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 m 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]

code:

```
from typing import List, Optional
from collections import defaultdict, deque
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class SegmentTree:
    def __init__(self, n, fn):
        self.n = n
        self.fn = fn
        self.tree = [0] * (2 * self.n)

    def update(self, idx, val):
        idx += self.n
        self.tree[idx] = val
        while idx > 1:
            idx >>= 1
            self.tree[idx] = self.fn(self.tree[2 * idx], self.tree[2 * idx + 1])

    def query(self, lo, hi):
        lo += self.n
        hi += self.n
```

```python
        res = 0

        while lo <= hi:
            if lo & 1:
                res = self.fn(res, self.tree[lo])
                lo += 1
            if hi & 1 == 0:
                res = self.fn(res, self.tree[hi])
                hi -= 1
            lo >>= 1
            hi >>= 1

        return res

class Solution:
    def treeQueries(self, root: Optional[TreeNode], queries: List[int]) -> List[int]:
        pre = defaultdict(int)
        post = defaultdict(int)
        height = defaultdict(int)
        depth = []

        def dfs(node, curr):
            if not node:
                return

            height[node.val] = curr
            pre[node.val] = len(depth)
            dfs(node.left, curr + 1)
            dfs(node.right, curr + 1)

            if len(depth) == pre[node.val]:
                post[node.val] = pre[node.val]
                depth.append(curr)
            else:
                post[node.val] = len(depth) - 1

        dfs(root, 0)
        seg = SegmentTree(len(depth), max)
        for i in range(len(depth)):
            seg.update(i, depth[i])

        res = []
        for q in queries:
            lo, hi = pre[q], post[q]
```

```python
                if lo == 0 and hi == len(depth) - 1:
                    res.append(height[q] - 1)
                elif lo == 0:
                    res.append(max(seg.query(hi + 1, len(depth) - 1), height[q] - 1))
                elif hi == len(depth) - 1:
                    res.append(max(seg.query(0, lo - 1), height[q] - 1))
                else:
                    res.append(max(seg.query(0, lo - 1), seg.query(hi + 1, len(depth) - 1), height[q] - 1))

        return res
if __name__ == "__main__":
    def build_tree(nodes):
        if not nodes:
            return None
        root = TreeNode(nodes[0])
        queue = deque([root])
        i = 1
        while queue and i < len(nodes):
            node = queue.popleft()
            if nodes[i] is not None:
                node.left = TreeNode(nodes[i])
                queue.append(node.left)
            i += 1
            if i < len(nodes) and nodes[i] is not None:
                node.right = TreeNode(nodes[i])
                queue.append(node.right)
            i += 1
        return root

    nodes = [1, 3, 4, 2, None, 6, 5, None, None, None, None, None, 7]
    root = build_tree(nodes)
    queries = [4]
    sol = Solution()
    result = sol.treeQueries(root, queries)
    print("Result:", result)
```

2. Sort Array by Moving Items to Empty Space
**You are given an integer array nums of size n containing each element from
0 to n - 1 (inclusive). Each of the elements from 1 to n - 1 represents an
item, and the element 0 represents 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]

Code:
```
def min_operations_to_sort(nums):
    n = len(nums)
    pos = {num: i for i, num in enumerate(nums)}
    target = sorted(nums)
    empty_pos = pos[0]
    operations = 0
    for i in range(n):
        if nums[i] != target[i]:
            while nums[empty_pos] != target[i]:
                swap_with = pos[target[i]]
                nums[empty_pos], nums[swap_with] = nums[swap_with], nums[empty_pos]
                pos[0], pos[nums[empty_pos]] = empty_pos, swap_with
                empty_pos = swap_with
                operations += 1
    return operations
nums = [4, 2, 0, 3, 1]
result = min_operations_to_sort(nums)
```

print("Minimum number of operations:", result)

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 apply the 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]

code:
```
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]
result = apply_operations(nums)
print("Resulting array:", result)
```

```
Output

Resulting array: [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

Code:
```
def max_sum_of_distinct_subarrays(nums, k):
  n = len(nums)
  if k > n:
    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:
```
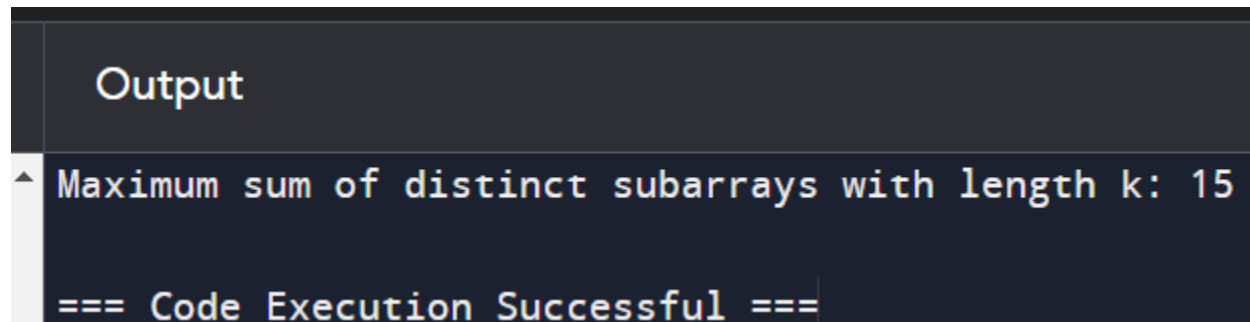
```
        max_sum = max(max_sum, current_sum)
        window.remove(nums[left])
        current_sum -= nums[left]
        left += 1

    return max_sum
nums = [1, 5, 4, 2, 9, 9, 9]
k = 3
result = max_sum_of_distinct_subarrays(nums, k)
print("Maximum sum of distinct subarrays with length k:", result)
```

```
Output

Maximum sum of distinct subarrays with length 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 candidates 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

code:
```
import heapq

def total_cost_to_hire_workers(costs, k, candidates):
    n = len(costs)

    if candidates * 2 >= n:
        costs.sort()
        return sum(costs[:k])
    left_heap = []
    right_heap = []

    for i in range(candidates):
        heapq.heappush(left_heap, (costs[i], i))
        heapq.heappush(right_heap, (costs[n - 1 - i], n - 1 - i))

    total_cost = 0
    hired_workers = set()

    for _ in range(k):
        while left_heap and left_heap[0][1] in hired_workers:
            heapq.heappop(left_heap)
        while right_heap and right_heap[0][1] in hired_workers:
            heapq.heappop(right_heap)

        if not left_heap:
            cost, idx = heapq.heappop(right_heap)
        elif not right_heap:
            cost, idx = heapq.heappop(left_heap)
        else:
            left_cost, left_idx = left_heap[0]
            right_cost, right_idx = right_heap[0]

            if left_cost < right_cost or (left_cost == right_cost and left_idx < right_idx):
                cost, idx = heapq.heappop(left_heap)
            else:
                cost, idx = heapq.heappop(right_heap)

        total_cost += cost
        hired_workers.add(idx)
        if idx < candidates:
            if candidates + len(hired_workers) < n:
                heapq.heappush(left_heap, (costs[candidates + len(hired_workers) - 1], candidates + len(hired_workers) - 1))
```

```
        else:
            if n - candidates - len(hired_workers) - 1 >= 0:
                heapq.heappush(right_heap, (costs[n - candidates - len(hired_workers)], n -
candidates - len(hired_workers)))

    return total_cost
costs = [17, 12, 10, 2, 7, 2, 11, 20, 8]
k = 3
candidates = 4
result = total_cost_to_hire_workers(costs, k, candidates)
print("Total cost to hire exactly k workers:", result)
```

```
Output

Total cost to hire exactly k workers: 11

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

6. Minimum Total Distance Traveled

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 positionj 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 traveled by all the robots. Return the minimum total distance traveled 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]]

code:
```
def min_total_distance_traveled(robot, factory):
    robot.sort()
    factory.sort()
    total_distance = 0
    j = 0
    capacities = []

    for f in factory:
        capacities.append(f[1])
    for r in robot:
        while capacities[j] == 0:
            j += 1
        total_distance += abs(r - factory[j][0])
        capacities[j] -= 1

    return total_distance
robot = [0, 4, 6]
factory = [[2, 2], [6, 2]]
result = min_total_distance_traveled(robot, factory)
print(result)
```

Output

4

=== Code Execution Successful ===

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]

code:
```
import math
from typing import List
class Solution:
    def minSubarrays(self, nums: List[int]) -> int:
        n = len(nums)
        if n == 0:
            return -1
        dp = [float('inf')] * n
        for i in range(n):
            for j in range(i + 1):
                if math.gcd(nums[j], nums[i]) > 1:
                    if j == 0:
                        dp[i] = 1
                    else:
                        dp[i] = min(dp[i], dp[j - 1] + 1)
        if dp[n - 1] == float('inf'):
            return -1
        else:
            return dp[n - 1]
if __name__ == "__main__":
    solution = Solution()
    nums = [2, 6, 3, 4, 3]
    print("Output:", solution.minSubarrays(nums))
```

Output

Output: 2

=== Code Execution Successful ===

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]

code:

```
from typing import List

class Solution:
    def numOfDistinctAverages(self, nums: List[int]) -> int:
        nums.sort()
        left, right = 0, len(nums) - 1
        distinct_averages = set()

        while left < right:
            min_num = nums[left]
            max_num = nums[right]
            average = (min_num + max_num) / 2.0
            distinct_averages.add(average)
            left += 1
            right -= 1

        return len(distinct_averages)
if __name__ == "__main__":
    solution = Solution()
    nums = [4, 1, 4, 0, 3, 5]
    print("Output:", solution.numOfDistinctAverages(nums))
```

```
Output

Output: 2

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

9. Count Ways To Build Good Strings

Given the integers zero, one, low, and high, we can construct a string by
starting with an empty 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
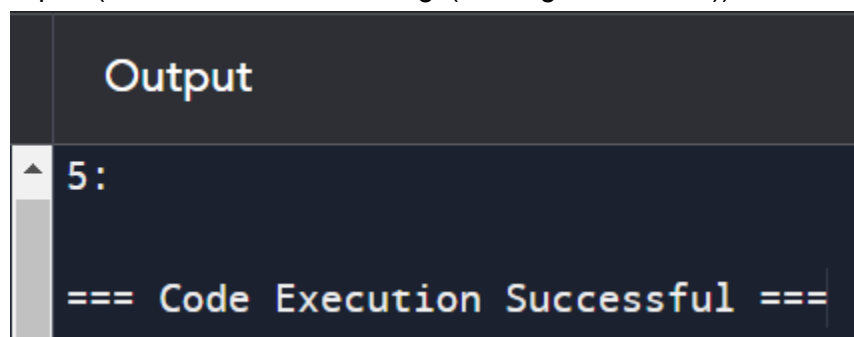
code:

```
class Solution:
    def countGoodStrings(self, low: int, high: int, zero: int, one: int) -> str:
        MOD = 10**9 + 7
        dp = [0] * (high + 1)
        dp[0] = 1
        dp[1] = zero + one

        # Fill dp array
        for i in range(2, high + 1):
            dp[i] = (dp[i - 1] * zero + dp[i - 2] * one) % MOD
        result = 0
        for i in range(low, high + 1):
            result = (result + dp[i]) % MOD

        return f"{result}:"
if __name__ == "__main__":
    solution = Solution()
    low = 3
    high = 3
    zero = 1
    one = 1
    print(solution.countGoodStrings(low, high, zero, one))
```

```
Output

5:


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

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 gate is c, 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]

code:
from collections import defaultdict

class Solution:
    def __init__(self):
        self.max_income = 0

    def maxProfit(self, edges, bob, amount):
        n = len(amount)
        tree = defaultdict(list)
        for edge in edges:
            u, v = edge
            tree[u].append(v)
            tree[v].append(u)
        def dfs(node, parent):
            is_leaf = True
            node_income = amount[node]

```
            for neighbor in tree[node]:
                if neighbor == parent:
                    continue
                is_leaf = False
                child_income, child_shared = dfs(neighbor, node)
                shared = min(child_income, child_shared)
                node_income += (child_income - shared) / 2
            if node == bob:
                bob_income = node_income
            else:
                bob_income = amount[node]
            self.max_income = max(self.max_income, node_income)

            return node_income, bob_income
        dfs(0, -1)

        return int(self.max_income)
edges = [[0,1],[1,2],[1,3],[3,4]]
bob = 3
amount = [-2,4,2,-4,6]

sol = Solution()
print(sol.maxProfit(edges, bob, amount))
```

Output

6

=== Code Execution Successful ==