|  |  |  |
| --- | --- | --- |
| A green and white logo  Description automatically generated­ |  | **ISM 6225 Application Development for Analytics** |

Assignment 2 – Computational Problem Solving

Hriday Reddy Purma (U35212569)

#### Questions:

### 1. Find Missing Numbers in Array:

**Description:**  
Given an unsorted integer array nums of size n containing numbers from 1 to n, find all the numbers that are missing from the array.

**Examples:**

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

**CODE:**

**public static IList<int> FindMissingNumbers(int[] nums)**

**{**

**try**

**{**

**// checking if the array is null or empty and returning an empty list**

**if (nums == null || nums.Length == 0)**

**{**

**return new List<int>(); // Return an empty list if the array is null or empty**

**}**

**int n = nums.Length;**

**// Using a HashSet to track numbers and the missing ones to return**

**HashSet<int> numSet = new HashSet<int>(nums);**

**// we are creating a list to store the missing numbers**

**List<int> missingNum = new List<int>();**

**// Looping through the length of the given array.**

**for (int i = 1; i <= n; i++)**

**{**

**// Checking if the number is present in the hashset or not.**

**if (!numSet.Contains(i))**

**{**

**// If it is not present, we can add it to a LIST of missing numbers.**

**missingNum.Add(i);**

**}**

**}**

**// Returning the list of missing numbers**

**// if there are no missing elements in the given array, it will return an empty list**

**return missingNum;**

**}**

**// IF there are any exceptions, we are throwing the exception**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial Thought Process:**

My initial idea was to use a List (List<int>) to store the missing numbers since we don't know how many values will be missing from the sequence. I planned to loop from 1 to the length of the array and for each number, check whether it is present in the array or not. If not found, I would add it to the list of missing numbers. However, I realized that this approach could be inefficient due to the repeated search operations over the array, which would result in a time complexity of O(n²) in the worst case.

**Copilot prompt** – How to iterate through an array to find the missing numbers?

**Suggestion**: Using Hashset. A Hashset allows us to store only the unique elements which are present in the given array, discarding the duplicates. Then we iterate through the length of the array and check for the missing elements which are then added to the list. Then we return the list as the output. I made sure to include exception handling, although the AI did not initially suggest it.

This suggestion with the help of AI Copilot code suggestions, I was able to come up with the code above. The exception handling as mentioned in the above code is helpful in this case for when the array is empty, the Code is supposed to return an empty list as the output.

This approach of using a Hashset to save the unique elements of an array helped in reducing the time complexity to O(n) from my initial thought process, where it would have been O(n^2). The Space Complexity for this code is O(n) as we are storing the numbers using a hashset.

**INPUT:**

Console.WriteLine("Question 1:");

int[] nums1 = { 4, 3, 2, 7, 8, 2, 3, 1 };

IList<int> missingNumbers = FindMissingNumbers(nums1);

**OUTPUT:**

Question 1:

5,6

### 2. Sort Array by Parity:

**Description:**  
Given an integer array nums, move all even integers to the beginning of the array followed by all odd integers. Return the array in-place.

**Examples:**

* Input: [3, 1, 2, 4], Output: [2, 4, 3, 1]
* Input: [0, 1, 2], Output: [0, 2, 1]

**CODE:**

**public static int[] SortArrayByParity(int[] nums)**

**{**

**try**

**{**

**// checking if the array is null or empty and returning the original array**

**if (nums == null || nums.Length == 0)**

**{**

**return nums; // Return the original array if it's null or empty**

**}**

**int left = 0, right = nums.Length - 1;**

**while (left < right)**

**{**

**// If the left number is even, we can move to the next number**

**if (nums[left] % 2 == 0)**

**{**

**left++;**

**}**

**// If the right number is odd, we can move to the previous number**

**else if (nums[right] % 2 != 0)**

**{**

**right--;**

**}**

**// If the left number is odd and the right number is even, we can swap them**

**else**

**{**

**// Swap the left and right numbers**

**int temp = nums[left];**

**nums[left] = nums[right];**

**nums[right] = temp;**

**left++;**

**right--;**

**}**

**}**

**return nums; // Placeholder**

**}**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial Thought Process:**

My initial idea was to create another array of the same size as the given array. I planned to loop through the array, find even numbers, and place them at the beginning, while moving odd numbers to the end. However, this approach has two major downsides, the increased space complexity in creating a new array and the increased time complexity in iterating through the whole new array.

**Copilot Prompt –**  How do I iterate through the elements of array in sorting the elements into even and odd without creating a new array?

**Suggestion :**  Copilot suggested to use a two-pointer approach to sort the elements in the array. If the element in the left is even go to the next element, similarly if the element of the right most pointer is odd go to the element in front of it, When there is a place where the left most pointer element is odd and the right most pointer element is even then we swap the elements places. We repeat this process until the pointers meet.

This suggestion with the help of AI copilot, I was able to come up with the above code. Checking for the exception case, if the array is null or empty, if it is so we return the initial array as the output.

This approach of using a 2-pointer approach to sort the array to begin with even and ending with odd, helped in reducing the time complexity to O(n) from my initial thought process, where it would have been O(n^2). The Space Complexity for this code is O(1) as we are not creating any new list/array but working on the current one only.

**INPUT:**

Console.WriteLine("Question 2:");

int[] nums2 = { 3, 1, 2, 4 };

int[] sortedArray = SortArrayByParity(nums2);

**OUTPUT:**

Question 2:

4,2,1,3

### 3. Two Sum (Find Two Numbers that Add to Target):

**Description:**  
Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to the target.

**Examples:**

* Input: nums = [2, 7, 11, 15], target = 9, Output: [0, 1]
* Input: nums = [3, 2, 4], target = 6, Output: [1, 2]

**CODE:**

**public static int[] TwoSum(int[] nums, int target)**

**{**

**try**

**{**

**Dictionary<int, int> newDict = new Dictionary<int, int>();**

**for(int i = 0; i < nums.Length; i++)**

**{**

**int com = target - nums[i];**

**if (newDict.ContainsKey(com))**

**{**

**return new int[] { newDict[com], i };**

**}**

**newDict[nums[i]] = i;**

**}**

**Console.WriteLine($"No two sum solution found for target {target}");**

**return new int[0]; // Placeholder**

**}**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial Thought Process:**

My initial thought was based on understanding that for any number in the array, if we subtract it from the target, the result should be the other number we're looking for. In other words, if target - nums[i] = x, then x must also exist in the array. Initially, I considered looping through the array twice: for each element, searching the rest of the array to find if this complementary value exists. This would have resulted in a time complexity of O(n²).

To improve upon this, I thought of using a dictionary to keep track of the numbers we've seen and their indices, as dictionaries allow constant-time lookups.

**Copilot Prompt** : How do I not store duplicate elements in a dictionary to not check for the element twice when I am working on the Two sum Problem?

**Suggestion** : Since each element where we are iterating through is saved in the dictionary along with the index of it and, they will be unique. The Copilot suggested that for each element we calculate its compliment and then search if the compliment (target – nums[i]) is in the array too. If it is there we return both the indexes of the compliment and that element. If we don’t find it, we store it to the dictionary and go through to the next element.

This suggestion with the help of the AI copilot, I was able to come up with the above code. I used a Dictionary<int, int> to store the number and its index for fast lookups. This approach reduces the time complexity from **O(n²)** to **O(n)** by eliminating the need for nested loops. The dictionary requires **O(n)** additional space, which is acceptable for the significant gain in performance.

**INPUT:**

int[] nums3 = { 2, 7, 11, 15 };

int target = 9;

int[] indices = TwoSum(nums3, target);

**OUTPUT:**

Question 3:

0,1

### 4. Find Maximum Product of Three Numbers:

**Description:**  
Given an integer array nums, find three numbers whose product is the maximum and return the product.

**Examples:**

* Input: [1, 2, 3], Output: 6
* Input: [1, 2, 3, 4], Output: 24

**CODE:**

**public static int MaximumProduct(int[] nums)**

**{**

**try**

**{**

**// Check if the array is null or has less than 3 elements**

**if (nums == null || nums.Length < 3)**

**{**

**throw new ArgumentException();**

**}**

**// Sort the array**

**Array.Sort(nums);**

**int n = nums.Length;**

**// Calculate the maximum product from either:**

**// - The product of the last three numbers**

**// - The product of the first two numbers (possibly negative) and the last number**

**int maxProduct = Math.Max(**

**nums[n - 1] \* nums[n - 2] \* nums[n - 3],**

**nums[0] \* nums[1] \* nums[n - 1]**

**);**

**return maxProduct;**

**}**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial Thought Process:**

The initial thought , I have gotten was to sort the array and then return the product of the last three numbers. To do this, we can use the Arrays.Sort in-built Method and sort the array making it simple. But the problem with this is, the use case where there are negative numbers in the given array. So, we might have to choose 2 negative and one positive number too if there are negative values in the array.

**Copilot Prompt** : How do I find the largest of two different functions(one is the product of the 3 largest numbers in the array and other is the product of the 2 negative values and one positive value overall) to give the maximum of the two in C#?

**Suggestion:**  To find the largest of the two functions first iterate through the two elements and then return the maximum of the two using the Math.Max ( ) in build function. With this function suggestion I was able to first sort the array from the lowest to the highest. And then see if the product of the three greatest numbers in the array is greater than the product of the two lowest numbers and the highest number. Here, if the number of elements is less than 3 in the array, we cannot find the maximum product among them, so if the number of elements is less than 3 we return an exception error.

This suggestion, with the help of the inbuilt GitHub copilot in Visual Studio, I was able to come up with the above code. The time complexity for this approach is O(n log n), because we are sorting the array first by iterating through all the elements of the array(O(n)) and then finding the maximum product of the last three elements (O(log n)). The space complexity is the optimal O(1) here because we are not using any extra space to store the numbers.

**INPUT:**

Console.WriteLine("Question 4:");

int[] nums4 = { 1, 2, 3, 4 };

int maxProduct = MaximumProduct(nums4);

**OUTPUT:**

Question 4:

24

### 5. Decimal to Binary Conversion:

**Description:**  
Write a function that converts a decimal number to its binary equivalent.

**Examples:**

* Input: 42, Output: 101010
* Input: 10, Output: 1010

**CODE:**

**public static string DecimalToBinary(int decimalNumber)**

**{**

**try**

**{**

**if (decimalNumber < 0)**

**{**

**throw new ArgumentException();**

**}**

**// Using the built-in Convert.ToString method to convert the decimal number to binary**

**return Convert.ToString(decimalNumber, 2);**

**}**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial thought process:**

My initial idea was to manually convert the decimal number to binary using repeated division by 2 and tracking the remainders. While this approach would work, I found it is very difficult to implement and prone to mistakes, especially when handling edge cases like zero or very large numbers. Instead of implementing this logic from scratch, I wondered if there was a more optimised way to achieve the same using an inbuilt methods.

**Copilot Prompt:** How do I convert a decimal number to a binary string?

**Recommendation:** Copilot recommended to use the in-built method Convert.ToString(decnum,2) which directly converts an integer to its binary representation as a string. The AI suggested adding a validation check to ensure the number is non-negative and throwing an exception otherwise.

This suggestion, with the help of the inbuilt GitHub copilot in Visual Studio, I was able to come up with the above code. My initial process of repeated divisor would have had the time complexity of O(log n) but the above code has a space complexity of O(1) since only a single function is called. The space complexity of the optimal code given above is O(1).

**INPUT:**

Console.WriteLine("Question 5:");

int decimalNumber = 142;

string binary = DecimalToBinary(decimalNumber);

**OUTPUT:**

Question 5:

10001110

### 6. Find Minimum in Rotated Sorted Array:

**Description:**  
Given a sorted array that has been rotated, find the minimum element.

**Examples:**

* Input: [3, 4, 5, 1, 2], Output: 1
* Input: [4, 5, 6, 7, 0, 1, 2], Output: 0

**CODE:**

**public static int FindMin(int[] nums)**

**{**

**try**

**{**

**// Check if the array is null or empty**

**if (nums == null || nums.Length == 0)**

**{**

**throw new ArgumentException("Array must not be null or empty.");**

**}**

**int left = 0, right = nums.Length - 1;**

**// Binary search to find the minimum in rotated sorted array**

**while (left < right)**

**{**

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

**// If middle element is greater than the rightmost, min is in right half**

**if (nums[mid] > nums[right])**

**{**

**left = mid + 1;**

**}**

**else**

**{**

**// Otherwise, it is in the left half (including mid)**

**right = mid;**

**}**

**}**

**// At the end of the loop, left == right, pointing to the minimum element**

**return nums[left];**

**}**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial thought Process:** My initial idea was to linearly traverse through the entire array and compare each element to find the minimum value. but this has a time complexity of O(n) which is not optimal, especially when the input array is large. Since the array is a rotated sorted array, I realized that a more efficient approach should leverage binarysearch, which is commonly used for sorted arrays.

**Copilot Prompt:** How to implement a binary search algorithm to find the minimum in a rotated sorted array?

**Suggestion:** The key idea is to compare the middle element with the rightmost element. If nums[mid] > nums[right], then the minimum must lie in the right half, If nums[mid] <= nums[right], the minimum must lie in the left half. Possibly at the mid too. The loop continues until the left and right converge on the smallest element.

This suggestion, with the help of the inbuilt GitHub copilot in Visual Studio, I was able to come up with the above code. First, I check for null or empty array inputs. Then I calculate the midpoint to avoid overflow. Then return the element at the converged index, which is the minimum. The time complexity of the code has improved from O(n) for my initial idea, to O(log n) for the optimal code. The space complexity is O(1).

**INPUT:**

Console.WriteLine("Question 6:");

int[] nums5 = { 3, 4, 5, 1, 2 };

int minElement = FindMin(nums5);

**OUTPUT:**

Question 6:

1

### Question 7: Palindrome Number

**Description:**  
Given an integer x, return true if x is a palindrome, and false otherwise.

A palindrome is a number that reads the same forward and backward.

**Examples:**

* Input: 121, Output: true
* Input: 10, Output: false (Explanation: Reads 01 from right to left. Therefore, it is not a palindrome.)

**CODE:**

**public static bool IsPalindrome(int x)**

**{**

**try**

**{**

**// Negative numbers are not palindromes**

**if (x < 0)**

**{**

**return false;**

**}**

**int original = x; // Storing the original number**

**int reversed = 0; // Variable to hold the reversed number**

**// Reversing the number**

**while (x > 0)**

**{**

**int digit = x % 10; // Getting the last digit**

**reversed = reversed \* 10 + digit; // Appending the digit to reversed**

**x /= 10; // Removing the last digit from x**

**}**

**// Comparing the original and reversed number**

**return original == reversed;**

**}**

**// If any exception occurs, rethrow it**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial Thought process:**

My initial idea was straightforward, to reverse the given number and check if the reversed number is equal to the original. If they are equal, then the number is a palindrome. Before that, I also considered the fact that negative numbers can never be palindromes, since the negative sign - is not mirrored. So, if the input number is negative, the function should immediately return false. To reverse the number, I planned to extract each digit using modulo 10, build the reversed number by multiplying the result so far by 10, and then strip the last digit of the original number by integer division.

**Copilot Prompt:**  How to check if an integer is a palindrome without converting it to a string?

**Suggestion:** Copilot confirmed my approach by suggesting the exact same idea — reverse the number mathematically rather than using string conversion. The suggested logic was to first initialise a variable reversed = 0, use a loop to extract digit and build the reverse number. After the loop, we compare number to the original input.

This suggestion, with the help of the inbuilt GitHub copilot in Visual Studio, I was able to come up with the above code. First, I check for negative values and return false for them. Then initialise a reverse variable to zero and then iterate to reverse the original input number to save it at reversed digit. Then compare that with the original and return true if same and false if it is not. The time complexity of the solution is O(log n). The space complexity of the solution is O(1).

**INPUT:**

Console.WriteLine("Question 7:");

int palindromeNumber = 121;

bool isPalindrome = IsPalindrome(palindromeNumber);

**OUTPUT:**

Question 7:

True

### 8. Question 8: Fibonacci Number

**Description:**  
The Fibonacci numbers, commonly denoted F(n), form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

* F(0) = 0, F(1) = 1
* F(n) = F(n - 1) + F(n - 2), for n > 1

Given n, calculate F(n).

**Examples:**

* Input: 2, Output: 1
* Input: 3, Output: 2
* Input: 4, Output: 3

**Constraints:**

* 0 <= n <= 30

**CODE:**

**public static int Fibonacci(int n)**

**{**

**try**

**{**

**if (n < 0)**

**{**

**throw new ArgumentException("Input must be a non-negative integer.");**

**}**

**if (n == 0)**

**{**

**return 0;**

**}**

**else if (n == 1)**

**{**

**return 1;**

**}**

**// Using an iterative approach to calculate Fibonacci**

**int a = 0, b = 1, c = 0;**

**for (int i = 2; i <= n; i++)**

**{**

**c = a + b; // Calculate the next Fibonacci number**

**a = b; // Update a to the previous Fibonacci number**

**b = c; // Update b to the current Fibonacci number**

**}**

**// Return the nth Fibonacci number**

**return b; // Placeholder**

**}**

**catch (Exception)**

**{**

**throw;**

**}**

**}**

**Initial Thought Process:** My initial idea was to calculate each Fibonacci number up to n and return the result. I considered using a recursive approach, but I realized that recursion would have a high time complexity and could cause stack overflow for large n. So, I shifted to an iterative solution, which is more efficient.  
First, I included a base condition to check if the input is non-negative, because Fibonacci is only defined for non-negative integers. Then I handled the base cases, F(0) =0 and F(1)=1. For n>=2, I planned to use two variables to track the last two Fibonacci numbers and update them in a loop until I reached the nth number.

**Copilot Prompt:** How to calculate the nth Fibonacci using iteration?

**Suggestion:** The suggestion was to use a loop with two variables – a and b to store the last two Fibonacci numbers. And then in each iteration compute c = a + b and then update a=b and b=c.

This suggestion, with the help of the inbuilt GitHub copilot in Visual Studio, I was able to come up with the above code. First I checked for invalid input which are negative values. Then I Handled the bases cases of F(0) = 0 and F(1) = 1. The used an iterative loop starting from i=2 to compute each Fibonacci number using the previous two. Returned the final value as the result of the nth Fibonacci Number. The time complexity of this solution is O(n) because we are using an iterative approach to calculate the Fibonacci values. The space complexity is O(1) because we are not using extra space.

**INPUT:**

Console.WriteLine("Question 8:");

int n = 4;

int fibonacciNumber = Fibonacci(n);

**OUTPUT:**

Question 8:

3

INPUT(‘MAIN’ FUNCTION) OF THE FOLLOWING PROGRAM.cs FILE:

*static void Main(string[] args)*

*{*

*// Question 1: Find Missing Numbers in Array*

*Console.WriteLine("Question 1:");*

*int[] nums1 = { 4, 3, 2, 7, 8, 2, 3, 1 };*

*IList<int> missingNumbers = FindMissingNumbers(nums1);*

*Console.WriteLine(string.Join(",", missingNumbers));*

*// Question 2: Sort Array by Parity*

*Console.WriteLine("Question 2:");*

*int[] nums2 = { 3, 1, 2, 4 };*

*int[] sortedArray = SortArrayByParity(nums2);*

*Console.WriteLine(string.Join(",", sortedArray));*

*// Question 3: Two Sum*

*Console.WriteLine("Question 3:");*

*int[] nums3 = { 2, 7, 11, 15 };*

*int target = 9;*

*int[] indices = TwoSum(nums3, target);*

*Console.WriteLine(string.Join(",", indices));*

*// Question 4: Find Maximum Product of Three Numbers*

*Console.WriteLine("Question 4:");*

*int[] nums4 = { 1, 2, 3, 4 };*

*int maxProduct = MaximumProduct(nums4);*

*Console.WriteLine(maxProduct);*

*// Question 5: Decimal to Binary Conversion*

*Console.WriteLine("Question 5:");*

*int decimalNumber = 142;*

*string binary = DecimalToBinary(decimalNumber);*

*Console.WriteLine(binary);*

*// Question 6: Find Minimum in Rotated Sorted Array*

*Console.WriteLine("Question 6:");*

*int[] nums5 = { 3, 4, 5, 1, 2 };*

*int minElement = FindMin(nums5);*

*Console.WriteLine(minElement);*

*// Question 7: Palindrome Number*

*Console.WriteLine("Question 7:");*

*int palindromeNumber = 121;*

*bool isPalindrome = IsPalindrome(palindromeNumber);*

*Console.WriteLine(isPalindrome);*

*// Question 8: Fibonacci Number*

*Console.WriteLine("Question 8:");*

*int n = 4;*

*int fibonacciNumber = Fibonacci(n);*

*Console.WriteLine(fibonacciNumber);*

*}*

OUTPUT OF THE program.cs FILE:

A computer screen with white text

AI-generated content may be incorrect.

LINK TO THE PUBLIC GITHUB REPOSITORY:

<https://github.com/hridayreddyp/ISM6225_Fall24_Assignment_2.git>

\*\* In the above Document when I mention “Initial Thought Process”, this was before I looked at AI help and wrote down the steps and ideas that I have gotten, after which I have confirmed with AI to check if my logic is correct and optimal. I have written down the most optimal code for each of the following and discarded my brute force approach code where necessary.