, and a **0-indexed** 2D array roads, where roads[i] = [u_i, v_i, w_i] represents the **undirected** road between branches u_i and v_i with length w_i.

Return *the number of possible sets of closing branches, so that any branch has a distance of at most* maxDistance *from any other*.

Note that, after closing a branch, the company will no longer have access to any roads connected to it.

Note that, multiple roads are allowed.

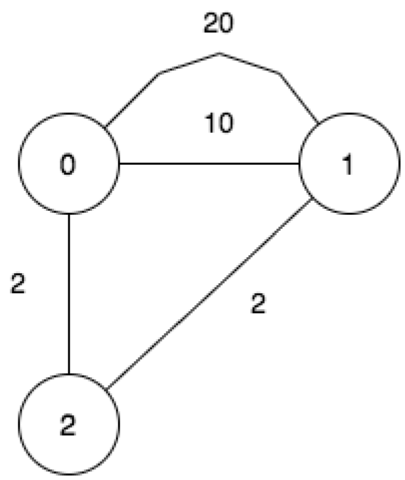
Example 1:



Input: n = 3, maxDistance = 5, roads = [[0,1,2],[1,2,10],[0,2,10]]
Output: 5

Explanation: The possible sets of closing branches are:
- The set [2], after closing, active branches are [0,1] and they are reachable to each other within distance 2.
- The set [0,1], after closing, the active branch is [2].
- The set [1,2], after closing, the active branch is [0].
- The set [0,2], after closing, the active branch is [1].
- The set [0,1,2], after closing, there are no active branches.
It can be proven, that there are only 5 possible sets of closing branches.

Example 2:



Input: n = 3, maxDistance = 5, roads = [[0,1,20],[0,1,10],[1,2,2],[0,2,2]]
Output: 7
Explanation: The possible sets of closing branches are:
- The set [], after closing, active branches are [0,1,2] and they are reachable to each other within distance 4.
- The set [0], after closing, active branches are [1,2] and they are reachable to each other within distance 2.
- The set [1], after closing, active branches are [0,2] and they are reachable to each other within distance 2.
- The set [0,1], after closing, the active branch is [2].
- The set [1,2], after closing, the active branch is [0].
- The set [0,2], after closing, the active branch is [1].
- The set [0,1,2], after closing, there are no active branches.
It can be proven, that there are only 7 possible sets of closing branches.

Example 3:

Input: n = 1, maxDistance = 10, roads = []
Output: 2
Explanation: The possible sets of closing branches are:
- The set [], after closing, the active branch is [0].
- The set [0], after closing, there are no active branches.
It can be proven, that there are only 2 possible sets of closing branches.

Constraints:

- 1 <= n <= 10
- 1 <= maxDistance <= 10^5
- 0 <= roads.length <= 1000
- roads[i].length == 3
- 0 <= u_i, v_i <= n - 1
- u_i != v_i
- 1 <= w_i <= 1000
- All branches are reachable from each other by traveling some roads.

Code:

```c
C

int numberOfSets(int n, int maxDistance, int** roads, int roadsSize, int* roadsColSize) {
    int count = 0;
    int INF = INT_MAX / 2;
    int dist[10][10];
    int total = 1 << n;
    for (int mask = 0; mask < total; ++mask) {
        bool active[10] = {0};
        int activeCount = 0;
        for (int i = 0; i < n; ++i) {
            if (!(mask & (1 << i))) {
                active[i] = true;
                activeCount++;
            }
        }
        if (activeCount == 0) {
            count++;
            continue;
        }
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                dist[i][j] = (i == j) ? 0 : INF;

        for (int i = 0; i < roadsSize; ++i) {
            int u = roads[i][0], v = roads[i][1], w = roads[i][2];
            if (active[u] && active[v] && w < dist[u][v]) {
                dist[u][v] = dist[v][u] = w;
            }
        }
        for (int k = 0; k < n; ++k) {
            if (!active[k]) continue;
            for (int i = 0; i < n; ++i) {
                if (!active[i]) continue;
                for (int j = 0; j < n; ++j) {
                    if (!active[j]) continue;
                    if (dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }
        bool valid = true;
        for (int i = 0; i < n && valid; ++i) {
            if (!active[i]) continue;
            for (int j = 0; j < n; ++j) {
                if (!active[j]) continue;
                if (dist[i][j] > maxDistance) {
                    valid = false;
                    break;
                }
            }
        }
        if (valid) count++;
    }
    return count;
}
```

Submission: