# Assignment 2 - Computational problem solving

*Prompt:* **How to find missing numbers in an array of 1 to n in C#?**

*Response*

To find missing numbers in an array of 1 to n in C#, you can use a HashSet to track the numbers that are present in the array and then iterate from 1 to n to find the missing numbers.

**Implementation:** I followed the suggestion and implemented the logic using HashSet<int> seen = new HashSet<int>(nums) and looped from 1 to n.

**Adjustments:** I added a result list and ensured the loop was i <= nums.Length. I also handled edge cases such as duplicate numbers and an empty array.

*Prompt:***How to implement Two Sum in C# using a dictionary for O(n) time complexity?**

*Response*

* To implement the Two Sum problem in C# using a dictionary for O(n) time complexity
* Use a dictionary to store the numbers and their indices as you iterate through the array.
* For each number, calculate its complement (i.e., target - current number).
* If the complement exists in the dictionary, you have found the two numbers that add up to the target.
* If the complement is not found, store the current number and its index in the dictionary.

**Implementation:**  
I followed the Copilot-recommended approach exactly. I initialized a Dictionary<int, int> to store the values and their corresponding indices. While iterating through the array, I calculated the complement (target - current number) and checked if it already existed in the dictionary. If found, I returned the current index and the stored index for the complement. This ensures an O(n) solution as each number is processed only once.

**Adjustments:**

* I added a condition to avoid inserting duplicate keys into the dictionary (although it wouldn't break logic, it keeps the implementation clean).
* I returned an empty array if no valid pair was found, to safely handle cases where no solution exists.
* I included detailed inline comments and edge case explanations to improve readability and maintainability.

*Prompt:* **what are the edge cases for maximum product?**

*Response*

1. Array with Exactly Three Elements:

• The array has exactly three elements, so the product is simply the product of these three numbers.

2. Array with All Positive Numbers:

• The maximum product will be the product of the three largest numbers.

3. Array with All Negative Numbers:

• The maximum product will be the product of the three least negative (i.e., largest in absolute value) numbers.

4. Array with Both Positive and Negative Numbers:

• The maximum product can be either:

• The product of the three largest numbers.

• The product of the two smallest numbers (which could be negative, resulting in a positive product) and the largest number.

5. Array with Zeroes:

• If the array contains zeroes, they can affect the product, especially if the array has negative numbers. Zeroes can reduce the product to zero if included.

7. Large Numbers:

• Ensure that the product does not overflow. In C#, the int type can handle up to 2,147,483,647, but if the product exceeds this, consider using long for larger ranges.

### ****Implementation :****

I followed the approach Copilot suggested, which involved **sorting the array first** to access both:

the three largest values (at the end), and the two smallest (most negative) values at the beginning.

The final product was determined by taking the **maximum** between:

the product of the three largest numbers, and the product of the two smallest numbers and the largest number.

### ****Adjustments:****

### The logic naturally supports this case after sorting. No special handling is needed.

* For positive numbers the Sorting guarantees the top three are at the end of the array, so product1 will be correct.
* For negative numbers The product of the three least negative numbers (end of the array) is correctly selected via product1.
* No special code needed, but sorting ensures zeros are grouped near the start, naturally handled in product2.

*Prompt:* **Find the index of the minimum element in a rotated array using binary search?**

*Response*

1. Initialize Pointers: Use two pointers, left and right, to represent the range of the array.

2. Binary Search Loop: Use a while loop to perform binary search until left is less than right.

3. Calculate Midpoint: Calculate the midpoint of the current range.

4. Compare Midpoint with Right: Compare the element at the midpoint with the element at the right pointer to determine which half of the array contains the minimum element.

5. Adjust Pointers: Adjust the left or right pointer based on the comparison.

6. Return Index: When the loop ends, left will be the index of the minimum element.

**Implementation :**

I implemented a binary search algorithm using two pointers, left and right, to narrow down the range. In each iteration, I calculated the midpoint and compared it with the rightmost element. If the middle value was greater than the right value, the minimum was in the right half; otherwise, it was in the left half (including mid). When the loop ended, left and right converged on the index of the smallest element. This approach ensures an efficient O(log n) solution.

**Adjustments:**

* I assumed that the array contains no duplicate elements, which aligns with the standard constraint in problems like this.
* The logic safely handles fully sorted arrays (i.e., not rotated), where the first element is already the minimum.
* It also works for arrays rotated at any position, including where the smallest element is at the end or beginning.
* For arrays with only one element, it returns that element correctly without errors.
* I added comments to clearly explain each step of the binary search process for readability and maintainability.