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 valuefrom 1 to n. You are also given an array queries of size m.You have to performmindependent 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] fromthe 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 performingthe 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 fromthe root tosome 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). Example 2: Input: root = [5,8,9,2,1,3,7,4,6], queries = [3,2,4,8] Output: [3,2,3,2] Explanation: We have the following queries: - Removing the subtree rooted at node with value 3. The height of the tree becomes 3(Thepath 5 -> 8 -> 2 -> 4). - Removing the subtree rooted at node with value 2. The height of the tree becomes 2(Thepath 5 -> 8 -> 1). - Removing the subtree rooted at node with value 4. The height of the tree becomes 3(Thepath 5 -> 8 -> 2 -> 6). - Removing the subtree rooted at node with value 8. The height of the tree becomes 2(Thepath 5 -> 9 -> 3). Constraints: ● The number of nodes in the tree is n. ● 2 <= n <= 105 ● 1 <= Node.val <= n ● All the values in the tree are unique. ● m == queries.length ● 1 <= m <= min(n, 104) ● 1 <= queries[i] <= n ● queries[i] != root.val

CODE:

class Solution:

def treeQueries(self, root: Optional[TreeNode], queries: List[int]) -> List[int]:

@lru\_cache(None)

def height(root: Optional[TreeNode]) -> int:

if not root:

return 0

return 1 + max(height(root.left), height(root.right))

# valToMaxHeight[val] := the maximum height without the node with `val`

valToMaxHeight = {}

# maxHeight := the maximum height without the current node `root`

def dfs(root: Optional[TreeNode], depth: int, maxHeight: int) -> None:

if not root:

return

valToMaxHeight[root.val] = maxHeight

dfs(root.left, depth + 1, max(maxHeight, depth + height(root.right)))

dfs(root.right, depth + 1, max(maxHeight, depth + height(root.left)))

dfs(root, 0, 0)

return [valToMaxHeight[query] for query in queries]

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 sortedif the numbers of all the items are in ascending order and the empty space is either at thebeginning 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 neededto 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. Example 2: Input: nums = [1,2,3,4,0] Output: 0 Explanation: nums is already sorted so return 0. Example 3: Input: nums = [1,0,2,4,3] Output: 2 Explanation: - Move item 2 to the empty space. Now, nums = [1,2,0,4,3]. - Move item 3 to the empty space. Now, nums = [1,2,3,4,0]. It can be proven that 2 is the minimum number of operations needed. Constraints: ● n == nums.length ● 2 <= n <= 105 ● 0 <= nums[i] < n ● All the values of nums

CODE:

class Solution:

def sortArray(self, nums: List[int]) -> int:

n = len(nums)

numToIndex = [0] \* n

for i, num in enumerate(nums):

numToIndex[num] = i

def minOps(numToIndex: List[int], zeroInBeginning: bool) -> int:

ops = 0

num = 1

# If zeroInBeginning, the correct index of each num is num.

# If not zeroInBeginning, the correct index of each num is num - 1.

offset = 0 if zeroInBeginning else 1

while num < n:

# 0 is in the correct index, so swap 0 with the first `numInWrongIndex`.

if zeroInBeginning and numToIndex[0] == 0 or \

not zeroInBeginning and numToIndex[0] == n - 1:

while numToIndex[num] == num - offset: # num is in correct position

num += 1

if num == n:

return ops

numInWrongIndex = num

# 0 is in the wrong index. e.g. numToIndex[0] == 2, that means 2 is not

# in nums[2] because nums[2] == 0.

else:

numInWrongIndex = numToIndex[0] + offset

numToIndex[0], numToIndex[numInWrongIndex] = \

numToIndex[numInWrongIndex], numToIndex[0]

ops += 1

return min(minOps(numToIndex.copy(), True),

minOps(numToIndex.copy(), False))

3. Apply Operations to an Array You are given a 0-indexed array nums of size n consisting of non-negative integers.Youneedto 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] to0. 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]. Example 2: Input: nums = [0,1] Output: [1,0] Explanation: No operation can be applied, we just shift the 0 to the end. Constraints: ● 2 <= nums.length <= 2000

CODE:

class Solution:

def applyOperations(self, nums: List[int]) -> List[int]:

ans = [0] \* len(nums)

for i in range(len(nums) - 1):

if nums[i] == nums[i + 1]:

nums[i] \*= 2

nums[i + 1] = 0

i = 0

for num in nums:

if num > 0:

ans[i] = num

i += 1

return ans

4. Maximum Sum of Distinct Subarrays With Length K You are given an integer array nums and an integer k. Find the maximumsubarray sumof 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 nosubarray meets the conditions, return 0. A subarray is a contiguous non-empty sequenceof 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 Example 2: Input: nums = [4,4,4], k = 3 Output: 0 Explanation: The subarrays of nums with length 3 are: - [4,4,4] which does not meet the requirements because the element 4 is repeated. We return 0 because no subarrays meet the conditions. Constraints: ● 1 <= k <= nums.length <= 105 ●

CODE:

class Solution:

def maximumSubarraySum(self, nums: List[int], k: int) -> int:

ans = 0

summ = 0

distinct = 0

count = collections.Counter()

for i, num in enumerate(nums):

summ += num

count[num] += 1

if count[num] == 1:

distinct += 1

if i >= k:

count[nums[i - k]] -= 1

if count[nums[i - k]] == 0:

distinct -= 1

summ -= nums[i - k]

if i >= k - 1 and distinct == k:

ans = max(ans, summ)

return ans

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 ithworker.You are also given two integers k and candidates. We want to hire exactly k workersaccording 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 fromeither 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 hiringsession, 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 havethesame 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 thelowest 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 commoninthe first and last four workers. The total hiring cost is 11. Example 2: Input: costs = [1,2,4,1], k = 3, candidates = 3 Output: 4 Explanation: We hire 3 workers in total. The total cost is initially 0. - In the first hiring round we choose the worker from [1,2,4,1]. The lowest cost is 1, andwebreak the tie by the smallest index, which is 0. The total cost = 0 + 1 = 1. Notice that workerswith index 1 and 2 are common in the first and last 3 workers. - In the second hiring round we choose the worker from [2,4,1]. The lowest cost is 1 (index2). The total cost = 1 + 1 = 2. - In the third hiring round there are less than three candidates. We choose the worker fromtheremaining workers [2,4]. The lowest cost is 2 (index 0). The total cost = 2 + 2 = 4. The total hiring cost is 4. Constraints: ● 1 <= costs.length <= 105 ● 1 <= costs[i] <= 105 ● 1 <= k, candidates <= costs.length

CODE:

class Solution:

def totalCost(self, costs: List[int], k: int, candidates: int) -> int:

ans = 0

i = 0

j = len(costs) - 1

minHeapL = [] # First half

minHeapR = [] # Second half

for \_ in range(k):

while len(minHeapL) < candidates and i <= j:

heapq.heappush(minHeapL, costs[i])

i += 1

while len(minHeapR) < candidates and i <= j:

heapq.heappush(minHeapR, costs[j])

j -= 1

if not minHeapL:

ans += heapq.heappop(minHeapR)

elif not minHeapR:

ans += heapq.heappop(minHeapL)

# Both `minHeapL` and `minHeapR` are not empty.

elif minHeapL[0] <= minHeapR[0]:

ans += heapq.heappop(minHeapL)

else:

ans += heapq.heappop(minHeapR)

return ans

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 factorywhere factory[j] = [positionj, limitj] indicates that positionj is the position of the jth factoryand 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. Notethat 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 couldbethe negative or the positive direction of the X-axis. When a robot reaches a factory that didnot 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 tominimize the total distance traveled by all the robots. Return the minimum total distance traveled by all the robots. The test cases are generatedsuch 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 donot 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 thefirst factory. - The third robot at position 6 will be repaired at the second factory. It does not need tomove. 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 abetter total distance than 4. Example 2: Input: robot = [1,-1], factory = [[-2,1],[2,1]] Output: 2 Explanation: As shown in the figure: - The first robot at position 1 moves in the positive direction. It will be repaired at the secondfactory. - The second robot at position -1 moves in the negative direction. It will be repaired at thefirst factory. The limit of the first factory is 1, and it fixed 1 robot. The limit of the second factory is 1, and it fixed 1 robot. The total distance is |2 - 1| + |(-2) - (-1)| = 2. It can be shown that we cannot achieve a better total distance than 2. Constraints: ● 1 <= robot.length, factory.length <= 100 ● factory[j].length == 2 ● -109 <= robot[i], positionj <= 109 ● 0 <= limitj <= robot.length ● The input will be generated such that it is always possible to repair every robot

CODE:

class Solution:

def minimumTotalDistance(self, robot: List[int], factory: List[List[int]]) -> int:

robot.sort()

factory.sort()

@functools.lru\_cache(None)

def dp(i: int, j: int, k: int) -> int:

"""

Returns the minimum distance to fix robot[i..n) with factory[j..n), where

factory[j] already fixed k robots.

"""

if i == len(robot):

return 0

if j == len(factory):

return math.inf

skipFactory = dp(i, j + 1, 0)

position, limit = factory[j]

useFactory = dp(i + 1, j, k + 1) + abs(robot[i] - position) \

if limit > k else math.inf

return min(skipFactory, useFactory)

return dp(0, 0, 0)

7. Minimum Subarrays in a Valid Split You are given an integer array nums.Splitting of an integer array nums into subarrays is validif: ● 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 validsubarray 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 commondivisor is 2, which is greater than 1. - The starting element of the 2nd subarray is 3 and the ending is 3. Their greatest commondivisor 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 validsplit. Example 2: Input: nums = [3,5] Output: 2 Explanation: We can create a valid split in the following way: [3] | [5]. - The starting element of the 1st subarray is 3 and the ending is 3. Their greatest commondivisor is 3, which is greater than 1. - The starting element of the 2nd subarray is 5 and the ending is 5. Their greatest commondivisor is 5, which is greater than 1. It can be proved that 2 is the minimum number of subarrays that we can obtain in a validsplit. Example 3: Input: nums = [1,2,1] Output: -1 Explanation: It is impossible to create valid split. Constraints: ● 1 <= nums.length <= 1000 ● 1 <= nums[i] <= 105

CODE:

class Solution:

def validSubarraySplit(self, nums: List[int]) -> int:

# dp[i] := the minimum number of subarrays to validly split nums[0..i]

dp = [math.inf] \* len(nums)

for i, num in enumerate(nums):

for j in range(i + 1):

if math.gcd(nums[j], num) > 1:

dp[i] = min(dp[i], 1 if j == 0 else dp[j - 1] + 1)

return -1 if dp[-1] == math.inf else dp[-1]

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 whenthere 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. Example 2: Input: nums = [1,100] Output: 1 Explanation: There is only one average to be calculated after removing 1 and 100, so we return 1. Constraints: ● 2 <= nums.length <= 100 ● nums.length is even. ● 0 <= nums[i] <= 100

CODE:

class Solution:

def distinctAverages(self, nums: List[int]) -> int:

n = len(nums)

sums = set()

nums.sort()

for i in range(n // 2):

sums.add(nums[i] + nums[n - 1 - i])

return len(sums)

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. Example 2: Input: low = 2, high = 3, zero = 1, one = 2 Output: 5 Explanation: The good strings are "00", "11", "000", "110", and "011". Constraints: ● 1 <= low <= high <= 105 ● 1 <= zero, one <= low

CODE:

class Solution:

def countGoodStrings(self, low: int, high: int, zero: int, one: int) -> int:

kMod = 1\_000\_000\_007

ans = 0

# dp[i] := the number of good strings with length i

dp = [1] + [0] \* high

for i in range(1, high + 1):

if i >= zero:

dp[i] = (dp[i] + dp[i - zero]) % kMod

if i >= one:

dp[i] = (dp[i] + dp[i - one]) % kMod

if i >= low:

ans = (ans + dp[i]) % kMod

return ans

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. Youaregiven a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that thereisan 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, whereamount[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 towardssome leaf node, while Bob moves towards node 0. ● For every node along their path, Alice and Bob either spend money to open the gateat that node, or accept the reward. Note that: ○ If the gate is already open, no price will be required, nor will there be anycashreward. ○ If Alice and Bob reach the node simultaneously, they share the price/rewardfor 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, hestops 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. Example 2: Input: edges = [[0,1]], bob = 1, amount = [-7280,2350] Output: -7280 Explanation: Alice follows the path 0->1 whereas Bob follows the path 1->0. Thus, Alice opens the gate at node 0 only. Hence, her net income is -7280. Constraints: ● 2 <= n <= 105 ● edges.length == n - 1 ● edges[i].length == 2 ● 0 <= ai, bi < n ● ai != bi ● edges represents a valid tree. ● 1 <= bob < n ● amount.length == n ● amount[i] is an even integer in the range [-104, 104]

CODE:

class Solution:

def mostProfitablePath(self, edges: List[List[int]], bob: int, amount: List[int]) -> int:

n = len(amount)

tree = [[] for \_ in range(n)]

parent = [0] \* n

aliceDist = [-1] \* n

for u, v in edges:

tree[u].append(v)

tree[v].append(u)

# Fills `parent` and `aliceDist`.

def dfs(u: int, prev: int, d: int) -> None:

parent[u] = prev

aliceDist[u] = d

for v in tree[u]:

if aliceDist[v] == -1:

dfs(v, u, d + 1)

dfs(0, -1, 0)

# Modify amount athe path from node bob to node 0.

# For each node,

# 1. If Bob reaches earlier than Alice does, change the amount to 0.

# 2. If Bob and Alice reach simultaneously, devide the amount by 2.

u = bob

bobDist = 0

while u != 0:

if bobDist < aliceDist[u]:

amount[u] = 0

elif bobDist == aliceDist[u]:

amount[u] //= 2

u = parent[u]

bobDist += 1

return self.\_getMoney(tree, 0, -1, amount)

def \_getMoney(self, tree: List[List[int]], u: int, prev: int, amount: List[int]) -> int:

# a leaf node

if len(tree[u]) == 1 and tree[u][0] == prev:

return amount[u]

maxPath = -math.inf

for v in tree[u]:

if v != prev:

maxPath = max(maxPath, self.\_getMoney(tree, v, u, amount))

return amount[u] + maxPath