RIPHAH INTERNATIONAL UNIVERSITY LAHORE CAMPUS

Riphah School of Computing & Innovation (RSCI)

Design & Analysis of Algorithm

BS Computer Science (5B) – Fall 2024

Semester Project

Student Information	
Name:	Muhammad Talha, Hafiza Shujia Yousaf
SAP ID:	46287, 47798
Section:	5-B
Project Name:	Banking Application

Final Project Report

Project Title: Banking Application

- **Introduction:**

In a banking application, efficient data processing is crucial to ensure fast retrieval and manipulation of customer records. The application must handle frequent operations, such as searching for customer information and sorting transaction data. Choosing the right algorithms for searching and sorting can significantly improve the application's performance and responsiveness. In this report, we will discuss the most suitable algorithms for these tasks, analyze their efficiency, and consider the consequences of using alternative algorithms. Detailed explanations of best, worst, and average case complexities will be provided to highlight the trade-offs between different algorithms.

1. Searching Algorithms for Customer Records:

1.1 TimSort:

- **Overview:**

1.1.1 Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort. It works by dividing the data into **runs** and sorting these runs using insertion sort, then merging them using a process similar to merge sort. It is the default sorting algorithm in Python and Android's Java.

- **Application in Mobile Banking**

1.1.2 **Transaction History**: Displaying a user's transaction history (sorted by date or amount).

1.1.3 **Account Listings**: Sorting a list of customer accounts by balance, account number, or name.

1.1.4 **Loan Applications**: Sorting loan applications based on interest rates, amounts, or approval status.

- **How Timsort Works:**

1.1.5 **Identify Natural Runs:** It scans through the data to find already sorted sub-sequences (runs). For example, transactions might already be sorted by date in parts.

1.1.6 **Sort Small Runs:** Uses insertion sort on small runs for better performance.

1.1.7 **Merge Runs:** Uses a merging technique similar to merge sort for larger data.

1.2 Introsort:

- **Overview:**

1.2.1 Introsort (Introspective Sort) combines quicksort, heapsort, and insertion sort. It starts with quicksort and switches to heapsort if the recursion depth exceeds a certain limit to avoid worst-case performance.

- **Application in Mobile Banking:**

1.2.2 **Sorting Customers:** When fetching and displaying customers based on balance or name.
1.2.3 **Card Transactions:** Sorting credit or debit card transactions based on the transaction date or amount.
1.2.4 **Interest Rates:** Sorting loan interest rates or deposit schemes.

- **How Introsort Works:**

1.2.5 **Initial Phase with Quicksort**: Uses quicksort for partitioning data.

1.2.6 **Switch to Heapsort**: If the recursion depth grows too large (a sign of potential worst-case behavior), it switches to heapsort.

1.2.7 **Insertion Sort for Small Partitions**: Handles small partitions with insertion sort for efficiency.

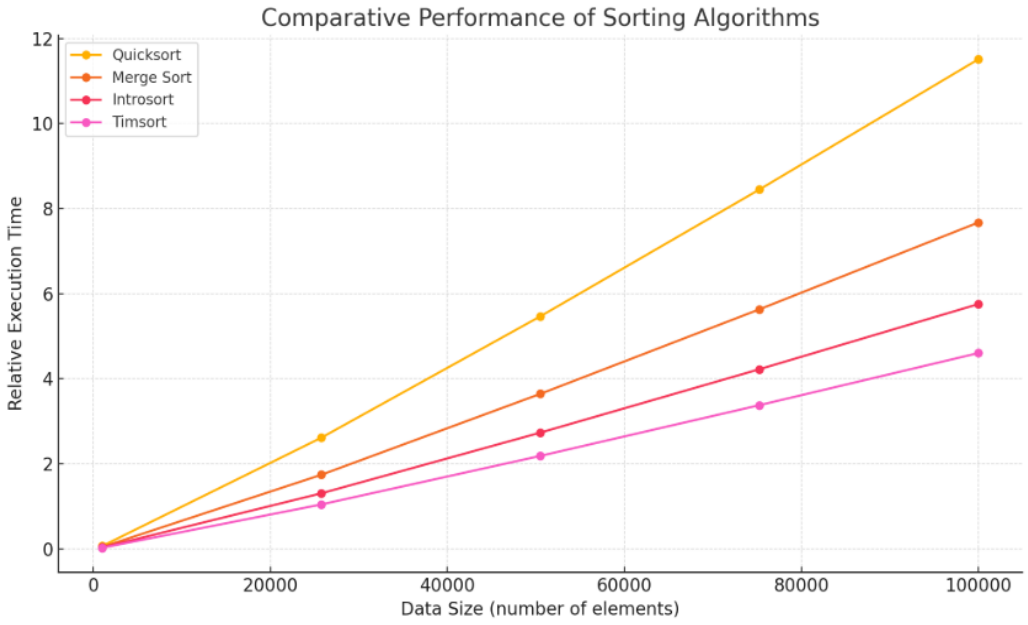
- **Why Introsort is Suitable:**

1.2.8 **Versatile:** Provides optimal performance across various input sizes.

1.2.9 **Predictable Time Complexity:** Avoids quicksort's worst-case behavior with heapsort.

1.2.10 **Efficient for Large Data**: Banking applications may handle large data like bulk customer records, making introsort's efficiency crucial.

Scenario	Timsort	Introsort
Transaction History (Sorted by Date)	Adaptable to partially sorted data	Faster if deep recursive sorting needed
Sorting Loan Applications by Rate	Stable sorting preserves order	Quick decision on large unsorted data
Customer Account Listings	Faster with pre-sorted lists	More efficient for random, large data



2. Sorting Algorithms for Customer Data:

2.1 Two Pointers Technique:

- **Overview:**
  - 2.1.1 This technique uses two pointers that traverse the data structure simultaneously, often from opposite ends or moving at different speeds. It is useful for:
  - 2.1.2 Searching pairs or finding subsequences with certain properties.
  - 2.1.3 Eliminating redundant loops to improve time complexity.

- **Searching for Transactions within a Range:**
  - 2.1.4 **Time Complexity**: O(n), much faster than a nested loop ($O(n^2)$).

- **Detecting Duplicate Transactions**
  - 2.1.5 By sorting the transactions first, a single pass with two pointers can identify duplicates in O(n log n + n) time.
  - 2.1.6 Merging Sorted Lists
  - 2.1.7 Combining debit and credit transaction histories sorted by date using two pointers. One pointer iterates over the debit transactions, and another over the credit transactions. The smaller date between the two moves forward in a merge-like process.

- **Advantages of Two Pointers Technique:**

2.1.8 Efficiency: Reduces time complexity from $O(n^2)$ to $O(n)$ or $O(n \log n)$.

2.1.9 Simple Implementation: Easier to implement than advanced search structures for sorted data.

2.1.10 Simple Implementation: Easier to implement than advanced search structures for sorted data.

Feature	Two Pointers Usage
Find transactions between two dates	Use one pointer to track the start date and another to find transactions until the end date.
Detect if a user exceeded a daily transaction limit	Move a pointer for each transaction within a 24-hour period to calculate the cumulative total.
Check for duplicate accounts	Compare sorted account numbers or IDs using two pointers to find duplicates.

- **Comparative Graph:**