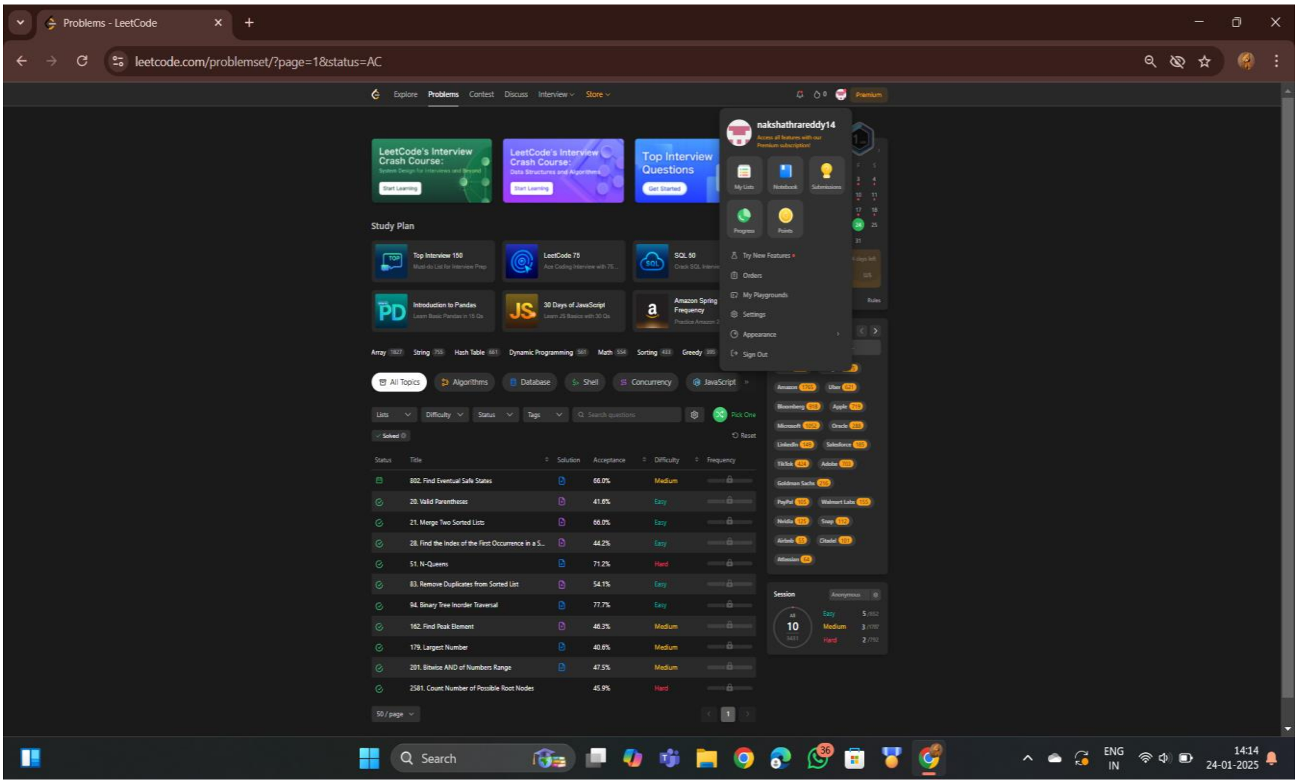
M.Nakshathra Reddy

2211CS020319

DAA HOLIDAY ASSIGNMENT

Develop the code for following Leet Code problems



M.Nakshathra Reddy

2211CS020319

Case Study:

1. Time and Space Complexities of Common Sorting Algorithms:

Quick Sort:

- **Time Complexity:**
  - Best case: O(nlogn)O(n \log n)O(nlogn) — when the pivot divides the array into roughly equal parts.
  - Average case: O(nlogn)O(n \log n)O(nlogn) — for random input.
  - Worst case: O(n2)O(n^2)O(n2) — when the pivot choice leads to unbalanced partitions (e.g., sorted or nearly sorted data).
- **Space Complexity:**
  - O(logn)O(\log n)O(logn) — due to the recursion stack in the best and average cases.
  - Worst case: O(n)O(n)O(n) — when the recursion stack is deep (which can happen in the worst case).

Merge Sort:

- **Time Complexity:**
  - Best, average, and worst case: O(nlogn)O(n \log n)O(nlogn) — as the algorithm always divides the list into two halves and merges them back.
- **Space Complexity:**
  - O(n)O(n)O(n) — because it requires additional space for the merged arrays.

2. Impact of Data Characteristics on Sorting Algorithm Choice:

- **Range of Prices:**
  - If the prices are within a narrow range or are integers, algorithms like **Counting Sort** or **Radix Sort** may be more efficient because their time complexity can be O(n)O(n)O(n), which is faster than comparison-based algorithms (like Quick Sort or Merge Sort) for such specific data.

- For a broad range of prices, **Quick Sort** or **Merge Sort** would be more appropriate as they offer O(nlogn)O(n \log n)O(nlogn) time complexity.

- **Product Name Lengths:**
    - For product names, which are strings, algorithms like **Quick Sort** and **Merge Sort** are commonly used because they compare strings lexicographically.
    - If the names are very short, the cost of comparison is low, so Quick Sort and Merge Sort are effective.
    - If names are long but have a fixed pattern (e.g., fixed length strings), **Radix Sort** may perform well as it can handle strings efficiently by processing character by character.
- **Data Distribution:**
    - If the data is nearly sorted or contains many duplicate values, **Insertion Sort** or **Bubble Sort** might be more efficient in practice despite their worst-case time complexities of O(n2)O(n^2)O(n2) since they can quickly detect already sorted sections.
    - For random or unsorted data, **Quick Sort** is usually preferred for its average case performance.

In conclusion, when dealing with millions of items, if the data characteristics are known (e.g., narrow price range), non-comparison-based algorithms (like Radix Sort or Counting Sort) may be a good choice. For more general use cases, **Quick Sort** or **Merge Sort** will usually be the best option, with Merge Sort providing more consistent performance in terms of time complexity.