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

Assignment 2 – Computational Problem Solving – U14273550

### 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]

Initial Code:

IList<int> missingNumbers = new List<int>();

for (int i =1; i <= nums.Length; i++)

{

if(nums.Contains(i) == false)

{

missingNumbers.Add(i);

}

}

return missingNumbers;

**AI Prompt:** What is the time and space complexity of this code, and could it be improved?

**AI Responses:** Copilot responded with the modified code with improved time and space complexity.

**Time Complexity**

* The outer loop runs n times, where n is the length of the array.
* The nums.Contains(i) operation runs in O(n) time in the worst case because it needs to check each element in the array. Therefore, the overall time complexity is O(n^2).

**Space Complexity**

* The space complexity is O(n) for storing the missingNumbers list.

**Improved Implementation**

We can improve the time complexity by using a HashSet to store the elements of the array. Checking for the existence of an element in a HashSet is O(1) on average.

**Improved Time Complexity**

* The outer loop still runs n times.
* The numSet.Contains(i) operation runs in O(1) time on average. Therefore, the overall time complexity is improved to O(n).

**Improved Space Complexity**

* The space complexity remains O(n) for storing the numSet and missingNumbers list. By using a HashSet, we have significantly improved the time complexity from O(n^2) to O(n).

### 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]

Initial Code

int[] evenNumbers = Array.FindAll(nums, n => n % 2 == 0);

int[] oddNumbers = Array.FindAll(nums, n => n % 2 != 0);

int[] sortedArray = new int[nums.Length];

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

{

sortedArray[i] = evenNumbers[i];

}

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

{

sortedArray[evenNumbers.Length + i] = oddNumbers[i];

}

return sortedArray;

**AI Prompt**: could this code be improved based on time and space complexity.

**AI Response**: Yes, the SortArrayByParity method can be improved in terms of both time and space complexity. The current implementation uses additional arrays to store even and odd numbers separately, which increases space complexity. We can achieve the same result with a single pass through the array and in-place swapping, which will improve both time and space complexity.

**Two-pointer approach**: We use two pointers, left starting from the beginning and right starting from the end of the array.

1. **Condition check and swap**: If the element at left is odd and the element at right is even, we swap them.
2. **Move pointers**: If the element at left is even, we move the left pointer to the right. If the element at right is odd, we move the right pointer to the left.
3. **Return the modified array**: The array is sorted by parity in-place.

This approach ensures that the array is sorted by parity with a single pass through the array, resulting in O(n) time complexity and O(1) space complexity.

int[] sortedArray = new int[nums.Length];

int evenIndex = 0;

int oddIndex = nums.Length - 1;

foreach (int num in nums)

{

if (num % 2 == 0)

{

sortedArray[evenIndex++] = num;

}

else

{

sortedArray[oddIndex--] = num;

}

}

return sortedArray;

Thoughts:

### 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]

**AI prompt: how can this code be improved based on time and space complexity.**

**AI Response:** Yes, the TwoSum method can be improved in terms of time complexity. The current implementation has a time complexity of O(n^2) due to the nested loops. We can improve it to O(n) by using a dictionary to store the numbers and their indices as we iterate through the array. This way, we can check if the complement (target - current number) exists in the dictionary in constant time.

**Explanation:**

**Dictionary Usage:** We use a dictionary (numDict) to store each number and its index as we iterate through the array**.**

**Complement Check:** For each number, we calculate its complement (target - nums[i]) and check if it exists in the dictionary.

**Return Indices:** If the complement exists, we return the indices of the complement and the current number.

**Add to Dictionary:** If the complement does not exist, we add the current number and its index to the dictionary.

**Exception Handling:** If no solution is found, we throw an exception.

This approach reduces the time complexity to O(n) and uses O(n) additional space for the dictionary.

### 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

Initial Code:

int maxProduct = 0;

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

{

for (int j = i + 1; j < nums.Length; j++)

{

for (int k = j + 1; k < nums.Length; k++)

{

int product = nums[i] \* nums[j] \* nums[k];

if (product > maxProduct)

{

maxProduct = product;

}

}

}

}

return maxProduct;

**AI Input:** What is the time and space complexity of this code and how can it be improved also check if all the corner cases are handled?

**AI Response:** The current implementation of the MaximumProduct method has a time complexity of O(n^3) and a space complexity of O(1). Let's break down why this is the case and how we can improve it.

Time Complexity

The method uses three nested loops to iterate through all possible triplets in the array. Each loop runs n times, where n is the length of the array. Therefore, the time complexity is O(n^3).

Space Complexity

The space complexity is O(1) because the method uses a constant amount of extra space regardless of the input size.

Corner Cases

1. Array Length Less Than 3: The current implementation does not handle arrays with fewer than three elements, which would result in an out-of-bounds error.
2. Negative Numbers: The current implementation does not consider the possibility of negative numbers, which could affect the maximum product.

Improved Solution

We can improve the time complexity to O(n log n) by sorting the array and then considering the maximum product of the last three elements or the product of the first two (most negative) and the last element (most positive).

Time Complexity of Improved Solution

Sorting the array takes O(n log n).

Calculating the products and comparing them takes O(1).

Therefore, the overall time complexity is O(n log n).

Space Complexity of Improved Solution

The space complexity remains O(1) as we are not using any additional space that scales with the input size.

Corner Cases Handled

Array Length Less Than 3: The improved code checks if the array has fewer than three elements and throws an exception.

Negative Numbers: The improved code considers the possibility of negative numbers by comparing the product of the last three elements and the product of the first two elements with the last element.

This improved solution is more efficient and handles all the corner cases effectively.

### 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.)

**AI Response:** Time and Space Complexity

Time Complexity

The Is Palindrome method converts the integer x to a string and then checks if the string is a palindrome. The time complexity can be broken down as follows:

1. Conversion to String: Converting an integer to a string takes O(d) time, where d is the number of digits in the integer.
2. Palindrome Check: The while loop runs for approximately d/2 iterations, where d is the number of digits. Each iteration involves a constant-time comparison.

Overall, the time complexity is O(d), where d is the number of digits in the integer.

Space Complexity

The space complexity is also O(d) due to the space required to store the string representation of the integer.

Handling Corner Cases

The method handles most typical cases, but there are a few corner cases to consider:

1. Negative Numbers: The current implementation does not explicitly handle negative numbers. By definition, negative numbers are not palindromes because of the negative sign.
2. Single Digit Numbers: Single digit numbers are palindromes by default, and the current implementation handles this correctly.
3. Zero: Zero is a palindrome, and the current implementation handles this correctly.

Suggested Improvements

To handle negative numbers explicitly, you can add a check at the beginning of the method.

**Thought on Using Copilot:**

Initially, even before writing code snippet, AI is prompting the entire code which does not help for someone who is trying to learn coding. However, it helps a lot for someone who knows how to write code but needs to write a better feasible code with considering time and space complexity.

Final Code Output:

A computer screen shot of a black screen

AI-generated content may be incorrect.

Git Hub Link: [ravitejamodupalli/ISM6225\_Fall24\_Assignment\_2](https://github.com/ravitejamodupalli/ISM6225_Fall24_Assignment_2)