**Assignment 2 – Computational problem solving**

**Problem 1 – Find Missing Numbers in Array**

**Ques:** How do I find all missing numbers in an array that contains numbers from 1 to n, possibly with duplicates?

**Ans:** You can use a boolean array of size n+1 to mark all numbers that appear in the input. Then loop from 1 to n and collect the numbers that weren’t marked as seen.

**Code:**

bool[] seen = new bool[nums.Length + 1];

foreach (int num in nums) {

if (num >= 1 && num <= nums.Length)

seen[num] = true;

}

List<int> result = new List<int>();

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

if (!seen[i]) result.Add(i);

}

return result;

**Ques:** What if the array has invalid numbers like 0 or n+2?

**Ans:** That’s a good edge case. The check if (num >= 1 && num <= nums.Length) ensures such values are ignored. You don’t want to mark out-of-bounds values.

**Ques:** One more thing—what if the array is empty?

**Ans:** In that case, you’ll get an empty result list since there are no numbers from 1 to n. It doesn’t throw errors because the loop won't run.

**Problem 2 – Sort Array by Parity**

**Ques:** How can I sort an array so that all even numbers come before all odd numbers in C#?

**Ans:** you can loop through the array twice:

1. First, add even numbers to a result list
2. Then, add odd numbers.  
   This maintains the relative order of even and odd numbers separately.

**Code:**

List<int> result = new List<int>();

foreach (int num in nums) {

if (num % 2 == 0)

result.Add(num);

}

foreach (int num in nums) {

if (num % 2 != 0)

result.Add(num);

}

return result.ToArray();

**Ques:** Yeah, this looks simple and clean. Does it guarantee stable ordering?

**Ans:** Yes, this approach preserves the order of appearance within even and odd groups. So [3,1,2,4] becomes [2,4,3,1].

**Ques:** Would a two-pointer method be better?

**Ans:** You could use two pointers (start and end) and build the array in-place if performance or space is critical. But for now, this solution is perfectly valid and readable.

**Problem 3 – Two Sum  
Ques:** How do I implement the Two Sum problem in C# using a dictionary?

**Ans:** Use a hash map (dictionary) to store numbers as you iterate. For each element, check if its complement (i.e., target - current number) already exists in the map. If yes, return the indices.

**Code:**

Dictionary<int, int> map = new Dictionary<int, int>();

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

int complement = target - nums[i];

if (map.ContainsKey(complement)) {

return new int[] { map[complement], i };

}

map[nums[i]] = i;

}

**Ques:** This looks good. But what if the same number appears twice?

**Ans:** Good question. You should only add a number to the map after checking for its complement. That way, you won't mistakenly match the same element with itself.

**Ques:** What happens if there's no solution?

**Ans:** You can return an empty array (new int[0]) to handle such cases safely.

**Problem 4 – Find Maximum Product of Three Numbers**

**Ques:** How can I find the maximum product of any three numbers in an array in C#?

**Ans:** The trick here is to sort the array first. Then check:

1. The product of the last three numbers (largest positives)
2. The product of the two smallest (possibly negative) numbers and the largest number

Return the maximum of the two.

**Code:**

Array.Sort(nums);

int n = nums.Length;

int product1 = nums[n - 1] \* nums[n - 2] \* nums[n - 3];

int product2 = nums[0] \* nums[1] \* nums[n - 1];

return Math.Max(product1, product2);

**Ques:** Ah okay, so this handles negative values too — like [-10, -10, 5, 2]?

**Ans:** Exactly. The two smallest values might be negative, and their product can be a large positive number when multiplied with the biggest number.

Makes sense. I used Array.Sort() and implemented it exactly this way. Added a try-catch like all other methods.

**Ques:** Should I validate input size?

**Ans:**  You could, but since the method assumes a minimum of 3 elements (as per problem description), it's okay to skip that check unless you're making the function more general.

**Problem 5 – Decimal to Binary Conversion**

**Ques:** What’s the easiest way to convert a decimal number to binary in C#?

**Ans:**  
The simplest way is to use the built-in method:

**Code:**

Convert.ToString(decimalNumber, 2);

This converts the integer to its binary representation as a string.

**Ques:** Oh that’s clean. Do I need to handle negative numbers?

**Ans:** C# will give you a two’s complement binary string for negative numbers. But if the use case is positive integers only, you're fine. You could always validate input if needed.

Okay, in my assignment the sample input is 42, so I’m assuming positives. I’ll use that method as-is. Added try-catch too.

**Ques:** Just curious — how would I do it manually?

**Ans:** You can repeatedly divide the number by 2 and build the binary string from remainders. But for now, built-in is perfect and more reliable.

**Problem 6 – Find Minimum in Rotated Sorted Array**

**Ques:** How can I find the minimum element in a rotated sorted array in C#?

**Ans:** You can use a binary search approach. At each step, compare the middle element with the rightmost one:

* If nums[mid] > nums[right], the minimum must be on the right.
* Otherwise, it’s on the left (including mid).

**Code:**

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

while (left < right) {

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

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

left = mid + 1;

else

right = mid;

}

return nums[left];

This makes sense. I added this directly into the method. No changes needed.

**Ques:** What if the array is not rotated at all?

**Ans:** Good catch. This logic still works — it will return the first element since nums[mid] <= nums[right] will always hold true.

Cool. Also added standard error handling with try-catch.

**Ques:** Is there a case where binary search fails?

**Ans:** Only if the array contains duplicates — but that’s a different version of the problem. For distinct values, this is optimal at O(log n) time.

**Problem 7 – Palindrome Number**

**Ques:** How can I check if a number is a palindrome in C#?

**Ans:** You can reverse the number digit by digit and compare it with the original value.

**Code:**

if (x < 0) return false;

int original = x;

int reversed = 0;

while (x != 0) {

int digit = x % 10;

reversed = reversed \* 10 + digit;

x /= 10;

}

return original == reversed;

**Ques:** Why do we return false for negative numbers?

**Ans:** Because negative numbers aren’t considered palindromes — the - sign would only appear on one side. For example, -121 ≠ 121-.

Got it. I implemented the exact logic and wrapped it in try-catch. It's clean and works even for numbers like 0 and 12321.

**Ques:** Is there a string-based way to do this?

**Ans:** Yes! You could convert the number to a string and compare characters from start and end. But reversing digits numerically is more efficient and avoids unnecessary conversions.

**Problem 8 – Fibonacci Number**

**Ques:** How can I compute the nth Fibonacci number in C# without using recursion?

**Ans:** You can use an iterative approach with two variables to avoid the overhead of recursion. **Code:**

if (n <= 1) return n;

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

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

c = a + b;

a = b;

b = c;

}

return b;

**Ques:** Makes sense. So a and b hold the last two Fibonacci values, and c is the current one?

**Ans:** Exactly. It’s a rolling update pattern. This method is both time and space efficient, running in O(n) time and O(1) space.

Alright, I used this logic as-is. Wrapped it in try-catch and tested with n = 4 and n = 0. Works well.

**Ques:** Would recursion be better in any case?

**Ans:** Not really — recursion without memorization has exponential time complexity. For large n, it’s inefficient and can cause stack overflows. Iteration is the way to go here.