Lab session

1. Given two integer arrays nums1 and nums2, return an array of their intersection . Each element in the result must be unique and you may return the result in any order. Example 1: Input: nums1 = [1,2,2,1], nums2 = [2,2] Output: [2] Example 2: Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4] Output: [9,4] Explanation: [4,9] is also accepted. Constraints: 1 <= nums1.length, nums2.length <= 1000 0 <= nums1[i], nums2[i] <= 1000

Program:

```
def intersection(nums1, nums2):
    # Convert both lists to sets to remove duplicates
    set1 = set(nums1)
    set2 = set(nums2)

    # Find the intersection of both sets
    intersection_set = set1 & set2

    # Convert the set to a list (optional, depending on desired output format)
    result = list(intersection_set)

    return result

# Example usage:
nums1 = [1, 2, 2, 1]
nums2 = [2, 2]
print(intersection(nums1, nums2))  # Output: [2]

nums1 = [4, 9, 5]
nums2 = [9, 4, 9, 8, 4]
print(intersection(nums1, nums2))  # Output: [9, 4] or [4, 9]
```

2. 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? Example 1: Input: nums = [3,2,1,5,6,4], k = 2 Output: 5 Example 2: Input: nums = [3,2,3,1,2,4,5,5,6], k = 4 Output: 4 Constraints: 1 <= k <= nums.length <= 105 -104 <= nums[i] <= 104

Program:

```
import heapq


def findKthLargest(nums, k):
    # Create a min-heap with the first k elements of nums
    heap = nums[:k]
    heapq.heapify(heap)

    # Iterate through the remaining elements of nums
    for num in nums[k:]:
        # If the current number is larger than the smallest element in the heap
        if num > heap[0]:
            # Remove the smallest element and add the current number to the heap
            heapq.heappop(heap)
            heapq.heappush(heap, num)

    # The root of the heap is the k-th largest element
    return heap[0]


# Example usage:
nums1 = [3, 2, 1, 5, 6, 4]
k1 = 2
print(findKthLargest(nums1, k1))  # Output: 5


nums2 = [3, 2, 3, 1, 2, 4, 5, 5, 6]
k2 = 4
print(findKthLargest(nums2, k2))  # Output: 4
```

3. Given the strings s1 and s2 of size n and the string evil, return the number of good strings. A good string has size n, it is alphabetically greater than or equal to s1, it is alphabetically smaller than or equal to s2, and it does not contain the string evil as a substring. Since the answer can be a huge number, return this modulo 109 + 7. Example 1: Input: n = 2, s1 = "aa", s2 = "da", evil = "b" Output: 51 Explanation: There are 25 good strings starting with 'a': "aa","ac","ad",...,"az". Then there are 25

good strings starting with 'c': "ca","cc","cd",...,"cz" and finally there is one good string starting with 'd': "da".

Program:

```
MOD = 10**9 + 7


def countGoodStrings(n, s1, s2, evil):
    # Precompute the KMP failure function for the evil string
    m = len(evil)
    kmp = [0] * m
    j = 0
    for i in range(1, m):
        while j > 0 and evil[i] != evil[j]:
            j = kmp[j - 1]
        if evil[i] == evil[j]:
            j += 1
        kmp[i] = j

    memo = {}

    def dp(pos, tight1, tight2, evil_len):
        if evil_len == m:
            return 0  # Found evil as a substring
        if pos == n:
            return 1  # Reached the end of the string

        if (pos, tight1, tight2, evil_len) in memo:
            return memo[(pos, tight1, tight2, evil_len)]

        # Calculate the range of the current character
        from_char = s1[pos] if tight1 else 'a'
        to_char = s2[pos] if tight2 else 'z'
```

```
        res = 0
        for char in range(ord(from_char), ord(to_char) + 1):
            new_tight1 = tight1 and (char == ord(s1[pos]))
            new_tight2 = tight2 and (char == ord(s2[pos]))

            # Update the evil_len using the KMP logic
            new_evil_len = evil_len
            while new_evil_len > 0 and chr(char) != evil[new_evil_len]:
                new_evil_len = kmp[new_evil_len - 1]
            if chr(char) == evil[new_evil_len]:
                new_evil_len += 1

            res = (res + dp(pos + 1, new_tight1, new_tight2, new_evil_len)) % MOD

        memo[(pos, tight1, tight2, evil_len)] = res
        return res

    return dp(0, True, True, 0)

# Example usage
n = 2
s1 = "aa"
s2 = "da"
evil = "b"
print(countGoodStrings(n, s1, s2, evil))  # Output: 51
```

4.Given an array nums of size n, return the majority element.The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array. Example 1: Input: nums = [3,2,3] Output: 3 Example 2: Input: nums = [2,2,1,1,1,2,2] Output: 2 Constraints: n == nums.length 1 <= n <= 5 * 104

program:

```
def majorityElement(nums):
```

```
    candidate = None
    count = 0

    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)

    return candidate

# Example usage:
nums1 = [3, 2, 3]
print(majorityElement(nums1))  # Output: 3

nums2 = [2, 2, 1, 1, 1, 2, 2]
print(majorityElement(nums2))  # Output: 2
```

5.Given a 2D integer array matrix, return the transpose of matrix. The transpose of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices. Example 1: Input: matrix = [[1,2,3],[4,5,6],[7,8,9]] Output: [[1,4,7],[2,5,8],[3,6,9]] Example 2: Input: matrix = [[1,2,3],[4,5,6]] Output: [[1,4],[2,5],[3,6]]

Program:

```
def transpose(matrix):
    # Get the number of rows and columns of the original matrix
    rows = len(matrix)
    cols = len(matrix[0])

    # Initialize the transposed matrix with dimensions cols x rows
    transposed = [[0] * rows for _ in range(cols)]

    # Fill the transposed matrix
    for i in range(rows):
        for j in range(cols):
```

```
            transposed[j][i] = matrix[i][j]

    return transposed

# Example usage:
matrix1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(transpose(matrix1))
# Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

matrix2 = [[1, 2, 3], [4, 5, 6]]
print(transpose(matrix2))
# Output: [[1, 4], [2, 5], [3, 6]]
```