**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.

**Bubble Sort in Farmington:**

**Concept:**

Bubble Sort repeatedly compares **adjacent orders** and **swaps them** if they are in the wrong order. This continues until the entire list is sorted.

**Use Case in Farmington:**

**Sort Orders by Quantity (Low to High)**

* Bubble sort would compare each order’s quantity and push the largest values to the end (like bubbles rising).

**✅ Advantage:**

* Easy to implement and understand.
* Works well on small datasets (e.g., sorting 10–20 recent orders for quick review).

**❌ Limitation:**

* Time complexity: **O(n²)** – inefficient for 1000s of orders.

**Insertion Sort in Farmington:**

**Concept:**

Insertion Sort builds the sorted list one item at a time by **inserting each order** into its correct position compared to the previous ones.

**Use Case in Farmington:**

**Sort Orders Alphabetically by Customer Name**

* Useful when orders arrive in mostly sorted order (like regular customers ordering daily).

**✅ Advantage:**

* Efficient for **nearly sorted** data.
* Performs better than bubble sort in practice.

**❌ Limitation:**

* Still **O(n²)** in the worst case, not suitable for bulk datasets.

**Selection Sort in Farmington:**

**Concept:**

Selection Sort repeatedly finds the **smallest (or largest)** order from the unsorted part and moves it to the front.

**Use Case in Farmington:**

**Sort Orders by Item Name**

* Useful for printing item-wise packing lists, ensuring items are grouped together for dispatch.

**✅ Advantage:**

* Fewer swaps than bubble sort.
* Good when memory write is expensive.

**❌ Limitation:**

* Time complexity: **O(n²)** – not ideal for large-scale order data.

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

| **Sorting Method** | **Best Case** | **Worst Case** | **Usefulness in Farmington** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | For small order logs or quick fixes |
| Insertion Sort | O(n) | O(n²) | For real-time insertion during logging |
| Selection Sort | O(n²) | O(n²) | For sorting fields with few unique values |

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

| **Scenario** | **Suggested Sort** | **Why?** |
| --- | --- | --- |
| Sorting 10 recent orders by quantity | Bubble Sort | Simple and quick for small dataset |
| Sorting order entries as they come in | Insertion Sort | Keeps list sorted in real-time |
| Sorting a small list alphabetically for reporting | Selection Sort | Few swaps, maintains stable order |
| Large list of 1000+ orders | ❌ Not ideal | Use Python’s built-in sorted() or heapq |

**Objectives of Sorting Algorithms in Farmington**

**1. Bubble Sort – Objective: Simple Sorting for Small, Recent Orders**

**Purpose in Farmington:**

* Sort recent customer orders based on **quantity**, **price**, or **timestamp** in a basic and easy-to-understand way.

**Key Objectives:**

* **Clean up the order list** quickly for small datasets (like last 10 orders).
* **Generate mini reports** like “Top 5 smallest quantity orders today.”
* **Provide a beginner-friendly approach** for internal training or demos.

**2. Insertion Sort – Objective: Efficient Real-Time Sorting During Input**

**Purpose in Farmington:**

* Maintain a **live-sorted list** as new orders come in (e.g., by customer name, delivery time, or quantity).

**Key Objectives:**

* **Keep data sorted on-the-go** as new orders are placed.
* **Prioritize VIP or repeat customers** by sorting as per profile or urgency.
* **Enable quicker search** in mostly-sorted datasets.

**Challenges in Sorting – Farmington Case Study**

**1. Performance on Large Datasets**

**Challenge:**

* Farmington may handle **thousands of orders daily**.
* Bubble, Insertion, and Selection Sort all have **O(n²)** time complexity.

**Impact:**

* Sorting 10,000+ orders becomes **slow and laggy**.
* Causes delays in generating reports or organizing orders.

**2. Real-Time Sorting Requirements**

**Challenge:**

* Orders are placed **dynamically in real-time**.
* Manual sorting after every new order is inefficient.

**Impact:**

* Continuous sorting with these algorithms causes **performance bottlenecks**.
* Real-time user dashboards may **freeze or lag**.

**3. Lack of Stability in Some Algorithms**

**Challenge:**

* Selection Sort is **not stable** (i.e., it can change the order of equal items).

**Impact:**

* Orders with the same item name might appear **disordered by customer name or time**.
* Affects **order tracking consistency**.

**4. Memory Inefficiency with Large Orders**

**Challenge:**

* Although these sorting algorithms don’t need extra space, the **long processing time** ties up system resources.

**Impact:**

* Causes **backend server slowdowns**.

**Order Management Issues and Solutions**

**1. Order Cancellation Handling** Challenge:

* Customers may request order cancellations, but the last order in the stack may not be the one needing cancellation. Solution:
* Use an auxiliary queue to search and remove specific orders efficiently.

**2. High Storage Usage** Challenge:

* Large volumes of order data can increase storage requirements. Solution:
* Compressing order records and archiving older data can optimize memory usage.

**3. Order Processing Delays** Challenge:

* Heavy load on the system can slow down order dispatch and tracking. Solution:
* Implementing multithreading with thread-safe stacks like queue.LifoQueue can improve performance.

**4. Slow Payment Verification** Challenge:

* Processing payments sequentially can delay order confirmation. Solution:
* Using a concurrent stack system to handle payments and order management in parallel.

**5. Stock Mismanagement** Challenge:

* Unprocessed orders can lead to incorrect stock updates. Solution:
* Use an inventory-linked stack that updates stock levels immediately upon order placement.

**Advantages of Using Sorting in Farmington's Order Management**

**1. Faster Access to Critical Orders**

**Advantage:**

* Sorted orders help **quickly identify urgent or high-priority orders** (e.g., nearest delivery deadlines or large quantities).

**Farmington Use Case:**

* Admins can **prioritize perishable items** or VIP customer deliveries.
* Enables **efficient packing and shipping** decisions.

**2. Improved Data Analysis & Reporting**

**Advantage:**

* Sorted data allows for **easier generation of trends**, charts, and summaries.

**Farmington Use Case:**

* Track **top-selling crops**, customer purchase frequency, or **most-demanded items** during a season.

**3. Better Inventory Planning**

**Advantage:**

* Sorted sales data reveals what is **selling fastest or slowest**.

**Farmington Use Case:**

* Restock based on demand trends.
* Prevent **out-of-stock issues** or **overstocking** of slow-moving items.

**🛒 4. Enhanced Customer Experience**

**Advantage:**

* Customers can view products sorted by **price**, **freshness**, or **popularity**, improving navigation and decision-making.

**Farmington Use Case:**

* Buyers find the best deal quickly, reducing **cart abandonment** and **increasing satisfaction**.

**5. Efficient Logistics & Dispatching**

**Advantage:**

* Sorting orders by **delivery zone**, **product type**, or **dispatch time** simplifies logistics.

**Farmington Use Case:**

* Easier to group orders going to the same location.

Code:

orders = []

def add\_order(order\_id, customer, item, quantity):

order = {

"id": order\_id,

"customer": customer,

"item": item,

"quantity": quantity

}

orders.append(order)

print(f"Order {order\_id} added successfully.")

def view\_orders(order\_list):

print("\nOrders:")

for order in order\_list:

print(order)

def bubble\_sort\_by\_quantity(order\_list):

sorted\_orders = order\_list[:]

n = len(sorted\_orders)

for i in range(n):

for j in range(0, n - i - 1):

if sorted\_orders[j]['quantity'] > sorted\_orders[j + 1]['quantity']:

sorted\_orders[j], sorted\_orders[j + 1] = sorted\_orders[j + 1], sorted\_orders[j]

return sorted\_orders

def insertion\_sort\_by\_quantity(order\_list):

sorted\_orders = order\_list[:]

for i in range(1, len(sorted\_orders)):

key = sorted\_orders[i]

j = i - 1

while j >= 0 and sorted\_orders[j]['quantity'] > key['quantity']:

sorted\_orders[j + 1] = sorted\_orders[j]

j -= 1

sorted\_orders[j + 1] = key

return sorted\_orders

def selection\_sort\_by\_quantity(order\_list):

sorted\_orders = order\_list[:]

n = len(sorted\_orders)

for i in range(n):

min\_index = i

for j in range(i + 1, n):

if sorted\_orders[j]['quantity'] < sorted\_orders[min\_index]['quantity']:

min\_index = j

sorted\_orders[i], sorted\_orders[min\_index] = sorted\_orders[min\_index], sorted\_orders[i]

return sorted\_orders

add\_order(101, "Akash", "Tomatoes", 10)

add\_order(102, "Preethi", "Carrots", 5)

add\_order(103, "Prithvi", "Onions", 8)

add\_order(104, "Kavya", "Beans", 12)

add\_order(105, "Ravi", "Potatoes", 7)

view\_orders(orders)

print("\n🔹 Bubble Sort by Quantity:")

view\_orders(bubble\_sort\_by\_quantity(orders))

print("\n🔹 Insertion Sort by Quantity:")

view\_orders(insertion\_sort\_by\_quantity(orders))

print("\n🔹 Selection Sort by Quantity:")

view\_orders(selection\_sort\_by\_quantity(orders))

**Conclusion**

In the evolving landscape of digital agriculture, efficient **order management** plays a vital role in ensuring customer satisfaction, smooth logistics, and business growth. **Sorting algorithms**—though simple in concept—add tremendous value to Farmington by organizing order data in meaningful ways. Whether it’s prioritizing urgent deliveries, analyzing customer trends, or grouping similar orders for dispatch, sorting enhances **clarity, speed, and efficiency** across the platform.