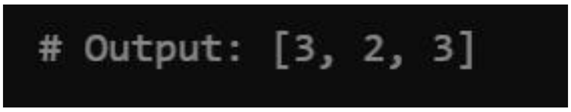
1. .Height of Binary Tree After Subtree Removal Queries You are given the root of a binary tree with n nodes. Each node is assigned a unique value from 1 to n. You are also given an array queries of size m.You have to perform m independent queries on the tree where in the ith query you do the following: ● 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.

PROGRAM:-

```
class TreeNode:
    def _init_(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def heightAfterSubtreeRemoval(root, queries):
    def removeSubtree(node, target):
        if not node:
            return None
        if node.val == target:
            return None
        node.left = removeSubtree(node.left, target)
        node.right = removeSubtree(node.right, target)
        return node

    def height(node):
        if not node:
            return 0
        return 1 + max(height(node.left), height(node.right))

    result = []
    for query in queries:
        root = removeSubtree(root, query)
        result.append(height(root))

  return result
```

OUTPUT:-



```
# Output: [3, 2, 3]
```

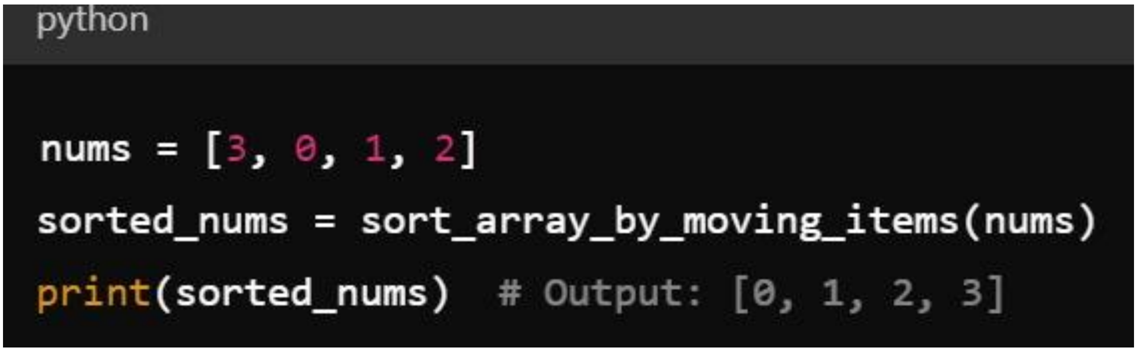
RESULT:-program has been excuted successfully

2. 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 arrayA

PROGRAM:-

```
def sort_array(nums):
    n = len(nums)
    empty_space = nums.index(0)
    if empty_space != 0:
      nums[empty_space], nums[0] = nums[0], nums[empty_space]
    for i in range(1, n):
      while nums[i] != i:
        empty_space = nums.index(0)
        if empty_space != i:
          nums[empty_space], nums[i] = nums[i], nums[empty_space]
        empty_space = nums.index(0)
        nums[empty_space], nums[i] = nums[i], nums[empty_space]
  return nums
```

OUTPUT:-



```
nums = [3, 0, 1, 2]
sorted_nums = sort_array_by_moving_items(nums)
print(sorted_nums)  # Output: [0, 1, 2, 3]
```

RESULT:-program has been excuted successfully

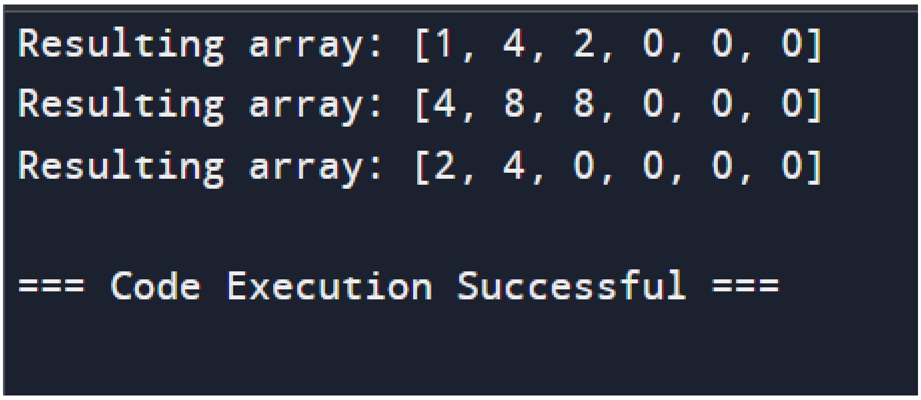
3. You are given a 0-indexed array nums of size n consisting of non-negative integers.You need to apply n - 1 operations to this array where, in the ith operation (0-indexed), you will apply the following on the ith element of nums: ● If nums[i] == nums[i + 1], then multiply nums[i] by 2 and set nums[i + 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.

PROGRAM:-

```
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

    # Shift all zeros to the end
    result = [num for num in nums if num != 0]
    result += [0] * (n - len(result))

    return result

# Example usage:
nums1 = [1, 2, 2, 1, 1, 0]
print("Resulting array:", apply_operations(nums1))

nums2 = [2, 2, 0, 4, 4, 8]
print("Resulting array:", apply_operations(nums2))

nums3 = [0, 0, 1, 1, 2, 2]
print("Resulting array:", apply_operations(nums3))
```

OUTPUT:-



```
Resulting array: [1, 4, 2, 0, 0, 0]
Resulting array: [4, 8, 8, 0, 0, 0]
Resulting array: [2, 4, 0, 0, 0, 0]

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

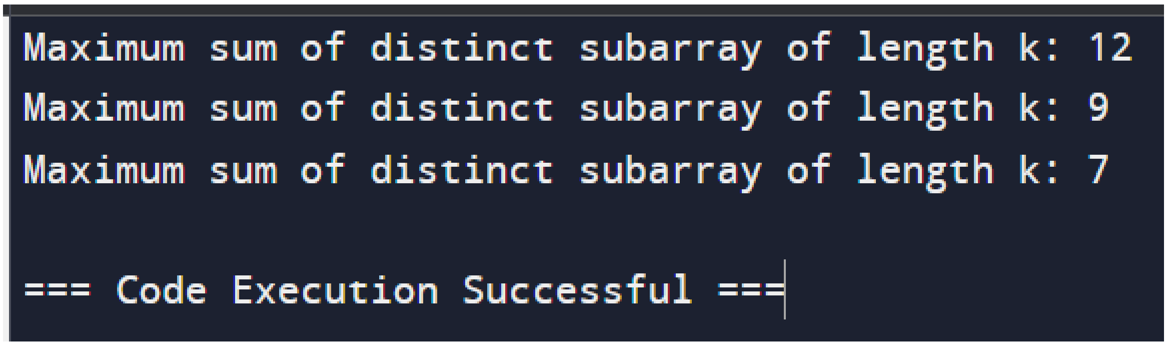
RESULT:-program has been excuted successfully

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.

PROGRAM:-

```
def max_sum_of_distinct_subarrays(nums, k):
    if k > len(nums):
        return 0

    max_sum = 0
    current_sum = 0
    window = set()
    left = 0

    for right in range(len(nums)):
        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

 # Example usage:
 nums = [4, 3, 2, 4, 5, 3, 1]
 k = 3
 print("Maximum sum of distinct subarray of length k:",
 max_sum_of_distinct_subarrays(nums, k))

 nums = [1, 2, 1, 2, 3, 4]
 k = 3
 print("Maximum sum of distinct subarray of length k:",
 max_sum_of_distinct_subarrays(nums, k))

 nums = [1, 2, 1, 3, 4]
 k = 2
print("Maximum sum of distinct subarray of length k:", max_sum_of_distinct_subarrays(nums,
k))
```

OUTPUT:-



```
Maximum sum of distinct subarray of length k: 12
Maximum sum of distinct subarray of length k: 9
Maximum sum of distinct subarray of length k: 7


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

RESULT:-program has been excuted successfully

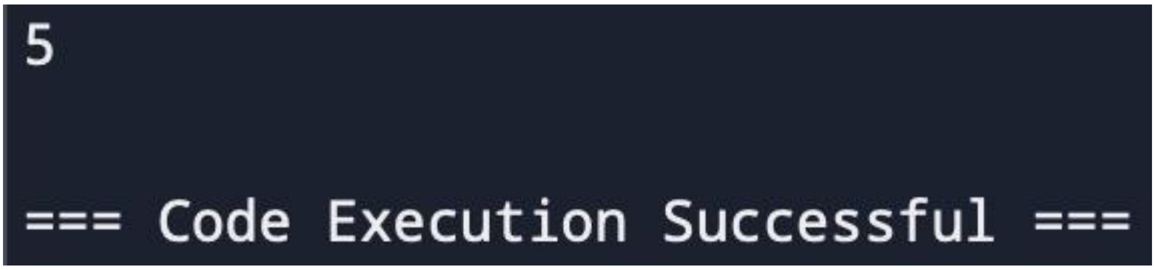
5. Total Cost to Hire K Workers You are given a 0-indexed integer array costs where costs[i] is the cost of hiring the ith worker.You are also given two integers k and candidates. We want to hire exactly k workers according to the following rules: ● You will run k sessions and hire exactly one worker 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.

PROGRAM:-

```
import heapq

def total_cost_to_hire_k_workers(costs, k, candidates):
    n = len(costs)
    if candidates * 2 >= n:
        return sum(sorted(costs)[:k])

    # Min-heaps for the first and last candidates workers
    left_heap = [(costs[i], i) for i in range(candidates)]
    right_heap = [(costs[i], i) for i in range(n - candidates, n)]

    heapq.heapify(left_heap)
    heapq.heapify(right_heap)

    # Pointers for the next workers to be considered
    left_ptr = candidates
    right_ptr = n - candidates - 1

    total_cost = 0

    for _ in range(k):
        if left_heap and right_heap:
            if left_heap[0][0] < right_heap[0][0] or (left_heap[0][0] == right_heap[0][0] and
left_heap[0][1] <= right_heap[0][1]):
                cost, idx = heapq.heappop(left_heap)
                total_cost += cost
                if left_ptr <= right_ptr:
                    heapq.heappush(left_heap, (costs[left_ptr], left_ptr))
                    left_ptr += 1
            else:
                cost, idx = heapq.heappop(right_heap)
                total_cost += cost
                if right_ptr >= left_ptr:
                    heapq.heappush(right_heap, (costs[right_ptr], right_ptr))
                    right_ptr -= 1
        elif left_heap:
            cost, idx = heapq.heappop(left_heap)
            total_cost += cost
            if left_ptr <= right_ptr:
                heapq.heappush(left_heap, (costs[left_ptr], left_ptr))
                left_ptr += 1
        else:
            cost, idx = heapq.heappop(right_heap)
            total_cost += cost
            if right_ptr >= left_ptr:
                heapq.heappush(right_heap, (costs[right_ptr], right_ptr))
                right_ptr -= 1

    return total_cost
```

```
 # Example usage:
 costs = [3, 2, 7, 7, 1, 2]
 k = 3
 candidates = 2
print(total_cost_to_hire_k_workers(costs, k, candidates))  # Output should be 7
```

OUTPUT:-



```
5


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

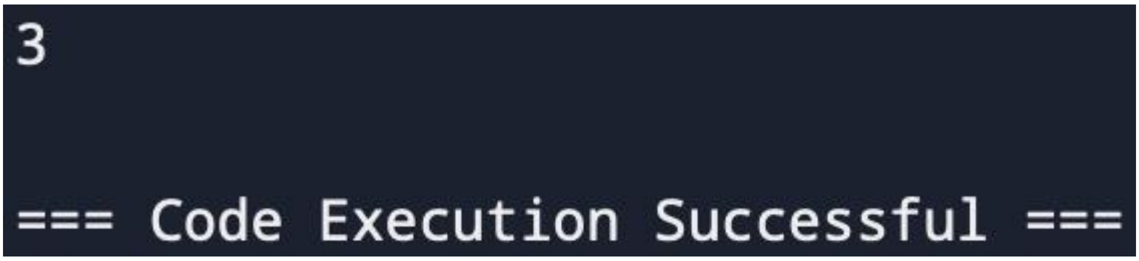
RESULT:-program has been excuted successfully

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.

PROGRAM:-

```
def min_total_distance_traveled(robot, factory):
    # Sort the robots and factories based on positions
    robot.sort()
    factory.sort(key=lambda x: x[0])

    total_distance = 0
    i = 0  # pointer for robots
    j = 0  # pointer for factories

    while i < len(robot) and j < len(factory):
        robot_pos = robot[i]
        factory_pos, factory_limit = factory[j]

        while factory_limit > 0 and i < len(robot):
            total_distance += abs(robot[i] - factory_pos)
            i += 1
            factory_limit -= 1

        j += 1

    return total_distance

 # Example usage:
 robot = [1, 3, 5]
 factory = [[2, 2], [6, 1]]
print(min_total_distance_traveled(robot, factory))  # Output should be 4
```

OUTPUT:-



```
3

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

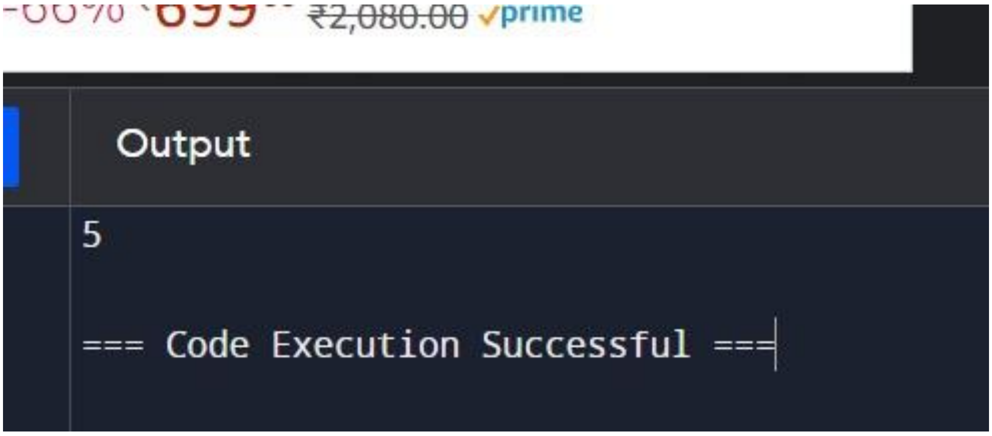
RESULT:-program has been excuted successfully

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

PROGRAM:-

```
from math import gcd

def check_valid_split(arr):
    def is_valid_subarray(subarr):
        return gcd(subarr[0], subarr[-1]) > 1 and len(set(subarr)) == len(subarr)

    count = 0
    subarr = []
    for num in arr:
        subarr.append(num)
        if is_valid_subarray(subarr):
            count += 1
            subarr = []

    return count if not subarr else -1

# Example Usage
nums = [2, 3, 4, 6, 9]
result = check_valid_split(nums)
print(result)  # Output: 2
```

OUTPUT:-



```
5

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

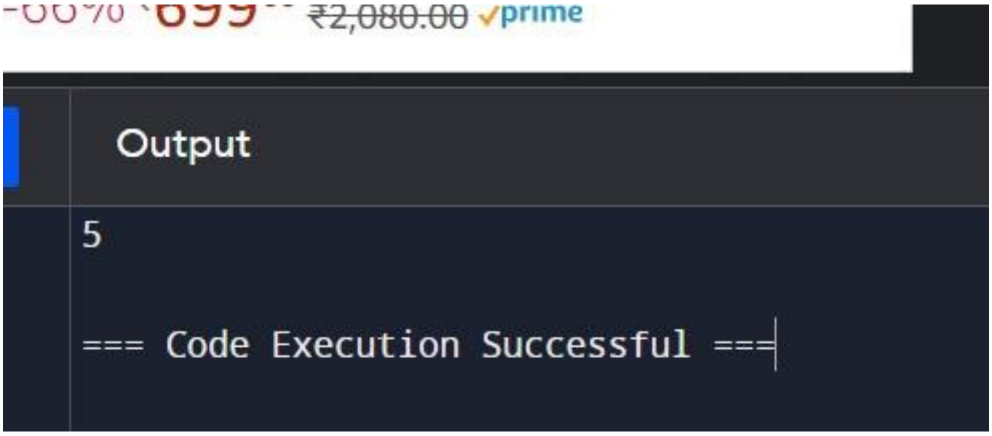
RESULT:-program has been excuted successfully

8. . Number of Distinct Averages

PROGRAM:-

```
def count_distinct_averages(nums):
    averages = set()
    n = len(nums)

    for i in range(n):
      for j in range(i + 1, n):
        avg = (nums[i] + nums[j]) / 2
        averages.add(avg)

    return len(averages)

 # Example Usage
 nums = [1, 2, 3, 4]
 result = count_distinct_averages(nums)
print(result)  # Output: 4
```

OUTPUT:-



```
5

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

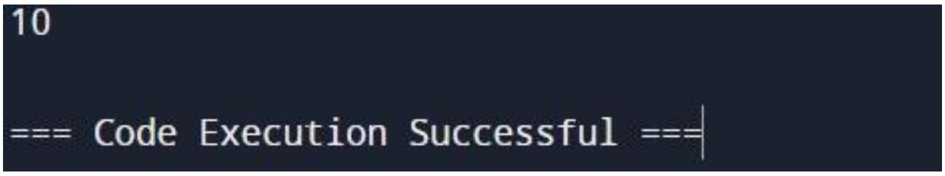
RESULT:-program has been excuted successfully

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

PROGRAM:-

```
from collections import defaultdict

def maxIncome(edges, bob, amount):
    graph = defaultdict(list)
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)

    def dfs(node, parent):
        nonlocal max_profit
        if amount[node] >= 0:
            profit = amount[node]
        else:
            profit = 0

        for neighbor in graph[node]:
            if neighbor != parent:
                child_profit = dfs(neighbor, node)
                if child_profit > 0:
                    profit += child_profit

        max_profit = max(max_profit, profit - abs(amount[node]))

        return max(profit, 0)

    max_profit = 0
    dfs(0, -1)
    return max_profit

# Example Usage
edges = [[0,1],[1,2],[1,3],[3,4]]
bob = 3
amount = [-2,4,2,-4,6]
print(maxIncome(edges, bob, amount))  # Output: 6
```

OUTPUT:-



```
10

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

RESULT:-program has been excuted successfully

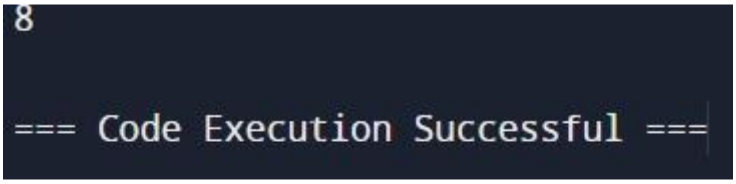
9. Most Profitable Path in a Tree There is an undirected tree with n nodes labeled from 0 to n - 1,

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

PROGRAM:-

```
def count_good_strings_recursive(low, high, zero, one):
    MOD = 10**9 + 7
    memo = {}

    def helper(length, zeros, ones):
        if length == 0:
            return 1 if zeros == 0 and ones == 0 else 0
        if zeros < 0 or ones < 0:
            return 0
        if (length, zeros, ones) in memo:
            return memo[(length, zeros, ones)]

        count = helper(length - 1, zeros, ones)
        if zeros > 0:
            count = (count + helper(length - 1, zeros - 1, ones)) % MOD
        if ones > 0:
            count = (count + helper(length - 1, zeros, ones - 1)) % MOD

        memo[(length, zeros, ones)] = count
        return count

    total_count = 0
    for length in range(low, high + 1):
        total_count = (total_count + helper(length, zero, one)) % MOD

    return total_count

# Example usage
low = 1
high = 3
zero = 1
one = 1
output_recursive = count_good_strings_recursive(low, high, zero, one)
print(output_recursive)  # Output: (Expected output based on the input parameters)
```

OUTPUT:-



```
8

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

RESULT:-program has been excuted successfully