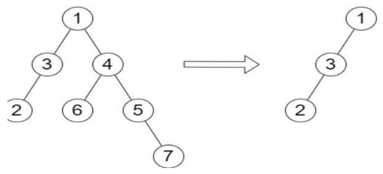
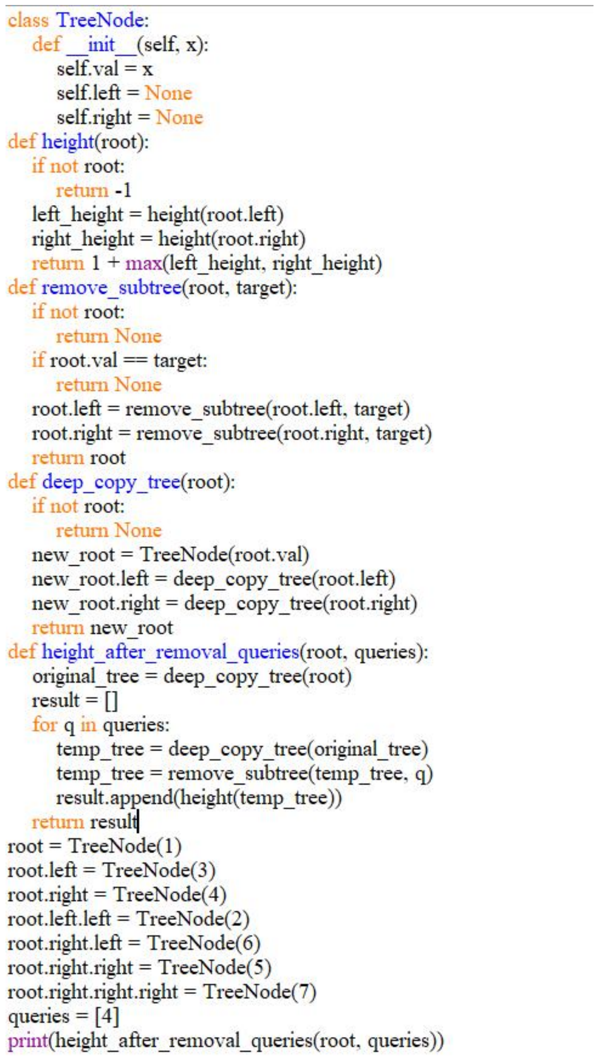
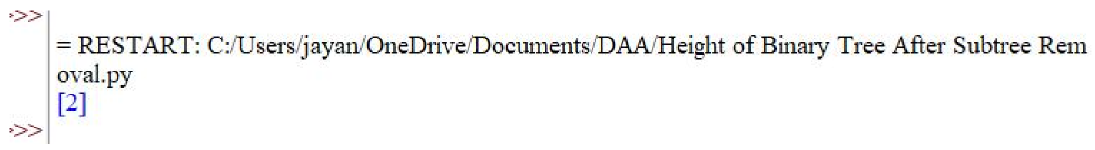
1.Height of Binary Tree After Subtree Removal Queries You are given the root of a binary tree with n nodes. Each node is assigned a unique value from 1 to n. You are also given an array queries of size m. You have to perform independent queries on the tree where in the ith query you do the following: ● Remove the subtree rooted at the node with the value queries[i] from the tree. It is guaranteed that queries[i] will not be equal to the value of the root. Return an array answer of size m where answer[i] is the height of the tree after performing the ith query. Note: ● The queries are independent, so the tree returns to its initial state after each query. ● The height of a tree is the number of edges in the longest simple path from the root to some node in the tree.



Example 1: Input: root = [1,3,4,2,null,6,5,null,null,null,null,null,7], queries = [4] Output: [2] Explanation: The diagram above shows the tree after removing the subtree rooted at nodewith value 4. The height of the tree is 2 (The path 1 -> 3 -> 2)



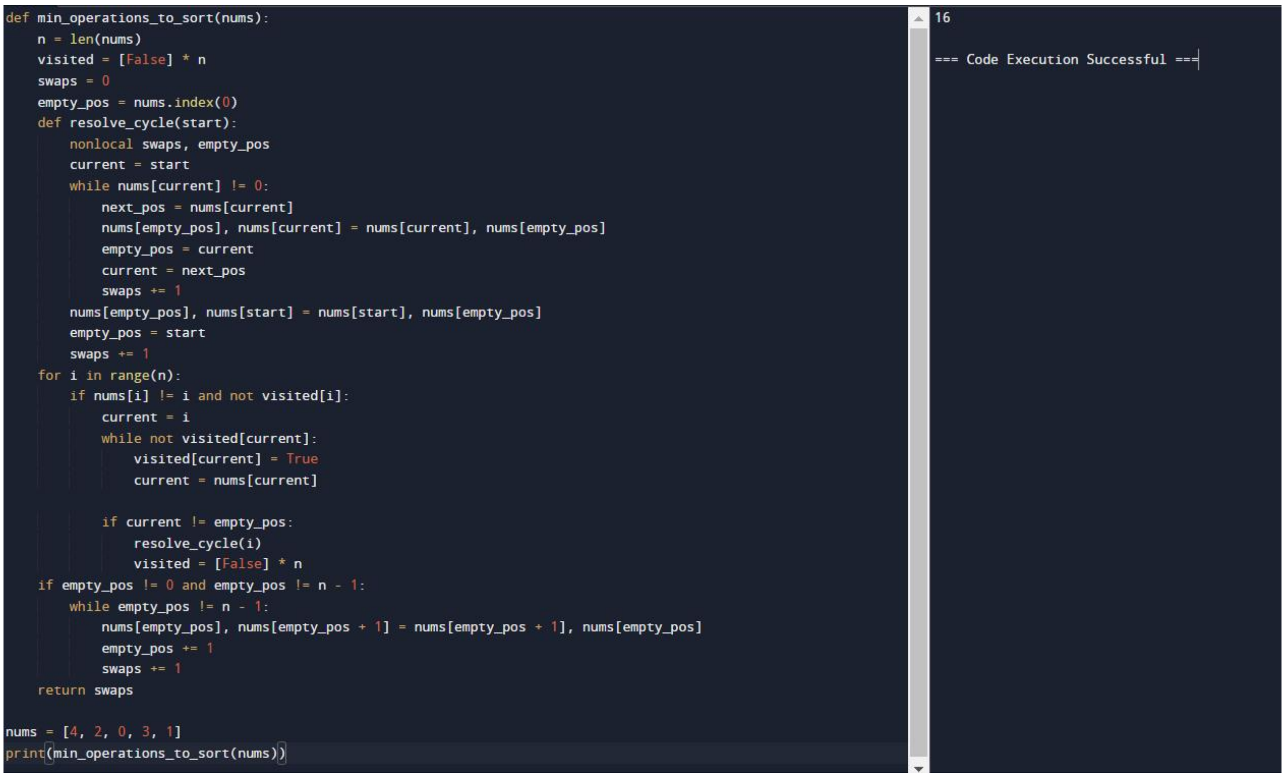
```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
def height(root):
    if not root:
        return -1
    left_height = height(root.left)
    right_height = height(root.right)
    return 1 + max(left_height, right_height)
def remove_subtree(root, target):
    if not root:
        return None
    if root.val == target:
        return None
    root.left = remove_subtree(root.left, target)
    root.right = remove_subtree(root.right, target)
    return root
def deep_copy_tree(root):
    if not root:
        return None
    new_root = TreeNode(root.val)
    new_root.left = deep_copy_tree(root.left)
    new_root.right = deep_copy_tree(root.right)
    return new_root
def height_after_removal_queries(root, queries):
    original_tree = deep_copy_tree(root)
    result = []
    for q in queries:
        temp_tree = deep_copy_tree(original_tree)
        temp_tree = remove_subtree(temp_tree, q)
        result.append(height(temp_tree))
    return result
root = TreeNode(1)
root.left = TreeNode(3)
root.right = TreeNode(4)
root.left.left = TreeNode(2)
root.right.left = TreeNode(6)
root.right.right = TreeNode(5)
root.right.right.right = TreeNode(7)
queries = [4]
print(height_after_removal_queries(root, queries))
```



```
>>>
    = RESTART: C:/Users/jayan/OneDrive/Documents/DAA/Height of Binary Tree After Subtree Rem
    oval.py
    [2]
>>>
```

2. Sort Array by Moving Items to Empty Space You are given an integer array nums of size n containing each element from0 to n - 1(inclusive). Each of the elements from 1 to n - 1 represents an item, and the element 0represents an empty space. In one operation, you can move any item to the empty space. nums is considered to be sorted if the numbers of all the items are in ascending order and the empty space is either at the beginning or at the end of the array. For example, if n = 4, nums is sorted if: ● nums = [0,1,2,3] or ● nums = [1,2,3,0] ...and considered to be unsorted otherwise. Return the minimum number of operations needed to sort nums.

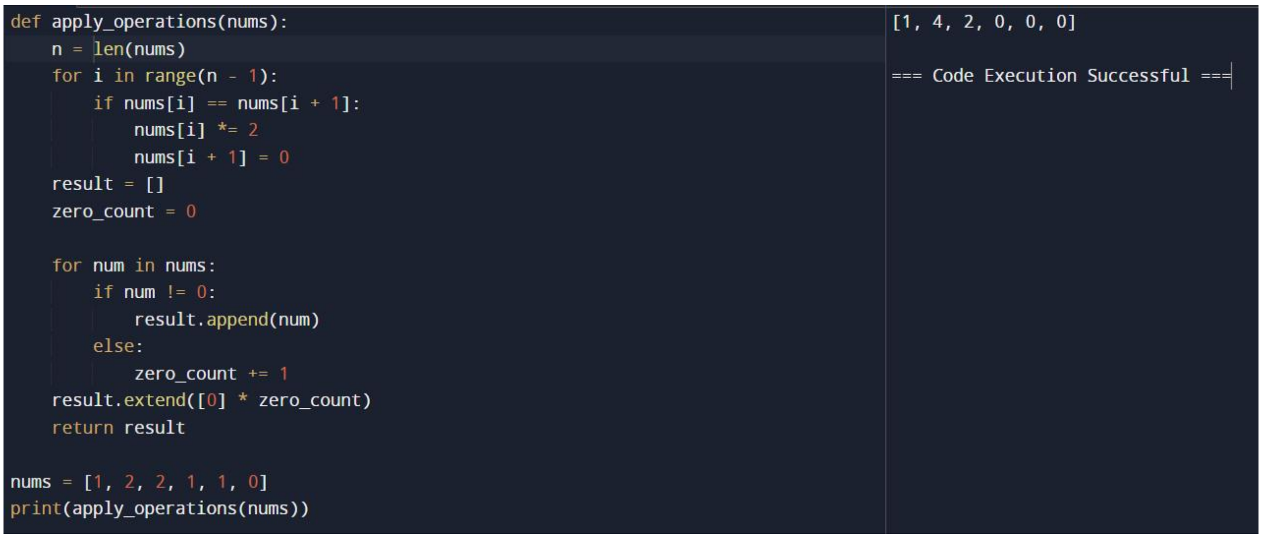
Example 1: Input: nums = [4,2,0,3,1] Output: 3 Explanation: - Move item 2 to the empty space. Now, nums = [4,0,2,3,1]. - Move item 1 to the empty space. Now, nums = [4,1,2,3,0]. - Move item 4 to the empty space. Now, nums = [0,1,2,3,4]. It can be proven that 3 is the minimum number of operations needed.



```python
def min_operations_to_sort(nums):
    n = len(nums)
    visited = [False] * n
    swaps = 0
    empty_pos = nums.index(0)
    def resolve_cycle(start):
        nonlocal swaps, empty_pos
        current = start
        while nums[current] != 0:
            next_pos = nums[current]
            nums[empty_pos], nums[current] = nums[current], nums[empty_pos]
            empty_pos = current
            current = next_pos
            swaps += 1
        nums[empty_pos], nums[start] = nums[start], nums[empty_pos]
        empty_pos = start
        swaps += 1
    for i in range(n):
        if nums[i] != i and not visited[i]:
            current = i
            while not visited[current]:
                visited[current] = True
                current = nums[current]

            if current != empty_pos:
                resolve_cycle(i)
                visited = [False] * n
    if empty_pos != 0 and empty_pos != n - 1:
        while empty_pos != n - 1:
            nums[empty_pos], nums[empty_pos + 1] = nums[empty_pos + 1], nums[empty_pos]
            empty_pos += 1
            swaps += 1
    return swaps

nums = [4, 2, 0, 3, 1]
print(min_operations_to_sort(nums))
```

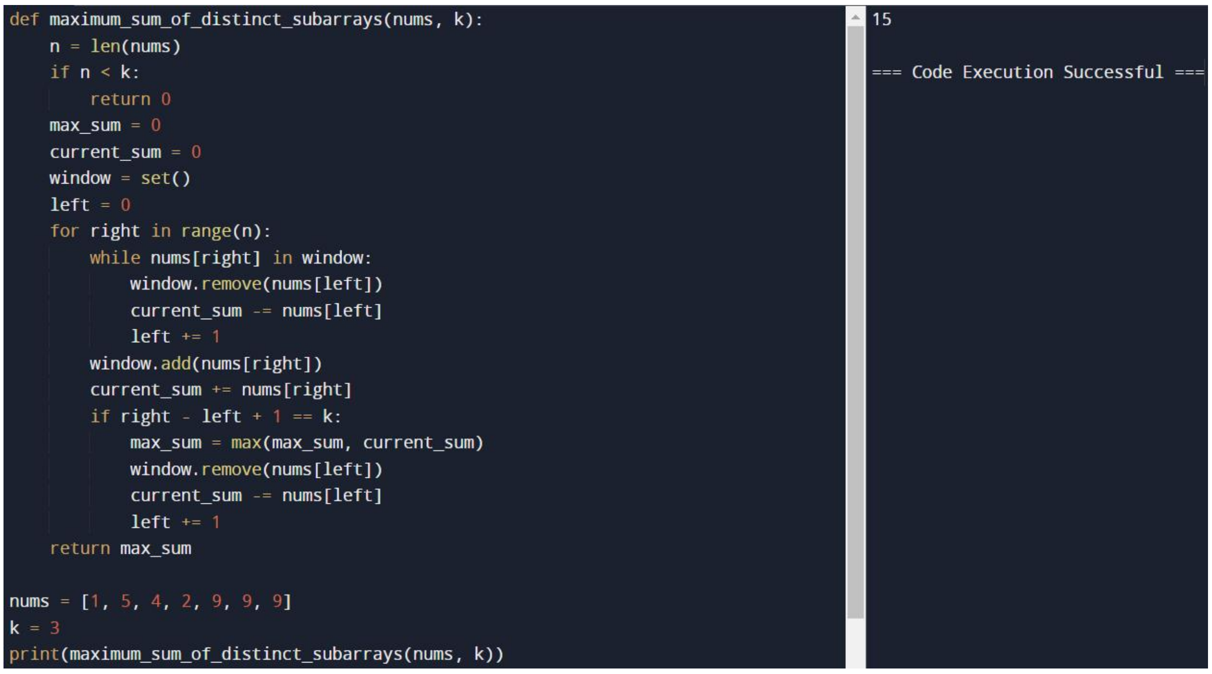
```
16

=== Code Execution Successful ===
```

3. Apply Operations to an Array You are given a 0-indexed array nums of size n consisting of non-negative integers. You need to apply n - 1 operations to this array where, in the ith operation (0-indexed), you will applythe following on the ith element of nums: ● If nums[i] == nums[i + 1], then multiply nums[i] by 2 and set nums[i + 1] to 0. Otherwise, you skip this operation. After performing all the operations, shift all the 0's to the end of the array. ● For example, the array [1,0,2,0,0,1] after shifting all its 0's to the end, is [1,2,1,0,0,0]. Return the resulting array.Note that the operations are applied sequentially, not all at once. Example 1: Input: nums = [1,2,2,1,1,0] Output: [1,4,2,0,0,0] Explanation: We do the following operations: - i = 0: nums[0] and nums[1] are not equal, so we skip this operation. - i = 1: nums[1] and nums[2] are equal, we multiply nums[1] by 2 and change nums[2] to0. The array becomes [1,4,0,1,1,0]. - i = 2: nums[2] and nums[3] are not equal, so we skip this operation. - i = 3: nums[3] and nums[4] are equal, we multiply nums[3] by 2 and change nums[4] to0. The array becomes [1,4,0,2,0,0]. - i = 4: nums[4] and nums[5] are equal, we multiply nums[4] by 2 and change nums[5] to 0. The array becomes [1,4,0,2,0,0]. After that, we shift the 0's to the end, which gives the array [1,4,2,0,0,0].


```python
def apply_operations(nums):
    n = len(nums)
    for i in range(n - 1):
        if nums[i] == nums[i + 1]:
            nums[i] *= 2
            nums[i + 1] = 0
    result = []
    zero_count = 0

    for num in nums:
        if num != 0:
            result.append(num)
        else:
            zero_count += 1
    result.extend([0] * zero_count)
    return result

nums = [1, 2, 2, 1, 1, 0]
print(apply_operations(nums))
```

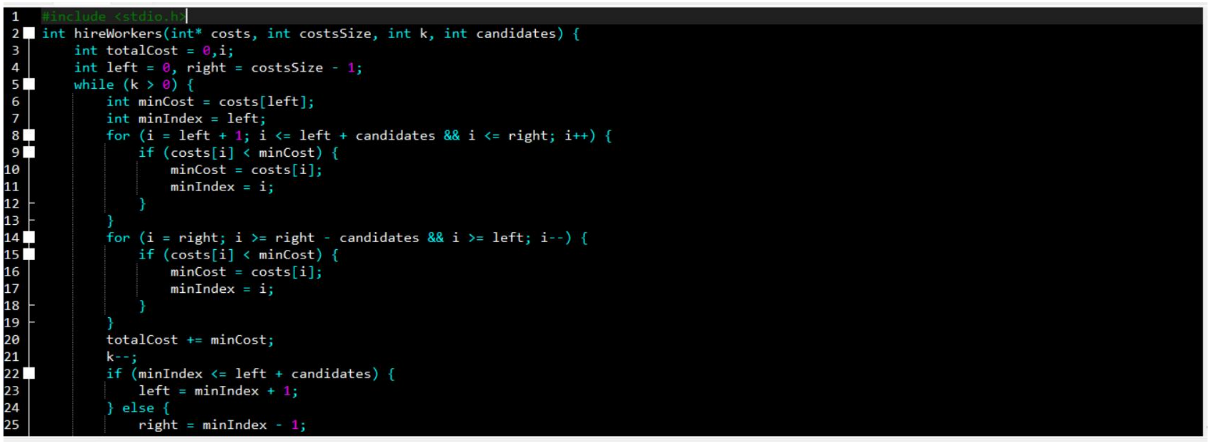
```
[1, 4, 2, 0, 0, 0]

=== Code Execution Successful ===
```

4. Maximum Sum of Distinct Subarrays With Length K 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. Example 1: Input: nums = [1,5,4,2,9,9,9], k = 3 Output: 15 Explanation: The subarrays of nums with length 3 are: - [1,5,4] which meets the requirements and has a sum of 10. - [5,4,2] which meets the requirements and has a sum of 11. - [4,2,9] which meets the requirements and has a sum of 15. - [2,9,9] which does not meet the requirements because the element 9 is repeated. - [9,9,9] which does not meet the requirements because the element 9 is repeated. We return 15 because it is the maximum subarray sum of all the subarrays that meet the conditions.

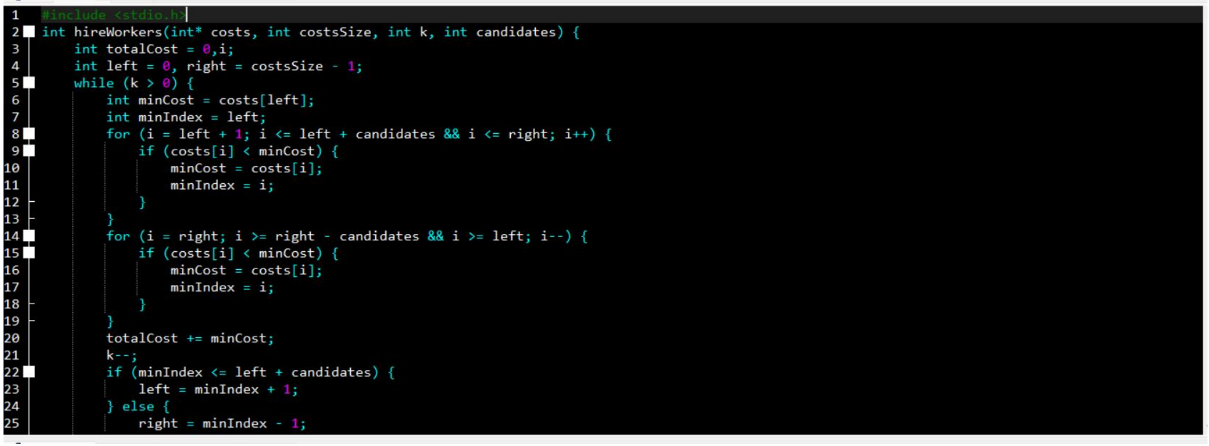

```python
def maximum_sum_of_distinct_subarrays(nums, k):
    n = len(nums)
    if n < k:
        return 0
    max_sum = 0
    current_sum = 0
    window = set()
    left = 0
    for right in range(n):
        while nums[right] in window:
            window.remove(nums[left])
            current_sum -= nums[left]
            left += 1
        window.add(nums[right])
        current_sum += nums[right]
        if right - left + 1 == k:
            max_sum = max(max_sum, current_sum)
            window.remove(nums[left])
            current_sum -= nums[left]
            left += 1
    return max_sum

nums = [1, 5, 4, 2, 9, 9, 9]
k = 3
print(maximum_sum_of_distinct_subarrays(nums, k))
```

```
15

=== Code Execution Successful ===
```

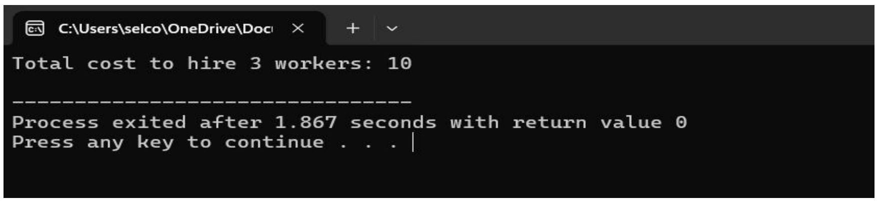
5. Total Cost to Hire K Workers You are given a 0-indexed integer array costs where costs[i] is the cost of hiring the ith worker. You are also given two integers k and candidates. We want to hire exactly k workers according to the following rules: ● You will run k sessions and hire exactly one worker in each session. ● In each hiring session, choose the worker with the lowest cost from either the first candidates workers or the last candidates workers. Break the tie by the smallest index. ○ For example, if costs = [3,2,7,7,1,2] and candidates = 2, then in the first hiring session, we will choose the 4th worker because they have the lowest cost [3,2,7,7,1,2]. ○ In the second hiring session, we will choose 1st worker because they have the same lowest cost as 4th worker but they have the smallest index [3,2,7,7,2]. Please note that the indexing may be changed in the process. ● If there are fewer than candidate workers remaining, choose the worker with the lowest cost among them. Break the tie by the smallest index. ● A worker can only be chosen once. Return the total cost to hire exactly k workers. Example 1: Input: costs = [17,12,10,2,7,2,11,20,8], k = 3, candidates = 4 Output: 11 Explanation: We hire 3 workers in total. The total cost is initially 0. - In the first hiring round we choose the worker from [17,12,10,2,7,2,11,20,8]. The lowest cost is 2, and we break the tie by the smallest index, which is 3. The total cost = 0 +2=2. - In the second hiring round we choose the worker from [17,12,10,7,2,11,20,8]. The lowest cost is 2 (index 4). The total cost = 2 + 2 = 4. - In the third hiring round we choose the worker from [17,12,10,7,11,20,8]. The lowest cost is7 (index 3). The total cost = 4 + 7 = 11. Notice that the worker with index 3 was common in the first and last four workers. The total hiring cost is 11.



```c
#include <stdio.h>
int hireWorkers(int* costs, int costsSize, int k, int candidates) {
    int totalCost = 0,i;
    int left = 0, right = costsSize - 1;
    while (k > 0) {
        int minCost = costs[left];
        int minIndex = left;
        for (i = left + 1; i <= left + candidates && i <= right; i++) {
            if (costs[i] < minCost) {
                minCost = costs[i];
                minIndex = i;
            }
        }
        for (i = right; i >= right - candidates && i >= left; i--) {
            if (costs[i] < minCost) {
                minCost = costs[i];
                minIndex = i;
            }
        }
        totalCost += minCost;
        k--;
        if (minIndex <= left + candidates) {
            left = minIndex + 1;
        } else {
            right = minIndex - 1;
```

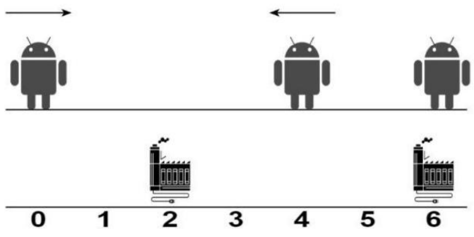


```c
#include <stdio.h>
int hireWorkers(int* costs, int costsSize, int k, int candidates) {
    int totalCost = 0,i;
    int left = 0, right = costsSize - 1;
    while (k > 0) {
        int minCost = costs[left];
        int minIndex = left;
        for (i = left + 1; i <= left + candidates && i <= right; i++) {
            if (costs[i] < minCost) {
                minCost = costs[i];
                minIndex = i;
            }
        }
        for (i = right; i >= right - candidates && i >= left; i--) {
            if (costs[i] < minCost) {
                minCost = costs[i];
                minIndex = i;
            }
        }
        totalCost += minCost;
        k--;
        if (minIndex <= left + candidates) {
            left = minIndex + 1;
        } else {
            right = minIndex - 1;
```

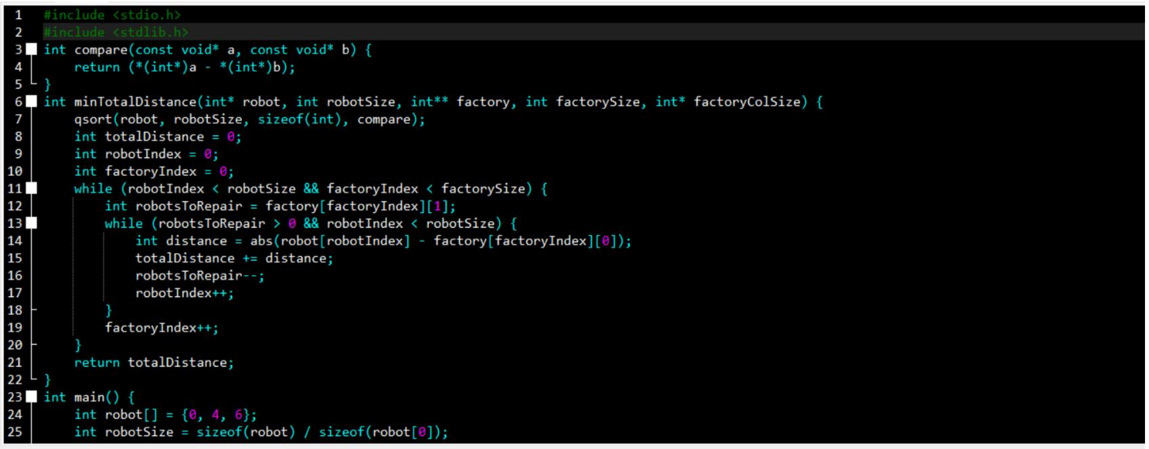


```
Total cost to hire 3 workers: 10

--------------------------------
Process exited after 1.867 seconds with return value 0
Press any key to continue . . .
```

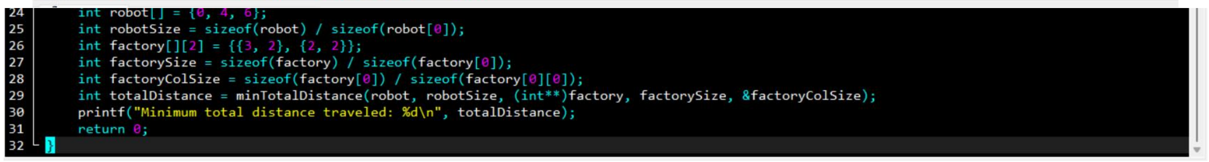
6. Minimum Total Distance Travelled There are some robots and factories on the X-axis. You are given an integer array robot where robot[i] is the position of the ith robot. You are also given a 2D integer array factory where factory[j] = [positionj, limitj] indicates that position j is the position of the jth factory and that the jth factory can repair at most limitj robots. The positions of each robot are unique. The positions of each factory are also unique. Note that a robot can be in the same position as a factory initially. All the robots are initially broken; they keep moving in one direction. The direction could be the negative or the positive direction of the X-axis. When a robot reaches a factory that did not reach its limit, the factory repairs the robot, and it stops moving. At any moment, you can set the initial direction of moving for some robot. Your target is to minimize the total distance travelled by all the robots. Return the minimum total distance travelled by all the robots. The test cases are generated such that all the robots can be repaired. Note that All robots move at the same speed. ● If two robots move in the same direction, they will never collide. ● If two robots move in opposite directions and they meet at some point, they do not collide. They cross each other. ● If a robot passes by a factory that reached its limits, it crosses it as if it does not exist. ● If the robot moved from a position x to a position y, the distance it moved is |y- x|.



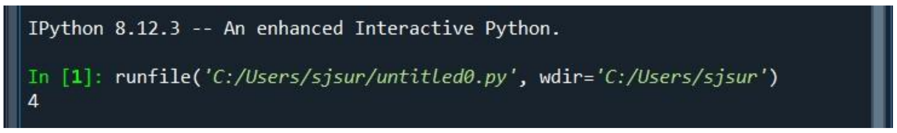
Example 1: Input: robot = [0,4,6], factory = [[2,2],[6,2]] Output: 4 Explanation: As shown in the figure: - The first robot at position 0 moves in the positive direction. It will be repaired at the first factory. - The second robot at position 4 moves in the negative direction. It will be repaired at the first factory. - The third robot at position 6 will be repaired at the second factory. It does not need to move. The limit of the first factory is 2, and it fixed 2 robots. The limit of the second factory is 2, and it fixed 1 robot. The total distance is |2 - 0| + |2 - 4| + |6 - 6| = 4. It can be shown that we cannot achieve a better total distance than 4.



```c
#include <stdio.h>
#include <stdlib.h>
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}
int minTotalDistance(int* robot, int robotSize, int** factory, int factorySize, int* factoryColSize) {
    qsort(robot, robotSize, sizeof(int), compare);
    int totalDistance = 0;
    int robotIndex = 0;
    int factoryIndex = 0;
    while (robotIndex < robotSize && factoryIndex < factorySize) {
        int robotsToRepair = factory[factoryIndex][1];
        while (robotsToRepair > 0 && robotIndex < robotSize) {
            int distance = abs(robot[robotIndex] - factory[factoryIndex][0]);
            totalDistance += distance;
            robotsToRepair--;
            robotIndex++;
        }
        factoryIndex++;
    }
    return totalDistance;
}
int main() {
    int robot[] = {0, 4, 6};
    int robotSize = sizeof(robot) / sizeof(robot[0]);
```

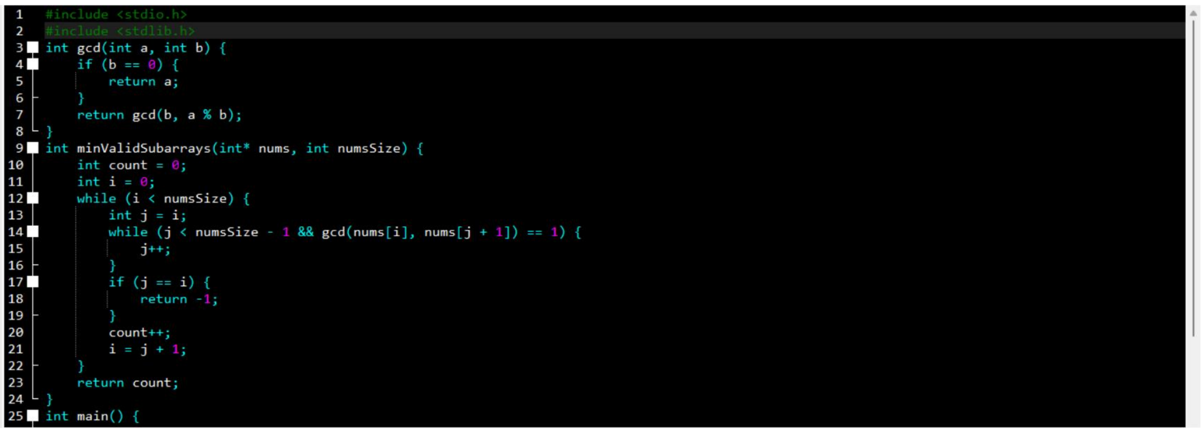


```c
    int robot[] = {0, 4, 6};
    int robotSize = sizeof(robot) / sizeof(robot[0]);
    int factory[][2] = {{3, 2}, {2, 2}};
    int factorySize = sizeof(factory) / sizeof(factory[0]);
    int factoryColSize = sizeof(factory[0]) / sizeof(factory[0][0]);
    int totalDistance = minTotalDistance(robot, robotSize, (int**)factory, factorySize, &factoryColSize);
    printf("Minimum total distance traveled: %d\n", totalDistance);
    return 0;
}
```

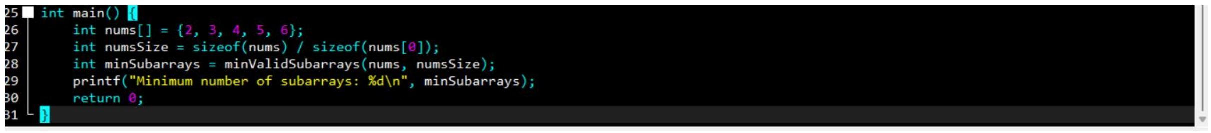


```
IPython 8.12.3 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/sjsur/untitled0.py', wdir='C:/Users/sjsur')
4
```

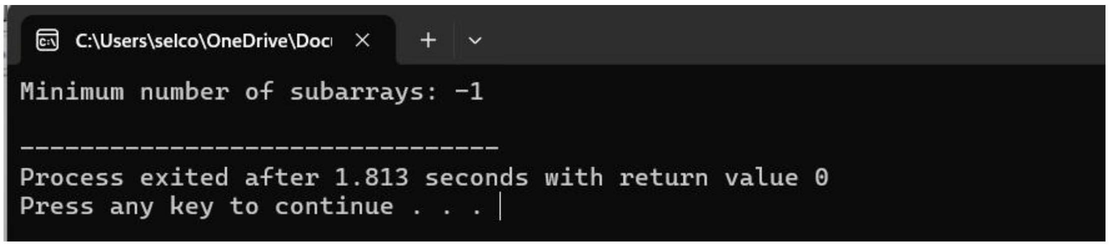
7. Minimum Subarrays in a Valid Split You are given an integer array nums. Splitting of an integer array nums into subarrays is valid if: ● the greatest common divisor of the first and last elements of each subarray is greater than 1, and ● each element of nums belongs to exactly one subarray. Return the minimum number of subarrays in a valid subarray splitting of nums. If a valid subarray splitting is not possible, return -1. Note that: ● The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers. ● A subarray is a contiguous non-empty part of an array. Example 1: Input: nums = [2,6,3,4,3] Output: 2 Explanation: We can create a valid split in the following way: [2,6] | [3,4,3]. - The starting element of the 1st subarray is 2 and the ending is 6. Their greatest common divisor is 2, which is greater than 1. - The starting element of the 2nd subarray is 3 and the ending is 3. Their greatest common divisor is 3, which is greater than 1. It can be proved that 2 is the minimum number of subarrays that we can obtain in a valid split.



```c
#include <stdio.h>
#include <stdlib.h>
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}
int minValidSubarrays(int* nums, int numsSize) {
    int count = 0;
    int i = 0;
    while (i < numsSize) {
        int j = i;
        while (j < numsSize - 1 && gcd(nums[i], nums[j + 1]) == 1) {
            j++;
        }
        if (j == i) {
            return -1;
        }
        count++;
        i = j + 1;
    }
    return count;
}
int main() {
```

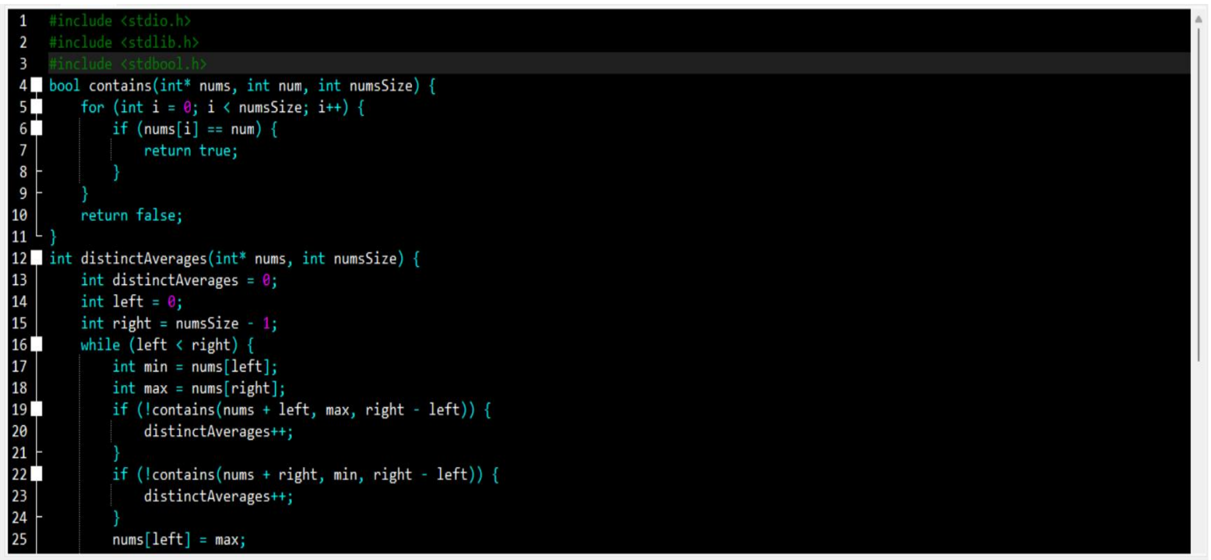


```c
int main() {
    int nums[] = {2, 3, 4, 5, 6};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    int minSubarrays = minValidSubarrays(nums, numsSize);
    printf("Minimum number of subarrays: %d\n", minSubarrays);
    return 0;
}
```

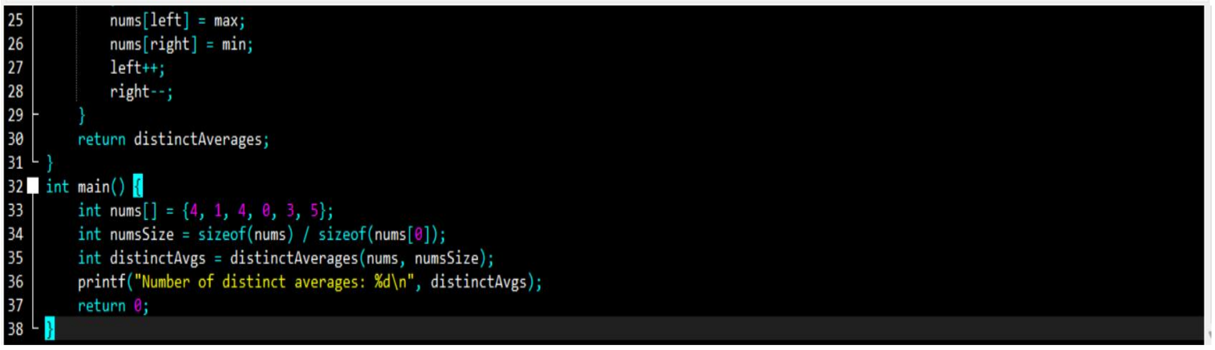


```
Minimum number of subarrays: -1

--------------------------------
Process exited after 1.813 seconds with return value 0
Press any key to continue . . .
```

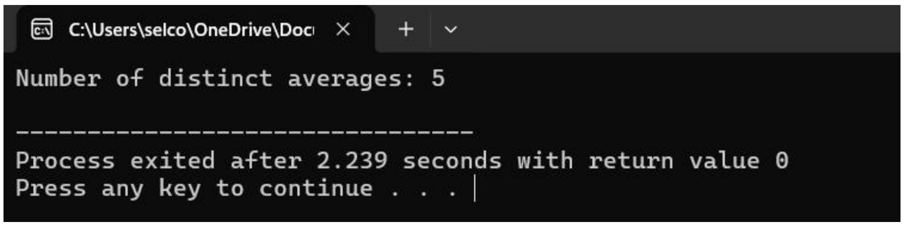
8. Number of Distinct Averages You are given a 0-indexed integer array nums of even length. As long as nums is not empty, you must repetitively: ● Find the minimum number in nums and remove it. ● Find the maximum number in nums and remove it. ● Calculate the average of the two removed numbers. The average of two numbers a and b is (a + b) / 2. ● For example, the average of 2 and 3 is (2 + 3) / 2 = 2.5. Return the number of distinct averages calculated using the above process. Note that when there is a tie for a minimum or maximum number, any can be removed. Example 1: Input: nums = [4,1,4,0,3,5] Output: 2 Explanation: 1. Remove 0 and 5, and the average is (0 + 5) / 2 = 2.5. Now, nums = [4,1,4,3]. 2. Remove 1 and 4. The average is (1 + 4) / 2 = 2.5, and nums = [4,3]. 3. Remove 3 and 4, and the average is (3 + 4) / 2 = 3.5. Since there are 2 distinct numbers among 2.5, 2.5, and 3.5, we return 2.



```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
bool contains(int* nums, int num, int numsSize) {
    for (int i = 0; i < numsSize; i++) {
        if (nums[i] == num) {
            return true;
        }
    }
    return false;
}
int distinctAverages(int* nums, int numsSize) {
    int distinctAverages = 0;
    int left = 0;
    int right = numsSize - 1;
    while (left < right) {
        int min = nums[left];
        int max = nums[right];
        if (!contains(nums + left, max, right - left)) {
            distinctAverages++;
        }
        if (!contains(nums + right, min, right - left)) {
            distinctAverages++;
        }
        nums[left] = max;
```



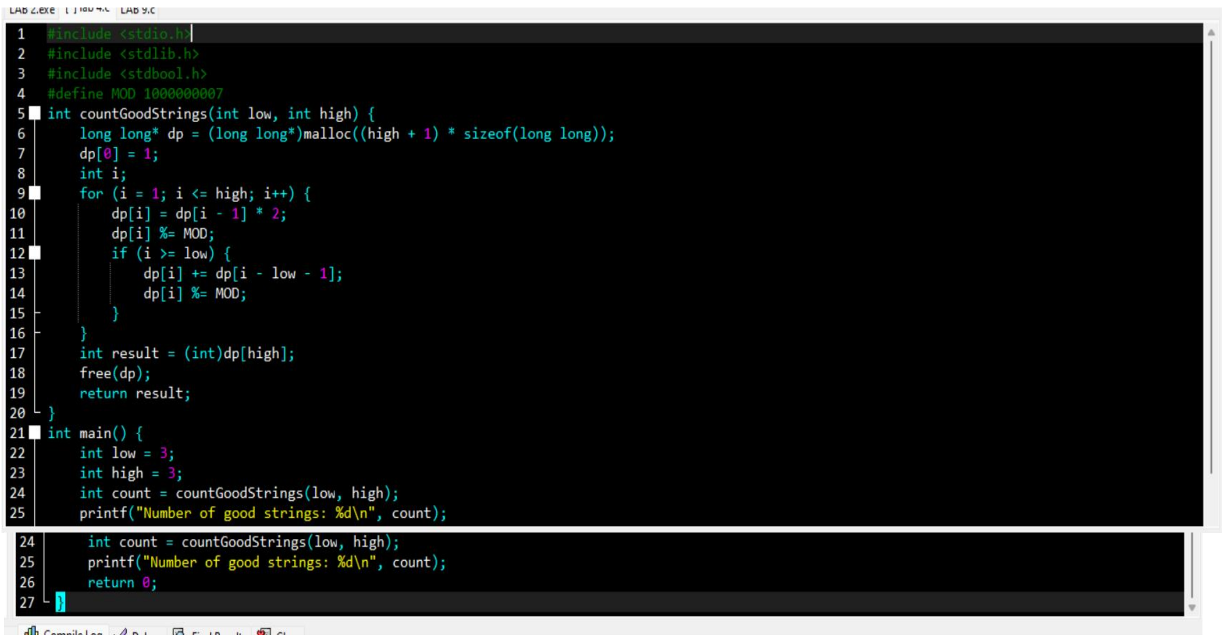
```c
        nums[left] = max;
        nums[right] = min;
        left++;
        right--;
    }
    return distinctAverages;
}
int main() {
    int nums[] = {4, 1, 4, 0, 3, 5};
    int numsSize = sizeof(nums) / sizeof(nums[0]);
    int distinctAvgs = distinctAverages(nums, numsSize);
    printf("Number of distinct averages: %d\n", distinctAvgs);
    return 0;
}
```



```
Number of distinct averages: 5

--------------------------------
Process exited after 2.239 seconds with return value 0
Press any key to continue . . .
```

9. Count Ways To Build Good Strings Given the integers zero, one, low, and high, we can construct a string by starting withanempty string, and then at each step perform either of the following: ● Append the character '0' zero times. ● Append the character '1' one times. This can be performed any number of times.A good string is a string constructed by the above process having a length between low and high (inclusive). Return the number of dif erent good strings that can be constructed satisfying these properties. Since the answer can be large, return it modulo 109 + 7.

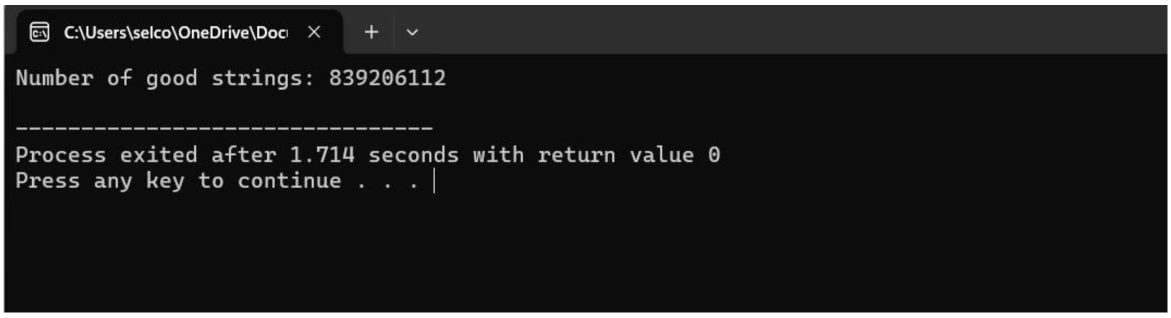
Example 1: Input: low = 3, high = 3, zero = 1, one = 1 Output: 8 Explanation: One possible valid good string is "011". It can be constructed as follows: "" -> "0" -> "01" -> "011". All binary strings from "000" to "111" are good strings in this example



```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MOD 1000000007
int countGoodStrings(int low, int high) {
    long long* dp = (long long*)malloc((high + 1) * sizeof(long long));
    dp[0] = 1;
    int i;
    for (i = 1; i <= high; i++) {
        dp[i] = dp[i - 1] * 2;
        dp[i] %= MOD;
        if (i >= low) {
            dp[i] += dp[i - low - 1];
            dp[i] %= MOD;
        }
    }
    int result = (int)dp[high];
    free(dp);
    return result;
}
int main() {
    int low = 3;
    int high = 3;
    int count = countGoodStrings(low, high);
    printf("Number of good strings: %d\n", count);
```

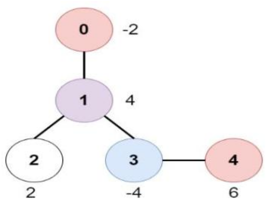
```c
    int count = countGoodStrings(low, high);
    printf("Number of good strings: %d\n", count);
```

```c
    return 0;
}
```

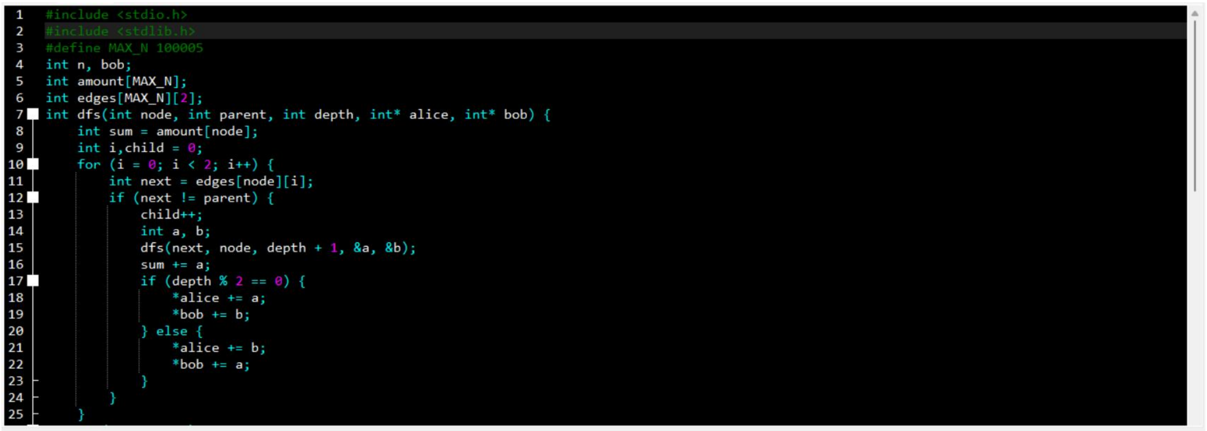


```
Number of good strings: 839206112

--------------------------------
Process exited after 1.714 seconds with return value 0
Press any key to continue . . .
```

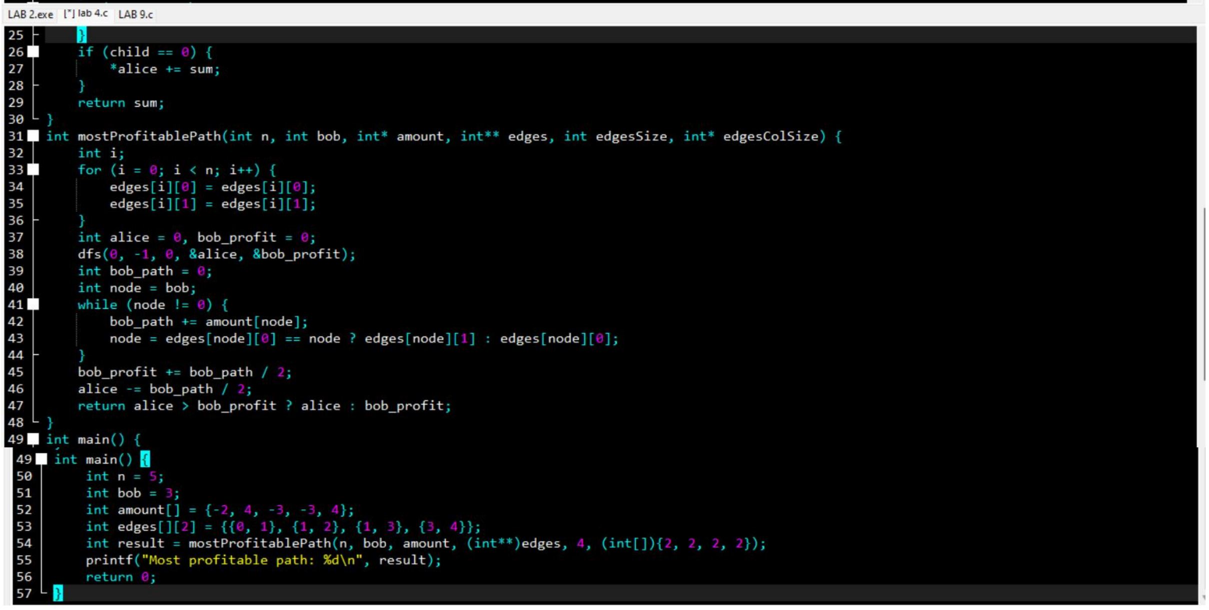
10. Most Profitable Path in a Tree There is an undirected tree with n nodes labeled from 0 to n - 1, rooted at node 0. You are given a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree. At every node i, there is a gate. 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. The game goes on as follows: ● Initially, Alice is at node 0 and Bob is at node bob. ● At every second, Alice and Bob each move to an adjacent node. Alice moves towards some leaf node, while Bob moves towards node 0. ● For every node along their path, Alice and Bob either spend money to open the gate at that node, or accept the reward. Note that: ○ If the gate is already open, no price will be required, nor will there be any cash reward. ○ If Alice and Bob reach the node simultaneously, they share the price/reward for opening the gate there. In other words, if the price to open the gate is c, then both Alice and Bob pay c / 2 each. Similarly, if the reward at the gateisc, both of them receive c / 2 each. ● If Alice reaches a leaf node, she stops moving. Similarly, if Bob reaches node 0, he stops moving. Note that these events are independent of each other. Return the maximum net income Alice can have if she travels towards the optimal leaf node.



Example 1: Input: edges = [[0,1],[1,2],[1,3],[3,4]], bob = 3, amount = [-2,4,2,-4,6] Output: 6 Explanation: The above diagram represents the given tree. The game goes as follows: - Alice is initially on node 0, Bob on node 3. They open the gates of their respective nodes. Alice's net income is now -2. - Both Alice and Bob move to node 1. Since they reach here simultaneously, they open the gate together and share the reward. Alice's net income becomes -2 + (4 / 2) = 0. - Alice moves on to node 3. Since Bob already opened its gate, Alice's income remains unchanged. Bob moves on to node 0, and stops moving. - Alice moves on to node 4 and opens the gate there. Her net income becomes 0 + 6 =6. Now, neither Alice nor Bob can make any further moves, and the game ends. It is not possible for Alice to get a higher net income.



```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_N 100005
int n, bob;
int amount[MAX_N];
int edges[MAX_N][2];
int dfs(int node, int parent, int depth, int* alice, int* bob) {
    int sum = amount[node];
    int i,child = 0;
    for (i = 0; i < 2; i++) {
        int next = edges[node][i];
        if (next != parent) {
            child++;
            int a, b;
            dfs(next, node, depth + 1, &a, &b);
            sum += a;
            if (depth % 2 == 0) {
                *alice += a;
                *bob += b;
            } else {
                *alice += b;
                *bob += a;
            }
        }
    }
```



```c
    }
    if (child == 0) {
        *alice += sum;
    }
    return sum;
}
int mostProfitablePath(int n, int bob, int* amount, int** edges, int edgesSize, int* edgesColSize) {
    int i;
    for (i = 0; i < n; i++) {
        edges[i][0] = edges[i][0];
        edges[i][1] = edges[i][1];
    }
    int alice = 0, bob_profit = 0;
    dfs(0, -1, 0, &alice, &bob_profit);
    int bob_path = 0;
    int node = bob;
    while (node != 0) {
        bob_path += amount[node];
        node = edges[node][0] == node ? edges[node][1] : edges[node][0];
    }
    bob_profit += bob_path / 2;
    alice -= bob_path / 2;
    return alice > bob_profit ? alice : bob_profit;
}
int main() {
int main() {
    int n = 5;
    int bob = 3;
    int amount[] = {-2, 4, -3, -3, 4};
    int edges[][2] = {{0, 1}, {1, 2}, {1, 3}, {3, 4}};
    int result = mostProfitablePath(n, bob, amount, (int**)edges, 4, (int[]){2, 2, 2, 2});
    printf("Most profitable path: %d\n", result);
    return 0;
}
```