Ramdeobaba University, Nagpur
Department of Computer Science and Engineering
Session: 2025-26

Subject: Design and Analysis of Algorithms (DAA) Lab Project

III Semester

LAB PROJECT REPORT

Group Members with Roll number and Section:

Manaswi Wasu	**A6_B1_05**
Dimpal Maskey	**A6_B3_43**
Anupama Sanjeevan	**A6_B3_49**

TITLE:

GREEDY METHOD FOR REVENUE OPTIMIZATION (Knapsack Problem)

Objectives:

- To apply the **Greedy Algorithm** approach to a real-world resource allocation problem (Train Seat Booking).
- To model the seat allocation problem as a variation of the **Knapsack Problem**, where seats are the capacity (Weight) and total fare is the profit.
- To verify that the Greedy Method provides an **optimal revenue** solution based on the chosen priority (Highest Fare).
- To demonstrate how the greedy strategy of choosing the highest-value item (group) maximizes the overall profit within a fixed capacity.
- To build a user-friendly system for dynamic group entry, allocation, and cancellation.

Introduction:

This project focuses on using the Greedy Algorithm to solve the challenge of optimally allocating a limited resource—the total number of train seats (TOTAL_SEATS = 100)—to maximize the total revenue. The system handles booking groups, where each group requires a certain number of seats (Weight) and generates a total fare (Profit) based on their route. The **greedy approach** works by prioritizing and accepting groups based on the highest total fare they generate. By always choosing the group that offers the most revenue first, the

algorithm ensures the maximum possible total revenue is secured before the train's capacity is reached. The system also includes a robust **cancellation handler** that updates the total revenue and frees up seats.

Algorithms/Technique used:

The core technique is a Greedy Selection Algorithm applied to a constrained optimization problem (Knapsack).

Task 1: Greedy Seat Allocation (allocate_seats)

- **Step 1:** Start.
- **Step 2:** Initialize Capacity: Set the available seats to TOTAL_SEATS (e.g., 100).
- **Step 3:** Sort all incoming booking groups (requests) in **decreasing order of their total fare** (Greedy Choice). For ties, a secondary rule (e.g., favoring smaller groups) may be used.
- **Step 4:** Create an empty list ALLOCATED_GROUPS.
- **Step 5:** For each group in the sorted list, do:
    - a) If group.members \le current_total_seats then:
        - Add a group to ALLOCATED_GROUPS.
        - Update current_total_seats \leftarrow current_total_seats - group.members.
        - Update current_total_revenue \leftarrow current_total_revenue + group.total_fare.
    - b) Else:
        - Add a group to WAITING_GROUPS.
- **Step 6:** Return ALLOCATED_GROUPS, WAITING_GROUPS, and final revenue.
- **Step 7:** Stop.

Task 2: User Input & Group Management

- **Step 1:** Start.
- **Step 2:** User inputs group details: Route (e.g., AB), Number of Members (Seats).
- **Step 3:** Create a new Group object, automatically calculating its total fare.
- **Step 4:** Store the group in the list of all requests.
- **Step 5:** Call allocate_seats to run the optimizer and display results.
- **Step 6:** Stop.

Task 3: Cancellation Handler (handle_cancellation)

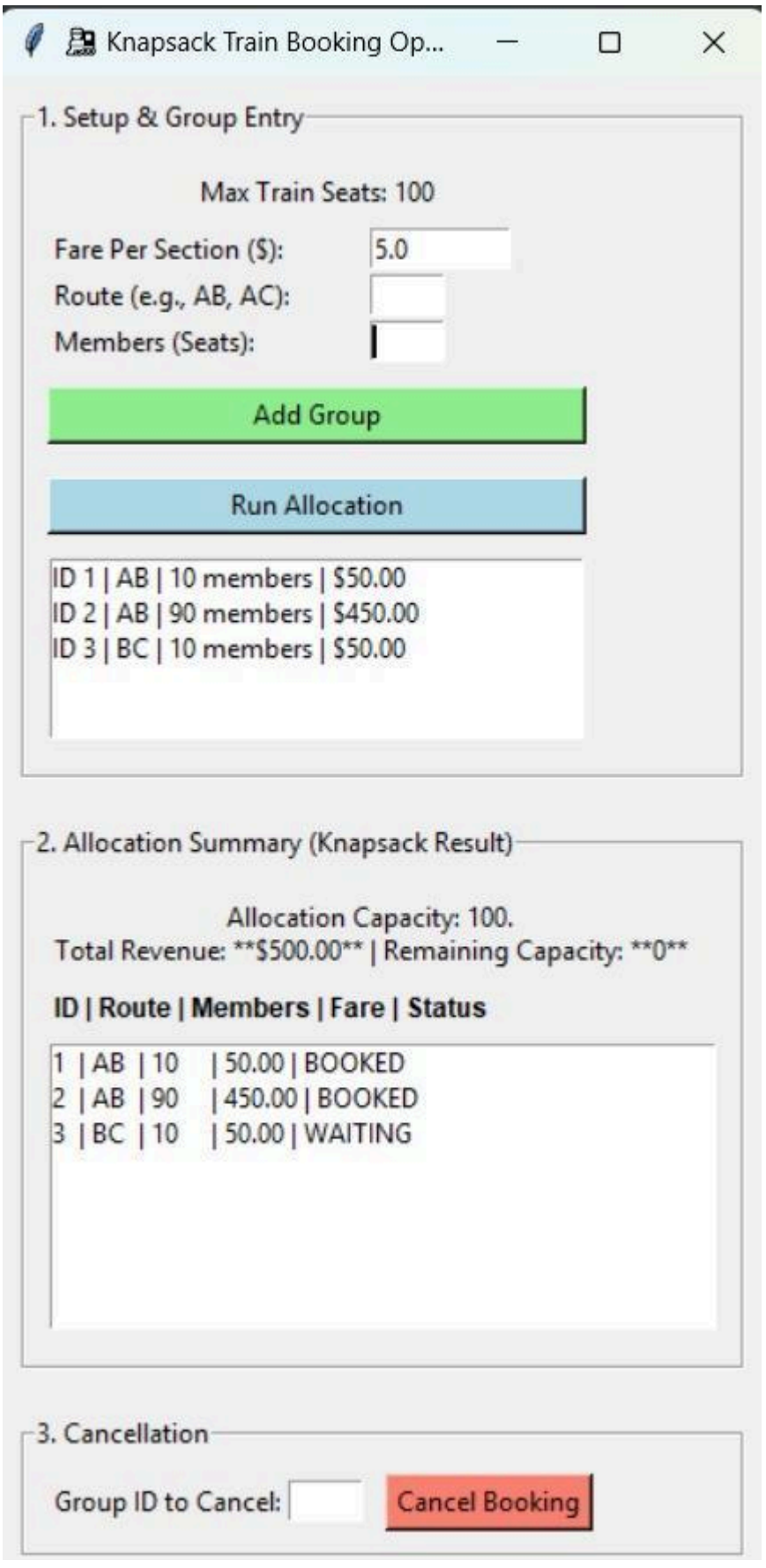
- **Step 1:** Start.
- **Step 2:** Read cancel_id.
- **Step 3:** Find the group with cancel_id.

- **Step 4:** Calculate refund_amount = fare \times$ (1 - CANCELLATION_FEE_PERCENTAGE).
- **Step 5:** Update state:
    - group.is_booked \leftarrow$ False.
    - current_total_seats \leftarrow current_total_seats + group.members.
    - current_total_revenue \leftarrow current_total_revenue - refund_amount.
- **Step 6:** Display status and new available seats/revenue.
- **Step 7:** Stop.

Time Complexity and its explanation:

Method	Time Complexity	Explanation
Greedy Allocation	O(n log n)	The dominant step is sorting n groups by fare, which takes $O(n \log n)$. The subsequent iteration (checking groups one-by-one) takes only $O(n)$. This makes it fast and scalable.
Brute Force (Conceptual)	O(2^n)	If a brute-force approach were used (like in the Event Scheduler), it would have to generate and check every possible combination of groups. This is computationally too slow (exponential) for practical booking systems.

Results:


Knapsack Train Booking Op...
1. Setup & Group Entry
Max Train Seats: 100
Fare Per Section ($):
5.0
Route (e.g., AB, AC):
Members (Seats):
Add Group
Run Allocation
ID 1 | AB | 10 members | $50.00
ID 2 | AB | 90 members | $450.00
ID 3 | BC | 10 members | $50.00
2. Allocation Summary (Knapsack Result)
Allocation Capacity: 100.
Total Revenue: **$500.00** | Remaining Capacity: **0**
ID | Route | Members | Fare | Status
1 | AB | 10 | 50.00 | BOOKED
2 | AB | 90 | 450.00 | BOOKED
3 | BC | 10 | 50.00 | WAITING
3. Cancellation
Group ID to Cancel:
Cancel Booking

UI and Output Discussion:

The application interface (ui_app.py) is built using tkinter and provides a simple way to interact with the booking logic.

- **Main Window (TrainBookingApp):** Sets up the layout for entering group details (Route, Members), displaying allocation results, and handling cancellations.
- **add_group:** This function takes the route and member count from the input fields, creates a new Group object (which calculates its own fare), and adds it to the list of pending groups.
- **run_allocation:** This is the main execution button. It calls the allocate_seats function from the logic file, which performs the greedy sorting and selection. It then updates the "Allocation Summary" listbox and the summary label to show the final revenue and remaining seats.
- **cancel_booking:** This takes a Group ID from the input field, calls the handle_cancellation function to update the logic, and then automatically re-runs run_allocation to refresh the results display with the newly freed seats and updated revenue.

Example Scenario (Conceptual):

- Group 1: 50 members, Fare: $1000
- Group 2: 20 members, Fare: $800
- Group 3: 40 members, Fare: $900
- TOTAL_SEATS = 100

1. **Sorted by Fare:** G1 ($1000), G3 ($900), G2 ($800).
2. **Allocate:** G1 (50 seats) is booked. Remaining seats: 50. Revenue: $1000.
3. **Allocate:** G3 (40 seats) is booked. Remaining seats: 10. Revenue: $1900.
4. **Allocate:** G2 (20 seats) is put on waiting list (20 > 10).
5. **Final Revenue: $1900.**

Conclusion: The Greedy method successfully selected the highest-value combination of groups (G1 and G3) to maximize revenue, providing an optimal solution for this specific problem.

Conclusion and future scope:

Conclusion :

The Train Booking Optimizer successfully applies the Greedy Algorithm to the Knapsack-like problem of seat allocation. By employing the greedy strategy of prioritizing groups by the highest total fare, the system efficiently maximizes the total revenue generated from the

fixed train capacity. The implementation is highly efficient with an O(n log n) time complexity, making it suitable for handling large numbers of booking requests quickly.

Future Scope

- **Dynamic Pricing Integration:** Instead of a fixed fare_per_section, integrate a dynamic system where the fare can change based on demand or time remaining before departure, influencing the 'profit' calculation.
- **Segment-based Allocation (Bin-Packing/Advanced Knapsack):** Currently, it uses a simple capacity check. A future version could track capacity per segment (e.g., A-B, B-C, C-D), requiring a more complex allocation algorithm to ensure no individual segment is overbooked.
- **Optimized Reallocation on Cancellation:** When a group cancels, automatically check the WAITING_GROUPS list to see if any high-value groups can now be booked with the newly freed seats.