# Experiment 5

**Student Name:** Tarun Baliyan **UID:** 22BET10184

**Branch:** IT **Section/Group:** 22BET\_IOT-702/A

**Semester:** 6th **Date of Performance:** 27/02/25

**Subject Name:** Advance Programming-II **Subject Code:** 22ITP-367

**Problem: 1.4.1: Merge Sorted Array**

**Problem Statement:** You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2, respectively. Modify nums1 in-place to be a single merged sorted array.

1. **Objective:** Merge nums2 into nums1 as one sorted array in non-decreasing order. The final sorted array should be stored inside nums1. Merge nums2 into nums1 as one sorted array in non-decreasing order. The final sorted array should be stored inside nums1.
2. **Code:**

## class Solution:

## def longestNiceSubstring(self, s: str) -> str:

## # Base case: if the string is empty or has only one character, return ""

## if len(s) < 2:

## return ""

## 

## # Check for invalid characters

## for i, ch in enumerate(s):

## if ch.swapcase() not in s:

## # Split around the invalid character and check both parts

## left = self.longestNiceSubstring(s[:i])

## right = self.longestNiceSubstring(s[i+1:])

## # Return the longer substring

## return left if len(left) >= len(right) else right

## 

## # If all characters are valid, return the entire string

## return s

## 3. Result:

## 

**Problem 1.4.2: Reverse Bits**

**Problem Statement:** You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

1. **Objective:** Find the smallest version number x such that isBadVersion(x) == true, using the minimum number of API calls.
2. **Code:**

class Solution {

public:

int firstBadVersion(int n) {

int left = 1, right = n;

while (left < right) {

int mid = left + (right - left) / 2; // Prevents integer overflow

if (isBadVersion(mid)) {

right = mid; // Search in the left half

} else {

left = mid + 1; // Search in the right half } }

return left; // First bad version }};

1. **Result:**

## 

**Problem 1.4.3: Kth Largest Element in an Array**

**Problem 1.4.3: Kth Largest Element in an Array**

**Problem Statement:** Given an integer array nums and an integer k, return the kth largest element in the array.

1. **Objective:** Find the **kth largest element** in the array using an efficient approach.
2. **Code:**

#include <vector>

#include <queue>

using namespace std;

class Solution {

public:

int findKthLargest(vector<int>& nums, int k) {

priority\_queue<int, vector<int>, greater<int>> minHeap; // Min heap

// Maintain a min-heap of size k

for (int num : nums) {

minHeap.push(num);

if (minHeap.size() > k) {

minHeap.pop(); // Remove smallest element

}

}

return minHeap.top(); // kth largest element

}

};

1. **Result:**

## 

**Problem 1.3.4: Find Peak Element**

**Problem Statement:** 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 there are multiple peaks, return any peak index.

You may assume nums[-1] = -∞ and nums[n] = -∞ (i.e., elements outside the bounds are always smaller)

1. **Objective:** Find any peak element efficiently using binary search.
2. **Code:**

#include <vector>

using namespace std;

class Solution {

public:

    int findPeakElement(vector<int>& nums) {

        int left = 0, right = nums.size() - 1;

        while (left < right) {

            int mid = left + (right - left) / 2;

            if (nums[mid] > nums[mid + 1]) {

                right = mid; // Peak is in the left half

            } else {

                left = mid + 1; // Peak is in the right half

            }

        }

        return left; // Index of the peak element

    }

};

1. **Result:**

## 

**Problem 1.4.5: Median of Two Sorted Arrays**

**Problem Statement:** Given two sorted arrays nums1 and nums2 of sizes m and n, return the median of the two sorted arrays.

The overall run-time complexity should be O(log (m + n)).

1. **Objective:** Find the median of two sorted arrays **efficiently** using **binary search**.
2. **Code:**

#include <vector>

using namespace std;

class Solution {

public:

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

if (nums1.size() > nums2.size()) {

return findMedianSortedArrays(nums2, nums1); // Ensure nums1 is smaller

}

int m = nums1.size(), n = nums2.size();

int left = 0, right = m, halfLen = (m + n + 1) / 2;

while (left <= right) {

int mid1 = left + (right - left) / 2;

int mid2 = halfLen - mid1;

int left1 = (mid1 > 0) ? nums1[mid1 - 1] : INT\_MIN;

int right1 = (mid1 < m) ? nums1[mid1] : INT\_MAX;

int left2 = (mid2 > 0) ? nums2[mid2 - 1] : INT\_MIN;

int right2 = (mid2 < n) ? nums2[mid2] : INT\_MAX;

if (left1 <= right2 && left2 <= right1) {

if ((m + n) % 2 == 0) {

return (max(left1, left2) + min(right1, right2)) / 2.0;

} else {

return max(left1, left2);

}

} else if (left1 > right2) {

right = mid1 - 1;

} else {

left = mid1 + 1;

}

}

return 0;

1. **Result:**

## 