CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

Experiment 5

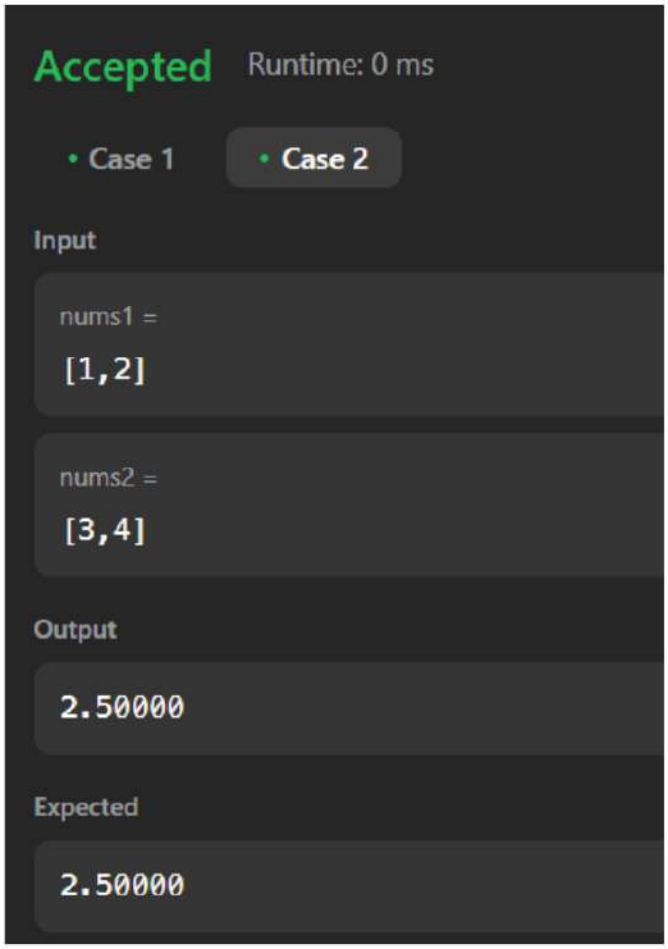
Name: Aaditya Maheshwari
UID: 22BET10094
Branch: IT
Section/Group: 22BET_IOT-702(A)
Semester: 6th
Date of Performance: 19/02/2025
Subject Name: Advance Programming
Subject Code: 22ITP-351

Aim: Median of Two Sorted Arrays
Objective: Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.
The overall run time complexity should be O(log (m+n)).
Code:

```
class Solution:
 def findMedianSortedArrays(self, nums1, nums2):
  if len(nums1) > len(nums2):
   nums1, nums2 = nums2, nums1

  x, y = len(nums1), len(nums2)
  low, high = 0, x

  while low <= high:
   partitionX = (low + high) // 2
   partitionY = (x + y + 1) // 2 - partitionX

   maxLeftX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
   minRightX = float('inf') if partitionX == x else nums1[partitionX]

   maxLeftY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
   minRightY = float('inf') if partitionY == y else nums2[partitionY]

   if maxLeftX <= minRightY and maxLeftY <= minRightX:
    if (x + y) % 2 == 0:
     return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2.0
    else:
     return float(max(maxLeftX, maxLeftY))
   elif maxLeftX > minRightY:
    high = partitionX - 1
```

```
else:
    low = partitionX + 1

solution = Solution()
nums1 = [1, 3]
nums2 = [2]
print(solution.findMedianSortedArrays(nums1, nums2))

nums1 = [1, 2]
nums2 = [3, 4]
print(solution.findMedianSortedArrays(nums1, nums2))
```

Output:



Aim: Kth smallest element in a sorted matrix

Objective: Given an n x n matrix where each of the rows and columns is sorted in ascending order, return the kth smallest element in the matrix.
Note that it is the kth smallest element in the sorted order, not the kth distinct element.
You must find a solution with a memory complexity better than O(n2).

Code:

```
import heapq

class Solution:
  def kthSmallest(self, matrix, k):
    n = len(matrix)
    min_heap = [(matrix[i][0], i, 0) for i in range(n)]
    heapq.heapify(min_heap)

    for _ in range(k - 1):
      value, r, c = heapq.heappop(min_heap)
      if c + 1 < n:
        heapq.heappush(min_heap, (matrix[r][c + 1], r, c + 1))

    return heapq.heappop(min_heap)[0]

solution = Solution()
matrix = [[1, 5, 9], [10, 11, 13], [12, 13, 15]]
k = 8
print(solution.kthSmallest(matrix, k))
```

Output:

Aim: Search a 2D Matrix II

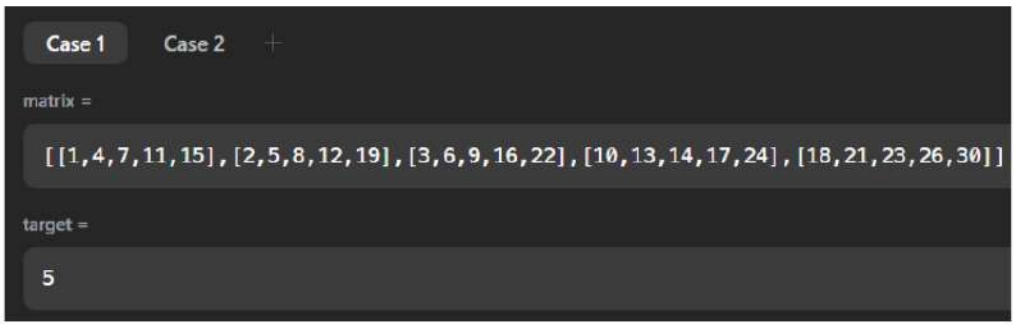
Objective: Write an efficient algorithm that searches for a value target in an m x n integer matrix matrix. This matrix has the following properties:

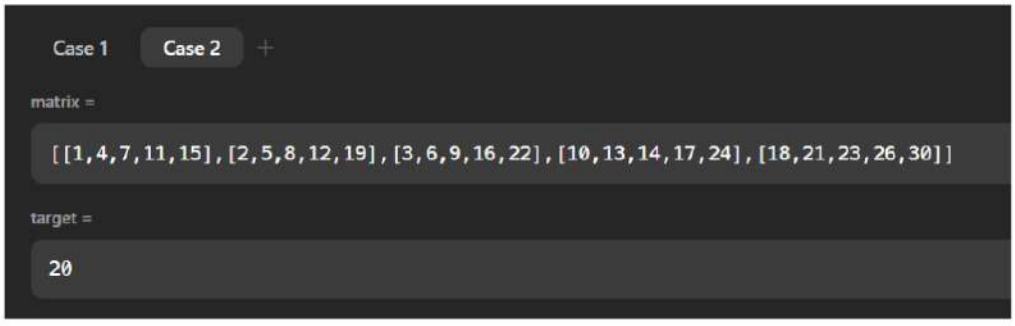
- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Code:

```
class Solution:
 def searchMatrix(self, matrix, target):
  if not matrix or not matrix[0]:
   return False

  rows, cols = len(matrix), len(matrix[0])
  row, col = 0, cols - 1

  while row < rows and col >= 0:
   if matrix[row][col] == target:
    return True
   elif matrix[row][col] > target:
    col -= 1
   else:
    row += 1

  return False
```

Output:





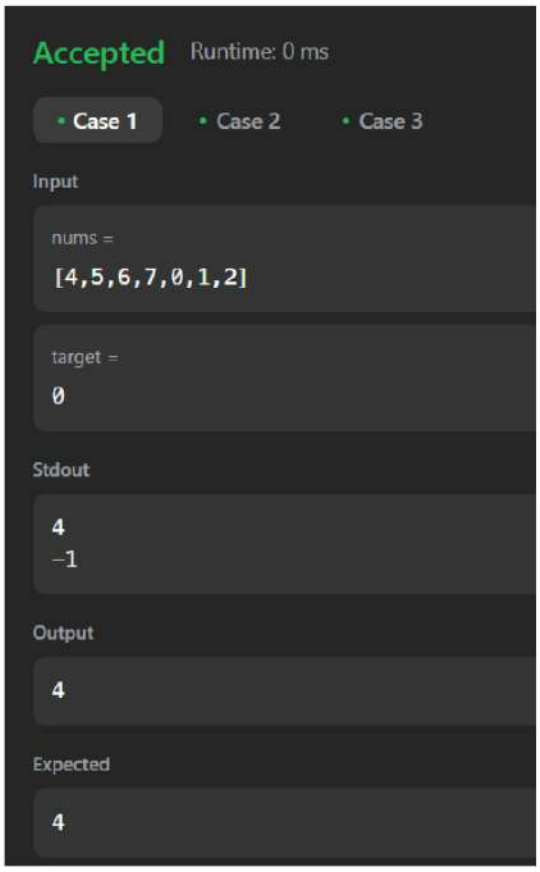
Aim: Search in Rotated Sorted Array

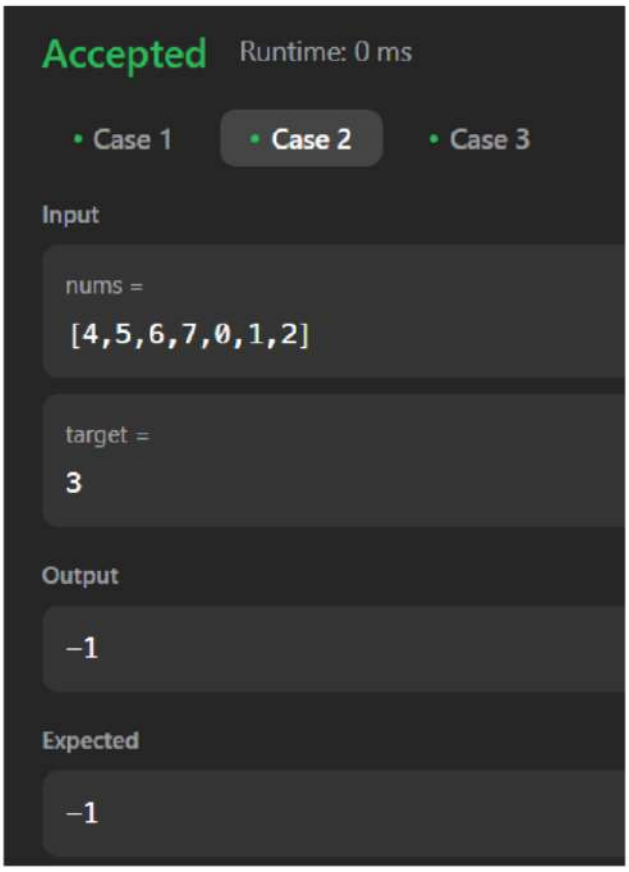
Objective: There is an integer array nums sorted in ascending order (with distinct values).
Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].
Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.
You must write an algorithm with O(log n) runtime complexity.

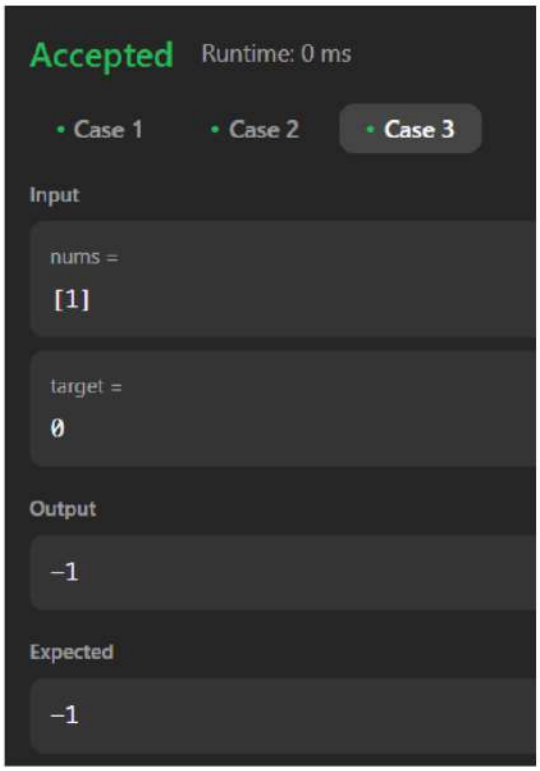
Code:

```
class Solution:
  def search(self, nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
      mid = (left + right) // 2

      if nums[mid] == target:
        return mid

      if nums[left] <= nums[mid]:
        if nums[left] <= target < nums[mid]:
          right = mid - 1
        else:
          left = mid + 1
      else:
        if nums[mid] < target <= nums[right]:
          left = mid + 1
        else:
          right = mid - 1

    return -1

solution = Solution()
nums = [4,5,6,7,0,1,2]
target = 0
print(solution.search(nums, target))

target = 3
print(solution.search(nums, target))
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[4,5,6,7,0,1,2]
target =
0
Stdout
4
-1
Output
4
Expected
4


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[4,5,6,7,0,1,2]
target =
3
Output
-1
Expected
-1


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[1]
target =
0
Output
-1
Expected
-1

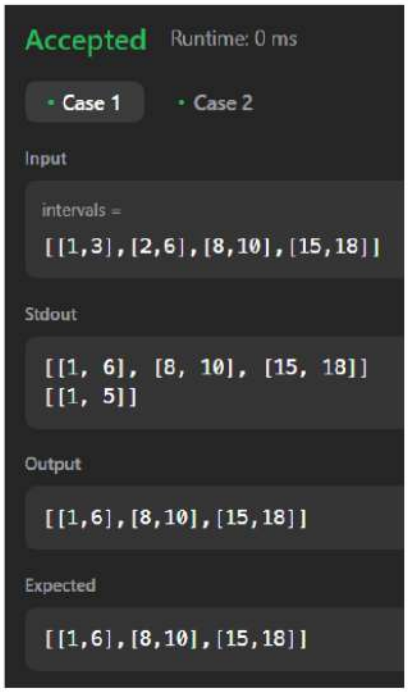
Aim: Merge Intervals

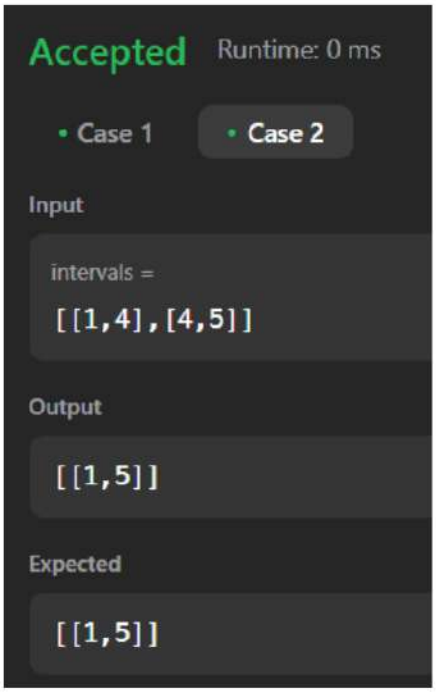
Objective: Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Code:

```
class Solution:
  def merge(self, intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []

    for interval in intervals:
      if not merged or merged[-1][1] < interval[0]:
        merged.append(interval)
      else:
        merged[-1][1] = max(merged[-1][1], interval[1])

    return merged

solution = Solution()
intervals = [[1,3],[2,6],[8,10],[15,18]]
print(solution.merge(intervals))

intervals = [[1,4],[4,5]]
print(solution.merge(intervals))
```

Output:





Aim: Find Peak Element

Objective: A peak element is an element that is strictly greater than its neighbors.
Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.
You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.
You must write an algorithm that runs in O(log n) time.

Code:

```
class Solution:
  def findPeakElement(self, nums):
    left, right = 0, len(nums) - 1

    while left < right:
      mid = (left + right) // 2
      if nums[mid] > nums[mid + 1]:
        right = mid
      else:
        left = mid + 1

    return left

solution = Solution()
nums = [1,2,3,1]
print(solution.findPeakElement(nums))

nums = [1,2,1,3,5,6,4]
print(solution.findPeakElement(nums))
```

Output:

Aim: Kth Largest element in an array

Objective: Given an integer array nums and an integer k, return the kth largest element in the array.
Note that it is the kth largest element in the sorted order, not the kth distinct element.
Can you solve it without sorting?

Code:

```
import heapq

class Solution:
  def findKthLargest(self, nums, k):
    return heapq.nlargest(k, nums)[-1]

solution = Solution()
nums = [3,2,1,5,6,4]
k = 2
print(solution.findKthLargest(nums, k))

nums = [3,2,3,1,2,4,5,5,6]
k = 4
print(solution.findKthLargest(nums, k))
```

Output: