BIG DATA ANALYTICS HOLIDAY ASSIGNMENT CASE STUDY QUESTION

2211CS020538

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 opcration needs to bc cfficicnt and scalable.

1.What are the time and space complexities of the commonly used Sorting algorithms (Quick Sort, Mergc Sort)?

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

1. Overview In an e-commerce platform, millions of products need to be sorted efficiently based on various attributes such as price, rating, and name. The implementation of an efficient and scalable sorting mechanism is critical for enhancing user experience and ensuring quick data retrieval. This case study analyzes the time and space complexities of commonly used sorting algorithms, such as Quick Sort and Merge Sort, and examines how data characteristics influence the choice of sorting algorithm.

2. Time and Space Complexities of Sorting Algorithms

Quick Sort:

- **Time Complexity:**
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n^2) (occurs when the pivot selection is poor, e.g., when the smallest or largest element is consistently chosen as the pivot)
- **Space Complexity:** O(log n) (due to the recursion stack)

Merge Sort:

- **Time Complexity:**
  - Best Case: O(n log n)
  - Average Case: O(n log n)
  - Worst Case: O(n log n) (consistent performance regardless of input)
- **Space Complexity:** O(n) (additional space required for temporary arrays)

3. Impact of Data Characteristics on Sorting Algorithm Choice

Range of Prices:

- When sorting by price, the range and distribution of prices significantly impact performance. For uniformly distributed prices, Quick Sort may perform efficiently.

However, if prices are skewed or have many duplicates, Merge Sort could be preferable due to its stable sorting nature.

Product Name Lengths:

- Sorting by product names involves handling strings of varying lengths. Merge Sort is more stable and handles strings better without adversely affecting performance. Quick Sort, on the other hand, may face performance issues due to repeated comparisons of string elements, especially if they are long and complex.

Case Studies and Observations:

- **Scenario 1: High Variability in Prices**
  - In cases where prices range widely, Quick Sort's average case performance is desirable. However, if a significant number of products share the same price, the risk of Quick Sort's worst-case performance increases.
  - **Recommendation:** Use Merge Sort to ensure consistent O(n log n) performance, as it handles duplicates and varying data distributions effectively.
- **Scenario 2: Uniformly Distributed Ratings**
  - Quick Sort performs well with uniformly distributed data. Pivot selection strategies like randomized or median-of-three can mitigate the risk of worst-case scenarios.
  - **Recommendation:** Implement Quick Sort with a robust pivot selection strategy.
- **Scenario 3: Long Product Names**
  - Sorting long strings can lead to inefficiencies in Quick Sort due to multiple comparisons. Merge Sort, with its consistent O(n log n) complexity, provides a more stable alternative.
  - **Recommendation:** Use Merge Sort for sorting by product names to leverage its stability and efficiency with string data.
- **Code Implementation**
- **Quick Sort Implementation in C++:**

```
#include <iostream>

#include <vector>

using namespace std;


void quickSort(vector<double>& arr, int left, int right) {

   if (left < right) {

      double pivot = arr[right];

      int i = left - 1;

      for (int j = left; j < right; j++) {

         if (arr[j] <= pivot) {
```

```
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[right]);
    int partitionIndex = i + 1;

    quickSort(arr, left, partitionIndex - 1);
    quickSort(arr, partitionIndex + 1, right);
  }
}

int main() {
  vector<double> products_price = {29.99, 19.99, 49.99, 9.99, 39.99};
  quickSort(products_price, 0, products_price.size() - 1);
  for (double price : products_price) {
    cout << price << " ";
  }
  return 0;
}
```

Merge Sort Implementation in C++:

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

void merge(vector<string>& arr, int left, int mid, int right) {
  int n1 = mid - left + 1;
  int n2 = right - mid;

  vector<string> L(n1), R(n2);
```

```
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(vector<string>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    vector<string> products_name = {"Apple", "Orange", "Banana", "Grapes",
"Pineapple"};
    mergeSort(products_name, 0, products_name.size() - 1);
    for (const string& name : products_name) {
        cout << name << " ";
    }
    return 0;
}
```