**Case Study: Sorting-Based Farmington**

**Introduction:**

Farmington's growing order database means that sorting becomes essential—whether it's sorting by order date, quantity, customer name, or item name. Sorting helps admins process orders efficiently, generate organized reports, and improve the user experience on the frontend.

**Why Sorting Is Needed in Farmington:**

1. **Admin Dashboard Views** – View orders sorted by most recent, quantity, or alphabetically.
2. **Inventory Planning** – Sort items by demand (quantity) to reorder popular ones first.
3. **Customer Service** – Quickly locate and organize customer-specific orders.
4. **Delivery Prioritization** – Sort orders by location or urgency.

**Merge Sort in Farmington:**

**Concept:**

Merge Sort is a divide-and-conquer algorithm that splits the list into smaller parts, sorts each part, and then merges them in sorted order.

**Use Case in Farmington:**

Sort Orders by Quantity (Low to High)

* Orders are split, sorted, and merged efficiently, even for large datasets.

✅ **Advantages:**

* Efficient for large datasets (O(n log n) complexity).
* Stable sort: maintains the relative order of equal elements.

❌ **Limitation:**

* Requires extra memory for merging, making it less space-efficient.

**Quick Sort in Farmington:**

**Concept:**

Quick Sort selects a pivot element, partitions orders around it, and sorts recursively.

**Use Case in Farmington:**

Sort Orders by Customer Name

* Helps in quickly locating customer orders during searches.

✅ **Advantages:**

* Faster in practice for large datasets.
* Works well with in-place sorting, reducing memory overhead.

❌ **Limitation:**

* Worst-case complexity is O(n²) if the pivot selection is poor.

**Heap Sort in Farmington:**

**Concept:**

Heap Sort builds a max/min heap and extracts elements in sorted order.

**Use Case in Farmington:**

Sort Orders by Item Name

* Useful for maintaining a priority queue of items based on demand.

✅ **Advantages:**

* Guarantees O(n log n) time complexity.
* Efficient for priority-based ordering.

❌ **Limitation:**

* Not a stable sort; equal elements may not retain their order.

**Time Complexity Comparison Table (Farmington Context):**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sorting Method** | **Best Case** | **Worst Case** | **Usefulness in Farmington** |
| Merge Sort | O(n log n) | O(n log n) | Best for large order lists |
| Quick Sort | O(n log n) | O(n²) | Fastest in most cases |
| Heap Sort | O(n log n) | O(n log n) | Good for priority-based sorting |

**Summary (When to Use What):**

|  |  |  |
| --- | --- | --- |
| Scenario | Suggested Sort | Why? |
| Sorting 10,000+ orders by quantity | Merge Sort | Stable, efficient for large data |
| Finding customer orders quickly | Quick Sort | Faster in practical use |
| Sorting by item demand | Heap Sort | Maintains a priority queue |

Code:

import heapq

def merge\_sort(order\_list):

if len(order\_list) > 1:

mid = len(order\_list) // 2

left\_half = order\_list[:mid]

right\_half = order\_list[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i]['quantity'] < right\_half[j]['quantity']:

order\_list[k] = left\_half[i]

i += 1

else:

order\_list[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

order\_list[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

order\_list[k] = right\_half[j]

j += 1

k += 1

return order\_list

def quick\_sort(order\_list):

if len(order\_list) <= 1:

return order\_list

pivot = order\_list[len(order\_list) // 2]

left = [x for x in order\_list if x['customer'] < pivot['customer']]

middle = [x for x in order\_list if x['customer'] == pivot['customer']]

right = [x for x in order\_list if x['customer'] > pivot['customer']]

return quick\_sort(left) + middle + quick\_sort(right)

def heap\_sort(order\_list):

heapq.heapify(order\_list)

return [heapq.heappop(order\_list) for \_ in range(len(order\_list))]

orders = [

{"id": 101, "customer": "Akash", "item": "Tomatoes", "quantity": 10},

{"id": 102, "customer": "Preethi", "item": "Carrots", "quantity": 5},

{"id": 103, "customer": "Prithvi", "item": "Onions", "quantity": 8},

{"id": 104, "customer": "Kavya", "item": "Beans", "quantity": 12},

{"id": 105, "customer": "Ravi", "item": "Potatoes", "quantity": 7}

]

print("\n🔹 Merge Sort by Quantity:")

print(merge\_sort(orders))

print("\n🔹 Quick Sort by Customer Name:")

print(quick\_sort(orders))

print("\n🔹 Heap Sort by Quantity:")

print(heap\_sort(orders))

**Conclusion**

Efficient order management is crucial in Farmington's digital agriculture business. Merge Sort, Quick Sort, and Heap Sort provide scalable solutions to sort large datasets, prioritize deliveries, and enhance overall efficiency. Implementing the right sorting algorithm improves customer satisfaction, logistics, and business intelligence.