**DSA – ASSIGNMENT 6**

💡 **Question 1** A permutation perm of n + 1 integers of all the integers in the range [0, n] can be represented as a string s of length n where:

* s[i] == 'I' if perm[i] < perm[i + 1], and
* s[i] == 'D' if perm[i] > perm[i + 1].

Given a string s, reconstruct the permutation perm and return it. If there are multiple valid permutations perm, return **any of them**.

**Example 1:**

**Input:** s = "IDID"

**Output:**

[0,4,1,3,2]

**Solution. :-**

* Initialize an empty list called perm to store the reconstructed permutation.
* Initialize a variable n to the length of the string s.
* Initialize two variables, low and high, to 0 and n respectively. These variables represent the current range of numbers available for selection.
* Iterate over each character c in the string s:
  + If c is 'I', append the value of low to perm and increment low by 1.
  + If c is 'D', append the value of high to perm and decrement high by 1.
* After the iteration, low and high will have the same value. Append this value to perm.
* Return the reconstructed permutation perm.

**def reconstructPermutation(s):**

**perm = []**

**n = len(s)**

**low, high = 0, n**

**for c in s:**

**if c == 'I':**

**perm.append(low)**

**low += 1**

**else:**

**perm.append(high)**

**high -= 1**

**perm.append(low)**

**return perm**

**s = "IDID"**

**result = reconstructPermutation(s)**

**print(result)**

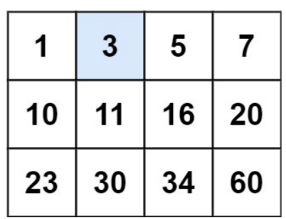
💡 **Question 2** You are given an m x n integer matrix with the following two properties:

* Each row is sorted in non-decreasing order.
* The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m \* n)) time complexity.

**Example 1:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Solution. :-**

* Initialize two variables, m and n, to the number of rows and columns in the matrix, respectively.
* Initialize two pointers, left and right, to the first and last indices of the flattened matrix.
  + left = 0
  + right = m \* n - 1
* While left <= right:
  + Calculate the middle index as mid = (left + right) // 2.
  + Convert the middle index mid back to the row and column indices using the formula:
    - row = mid // n
    - col = mid % n
  + Compare the value at the matrix[row][col] with the target:
    - If the value is equal to the target, return True.
    - If the value is greater than the target, update right = mid - 1 to search in the left half.
    - If the value is less than the target, update left = mid + 1 to search in the right half.
* If the target is not found after the loop, return False.

**def searchMatrix(matrix, target):**

**m = len(matrix)**

**n = len(matrix[0])**

**left = 0**

**right = m \* n - 1**

**while left <= right:**

**mid = (left + right) // 2**

**row = mid // n**

**col = mid % n**

**value = matrix[row][col]**

**if value == target:**

**return True**

**elif value > target:**

**right = mid - 1**

**else:**

**left = mid + 1**

**return False**

**matrix = [**

**[1, 3, 5, 7],**

**[10, 11, 16, 20],**

**[23, 30, 34, 60]**

**]**

**target = 3**

**result = searchMatrix(matrix, target)**

**print(result)**

💡 **Question 3** Given an array of integers arr, return *true if and only if it is a valid mountain array*.

Recall that arr is a mountain array if and only if:

* arr.length >= 3
* There exists some i with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
  + arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

**Example 1:**

**Input:** arr = [2,1]

**Output:**

False

**Solution. :-**

* Check if the length of arr is less than 3. If it is, return False since a mountain array must have at least three elements.
* Initialize two variables, i and n, to 0 and the length of arr minus 1, respectively.
* Increment i until arr[i] < arr[i + 1]. This will find the peak of the mountain.
* If i is 0 or i is equal to n, return False since a mountain array must have a peak that is not at the beginning or end.
* Increment i until arr[i] > arr[i + 1]. If the loop reaches the end of the array, return True, indicating that it is a valid mountain array.
* If the loop does not reach the end of the array, return False.

**def validMountainArray(arr):**

**n = len(arr)**

**i = 0**

**# Find the peak of the mountain**

**while i < n - 1 and arr[i] < arr[i + 1]:**

**i += 1**

**# Check if the peak is at the beginning or end**

**if i == 0 or i == n - 1:**

**return False**

**# Check if the array decreases after the peak**

**while i < n - 1 and arr[i] > arr[i + 1]:**

**i += 1**

**# Return True if the loop reaches the end of the array**

**return i == n – 1**

**arr = [2, 1]**

**result = validMountainArray(arr)**

**print(result)**

💡 **Question 4** Given a binary array nums, return *the maximum length of a contiguous subarray with an equal number of* 0 *and* 1.

**Example 1:**

**Input:** nums = [0,1]

**Output:** 2

**Explanation:**

[0, 1] is the longest contiguous subarray with an equal number of 0 and 1.

**Solution. :-**

* Initialize a dictionary count to store the running count of 0s and 1s encountered during the iteration. The dictionary will store the cumulative count of 0s and 1s encountered so far as the key and the index at which it was encountered as the value.
  + Initialize count[0] = -1 to account for the case when an equal number of 0s and 1s are encountered starting from the beginning.
* Initialize a variable max\_len to 0 to store the maximum length of the subarray.
* Initialize a variable cur\_sum to 0 to store the running count of 0s and 1s encountered so far.
* Iterate over the binary array nums, starting from index 0:
  + If the current element is 0, increment cur\_sum by -1. Otherwise, increment it by 1.
  + If cur\_sum is already in the count dictionary, calculate the length of the subarray as i - count[cur\_sum], where i is the current index.
  + If the length of the subarray is greater than max\_len, update max\_len with the new length.
  + If cur\_sum is not in the count dictionary, add it to the dictionary with the current index as the value.
* Return max\_len, which represents the maximum length of a contiguous subarray with an equal number of 0s and 1s.

**def findMaxLength(nums):**

**count = {0: -1}**

**max\_len = 0**

**cur\_sum = 0**

**for i in range(len(nums)):**

**cur\_sum += -1 if nums[i] == 0 else 1**

**if cur\_sum in count:**

**subarray\_len = i - count[cur\_sum]**

**max\_len = max(max\_len, subarray\_len)**

**else:**

**count[cur\_sum] = i**

**return max\_len**

**nums = [0, 1]**

**result = findMaxLength(nums)**

**print(result)**

💡 **Question 5** The **product sum** of two equal-length arrays a and b is equal to the sum of a[i] \* b[i] for all 0 <= i < a.length (**0-indexed**).

* For example, if a = [1,2,3,4] and b = [5,2,3,1], the **product sum** would be 1*5 + 2*2 + 3*3 + 4*1 = 22.

Given two arrays nums1 and nums2 of length n, return *the* ***minimum product sum*** *if you are allowed to* ***rearrange*** *the* ***order*** *of the elements in* nums1.

**Example 1:**

**Input:** nums1 = [5,3,4,2], nums2 = [4,2,2,5]

**Output:** 40

**Explanation:**

We can rearrange nums1 to become [3,5,4,2]. The product sum of [3,5,4,2] and [4,2,2,5] is 3*4 + 5*2 + 4*2 + 2*5 = 40.

**Solution. :-**

* Sort both nums1 and nums2 arrays in non-decreasing order.
* Initialize a variable min\_product\_sum to 0 to store the minimum product sum.
* Iterate over the sorted arrays nums1 and nums2 simultaneously:
  + Multiply the current element of nums1 with the corresponding element of nums2.
  + Add the result to min\_product\_sum.
* Return min\_product\_sum.

**def minProductSum(nums1, nums2):**

**nums1.sort()**

**nums2.sort(reverse=True)**

**min\_product\_sum = 0**

**for i in range(len(nums1)):**

**min\_product\_sum += nums1[i] \* nums2[i]**

**return min\_product\_sum**

**nums1 = [5, 3, 4, 2]**

**nums2 = [4, 2, 2, 5]**

**result = minProductSum(nums1, nums2)**

**print(result)**

💡 **Question 6** An integer array original is transformed into a **doubled** array changed by appending **twice the value** of every element in original, and then randomly **shuffling** the resulting array.

Given an array changed, return original *if* changed *is a* ***doubled*** *array. If* changed *is not a* ***doubled*** *array, return an empty array. The elements in* original *may be returned in* ***any*** *order*.

**Example 1:**

**Input:** changed = [1,3,4,2,6,8]

**Output:** [1,3,4]

**Explanation:** One possible original array could be [1,3,4]:

* Twice the value of 1 is 1 \* 2 = 2.
* Twice the value of 3 is 3 \* 2 = 6.
* Twice the value of 4 is 4 \* 2 = 8.

Other original arrays could be [4,3,1] or [3,1,4].

**Solution. :-**

* Create an empty dictionary called freq to store the frequency of each element in the changed array.
* Iterate over each element num in the changed array and update its frequency in the freq dictionary.
* Iterate over each element num in the changed array again:
  + If freq[num] is 0, continue to the next iteration since num is not present in the original array.
  + Decrement freq[num] by 1.
  + If freq[2 \* num] is 0, return an empty array since the corresponding element 2 \* num is not present in the original array.
  + Decrement freq[2 \* num] by 1.
* If all elements in the changed array are processed successfully, create an empty list called original.
* Iterate over each element num in the changed array again:
  + If freq[num] is greater than 0, append num to the original list freq[num] times.
* Return the original list, which represents one possible original array.

**from collections import defaultdict**

**def findOriginalArray(changed):**

**freq = defaultdict(int)**

**for num in changed:**

**freq[num] += 1**

**original = []**

**for num in changed:**

**if freq[num] == 0:**

**continue**

**freq[num] -= 1**

**if freq[2 \* num] == 0:**

**return []**

**freq[2 \* num] -= 1**

**original.append(num)**

**return original**

**changed = [1, 3, 4, 2, 6, 8]**

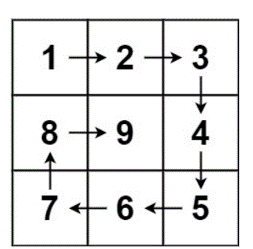
**result = findOriginalArray(changed)**

**print(result)**

💡 **Question 7**

Given a positive integer n, generate an n x n matrix filled with elements from 1 to n2 in spiral order.

**Example 1:**



**Input:** n = 3

**Output:** [[1,2,3],[8,9,4],[7,6,5]]

**Solution. :-**

* Initialize the matrix result with zeros of size n x n.
* Initialize variables row\_start, row\_end, col\_start, and col\_end to keep track of the boundaries of the matrix.
  + row\_start is the starting row index.
  + row\_end is the ending row index.
  + col\_start is the starting column index.
  + col\_end is the ending column index.
* Initialize a variable num to 1, which represents the current number to be filled in the matrix.
* Run a while loop until num reaches n^2.
  + Iterate from col\_start to col\_end (left to right) and fill the matrix with the current value of num. Increment num after each element is filled.
  + Increment row\_start by 1 to move to the next row.
  + Iterate from row\_start to row\_end (top to bottom) and fill the matrix with the current value of num. Increment num after each element is filled.
  + Decrement col\_end by 1 to move to the previous column.
  + Iterate from col\_end to col\_start (right to left) and fill the matrix with the current value of num. Increment num after each element is filled.
  + Decrement row\_end by 1 to move to the previous row.
  + Iterate from row\_end to row\_start (bottom to top) and fill the matrix with the current value of num. Increment num after each element is filled.
  + Increment col\_start by 1 to move to the next column.
* Return the resulting matrix result.

**def generateMatrix(n):**

**result = [[0] \* n for \_ in range(n)]**

**row\_start, row\_end = 0, n - 1**

**col\_start, col\_end = 0, n - 1**

**num = 1**

**while num <= n \* n:**

**for j in range(col\_start, col\_end + 1):**

**result[row\_start][j] = num**

**num += 1**

**row\_start += 1**

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

**result[i][col\_end] = num**

**num += 1**

**col\_end -= 1**

**for j in range(col\_end, col\_start - 1, -1):**

**result[row\_end][j] = num**

**num += 1**

**row\_end -= 1**

**for i in range(row\_end, row\_start - 1, -1):**

**result[i][col\_start] = num**

**num += 1**

**col\_start += 1**

**return result**

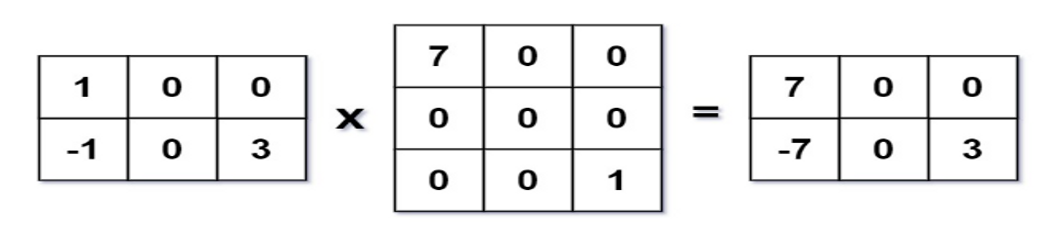
**n = 3**

**result = generateMatrix(n)**

**print(result)**

💡 **Question 8** Given two [sparse matrices](https://en.wikipedia.org/wiki/Sparse_matrix) mat1 of size m x k and mat2 of size k x n, return the result of mat1 x mat2. You may assume that multiplication is always possible.

**Example 1:**



**Input:** mat1 = [[1,0,0],[-1,0,3]], mat2 = [[7,0,0],[0,0,0],[0,0,1]]

**Output:**

[[7,0,0],[-7,0,3]]

**Solution. :-**

* Initialize an empty result matrix result of size m x n with all elements set to 0.
* Iterate over each row i in mat1 and each column j in mat2:
  + Initialize a variable temp to 0 to store the temporary sum of the dot product of row i in mat1 and column j in mat2.
  + Iterate over each element x in row i of mat1 and each element y in column j of mat2:
    - If the value of mat1[i][x] or mat2[x][j] is 0, continue to the next iteration.
    - Update temp by adding the product of mat1[i][x] and mat2[x][j].
  + Set result[i][j] to the value of temp.
* Return the resulting matrix result.

**def multiply(mat1, mat2):**

**m, k, n = len(mat1), len(mat1[0]), len(mat2[0])**

**result = [[0] \* n for \_ in range(m)]**

**for i in range(m):**

**for j in range(n):**

**temp = 0**

**for x in range(k):**

**if mat1[i][x] == 0 or mat2[x][j] == 0:**

**continue**

**temp += mat1[i][x] \* mat2[x][j]**

**result[i][j] = temp**

**return result**

**mat1 = [[1, 0, 0], [-1, 0, 3]]**

**mat2 = [[7, 0, 0], [0, 0, 0], [0, 0, 1]]**

**result = multiply(mat1, mat2)**

**print(result)**