



INTERNSHIP REPORT

WEEK 2 – Leet Code Documentation

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

RECRUITMENT LEAD

SUBMITTED BY

SANJEEV REDDY SIRIPINANE

All My Submissions

Time Submitted	Question	Status	Runtime	Language
9 hours ago	Minimum Path Sum	Accepted	4 ms	java
13 hours, 35 minutes ago	Unique Paths	Accepted	0 ms	java
1 day, 1 hour ago	Unique Binary Search Trees	Accepted	0 ms	java
1 day, 5 hours ago	Edit Distance	Accepted	5 ms	java
1 day, 8 hours ago	Unique Binary Search Trees II	Accepted	1 ms	java
1 day, 9 hours ago	Maximum Subarray	Accepted	1 ms	java
1 day, 10 hours ago	Generate Parentheses	Accepted	1 ms	java
1 day, 11 hours ago	Longest Path With Different Adjacent Characters	Accepted	86 ms	java
1 day, 13 hours ago	Number of Good Paths	Accepted	115 ms	java
2 days, 7 hours ago	Find Minimum Diameter After Merging Two Trees	Accepted	323 ms	java
2 days, 7 hours ago	Find Minimum Diameter After Merging Two Trees	Wrong Answer	N/A	java
2 days, 7 hours ago	Find Minimum Diameter After Merging Two Trees	Runtime Error	N/A	java
2 days, 8 hours ago	Validate Binary Tree Nodes	Accepted	4 ms	java
2 days, 10 hours ago	Reachable Nodes With Restrictions	Accepted	69 ms	java
2 days, 12 hours ago	Most Profitable Path in a Tree	Accepted	53 ms	java
2 days, 13 hours ago	Minimum Fuel Cost to Report to the Capital	Accepted	52 ms	java
2 days, 14 hours ago	Time Taken to Mark All Nodes	Time Limit Exceeded	N/A	java
3 days, 5 hours ago	Time Taken to Mark All Nodes	Time Limit Exceeded	N/A	java
3 days, 5 hours ago	Time Taken to Mark All Nodes	Time Limit Exceeded	N/A	java
3 days, 8 hours ago	Maximum Sum of Distinct Subarrays With Length K	Accepted	32 ms	java

Maximum Sum of Distinct Subarrays with Length K:

Testcase

Test Result

Accepted

0 ms

Case 1

Case 2

Input

nums =

[1,5,4,2,9,9,9]

k =

3

Output

15

Expected

15

Contribute a testcase

Description

Editorial

Solutions

Submit

2461. Maximum Sum of Distinct Subarrays With Length K

Medium

Topics

Companies

Hint

You are given an integer array `nums` and an integer `k`. Find the maximum subarray sum of all the subarrays of `nums` that meet the following conditions:

- The length of the subarray is `k`, and
- All the elements of the subarray are **distinct**.

Return the maximum subarray sum of all the subarrays that meet the conditions. If no subarray meets the conditions, return `0`.

A **subarray** is a contiguous non-empty sequence of elements within an array.

1.9K

188

1743 Online

Code

```

1 import java.util.HashSet;
2 class Solution {
3     public long maximumSubarraySum(int[] nums, int k) {
4         long maxAdd = 0;
5         long currentAdd = 0;
6         int start = 0;
7         HashSet<Integer> set = new HashSet<>();
8         for (int end = 0; end < nums.length; end++) {
9             while (set.contains(nums[end])) {
10                 set.remove(nums[start]);
11                 currentAdd -= nums[start];
12                 start++;
13             }
14             set.add(nums[end]);
15             currentAdd += nums[end];
16
17             if (end - start + 1 == k) {
18                 maxAdd = Math.max(maxAdd, currentAdd);
19                 set.remove(nums[start]);
20                 currentAdd -= nums[start];
21                 start++;
22             }
23         }
24     }
25 }
```

Ln 1, Col 26

Saved

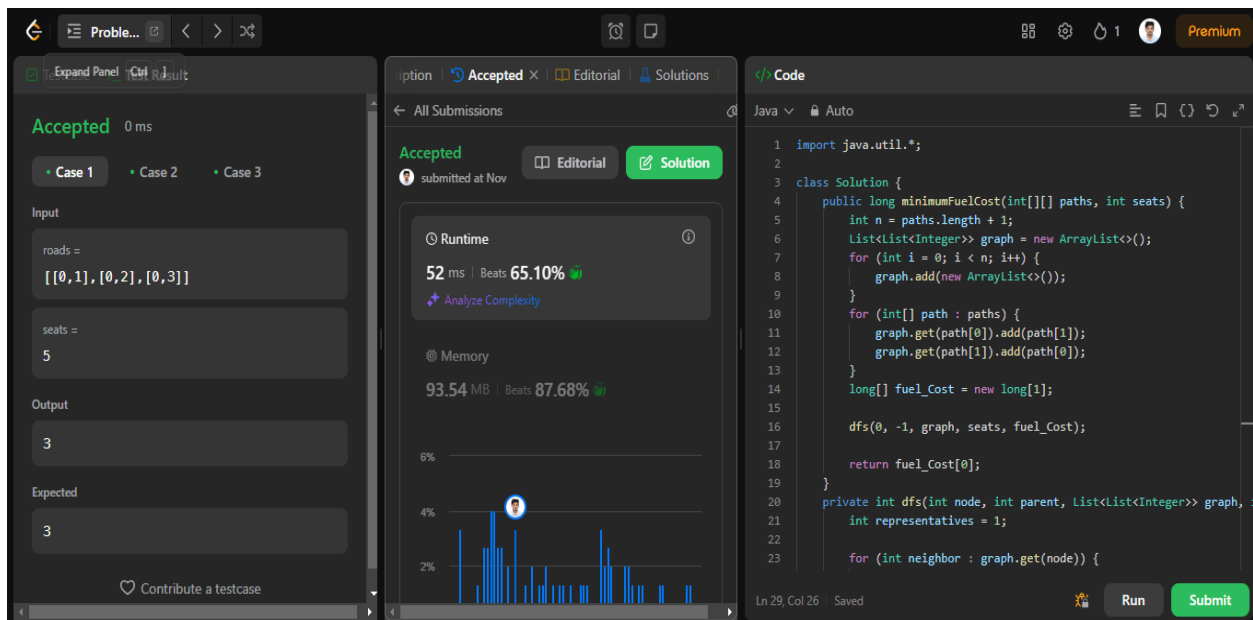
Run

Submit

Approach:

This algorithm aims to find the maximum sum of a subarray of length k with no duplicate values. It uses a sliding window approach to efficiently explore all potential subarrays. The idea is to maintain a window of size k while ensuring all elements in the window are unique. To achieve this, a `HashSet` is used to track the elements within the current window. As the end pointer moves through the array, it checks if the current element is already in the set. If it is, the start pointer is moved forward to shrink the window from the left, removing the duplicate and adjusting the sum accordingly. The `currentAdd` variable tracks the sum of the elements in the current window, and whenever the window reaches the desired length of k , the algorithm compares the current sum to the maximum sum encountered so far (`maxAdd`). By continuously updating the window and ensuring the uniqueness of its elements, the algorithm efficiently computes the maximum sum of the subarray of length k without duplicates.

Maximum FuelCost:



The screenshot displays the LeetCode submission interface for the 'Maximum FuelCost' problem. The problem status is 'Accepted' with a runtime of 0 ms. The input parameters are: roads = [[0,1],[0,2],[0,3]], seats = 5, and the expected output is 3. The performance graph shows a runtime of 52 ms, beating 65.10% of submissions, and a memory usage of 93.54 MB, beating 87.68% of submissions. The code is written in Java and uses a Depth-First Search (DFS) approach to calculate the minimum fuel cost.

```
1 import java.util.*;
2
3 class Solution {
4     public long minimumFuelCost(int[][] paths, int seats) {
5         int n = paths.length + 1;
6         List<List<Integer>> graph = new ArrayList<>();
7         for (int i = 0; i < n; i++) {
8             graph.add(new ArrayList<>());
9         }
10        for (int[] path : paths) {
11            graph.get(path[0]).add(path[1]);
12            graph.get(path[1]).add(path[0]);
13        }
14        long[] fuel_Cost = new long[1];
15
16        dfs(0, -1, graph, seats, fuel_Cost);
17
18        return fuel_Cost[0];
19    }
20    private int dfs(int node, int parent, List<List<Integer>> graph,
21        int representatives = 1;
22
23        for (int neighbor : graph.get(node)) {
```

Approach:

The algorithm calculates the minimum fuel cost required to transport people across a network of cities connected by roads, where each vehicle can carry a maximum of `seats` people. It represents the cities and roads as a graph and uses a Depth-First Search (DFS) to explore the cities. For each city, it recursively calculates the number of trips needed to transport people to its neighbors. The number of trips is determined by dividing the number of people in each subtree by the vehicle's capacity, and the fuel cost accumulates by

summing the trips across all cities. The final result is the total fuel cost required to transport all people efficiently.

Most Profitable Path in a Tree:

The screenshot displays a coding platform interface for the problem "2467. Most Profitable Path in a Tree".

Test Result Panel (Left):

- Status: Accepted (0 ms)
- Case 1 is selected.
- Input:
 - edges = `[[0,1],[1,2],[1,3],[3,4]]`
 - bob = `3`
 - amount = `[-2,4,2,-4,6]`
- Output: `6`
- Expected: (empty field)

Problem Description (Center):

2467. Most Profitable Path in a Tree Solved

Medium Topics Companies Hint

There is an undirected tree with n nodes labeled from 0 to $n - 1$, rooted at node 0 . You are given a 2D integer array `edges` of length $n - 1$ where `edges[i] = [ai, bi]` indicates that there is an edge between nodes a_i and b_i in the tree.

At every node i , there is a gate. You are also given an array of even integers `amount`, where `amount[i]` represents:

- the price needed to open the gate at node i , if `amount[i]` is negative, or,
- the cash reward obtained on opening the gate at node i , otherwise.

Code Editor (Right):

```
1 class Solution {
2     private List<Integer>[] adjacencyList;
3     private int[] nodeProfits;
4     private int[] nodeTimestamps;
5     private int highestProfit = Integer.MIN_VALUE;
6
7     public int mostProfitablePath(int[][] edges, int bobStartNode, int
8
9         int nodeCount = edges.length + 1;
10        adjacencyList = new List<Integer>[nodeCount];
11        nodeTimestamps = new int[nodeCount];
12        nodeProfits = profits;
13        Arrays.setAll(adjacencyList, i -> new ArrayList<>());
14        Arrays.fill(nodeTimestamps, nodeCount);
15        for (var edge : edges) {
16            int u = edge[0], v = edge[1];
17            adjacencyList[u].add(v);
18            adjacencyList[v].add(u);
19        }
20        assignTimestamps(bobStartNode, -1, 0);
21        nodeTimestamps[bobStartNode] = 0;
22        calculateProfit(0, -1, 0, 0);
23        return highestProfit;
24    }
```

Approach:

This algorithm finds the most profitable path in a tree of cities, starting from the root city (node 0). It considers both the main traveler and Bob, who starts from a different node, with Bob's arrival times affecting the profit. The graph is represented using an adjacency list, and each city has an associated profit. The algorithm uses a depth-first search (DFS) to compute the earliest time Bob reaches each city and adjusts the profit accordingly: if the main traveler arrives before Bob, the full profit is added; if Bob arrives first, the profit is halved. The algorithm recursively explores the graph, accumulating profits and tracking the highest possible profit along the way.

Reachable Nodes with Restrictions:

The screenshot shows the LeetCode interface for the problem "2368. Reachable Nodes With Restrictions". The problem is marked as "Solved". The input is $n = 7$, $edges = [[0, 1], [1, 2], [3, 1], [4, 0], [0, 5], [5, 6]]$, and $restricted = [4, 5]$. The output is 4. The description states: "There is an undirected tree with n nodes labeled from 0 to $n - 1$ and $n - 1$ edges. You are given a 2D integer array $edges$ of length $n - 1$ where $edges[i] = [a_i, b_i]$ indicates that there is an edge between nodes a_i and b_i in the tree. You are also given an integer array $restricted$ which represents $restricted$ nodes. Return the **maximum** number of nodes you can reach from node 0 without visiting a restricted node. Note that node 0 will **not** be a restricted node." The code editor shows a Java solution using Depth-First Search (DFS) to explore the graph from node 0, avoiding restricted nodes and marking visited nodes.

```
1 import java.util.*;
2
3 class Solution {
4     public int reachableNodes(int n, int[][] edges, int[] restricted) {
5         List<Integer>[] graph = new List[n];
6         for (int i = 0; i < n; i++) {
7             graph[i] = new ArrayList<>();
8         }
9
10        for (int[] edge : edges) {
11            int u = edge[0];
12            int v = edge[1];
13            graph[u].add(v);
14            graph[v].add(u);
15        }
16
17        Set<Integer> restrictedSet = new HashSet<>();
18        for (int r : restricted) {
19            restrictedSet.add(r);
20        }
21
22        boolean[] visited = new boolean[n];
23        int reachableCount = dfs(0, graph, visited, restrictedSet);
24    }
25}
```

Approach:

This algorithm calculates the number of reachable nodes in a graph, starting from node 0, while avoiding certain restricted nodes. The graph is represented by an adjacency list where each node has a list of its connected neighbors. The algorithm first builds the graph and converts the list of restricted nodes into a set for fast lookup. Then, it uses Depth-First Search (DFS) to explore the graph from node 0. As it traverses the graph, it avoids visiting restricted nodes and ensures each node is visited only once by marking them as visited. For every node that can be reached (and isn't restricted), it increments a counter. The algorithm returns the total count of reachable nodes from the starting node (0).

Find Minimum Diameter after Merging two trees:

The screenshot shows the LeetCode interface for problem 3203. The left panel displays the test result for Case 2, which is 'Accepted' with 0 ms execution time. The input consists of two edge lists: edges1 = [[0,1], [0,2], [0,3], [2,4], [2,5], [3,6], [2,7]] and edges2 = [[0,1], [0,2], [0,3], [2,4], [2,5], [3,6], [2,7]]. The output is 5, which matches the expected result. The middle panel shows the problem description, which states that two undirected trees with n and m nodes are given, and the goal is to find the minimum possible diameter of the resulting tree after merging them by adding one edge. The right panel shows the Java code solution, which uses a two-step BFS approach to find the diameter of each tree, calculate their radii, and then determine the minimum diameter of the merged tree by either connecting the two trees at their radii or by connecting them at a node that minimizes the maximum distance from the connection point to the farthest node in either tree.

```
import java.util.*;

class Solution {
    public int minimumDiameterAfterMerge(int[][] treeEdges1, int[][] treeEdges2) {
        if (treeEdges1.length == 0 && treeEdges2.length == 0) {
            return 1;
        }

        Map<Integer, List<Integer>> treeGraph1 = createAdjacencyList(treeEdges1);
        Map<Integer, List<Integer>> treeGraph2 = createAdjacencyList(treeEdges2);

        int tree1Diameter = findTreeDiameter(treeGraph1);
        int tree2Diameter = findTreeDiameter(treeGraph2);

        int tree1Radius = (tree1Diameter + 1) / 2;
        int tree2Radius = (tree2Diameter + 1) / 2;

        int mergedTreeDiameter = Math.max(tree1Diameter, Math.max(tree2Diameter, tree1Radius + tree2Radius + 1));

        return mergedTreeDiameter;
    }

    private Map<Integer, List<Integer>> createAdjacencyList(int[][] edges) {
        Map<Integer, List<Integer>> graph = new HashMap<>();
        for (int[] edge : edges) {
            graph.putIfAbsent(edge[0], new ArrayList<>());
            graph.putIfAbsent(edge[1], new ArrayList<>());
            graph.get(edge[0]).add(edge[1]);
            graph.get(edge[1]).add(edge[0]);
        }
        return graph;
    }

    private int findTreeDiameter(Map<Integer, List<Integer>> graph) {
        if (graph.isEmpty()) return 0;
        int start = graph.keySet().iterator().next();
        return bfs(start, graph);
    }

    private int bfs(int start, Map<Integer, List<Integer>> graph) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);
        Map<Integer, Integer> dist = new HashMap<>();
        dist.put(start, 0);

        while (!queue.isEmpty()) {
            int u = queue.poll();
            for (int v : graph.get(u)) {
                if (!dist.containsKey(v)) {
                    dist.put(v, dist.get(u) + 1);
                    queue.add(v);
                }
            }
        }

        int maxDist = 0;
        for (int dist : dist.values()) {
            maxDist = Math.max(maxDist, dist);
        }

        return maxDist;
    }
}
```

Approach:

This algorithm calculates the minimum possible diameter of a merged tree formed by connecting two given trees. It starts by constructing adjacency lists for each tree, then calculates the diameter of each tree by finding the longest path between any two nodes using a two-step BFS approach. The diameter of each tree is used to compute the radius, which is half the diameter, rounded up. Finally, the algorithm merges the two trees by adding an edge between them and calculates the new diameter as the maximum of the individual tree diameters or the sum of their radii plus one. This approach ensures that the resulting tree has the smallest possible diameter after the merge.

Number Of Good Paths:

The screenshot shows the LeetCode interface for problem 2421. The left panel displays the test result for Case 1, which is 'Accepted' with 0 ms execution time. The input consists of a values array [1, 3, 2, 1, 3] and an edges array [[0,1], [0,2], [2,3], [2,4]]. The output is 6, which matches the expected result. The middle panel shows the problem description, which states that a tree is given with n nodes and n-1 edges, and the goal is to find the number of good paths. A good path is defined as a simple path where the starting and ending nodes have the same value. The right panel shows the Java code solution, which uses a DFS approach to traverse the tree and count the number of good paths by maintaining a map of the frequency of each value encountered along the path.

```
import java.util.*;

class Solution {
    public int numberOfGoodPaths(int[] vals, int[][] edges) {
        int n = vals.length;

        Map<Integer, List<Integer>> valueToNodes = new TreeMap<>();
        for (int i = 0; i < n; i++) {
            valueToNodes.computeIfAbsent(vals[i], v -> new ArrayList<>());
            valueToNodes.get(vals[i]).add(i);
        }

        List<Integer>[] graph = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            graph[i] = new ArrayList<>();
        }

        for (int[] edge : edges) {
            int u = edge[0], v = edge[1];
            graph[u].add(v);
            graph[v].add(u);
        }

        int[] parent = new int[n];
        int[] size = new int[n];

        // DFS to count good paths
        return dfs(0, -1, graph, vals, valueToNodes, parent, size);
    }

    private int dfs(int node, int parent, List<Integer>[] graph, int[] vals, Map<Integer, List<Integer>> valueToNodes, int[] parent, int[] size) {
        List<Integer> nodes = valueToNodes.get(vals[node]);
        int count = 1; // Count the current node as a good path of length 1
        for (int neighbor : graph[node]) {
            if (neighbor == parent) continue;
            count += dfs(neighbor, node, graph, vals, valueToNodes, parent, size);
        }

        // Count paths where the current node is the root of a good path
        for (int neighbor : graph[node]) {
            if (neighbor == parent) continue;
            if (vals[neighbor] == vals[node]) {
                count++;
            }
        }

        return count;
    }
}
```

Approach:

This solution aims to calculate the number of "good paths" in a graph, where a good path is defined as a path where the values of the nodes along the path are non-decreasing. The approach first groups nodes by their values, then processes them in increasing order of node values. For each value, the algorithm connects nodes that have a smaller or equal value to their neighbors using a union-find (disjoint-set) data structure. This allows us to efficiently track connected components of nodes that are part of a valid path. After processing the nodes with each value, the algorithm counts the number of good paths in each component and accumulates them. The number of good paths in a component of size k is given by $k * (k + 1) / 2$. The union-find operations help in managing the dynamic connectivity of nodes as the graph is processed by node values.

Longest Path With Different Adjacent Character:

The screenshot shows the LeetCode interface for problem 2246. On the left, the 'Testcase' tab shows 'Accepted' status with 0 ms execution time. The input is `parent = [-1,0,0,1,1,2]` and `s = "abacbe"`. The output is `3`, which matches the expected result. The main panel shows the problem description: 'You are given a tree (i.e. a connected, undirected graph that has no cycles) rooted at node 0 consisting of nodes numbered from 0 to n - 1. The tree is represented by a 0-indexed array parent of size n where parent[i] is the parent of node i. Since 0 is the root, parent[0] = -1. You are also given a string s of length n, where s[i] is the character assigned to node i. Return the length of the longest path in the tree such that no pair of adjacent nodes on the path have the same character assigned to them.' The right panel shows the Java code for the solution.

```
1 import java.util.*;
2
3 class Solution {
4     private int longestPathLength = 0;
5
6     public int longestPath(int[] parent, String labels) {
7         int nodeCount = parent.length;
8         List<Integer>[] tree = new List<Integer>[nodeCount];
9         Arrays.setAll(tree, i -> new ArrayList<>());
10
11         for (int child = 1; child < nodeCount; child++) {
12             int parentNode = parent[child];
13             tree[parentNode].add(child);
14         }
15         findLongestPath(0, tree, labels);
16         return longestPathLength;
17     }
18
19     private int findLongestPath(int currentNode, List<Integer>[] tree, String labels) {
20         int longestBranch = 0;
21         int secondLongestBranch = 0;
22
23         for (int child : tree[currentNode]) {
24             int childPathLength = findLongestPath(child, tree, labels);
```

Approach:

This solution computes the longest path in a tree, where a valid path is one that only includes nodes with different labels. The tree is represented by an array `parent` where each element specifies the parent of a node. The `longestPath` method builds an adjacency list tree to represent the tree structure and then calls the helper function `findLongestPath` starting from the root node.

The helper function recursively traverses the tree, calculating the longest path through each node. For each node, it compares the labels of the current node and its children. If the labels are different, it updates the longest and second-longest branches found among the children, ensuring that only paths with different labels are considered. The result for each node is the sum of the two longest branches, plus one for the current node. The

longestPathLength variable is updated as the maximum of the current longest path and the calculated path for each node.

Dynamic Programming

Generate Parentheses

The screenshot displays the LeetCode interface for the '22. Generate Parentheses' problem. The left sidebar shows the 'Testcase' tab with 'Case 1' selected, indicating an 'Accepted' status with a runtime of 0 ms. The input for Case 1 is `n = 3`. The output is a list of strings: `["((()))", "(()())", "(())()", "()(())", "()()()"]`. The expected output is the same list. The main area shows the problem description: 'Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.' It includes two examples: Example 1 with input `n = 3` and output `["((()))", "(()())", "(())()", "()(())", "()()()"]`; and Example 2 with input `n = 1` and output `["()"]`. The right sidebar shows the 'Code' tab with a Java solution. The solution uses a recursive approach with a helper method `buildCombinations` that takes a list of strings, a current string, and counts of open and close parentheses. It adds an open parenthesis if the open count is less than `maxPairs` and a close parenthesis if the close count is less than the open count. When the current string length equals `maxPairs * 2`, it adds the current string to the combinations list and returns.

```
1 import java.util.*;
2
3 class Solution {
4     public List<String> generateParenthesis(int n) {
5         List<String> combinations = new ArrayList<>();
6         buildCombinations(combinations, "", 0, 0, n);
7         return combinations;
8     }
9
10    private void buildCombinations(List<String> combinations, String current,
11                                   int openCount, int closeCount, int maxPairs) {
12        if (current.length() == maxPairs * 2) {
13            combinations.add(current);
14            return;
15        }
16        if (openCount < maxPairs) {
17            buildCombinations(combinations, current + "(", openCount + 1, closeCount, maxPairs);
18        }
19        if (closeCount < openCount) {
20            buildCombinations(combinations, current + ")", openCount, closeCount + 1, maxPairs);
21        }
22    }
23 }
```

Approach:

This solution generates all possible valid combinations of parentheses for a given number `n` of pairs. The process starts by recursively building combinations of parentheses. At each step, the algorithm adds an open parenthesis if the number of open parentheses is less than `n` and adds a close parenthesis if it's allowed (i.e., there are more open parentheses than close ones). Once a combination reaches the length of $2 * n$, it's considered valid and added to the result list. By ensuring that the parentheses are balanced at every step, this approach efficiently generates all well-formed parentheses combinations.

Maximum Subarray:

The screenshot shows a coding platform interface for the "53. Maximum Subarray" problem. The left panel displays the "Test Result" section, indicating the solution is "Accepted" with a runtime of 0 ms. It shows the input array `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]` and the output `6`. The middle panel contains the problem description, which asks to find the subarray with the largest sum. It includes two examples: Example 1 with input `nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]` and output `6`, and Example 2 with input `nums = [1]` and output `1`. The right panel shows the Java code for the solution, which implements Kadane's Algorithm. The code is as follows:

```
1 class Solution {
2     public int maxSubArray(int[] nums) {
3         int currentSubarraySum = 0;
4         int maxSubarraySum = nums[0];
5
6         for (int num : nums) {
7             currentSubarraySum = Math.max(num, currentSubarraySum + num);
8             maxSubarraySum = Math.max(maxSubarraySum, currentSubarraySum);
9         }
10        return maxSubarraySum;
11    }
12 }
13
14
```

Approach:

This solution uses Kadane's Algorithm to efficiently find the maximum sum of a contiguous subarray within an integer array. It works by iterating through each element and deciding whether to start a new subarray with the current element or to add the current element to the existing subarray. The algorithm keeps track of the maximum sum encountered so far, updating it as needed. By the end of the loop, it returns the largest sum found. This approach ensures that the solution runs in linear time, making it both fast and effective for this problem.

Unique Binary Search Trees:

The screenshot shows a coding platform interface for the "95. Unique Binary Search Trees II" problem. The left panel displays the "Test Result" section, indicating the solution is "Accepted" with a runtime of 0 ms. It shows the input `n = 3` and the output, which is a list of all structurally unique BSTs for `n = 3`. The middle panel contains the problem description, which asks to return all the structurally unique BST's (binary search trees) which has exactly `n` nodes of unique values from `1` to `n`. It includes an example with `n = 3` and a diagram showing the output. The right panel shows the Java code for the solution, which uses a recursive approach to generate all unique BSTs. The code is as follows:

```
16 import java.util.*;
17
18 public class Solution {
19
20     public List<TreeNode> generateTrees(int n) {
21         if (n == 0) {
22             return new ArrayList<>();
23         }
24         return generateTrees(1, n);
25     }
26
27     private List<TreeNode> generateTrees(int start, int end) {
28         List<TreeNode> allTrees = new ArrayList<>();
29         if (start > end) {
30             allTrees.add(null);
31             return allTrees;
32         }
33         for (int rootValue = start; rootValue <= end; rootValue++) {
34             List<TreeNode> leftSubtrees = generateTrees(start, rootValue - 1);
35             List<TreeNode> rightSubtrees = generateTrees(rootValue + 1, end);
36             for (TreeNode left : leftSubtrees) {
37                 for (TreeNode right : rightSubtrees) {
38                     TreeNode root = new TreeNode(rootValue);
39                     root.left = left;
40                     root.right = right;
41                     allTrees.add(root);
42                 }
43             }
44         }
45         return allTrees;
46     }
47 }
```

Approach:

The goal is to construct all unique binary search trees (BSTs) that may be built with integers ranging from 1 to n. The method entails iteratively selecting each integer as the root within the specified range. It produces all feasible left and right subtrees for each root by combining the remaining integers. Once the subtrees have been produced, they are joined with the current root to create various BSTs. If n equals zero, the solution produces an empty list since no trees can be created. The output is a list of all conceivable unique BSTs.

Edit Distance:

The screenshot shows a LeetCode interface for the problem "72. Edit Distance". On the left, the "Testcase" tab shows "Accepted" status with 0 ms execution time. The input is word1 = "horse" and word2 = "ros", with an expected output of 3. The middle panel shows the problem description: "Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2." The allowed operations are Insert a character, Delete a character, and Replace a character. An example is provided: Input: word1 = "horse", word2 = "ros", Output: 3. On the right, the "Code" tab shows a Java solution using dynamic programming. The code defines a minDistance method that uses a 2D DP array to calculate the minimum edit distance between two strings.

```
1 class Solution {
2     public int minDistance(String word1, String word2) {
3         int length1 = word1.length();
4         int length2 = word2.length();
5
6         int[][] dp = new int[length1 + 1][length2 + 1];
7
8         for (int i = 0; i <= length1; i++) {
9             dp[i][0] = i;
10        }
11        for (int j = 0; j <= length2; j++) {
12            dp[0][j] = j;
13        }
14
15        for (int i = 1; i <= length1; i++) {
16            for (int j = 1; j <= length2; j++) {
17                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
18                    dp[i][j] = dp[i - 1][j - 1];
19                } else {
20                    dp[i][j] = Math.min(dp[i - 1][j],
21                                     Math.min(dp[i][j - 1],
22                                               dp[i - 1][j - 1]))
23                        + 1;
24            }
25        }
26    }
27 }
```

Approach:

The aim is to determine the smallest number of operations needed to convert one string to another, where the allowable operations are insertions, deletions, and replacements. The solution uses dynamic programming to generate a table, with each entry representing the smallest number of operations required to convert one substring of the first word into a substring of the second word. Starting with base cases (empty strings), the method checks characters at each point, updating the table using the three operations. The final entry in the table specifies the minimum number of operations required for the whole transformation.

Unique Binary Search Trees:

The screenshot shows the LeetCode interface for problem 96, 'Unique Binary Search Trees'. The left panel shows the test result for Case 1, which is 'Accepted' with a runtime of 0 ms. The input is n = 3 and the output is 5. The middle panel shows the problem description: 'Given an integer n, return the number of structurally unique BST's (binary search trees) which has exactly n nodes of unique values from 1 to n.' It includes an example diagram for n=3 showing 5 unique trees. The right panel shows the code editor with a Java solution using dynamic programming.

```
class Solution {
    public int numTrees(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            for (int j = 1; j <= i; j++) {
                dp[i] += dp[j - 1] * dp[i - j];
            }
        }
        return dp[n];
    }
}
```

Approach:

This approach uses dynamic programming to determine the number of distinct Binary Search Trees (BSTs) that may be created with n nodes. The goal is to take advantage of the fact that at any node i may serve as the root of a BST, with nodes to its left and right forming smaller subtrees. The dynamic programming array $dp[i]$ contains the number of distinct BSTs that may be created with i nodes. Beginning with base instances where $dp[0]$ and $dp[1]$ are both 1 (representing one empty tree and one tree with a single node, respectively), the method repeatedly computes the number of trees for higher n .

For each i , it evaluates all potential root nodes j and adds the products of the number of left and right subtrees, resulting in the total number of unique BSTs. The final result is stored in $dp[n]$.

Unique Paths:

Approach:

This solution calculates the number of unique paths in a grid from the top-left corner to the bottom-right corner, where you can only move either down or right. It uses dynamic programming, starting by initializing the first row and the first column to 1, since there's only one way to move along those edges — either by always going right for the first row or always going down for the first column. Then, for each remaining cell in the grid, the number of ways to reach that cell is the sum of the number of ways to reach the cell directly above it and the cell directly to the left of it. This way, the solution builds up the

total number of ways to reach each cell, and finally, the number of unique paths is found at the bottom-right corner of the grid.

62. Unique Paths Solved

Medium Topics Companies

There is a robot on an $m \times n$ grid. The robot is initially located at the **top-left corner** (i.e. `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e. `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers m and n , return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to $2 * 10^5$.

Example 1:

```

class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];

        for (int i = 0; i < m; i++) {
            dp[i][0] = 1;
        }
        for (int j = 0; j < n; j++) {
            dp[0][j] = 1;
        }

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

```

Minimum Path Sum:

Minimum Path Sum

Accepted Runtime: 0 ms

Case 1 Case 2

Input

grid =
[[1,3,1],[1,5,1],[4,2,1]]

Output

7

Expected

7

Contribute a testcase

Accepted submitted at Nov

Runtime
4 ms | Beats 62.51%

Memory
47.54 MB | Beats 45.39%

```

class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;

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

        for (int i = 1; i < m; i++) {
            grid[i][0] += grid[i - 1][0];
        }

        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                grid[i][j] += Math.min(grid[i - 1][j], grid[i][j - 1]);
            }
        }
        return grid[m - 1][n - 1];
    }
}

```

Approach:

This approach calculates the minimal path total from a grid's top-left corner to the bottom-right corner, with each step moving either right or down. The solution is built using dynamic programming. First, it updates the first row and column since there is only one method to

access each cell in those areas: go right for the row and down for the column. The value of each remaining cell is then updated by adding the current cell's value to the least of the values from the cell above it or the cell to its left, ensuring that the route sum is reduced at each step. Finally, the minimal path total is in the bottom-right corner.