Question1:

Prompt: Could this be implemented in a better way to improve time and space complexity?

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

{

if (nums[i] % 2 == 0) //checking if num is even

{

result[left] = nums[i]; // if even, append to left of array

left++;

} else

{ result[right] = nums[i]; // if odd, append to the right

right--; } }

Response:

Your logic works conceptually, but you're not truly doing it *in-place*. You are using a separate array result, which means you're using **O(n)** extra space. The question specifically asks to do it *in-place*, which means modifying the original nums array without allocating another array.

public void SortArrayByParity(int[] nums)

{

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

while (left < right)

{

// If left is even, just move forward

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

{

left++;

}

// If right is odd, just move backward

else if (nums[right] % 2 == 1)

{

right--;

}

// Swap odd on left with even on right

else

{

int temp = nums[left];

nums[left] = nums[right];

nums[right] = temp;

left++;

right--;

}

}

}

Question 2:

Prompt: Does it covers all the edge cases?

The SortArrayByParity method does not cover all edge cases. Specifically, it lacks handling for null or empty arrays, and it does not initialize the **left** and **right** pointers. Here is an improved version of the method that addresses these issues:

**Changes made:**

1. **Null or Empty Array Check**: Added a check to return an empty array if the input is null or empty.
2. **Initialize Pointers**: Initialized **left** to 0 and **right** to **nums.Length - 1**.

This ensures that the method handles edge cases properly and functions as expected.

Question 3:  
Prompt: Update the code covering all possible edge cases.

Responses:

try

{

// Edge Case: Null or insufficient length

if (nums == null || nums.Length < 2)

return new int[0];

// Use dictionary to store previously seen numbers and their indices

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

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

{

int complement = target - nums[i];

// Check if the complement is already seen

if (mapping.ContainsKey(complement))

{

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

}

// Store current number and its index

mapping[nums[i]] = i;

}

return new int[0]; // No solution found

}

catch (Exception)

{

throw;

}

Question 4: Below is my logic to get find maximum product of three numbers. Modify the code for the best time and space complexity.

// Write your code here

int[] ThreeNums = new int[3];

ThreeNums[0] = nums[0];

ThreeNums[1] = nums[1];

ThreeNums[2] = nums[2];

for ( int i = 3; i < nums.Length; i++)

{

if (nums[i] > ThreeNums.Min())

{

ThreeNums[Array.IndexOf(ThreeNums, ThreeNums.Min())] = nums[i];

}

}

return ThreeNums[0] \* ThreeNums[1] \* ThreeNums[2];

public static int MaximumProduct(int[] nums)

{

try

{

// Edge Case: Less than 3 elements

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

throw new ArgumentException("At least 3 numbers required");

Array.Sort(nums); // Sort array to access smallest/largest values

int n = nums.Length;

// Two candidates: 3 largest numbers, or 2 smallest and largest

return Math.Max(nums[n - 1] \* nums[n - 2] \* nums[n - 3],

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

}

catch (Exception)

{

throw;

}

}

Question 5:

Prompt: Find out possible edge cases and resolve.

Response:

try

{

// Edge Case: 0 input

if (decimalNumber == 0)

return "0";

// Edge Case: Negative number

if (decimalNumber < 0)

throw new ArgumentException("Input must be non-negative");

string binary = "";

// Convert decimal to binary by repeated division

while (decimalNumber > 0)

{

binary = (decimalNumber % 2) + binary;

decimalNumber /= 2;

}

return binary;

}

catch (Exception)

{

throw;

}

Question 6:  
Prompt: I'm trying to implement a function in C# that finds the minimum element in a rotated sorted array (with no duplicates). I want to use binary search for efficiency. Can you help me write and explain a well-structured implementation with error handling?

public static int FindMin(int[] nums)

{

try

{

int left = 0;

int right = nums.Length - 1;

// Binary search to find the smallest element

while (left < right)

{

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

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

{

// Minimum must be on the right

left = mid + 1;

}

else

{

// Minimum is at mid or to the left

right = mid;

}

}

return nums[left];

}

catch (Exception)

{

throw;

}

}

Question 7:  
Prompt: I’m trying to implement a function in C# to check if a number is a palindrome without converting it to a string. I’ve written some code, but I’m not sure if reversing the number using math operations is the right approach. Can you explain how that logic works and what to watch out for?

**Response:**

Yes, reversing a number using math operations is a common and valid approach. Here’s how it works:

* You extract the last digit of the number using x % 10.
* You build the reversed number by multiplying the current reversed value by 10 and adding the digit.
* Then, remove the last digit from the original number using integer division (x /= 10).

Repeat this until x becomes 0. Finally, compare the reversed number to the original to check for a palindrome.

Question 8:  
Prompt:

I'm trying to write an efficient C# function to calculate the nth Fibonacci number without using recursion. I used a loop with a few variables to store the previous terms. Can you explain if this approach is efficient and commonly used?

**Response:**

Yes, using an iterative approach with variables to track previous terms is both efficient and commonly used for calculating Fibonacci numbers.

**Explanation**:

* This method avoids recursion and stack overflow issues.
* Time complexity is **O(n)** and space complexity is **O(1)**.
* It uses only three variables to track current and previous values, making it optimal for most practical use cases.