**QUESTION 1**  
**Prompt** : How can I find missing numbers in an array of size n where numbers are in the range [1, n] with duplicates?

**Response** : You can iterate through the array and mark indices as visited by negating the number at the index (value - 1). Then the positive indices are the missing ones.

**Implementation Details:**

To find the missing numbers in an array where the numbers range from 1 to *n* and may include duplicates, I used a clever in-place marking strategy. I looped through the array and for each number, I treated its absolute value minus one as an index and marked the value at that index as negative (if it wasn’t already). This allowed me to use the array itself to track which numbers were present. After marking, I did a second loop through the array: any index with a positive number meant the number corresponding to that index (index + 1) was missing, so I added it to the result list.

This solution uses **O(n)** time and **O(1)** extra space (excluding the output list), meeting the efficiency requirements.

**Edge cases**// Edge Case 1: Empty array → return empty list

// Covered implicitly: the loops won't run, and result will be empty.

// Edge Case 2: All numbers present (no missing numbers)

// Example: [1, 2, 3, 4] → expected result: []

// Covered by marking and checking signs at end

// Edge Case 3: All numbers missing (e.g., array filled with duplicates of one number)

// Example: [2, 2, 2, 2] → expected result: [1, 3, 4]

// Covered by marking logic that allows duplicate detection

// Edge Case 4: Duplicates are present (e.g., [4,3,2,7,8,2,3,1])

// Must still detect missing numbers correctly

// We use Math.Abs to avoid double-negating previously marked values

// Edge Case 5: Array contains maximum allowed values (e.g., nums[i] = n)

// Handled by indexing at nums[i] - 1 which safely maps [1, n] to [0, n-1]

**QUESTION 2**

**Prompt**: How to sort an integer array so that all even numbers come before all odd numbers?

**Response**: You can loop through the array and put even numbers in one list and odd numbers in another, then combine them.

**Implementation details :**

To solve the problem of sorting an array by parity (moving all even numbers to the beginning), I used a two-pointer approach. I maintained a pointer called left to keep track of where the next even number should be placed. Then I iterated through the array once, and whenever I encountered an even number, I swapped it with the number at the left pointer and then moved the pointer forward. This way, even numbers are brought to the front in-place, while odd numbers automatically shift to the end as we process the array.

This approach has a time complexity of **O(n)** and performs sorting in-place with **O(1)** extra space, which is optimal for this task.

**Edge Cases Identified:**

* **Empty array** → returns empty array without error.
* **Single element array** → directly returned.
* **All even numbers** → remains unchanged.
* **All odd numbers** → order changes but result is still correct.
* **Mixed or unordered input** → grouped by parity correctly, no need to preserve relative order.

**Adjustments Made:**

* Checked for empty/single element arrays at the start to return early.
* Used two separate loops: first for even, then for odd, to enforce ordering.
* Used List<int> to dynamically build result and then convert to array.

**QUESTION3**

**Prompt** : How can I find indices of two numbers in an array that add up to a given target value?

**Response :** Use two nested for loops to iterate through the array. In the inner loop, check if the sum of the two numbers at positions i and j equals the target. If they do, return their indices.

**Prompt** : Can you show me how to handle edge cases like negative numbers, duplicates, or arrays with less than 2 elements?

**Response :** The solution will handle negative numbers by simply comparing their sum to the target. For duplicates, the solution will automatically skip the same indices since j starts at i+1. For arrays with fewer than 2 elements, you should return an empty array at the beginning.

**Implementation Details:**

I chose to implement the brute-force method as shown below:

* Iterate through the array using a nested for loop.
* For each pair (i, j) such that i < j, check if nums[i] + nums[j] == target.
* If so, return the indices [i, j].

**Adjustments:**

* Array Length Check: I modified the code to check if the array has fewer than 2 elements and return an empty array in such cases. This was added as a safeguard.
* Return Empty Array if No Solution: I ensured the code returns an empty array if no valid pair is found, even though the problem guarantees that a solution exists. This is just for completeness and to prevent errors in edge cases where input might deviate.

**Edge cases:**

1.Less than 2 elements  
*No pair possible*

Example: nums = [5], target = 5 → Output: []

2.Includes negative numbers  
 *Still works as addition is valid for negatives*  
 Example: nums = [-3, 4, 1], target = 1 → Output: [0, 1]

3.Multiple valid pairs  
 *Returns the first one it finds*  
 Example: nums = [1, 2, 3, 4], target = 5 → Could return [0, 3] or [1, 2]

**QUESTION 4**

**Prompt**: How do I find the maximum product of three numbers in an array using a single pass?

**Response**: Use variables to track the three largest and two smallest numbers as you iterate through the array. After the loop, calculate the maximum product using these variables.

**Implementation:**

1. Initialization:  
   Initialize variables to store the top 3 largest and bottom 2 smallest numbers (max1, max2, max3, min1, min2).
2. Iteration:  
   Loop through the array and update the largest and smallest values based on the current number.
3. Return the maximum product:  
   After the loop, calculate the maximum product from the possible combinations:
   * Product of three largest numbers.
   * Product of largest number and two smallest numbers.

**Adjustments Made:**

* Array Length Check:  
  Added a check to throw an exception if the array has fewer than 3 elements:

if (nums.Length < 3)

{

throw new ArgumentException("Array must contain at least three numbers.");

}

**Edge Cases**

1. < 3 elements: Not enough to calculate product → throw error  
 nums = [1, 2] → Invalid

2. All positives:  
 nums = [1, 2, 3, 4] → 4 \* 3 \* 2 = 24

3.Includes negatives:  
 nums = [-10, -10, 5, 2] → (-10) \* (-10) \* 5 = 500

**QUESTION 5**

**Prompt** : How can I convert a decimal number to binary in C#?

**Response** : Use a loop to repeatedly divide the number by 2 and collect the remainders. Reverse the result to get the binary string. Handle edge cases like 0 and negative numbers.

**Implementation Details** for**:**

1. Check Zero:  
   If the input is 0, directly return "0".
2. Store Remainders:  
   Use a List<int> to store remainders of division by 2 (i.e., binary digits).
3. Divide Iteratively:  
   Keep dividing the number by 2 and storing remainders until it becomes 0.
4. Reverse the List:  
   Since the binary digits are collected in reverse order, reverse them to get the final binary value.
5. Negative Numbers:  
   Convert the absolute value and prefix "-" to the result if input is negative.
6. Join Digits:  
   Convert the list of digits into a string using string.Join().

**Edge Cases** :

1.When the input is 0, the function should immediately return "0" without further processing, as 0 in binary is simply 0.

2.For negative numbers, the function should compute the binary of the absolute value and then add a negative sign ("-") to the result.

3.For positive numbers like 1, 10, or 42, the function correctly performs iterative division by 2 and forms the binary string as expected.

4.If the input is already 1, it’s a minimal edge case where only one digit ("1") is returned.

5.The function also handles larger numbers efficiently due to its loop-based division logic.

6.Unusual inputs like large negatives or extremely high positives are handled using Math.Abs() and string reversal.

**QUESTION 6**

**Prompt Used:** *How do I find the minimum element in a rotated sorted array using binary search?*

**Response Received:**

Use binary search:

* Compare middle element with the rightmost element.
* If mid > right, the minimum is in the right half.
* Else, it's in the left half.  
  Repeat until left == right.

**Implementation Details:**

* Applied binary search approach for O(log n) time.
* Used left, right, and mid pointers.
* At each step, narrowed down the search space depending on nums[mid] > nums[right].

**Edge Cases :**

* No rotation: Input like [1, 2, 3] returns first element.
* Rotated at various points: Works for any pivot rotation.
* Minimum at the end or start: Binary search still finds it.
* Single element: Returns that element.
* Empty array: Throws exception.

**QUESTION 7**

**Prompt Used:** *How do I check if a number is a palindrome without converting it to a string?*

**Response Received:**

Check digits by reversing half the number:

* If reversed half equals the remaining part, it's a palindrome.
* Avoid full reversal to prevent overflow0.

**Implementation Details:**

* Avoided string conversion.
* Reversed the last half of the digits (e.g., x % 10, then reversed = reversed \* 10 + digit).
* Checked if x == reversed or x == reversed / 10 (for odd digits).

**Adjustments Made:**

* Added early return if x < 0 (negative numbers can't be palindromes).
* Skipped cases where number ends with 0 (e.g., 10, not a palindrome unless the number is 0 itself).
* Handled both even and odd digit lengths.

**Edge Cases :**

* **Negative numbers**: Return false (e.g., -121).
* **Single-digit numbers**: Always true.
* **Numbers ending with 0 (but not 0 itself)**: Always false (e.g., 10).
* **Very large palindromes**: Still works due to half reversal.
* **Odd number of digits**: Handled using reversed / 10.

**QUESTION 8**

**Prompt Used:** *How can I compute the n-th Fibonacci number efficiently without recursion?*

**Response Received:**

Use iterative dynamic programming (bottom-up approach) to avoid stack overflow and redundant computations. Maintain two variables to track the previous two Fibonacci numbers and update them iteratively.

**Implementation Details:**

* Initialized a = 0 and b = 1 to represent F(0) and F(1).
* Loop from 2 to n, updating the next Fibonacci number using:  
  next = a + b, then a = b, b = next.
* Returned a if n == 0, b otherwise.
* This approach has O(n) time complexity and O(1) space complexity.

**Adjustments Made:**

* Avoided recursion to ensure scalability for large n.
* Handled base cases explicitly:
  + If n == 0, return 0
  + If n == 1, return 1

**Edge Cases :**

* n = 0 or 1: Should return 0 and 1 respectively.
* Negative n: Not valid in the classic Fibonacci definition (can throw an exception or return -1).
* Large n (e.g., n > 40): Iterative method works efficiently, unlike recursive which can crash.
* Memory efficiency: Constant space used regardless of input size.