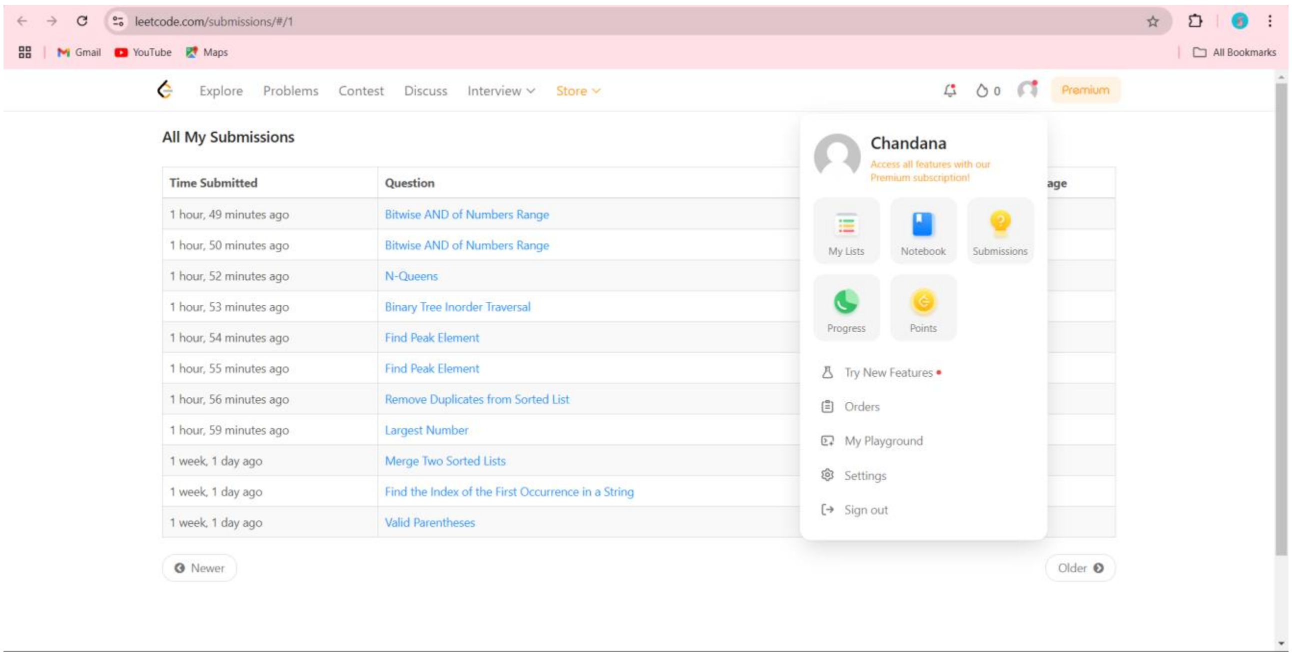
DAA HOLIDAY ASSIGNMENT



All My Submissions

Time Submitted	Question	age
1 hour, 49 minutes ago	Bitwise AND of Numbers Range	
1 hour, 50 minutes ago	Bitwise AND of Numbers Range	
1 hour, 52 minutes ago	N-Queens	
1 hour, 53 minutes ago	Binary Tree Inorder Traversal	
1 hour, 54 minutes ago	Find Peak Element	
1 hour, 55 minutes ago	Find Peak Element	
1 hour, 56 minutes ago	Remove Duplicates from Sorted List	
1 hour, 59 minutes ago	Largest Number	
1 week, 1 day ago	Merge Two Sorted Lists	
1 week, 1 day ago	Find the Index of the First Occurrence in a String	
1 week, 1 day ago	Valid Parentheses	

<u>CASE STUDY</u>

Scenario:

An e-commerce platform is implementing a feature where products need to be sorted by various attributes (e.g., price, rating, and name). The product list contains millions of items, and the sorting operation needs to be efficient and scalable.

1. **What are the time and space complexities of the commonly used sorting algorithms (Quick Sort, Merge Sort)?**

2. **How do the characteristics of the data (e.g., range of prices, product name lengths) impact the choice of sorting algorithm? in c++**

1. Time and Space Complexities of Common Sorting Algorithms

Quick Sort:

1. **Time Complexity**:

- **Best Case**: O(nlogn)
  O(n \log n)
  O(nlogn)
- **Average Case**: O(nlogn)
  O(n \log n)
  O(nlogn)
- **Worst Case**: O
  (n2)

  O(n^2)
  O(n2) (This happens when the pivot selection is poor, such as always selecting the smallest or largest element)

2. **Space Complexity**:

- **Best/Average Case**: O(logn)
  O(\log n)
  O(logn) (This is due to the recursion stack in the case of balanced partitioning)
- **Worst Case**: O(n)
  O(n)
  O(n) (In the worst case, the recursion stack could grow to the height of the array if there is no good pivot selection)

Quick Sort is generally considered efficient for large datasets, particularly when implemented with a good pivot strategy (e.g., random pivot selection or the median of three).

CODE:

```
#include <iostream>

#include <vector>

using namespace std;
```

```
partition(vector<int>& arr, int low, int high) {
int pivot = arr[high];     int i = low - 1;
 for (int j = low; j <= high - 1; j++) {
if (arr[j] <= pivot) { i++;
swap(arr[i], arr[j]);
        }
    }     swap(arr[i + 1],
arr[high]);     return i + 1;
}

function void quickSort(vector<int>& arr, int low,
int high) {     if (low < high) {         int pi =
partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Left partition
quickSort(arr, pi + 1, high); // Right partition
    }
}

int main() {     vector<int> arr = {1, 7,
8, 4, 1, 5};     int n = arr.size();
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
for (int num : arr) {
cout << num << " ";
    }
return 0;
}
```

Merge Sort:

Time Complexity:

- **Best Case**: O(nlogn)
  O(n \log n)
  O(nlogn)
- **Average Case**: O(nlogn)
  O(n \log n)
  O(nlogn)
- **Worst Case**: O(nlogn)
  O(n \log n)
  O(nlogn)

Space Complexity:

- **Worst Case**: O(n)
  O(n)
  O(n) (Merge Sort requires additional space to store the temporary arrays during the merging process)

Merge Sort is highly predictable in terms of performance, making it suitable for large datasets. Its space complexity can be a drawback in systems with limited memory.

CODE:

```
#include <iostream>
#include <vector>
using namespace std;

// Merge function to merge two sorted subarrays void
merge(vector<int>& arr, int left, int mid, int right) {
int n1 = mid - left + 1;
   int n2 = right - mid;

   vector<int> leftArr(n1), rightArr(n2);

   for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

   int i = 0, j = 0, k = left;

   while (i < n1 && j < n2) {
if (leftArr[i] <= rightArr[j]) {
arr[k] = leftArr[i];            i++;
} else {            arr[k] =
rightArr[j];            j++;        }
k++;
   }
```

```
    while (i < n1) {
arr[k] = leftArr[i];
i++;         k++;
    }

    while (j < n2) {
arr[k] = rightArr[j];
j++;         k++;
    }
function void mergeSort(vector<int>& arr, int left,
int right) {     if (left < right) {         int mid = left +
(right - left) / 2;

        mergeSort(arr, left, mid);  // Left part
mergeSort(arr, mid + 1, right); // Right part

        merge(arr, left, mid, right);  // Merge both parts
    }
}

int main() {     vector<int> arr = {12, 11,
13, 5, 6, 7};     int n = arr.size();
 mergeSort(arr, 0, n - 1);
```

```
    cout << "Sorted array: ";
for (int num : arr) {
cout << num << " ";
    }
return 0;
}
```

2.Impact of Data Characteristics on Sorting Algorithm Choice

When dealing with large datasets on an e-commerce platform (e.g., millions of products), choosing the right sorting algorithm depends on several data characteristics:

a) Range of Prices:

- **Limited Range of Prices**:
  - **Bucket Sort** or **Radix Sort** might be efficient if the prices fall within a known and narrow range. These algorithms can sort in linear time O(n)

    O(n)

    O(n) when the range of elements is small and the distribution is known (e.g., products priced between $0 to $1000).
  - **Comparison Sorts (e.g., Quick Sort, Merge Sort)** would not take advantage of the range, and they would run in O(logn)

    O(n \log n)

    O(nlogn).
- **Large Range of Prices**:
  - **Quick Sort** and **Merge Sort** would be good choices if the price range is wide. These algorithms work well with data that is uniformly distributed, even when the range is large.

b) Product Name Lengths:

- If product names are **short** (e.g., less than 20 characters):

  - **Quick Sort** and **Merge Sort** will perform well, as their time complexities are not dependent on the individual lengths of strings but rather on the total number of products.
- If product names are **long** (e.g., thousands of characters):
  - Sorting algorithms like **Merge Sort** (which can perform better with string sorting due to the merging of smaller subarrays) can still work effectively. However, the choice of sorting algorithm will also depend on whether you are sorting by the entire string or just parts (like name prefixes).
  - **Radix Sort** can also be useful for sorting strings, especially if the strings are of equal or bounded length. Radix Sort can sort strings based on characters or bytes in multiple passes.

c) Distribution of Product Ratings:

- If ratings are **uniformly distributed** (e.g., integer ratings from 1 to 5), specialized algorithms like **Counting Sort** might be useful, as it can sort in linear time O(n)

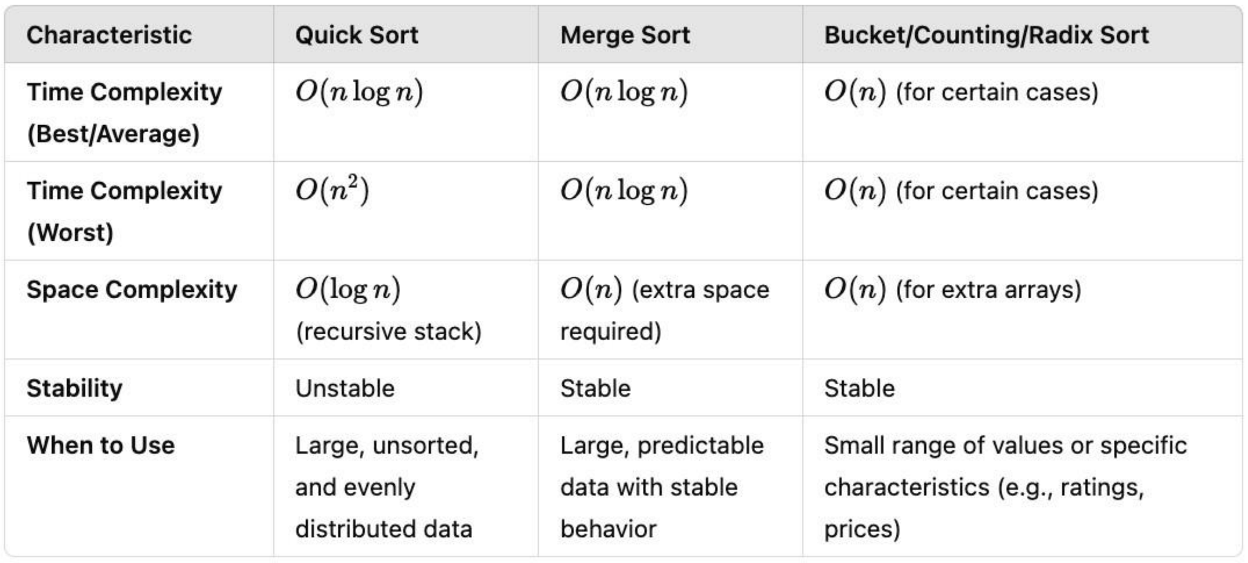
  O(n)

  O(n) when the number of distinct ratings is small.
- If the ratings are **highly varied** (e.g., float values with many decimal places), **Quick Sort** or **Merge Sort** would be better suited.

d) Stability:

- If you need to maintain the relative order of products that have the same price, rating, or other attributes, a **stable sorting algorithm** is required. **Merge Sort** is stable, while **Quick Sort** is not stable by default (though a stable version can be implemented).

3. Summary of Sorting Algorithm Characteristics and When to Use Them



Characteristic	Quick Sort	Merge Sort	Bucket/Counting/Radix Sort
Time Complexity (Best/Average)	$O(n \log n)$	$O(n \log n)$	$O(n)$ (for certain cases)
Time Complexity (Worst)	$O(n^2)$	$O(n \log n)$	$O(n)$ (for certain cases)
Space Complexity	$O(\log n)$ (recursive stack)	$O(n)$ (extra space required)	$O(n)$ (for extra arrays)
Stability	Unstable	Stable	Stable
When to Use	Large, unsorted, and evenly distributed data	Large, predictable data with stable behavior	Small range of values or specific characteristics (e.g., ratings, prices)