# DAA Case Study

## 2211CS020512

**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)?**

**Time and Space Complexities of Commonly Used Sorting Algorithms**

- **Quick Sort**
  - **Best and Average Time Complexity**: O(nlog⁡n)O(n \log n)O(nlogn)
  - **Worst Time Complexity**: O(n2)O(n^2)O(n2) (occurs when pivot selection is poor, but can be minimized with good pivot strategies like median-of-three)
  - **Space Complexity**: O(log⁡n)O(\log n)O(logn) for recursive call stack
- **Merge Sort**
  - **Time Complexity**: Always O(nlog⁡n)O(n \log n)O(nlogn)
  - **Space Complexity**: O(n)O(n)O(n) due to auxiliary arrays for merging.

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

**Impact of Data Characteristics on Sorting Algorithm Choice**

- **Range of Prices**:
  If the range of prices is small and bounded, **counting sort** or **bucket sort** can be more efficient than general-purpose sorts like quick sort or merge sort. However, these specialized sorts require additional memory.

- **Distribution of Ratings**:
  Ratings often have a small number of discrete values (e.g., 1 to 5 stars). **Counting sort** can handle this very efficiently since ratings are integers within a fixed range. Alternatively, **bucket sort** might be useful if ratings are more granular and continuous.

- **Length of Product Names**:
  Sorting by product names involves lexicographical ordering. In this case:

  - **Quick sort** is typically suitable due to its in-place nature and average-case efficiency.

  - If names are highly repetitive, **merge sort** or **radix sort (for strings)** can provide stable and predictable performance. Radix sort works well for fixed-length strings.

- **Size of Data**:
  When handling millions of items:
    - **Quick sort** is preferred for its in-place sorting and better cache locality, but careful pivot selection is needed to avoid worst-case performance.
    - **Merge sort** guarantees O(nlog⁡n)O(n \log n)O(nlogn) performance but requires extra space, which may be a drawback for very large datasets unless optimized for external sorting.

**CODE:**

**Sorting By Price(quick sort)**

```
#include <iostream>

#include <vector>

int partition(std::vector<int>& prices, int low, int high) {

    int pivot = prices[high];

    int i = low - 1;

    for (int j = low; j < high; ++j) {

        if (prices[j] < pivot) {

            std::swap(prices[++i], prices[j]);

        }

    }

    std::swap(prices[i + 1], prices[high]);

    return i + 1;

}

void quickSort(std::vector<int>& prices, int low, int high) {

    if (low < high) {

        int pi = partition(prices, low, high);

        quickSort(prices, low, pi - 1);

        quickSort(prices, pi + 1, high);

    }
}

int main() {
    std::vector<int> prices = {300, 100, 500, 200, 400};
    quickSort(prices, 0, prices.size() - 1);
    for (int price : prices) {
        std::cout << price << " ";
    }
    return 0;
}
```

**Sorting by name(merge sort)**

```
#include <iostream>
#include <vector>
#include <string>
void merge(std::vector<std::string>& names, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    std::vector<std::string> L(names.begin() + left, names.begin() + mid + 1);
    std::vector<std::string> R(names.begin() + mid + 1, names.begin() + right + 1);
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            names[k++] = L[i++];
        } else {
            names[k++] = R[j++];
        }
    }
    while (i < n1) names[k++] = L[i++];

    while (j < n2) names[k++] = R[j++];

}

void mergeSort(std::vector<std::string>& names, int left, int right) {

    if (left < right) {

        int mid = left + (right - left) / 2;

        mergeSort(names, left, mid);

        mergeSort(names, mid + 1, right);

        merge(names, left, mid, right);

    }

}

int main() {

    std::vector<std::string> names = {"Laptop", "Phone", "Tablet", "Monitor", "Keyboard"};

    mergeSort(names, 0, names.size() - 1);

    for (const std::string& name : names) {

        std::cout << name << " ";

    }

    return 0;

}
```

**Summary:**

**For price sorting:** Use quick sort or bucket sort depending on range and distribution.

**For rating sorting:** Use counting sort or quick sort if ratings are not discrete.

**For name sorting**: Use merge sort (for stability) or quick sort (for memory efficiency).
The exact choice depends on constraints related to memory availability, data patterns, and scalability requirements.