LAB PROGRMS-5

08/05/2024

1. Given binary array k and integer k written true if all 1’s are last k places away from each other otherwise return false

CODE:

def kLengthApart(nums, k):

pos=0

count=0

while (pos<len(nums) and nums[pos]==0):

pos+=1

for i in range(pos+1,len(nums)):

if nums[i]==0:

count+=1

else :

if count<k:

return False

count = 0

return True

if \_name\_ == "\_main\_":

nums = [ 1, 0, 0, 0, 1, 0, 1, 0, 0 ]

k=2

print(kLengthApart(nums, k))

2. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

Given an array of integers nums and an integer limit, return the size of the longest non-empty subarray such that the absolute difference between any two elements of this subarray is less than or equal to limit.

Example 1:

Input: nums = [8,2,4,7], limit = 4

Output: 2

Explanation: All subarrays are:

[8] with maximum absolute diff |8-8| = 0 <= 4.

[8,2] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4] with maximum absolute diff |8-2| = 6 > 4.

[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.

[2] with maximum absolute diff |2-2| = 0 <= 4.

[2,4] with maximum absolute diff |2-4| = 2 <= 4.

[2,4,7] with maximum absolute diff |2-7| = 5 > 4.

[4] with maximum absolute diff |4-4| = 0 <= 4.

[4,7] with maximum absolute diff |4-7| = 3 <= 4.

[7] with maximum absolute diff |7-7| = 0 <= 4.

Therefore, the size of the longest subarray is 2.

Example 2:

Input: nums = [10,1,2,4,7,2], limit = 5

Output: 4

Explanation: The subarray [2,4,7,2] is the longest since the maximum absolute diff is |2-7| = 5 <= 5.

Example 3:

Input: nums = [4,2,2,2,4,4,2,2], limit = 0

Output: 3

Constraints:

● 1 <= nums.length <= 105

● 1 <= nums[i] <= 109

● 0 <= limit <= 109

CODE:

from collections import deque

def longest\_subarray(arr, X):

min\_queue, max\_queue = deque(), deque()

n, start, end = len(arr), 0, 0

ans\_start, ans\_end = 0, 0

while end < n:

while min\_queue and arr[min\_queue[-1]] > arr[end]:

min\_queue.pop()

while max\_queue and arr[max\_queue[-1]] < arr[end]:

max\_queue.pop()

min\_queue.append(end)

max\_queue.append(end)

while arr[max\_queue[0]] - arr[min\_queue[0]] > X:

if start == min\_queue[0]:

min\_queue.popleft()

if start == max\_queue[0]:

max\_queue.popleft()

start += 1

if end - start > ans\_end - ans\_start:

ans\_start, ans\_end = start, end

end += 1

for i in range(ans\_start, ans\_end + 1):

print(arr[i], end=" ")

arr = [15, 10, 1, 2, 4, 7, 2]

X = 5

longest\_subarray(arr, X)

3. Find the Kth Smallest Sum of a Matrix With Sorted Rows

You are given an m x n matrix mat that has its rows sorted in non-decreasing order and an integer k.

You are allowed to choose exactly one element from each row to form an array.

Return the kth smallest array sum among all possible arrays.

Example 1:

Input: mat = [[1,3,11],[2,4,6]], k = 5

Output: 7

Explanation: Choosing one element from each row, the first k smallest sum are:

[1,2], [1,4], [3,2], [3,4], [1,6]. Where the 5th sum is 7.

Example 2:

Input: mat = [[1,3,11],[2,4,6]], k = 9

Output: 17

Example 3:

Input: mat = [[1,10,10],[1,4,5],[2,3,6]], k = 7

Output: 9

Explanation: Choosing one element from each row, the first k smallest sum are:

[1,1,2], [1,1,3], [1,4,2], [1,4,3], [1,1,6], [1,5,2], [1,5,3]. Where the 7th sum is 9.

Constraints:

● m == mat.length

● n == mat.length[i]

● 1 <= m, n <= 40

● 1 <= mat[i][j] <= 5000

● 1 <= k <= min(200, nm)

● mat[i] is a non-decreasing array.

CODE:

import heapq

def kthSmallest(arr, n, k):

pq = []

for i in range(n):

for j in range(n):

heapq.heappush(pq, arr[i][j])

c = 0

while pq:

temp = heapq.heappop(pq)

c += 1

if c == k:

return temp

return -1

if \_name\_ == "\_main\_":

mat = [

[10, 20, 30, 40],

[15, 25, 35, 45],

[25, 29, 37, 48],

[32, 33, 39, 50]

]

res = kthSmallest(mat, 4, 8)

print("7th smallest element is", res)

4. import heapq

def kthSmallest(arr, n, k):

pq = []

for i in range(n):

for j in range(n):

heapq.heappush(pq, arr[i][j])

c = 0

while pq:

temp = heapq.heappop(pq)

c += 1

if c == k:

return temp

return -1

if \_name\_ == "\_main\_":

mat = [

[10, 20, 30, 40],

[15, 25, 35, 45],

[25, 29, 37, 48],

[32, 33, 39, 50]

]

res = kthSmallest(mat, 4, 8)

print("7th smallest element is", res)

CODE:

def xor\_triplet(arr, n):

ans = 0

for i in range(n):

for j in range(i + 1, n):

for k in range(j, n):

xor1 = 0

xor2 = 0

for x in range(i, j):

xor1 ^= arr[x]

for x in range(j, k + 1):

xor2 ^= arr[x]

if (xor1 == xor2):

ans += 1

return ans

if \_name\_ == '\_main\_':

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

n = len(arr)

print(xor\_triplet(arr, n))

5. Minimum Time to Collect All Apples in a Tree

Given an undirected tree consisting of n vertices numbered from 0 to n-1, which has some apples in their vertices. You spend 1 second to walk over one edge of the tree. Return the minimum time in seconds you have to spend to collect all apples in the tree, starting at vertex 0 and coming back to this vertex.

The edges of the undirected tree are given in the array edges, where edges[i] = [ai, bi] means that exists an edge connecting the vertices ai and bi. Additionally, there is a boolean array hasApple, where hasApple[i] = true means that vertex i has an apple; otherwise, it does not have any apple.

Example 1:

Input: n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]], hasApple = [false,false,true,false,true,true,false]

Output: 8

Explanation: The figure above represents the given tree where red vertices have an apple. One optimal path to collect all apples is shown by the green arrows.

Example 2:

Input: n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]], hasApple = [false,false,true,false,false,true,false]

Output: 6

Explanation: The figure above represents the given tree where red vertices have an apple. One optimal path to collect all apples is shown by the green arrows.

Example 3:

Input: n = 7, edges = [[0,1],[0,2],[1,4],[1,5],[2,3],[2,6]], hasApple = [false,false,false,false,false,false,false]

Output: 0

Constraints:

● 1 <= n <= 105

● edges.length == n - 1

● edges[i].length == 2

● 0 <= ai < bi <= n - 1

● fromi < toi

● hasApple.length == n

CODE:

N = 100005

ans = [0 for i in range(N)]

flag = [0 for i in range(N)]

def minimumTime(u, par, hasApple, adj):

if (hasApple[u] == True):

flag[u] = 1

for it in adj[u]:

if (it != par):

minimumTime(it, u, hasApple, adj)

if (flag[it] > 0):

ans[u] += (ans[it] + 2)

flag[u] |= flag[it]

if \_name=='main\_':

n = 7

hasApple = [ False, False, True,

False, True, True, False ]

edges = []

edges.append([0, 1])

edges.append([0, 2])

edges.append([1, 4])

edges.append([1, 5])

edges.append([2, 3])

edges.append([2, 6])

adj = [[] for i in range(n)]

for i in range(len(edges)):

source\_node = edges[i][0]

destination\_node = edges[i][1]

adj[source\_node].append(destination\_node)

adj[destination\_node].append(source\_node)

minimumTime(0, -1, hasApple, adj);

print(ans[0])

6. Number of Ways of Cutting a Pizza

Given a rectangular pizza represented as a rows x cols matrix containing the following characters: 'A' (an apple) and '.' (empty cell) and given the integer k. You have to cut the pizza into k pieces using k-1 cuts.

For each cut you choose the direction: vertical or horizontal, then you choose a cut position at the cell boundary and cut the pizza into two pieces. If you cut the pizza vertically, give the left part of the pizza to a person. If you cut the pizza horizontally, give the upper part of the pizza to a person. Give the last piece of pizza to the last person.

Return the number of ways of cutting the pizza such that each piece contains at least one apple. Since the answer can be a huge number, return this modulo 10^9 + 7.

Example 1:

Input: pizza = ["A..","AAA","..."], k = 3

Output: 3

Explanation: The figure above shows the three ways to cut the pizza. Note that pieces must contain at least one apple.

Example 2:

Input: pizza = ["A..","AA.","..."], k = 3

Output: 1

Example 3:

Input: pizza = ["A..","A..","..."], k = 1

Output: 1

Constraints:

● 1 <= rows, cols <= 50

● rows == pizza.length

● cols == pizza[i].length

● 1 <= k <= 10

● pizza consists of characters 'A' and '.' only.

CODE:

from typing import List

from functools import lru\_cache

class Solution:

def ways(self, pizza: List[str], K: int) -> int:

m, n, MOD = len(pizza), len(pizza[0]), 10 \*\* 9 + 7

preSum = [[0] \* (n + 1) for \_ in range(m + 1)]

# Calculate the prefix sum array

for r in range(m - 1, -1, -1):

for c in range(n - 1, -1, -1):

preSum[r][c] = (preSum[r][c + 1] + preSum[r + 1][c] -

preSum[r + 1][c + 1] + (pizza[r][c] == 'A'))

@lru\_cache(None)

def dp(k, r, c):

if preSum[r][c] == 0:

return 0

if k == 0:

return 1

ans = 0

for nr in range(r + 1, m):

if preSum[r][c] - preSum[nr][c] > 0:

ans = (ans + dp(k - 1, nr, c)) % MOD

for nc in range(c + 1, n):

if preSum[r][c] - preSum[r][nc] > 0:

ans = (ans + dp(k - 1, r, nc)) % MOD

return ans

return dp(K - 1, 0, 0)

# Example usage:

solution = Solution()

pizza = ["A..","AAA","..."]

K = 3

print(solution.ways(pizza, K)) # Output should match the expected result