**Sorting**

**1.** <https://leetcode.com/problems/height-checker/>

**A screenshot of a computer

Description automatically generated**To solve this problem, I first needed to understand that we are comparing the current order of students' heights with the order that would be expected if they were lined up correctly by height. The basic idea was to sort the list of heights, which would give us the expected order. Then, by comparing the original list with this sorted version, I counted how many positions in the list had different values. Those differences indicate the students who are not standing in the correct order. This approach is straightforward because it leverages sorting and a simple comparison, making it both easy to implement and understand. The main challenge here is ensuring the comparison is accurate, but using the zip function simplifies this process by letting me pair up elements from both lists directly.

**2.** <https://leetcode.com/problems/relative-sort-array/>

To tackle this problem, I needed to ensure that the elements in `arr1` were sorted according to the specific order defined by `arr2`. The elements in `arr2` are distinct and must appear in the same order in `arr1`, while the remaining elements not in `arr2` should be sorted at the end in ascending order.

To achieve this, I started by creating a dictionary that mapped each element in `arr2` to its index. This mapping allowed me to know the exact order in which elements from `arr2` should appear. Then, I defined a custom sorting function that first checked if an element was in `arr2`. If it was, the function returned a tuple where the first value ensured these elements were sorted based on their appearance order in `arr2`. If an element wasn't in `arr2`, it was sorted by its value and placed at the end. Finally, I applied this custom sorting function to `arr1`, which gave me the desired order, respecting both the order specified by `arr2` and placing any other elements in ascending order at the end. This approach efficiently sorts the array while maintaining the required order.

**A screenshot of a computer program

Description automatically generated**

**3.**[**https://leetcode.com/problems/maximum-gap/**](https://leetcode.com/problems/maximum-gap/)

**A screenshot of a computer

Description automatically generated**

To solve this problem, I started by recognizing that the goal was to find the maximum gap between any two successive elements in the sorted version of the array. If the array has fewer than two elements, there's no meaningful gap to consider, so the result should be 0.

First, I sorted the array, which rearranged the elements in increasing order. With the sorted array, I could then compute the differences between each pair of successive elements. I stored these differences in a list, as this would allow me to easily identify the maximum difference by simply taking the maximum value from this list.

Finally, I checked if the array had fewer than two elements, returning 0 in that case. Otherwise, I returned the largest difference I had found. This approach effectively identifies the maximum gap after sorting the array, making sure that all possible gaps are considered.

**4.**[**https://leetcode.com/problems/query-kth-smallest-trimmed-number/**](https://leetcode.com/problems/query-kth-smallest-trimmed-number/)

**A screenshot of a computer

Description automatically generated**

To solve this problem, I needed to process each query by trimming the numbers in the array nums according to the specified number of digits, then determine the k-th smallest trimmed number based on its value and original index.

For each query, I extracted the necessary number of digits from the right of each number in nums by using slicing. This allowed me to focus on the rightmost digits as required by the query. I then created a temporary list to store these trimmed numbers alongside their original indices, so I could easily track their positions in the original array.

Next, I sorted this list using insertion sort, where I inserted each trimmed number into its correct position based on its value. If two trimmed numbers were the same, their order in the original list was preserved due to how I handled the sorting process. This ensured that the trimmed number with the lower index in the original list would be considered smaller.

After sorting, I simply retrieved the index of the k-th smallest trimmed number from the sorted list and stored it as the answer for the current query. Finally, I returned all the collected answers as the result. This approach allowed me to handle each query individually, ensuring that the trimmed numbers were correctly sorted and the k-th smallest was identified efficiently.

a challenge I faced is handling the sorting efficiently of trimmed numbers while preserving their original indices, especially when dealing with numbers that had identical trimmed values.

**5.**<https://leetcode.com/problems/sort-an-array/>

A screenshot of a computer

Description automatically generated

To implement merge sort, I started by dividing the array into two halves recursively until each subarray had only one or zero elements, which are already sorted. Then, I focused on merging these sorted halves. By comparing the smallest elements from each half, I placed the smaller one into a temporary buffer. Once the entire subarray was merged, I copied the sorted elements back into the original array. I repeated this process for all levels of recursion until the entire array was sorted. This approach efficiently sorts the array while maintaining a clear structure for handling each step.