

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: $s = \text{"babad"}$

Output: "bab"

Explanation: "aba" is also a valid answer.

Example 2:

Input: $s = \text{"cbbd"}$

Output: "bb"

Constraints:

- $1 \leq s.length \leq 1000$
- s consist of only digits and English letters.

Code:

```
Time Complexity =  $O(N^2)$ , Space Complexity =  $O(N^2)$  (aching will substrng)
//
class Solution
{
public:
    string longestPalindromicSubstring(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];

        //Initialization 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 be 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:

5. Longest Palindromic Substring

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

Example 1:
Input: $s = "babad"$
Output: $"bab"$
Explanation: $"aba"$ and $"bab"$ are longest palindromic substrings.

Example 2:
Input: $s = "cbbd"$
Output: $"bb"$

Constraints:

- $1 \leq s.length \leq 1000$
- s consists of only digits and English letters.

Accepted: 42,87,128 / 43 Submissions: 36.7%

Submission Status: Accepted (100%)

Runtime: 70 ms (28.56%)

Memory: 90.26 MB (90.00%)

```
class Solution {
public:
    string longestPalindromicSubstr(string s) {
        int n = s.length();
        if (n == 0) return "";

        bool dp[n][n];
        memset(dp, 0, sizeof(dp));

        for (int i = 0; i < n; i++) dp[i][i] = true;

        string ans = s[0];

        for (int i = n - 1; i >= 0; i--)
            for (int j = i + 1; j < n; j++)
                if (s[i] == s[j] && (j - i == 1 || dp[i + 1][j - 1]))
                    dp[i][j] = true;
                    if (j - i + 1 > ans.length()) ans = s.substr(i, j - i + 1);

        return ans;
    }
};
```

2. Dynamic Programming

124. Binary Tree Maximum Path Sum

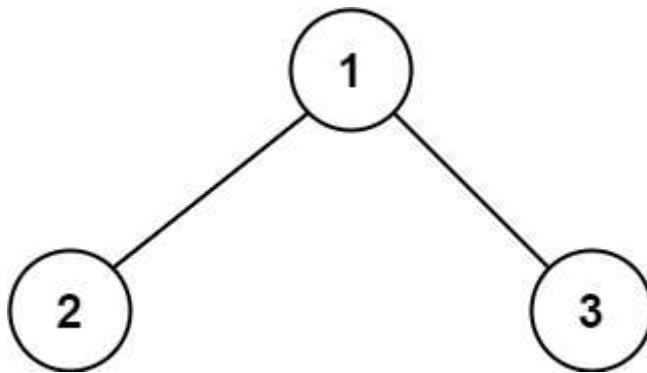
Level: **Hard**

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:

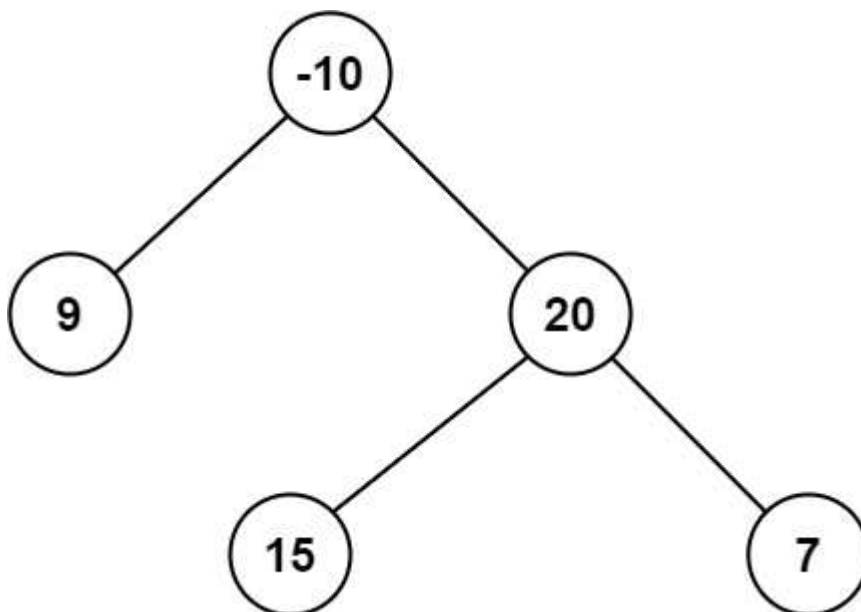


Input: root = [1,2,3]

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 $[1, 3 * 10^4]$.
- $-1000 \leq \text{Node.val} \leq 1000$

Code:

```
/**
 * 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:

The screenshot displays the LeetCode interface for the "124. Binary Tree Maximum Path Sum" problem. The left sidebar contains the problem description, which defines a path as a sequence of nodes where each pair of adjacent nodes is connected by an edge, and no node appears more than once. It includes two examples: Example 1 with a tree structure [1, 2, 3] and Example 2 with a tree structure [-10, 9, 20, null, null, 15, 7]. The right sidebar shows the submission results, indicating a 100% success rate and a bar chart comparing the submission's performance to other users. The code for the submission is visible in the bottom right pane.

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]]`

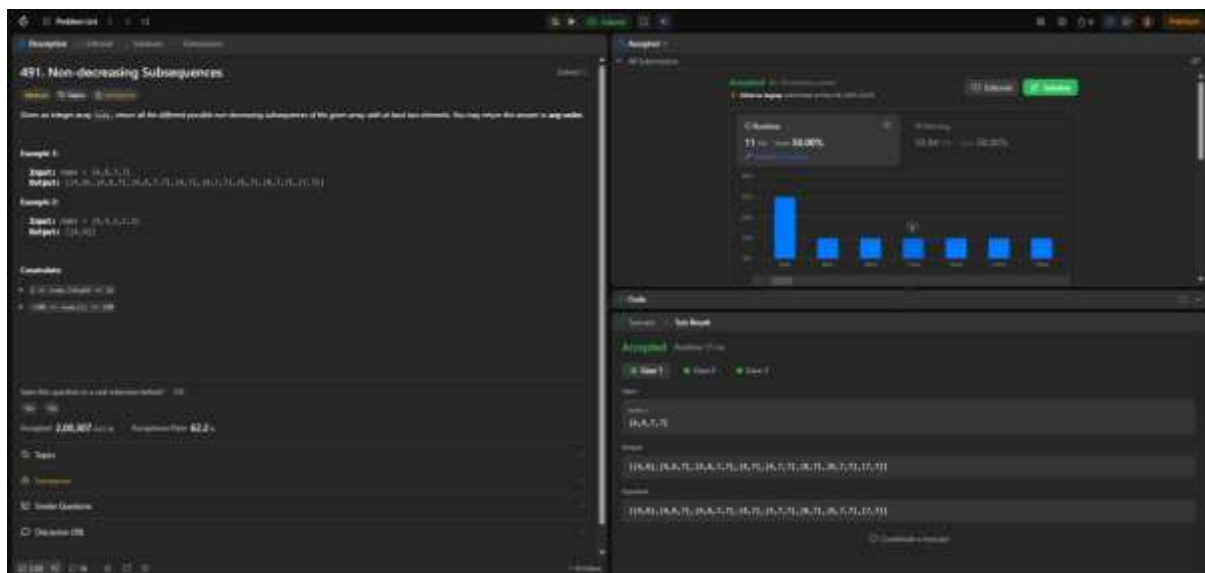
Constraints:

- $1 \leq \text{nums.length} \leq 15$
- $-100 \leq \text{nums}[i] \leq 100$

Topics:

- Array
- Hash Table
- Backtracking
- Bit Manipulation

Submission:



Code:

```
1  /**
2   * Return an array of arrays of size *returnSize.
3   * The sizes of the arrays are returned as *returnColumnSizes array.
4   * Note: Both returned array and *returnColumnSizes array must be malloced, assume caller calls free().
5   */
6  #define INITIAL_CAPACITY 1024
7  #define MAX_LEN 15
8
9  int** res;
10 int resSize;
11 int resCapacity;
12 int* colSizes;
13
14 int path[MAX_LEN];
15 int pathSize;
16
17 void ensureCapacity() {
18     if (resSize == resCapacity) {
19         resCapacity *= 2;
20         res = (int**)realloc(res, resCapacity * sizeof(int*));
21         colSizes = (int*)realloc(colSizes, resCapacity * sizeof(int));
22     }
23 }
24
25 void backtrack(int* nums, int numsSize, int startIndex) {
26     if (pathSize >= 2) {
27         ensureCapacity();
28         res[resSize] = (int*)malloc(pathSize * sizeof(int));
29         memcpy(res[resSize], path, pathSize * sizeof(int));
30         colSizes[resSize++] = pathSize;
31     }
32
33     int used[201] = {0}; // for duplicates in current recursion level
34
35     for (int i = startIndex; i < numsSize; i++) {
36         if ((pathSize > 0 && nums[i] < path[pathSize - 1]) || used[nums[i] + 100])
37             continue;
38
39         used[nums[i] + 100] = 1;
40         path[pathSize++] = nums[i];
41
42         backtrack(nums, numsSize, i + 1);
43
44         pathSize--;
45     }
46 }
47
48 int** findSubsequences(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
49     resCapacity = INITIAL_CAPACITY;
50     res = (int**)malloc(resCapacity * sizeof(int*));
51     colSizes = (int*)malloc(resCapacity * sizeof(int));
52     resSize = 0;
53     pathSize = 0;
54
55     backtrack(nums, numsSize, 0);
56
57     *returnSize = resSize;
58     *returnColumnSizes = (int*)malloc(resSize * sizeof(int));
59     memcpy(*returnColumnSizes, colSizes, resSize * sizeof(int));
60
61     return res;
62 }
63 }
```

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.

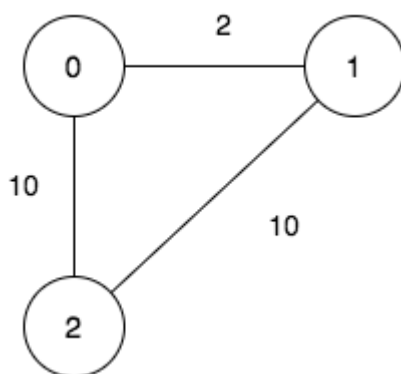
You are given integers n , maxDistance , and a **0-indexed** 2D array `roads`, where `roads[i] = [ui, vi, wi]` represents the **undirected** road between branches u_i and v_i with length w_i .

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: $n = 3$, $\text{maxDistance} = 5$, `roads = [[0,1,2],[1,2,10],[0,2,10]]`

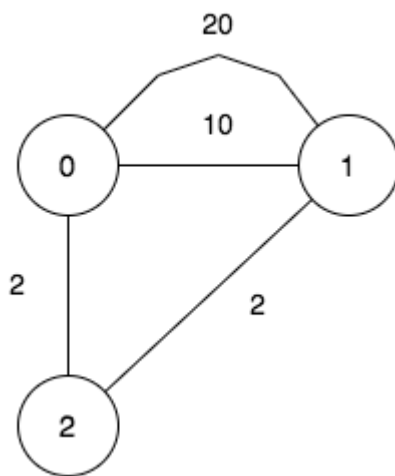
Output: 5

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: $n = 3$, $\text{maxDistance} = 5$, $\text{roads} = [[0,1,20],[0,1,10],[1,2,2],[0,2,2]]$

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: $n = 1$, $\text{maxDistance} = 10$, $\text{roads} = []$

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:

- $1 \leq n \leq 10$
- $1 \leq \text{maxDistance} \leq 10^5$
- $0 \leq \text{roads.length} \leq 1000$
- $\text{roads}[i].\text{length} == 3$
- $0 \leq u_i, v_i \leq n - 1$
- $u_i \neq v_i$
- $1 \leq w_i \leq 1000$
- All branches are reachable from each other by traveling some roads.

Code:

```
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)) > 0) {
                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] && 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:

2559. Number of Possible Sets of Closing Branches

There is a company with n branches across the country, some of which are connected by roads. Initially, all branches are not linked from each other by creating some roads.

The company has various road files are opened up for execution around of their existing branches. For instance, if a road i , they have described the start point s_i of their branches, s_i and t_i , they want to ensure that the existing branches have a distance of at least k between s_i and t_i .

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

For any given query s_i, t_i, k , and n , the company will ask you to find the number of possible sets of closing branches, s_i and t_i , with length k .

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

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

Note that, multiple roads are allowed.

Example 1:

Input: $n = 5, k = 2, roads = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 0], [0, 3], [1, 4]]$

Output: 5

Explanation: The possible sets of closing branches are:

- The set $\{0, 1\}$, after closing, active branches are $\{2, 3, 4\}$ and they are reachable to each other within distance 2.
- The set $\{1, 2\}$, after closing, the active branches are $\{0, 3, 4\}$.
- The set $\{2, 3\}$, after closing, the active branches are $\{0, 1, 4\}$.
- The set $\{3, 4\}$, after closing, there are no active branches.
- If you go forward, then there are only 5 possible sets of closing branches.

Example 2:

Input: $n = 5, k = 2, roads = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 0], [0, 3], [1, 4]]$

Output: 5

Explanation: The possible sets of closing branches are:

- The set $\{0, 1\}$, after closing, active branches are $\{2, 3, 4\}$ and they are reachable to each other within distance 2.
- The set $\{1, 2\}$, after closing, the active branches are $\{0, 3, 4\}$.
- The set $\{2, 3\}$, after closing, the active branches are $\{0, 1, 4\}$.
- The set $\{3, 4\}$, after closing, there are no active branches.
- If you go forward, then there are only 5 possible sets of closing branches.