CS 2009 – Design and Analysis of Algorithms

Semester Project

Title	Project & Rubric
Academic Year	2025
Semester	Spring
Assignment Title	-
Issue Date	As per GCR
Submission Date	As per GCR
Plagiarism Check	■ Yes ☐ No
Viva Based	■ Yes ☐ No
Instructor Name	Dr. Shamsa Abid, Dr. Sahar Ajmal, Muhammad Adeel
Sections	**All sections** (CS & SE)
Total Marks	100

Student Declaration

I solemnly declare that the work submitted for this assignment is my own and research sources are fully acknowledged. I fully understand the consequences of plagiarism and I understand that making a false declaration is a form of malpractice.

1. **Originality of Work:** All submissions must reflect my own understanding and work. Copy-pasting from online sources, textbooks, peers or previous assignments is strictly not followed, in such a case a 50% penalty can be applied on my assignment if proven.
2. **Use of Tools and References:** My work does not involve unauthorized assistance, including AI-generated content (e.g., ChatGPT) or plagiarism from other sources.

Note: GPT can be used for helping or understanding things, but a straight copy paste will lead to a 0, all content can be checked for AI generation and shall not be accepted in such case.

Instructions: Using & Submitting Your Project Self-Evaluation Rubric

This rubric is designed to help you critically evaluate your own project work before final submission. Follow these steps:

1. **Access the Rubric:** Locate and download the specific Excel self-evaluation rubric file for your project assignment from Google Classroom.
2. **Understand the Rubric:** Open the Excel file. Carefully read the "How to Use" instructions, the descriptions of the test cases, the criteria descriptions, and the detailed scoring guide under each project description.
3. **Test Your Work:** Apply the test cases described in the rubric (or create equivalent ones if instructed) to your project code. Observe how your program performs.
4. **Evaluate and Score:** Based on your project's performance on the tests and the descriptions in the scoring guide columns (Needs Improvement, Satisfactory, Good, Excellent), determine and enter your honest self-score for each criterion in the far-right column (usually labeled "Self-Score").
5. **Save Your Work:** Save the Excel file on your computer, ensuring your self-scores are saved.

6. **Submit:** Return to the project assignment page in Google Classroom and attach/upload your saved Excel rubric file. Submit the assignment by the deadline.

This self-evaluation helps you identify areas for improvement and provides insight into your understanding of the project requirements and algorithms.

Project 1: Smart Drone Delivery

Rubric Instructions

Purpose: This rubric helps you test and evaluate your own program for the Smart Drone Delivery Planner project. By applying these tests and assessing your work against the criteria, you can identify areas for improvement before final submission.

How to Use:

1. Run your program using the provided test cases (or create similar ones based on the descriptions below).
2. Carefully observe the output for each test case.
3. Attach the screenshots against each test case and submit this file carefully along with your project.
4. Evaluate your program against each criterion in the scoring guide below based on its performance on the test cases and your code structure.
5. Use the scoring guide to assign yourself points for each criterion.
6. Total your points and refer to the interpretation section.

Test Cases for Self-Evaluation:

Apply your program to at least these types of scenarios:

1. **Basic Functionality Test:**
   - **Input:** A small map (e.g., 3-4 cities, simple routes), 2-3 packages (varying weights/values), 1 drone with generous limits (high max weight, high max distance).
   - **Check For:** Does the program read input? Find a path? Select packages? Plan a route? Produce *any* output? Is the output format correct?
   - **Expected Outcome:** The drone should deliver some packages. The route should be simple.
2. **Constraint Handling Test:**
   - **Input:** A map with more cities (5-7), 5-7 packages (include some very heavy ones, some very valuable ones, some requiring long routes), 1-2 drones with tighter, realistic limits.
   - **Check For:** Does the program respect the drone's max weight? Does it respect the drone's max distance per trip? Does it choose packages considering value *within* constraints? Does it generate a route that fits the distance limit?
   - **Expected Outcome:** The drone might not be able to take all packages in one trip. Packages exceeding weight or distance limits for a single trip should be left behind or planned for a separate trip.
3. **Optimization & Edge Case Test:**
   - **Input:** A map with potential dead ends or inefficient routes, a mix of packages requiring trips of varying complexity, possibly a scenario where no package can be delivered within a drone's constraints, or a map with disconnected sections.
   - **Check For:** Does the program try to maximize value *and* minimize distance for the selected packages? How does it handle scenarios where no valid trip is possible? Does shortest path still work on complex maps? (Optional: Does it handle the bonus feature if implemented?)
   - **Expected Outcome:** The plan should seem reasonably efficient (high value packages preferred, route seems logical). The program should gracefully handle impossible scenarios (e.g., report "No feasible trips found").

Interpreting Your Score:

- **80-100 (Excellent/Good):** Your program is largely complete, correct, and meets requirements well. Focus on refining edge cases, improving code quality, and perhaps enhancing optimization or documenting further.
- **60-79 (Satisfactory):** Your program has the core functionality but may have bugs, fail certain tests, or not be well-optimized. Review the criteria where you scored lower. Re-run tests, debug, and refine your algorithms and logic.
- **40-59 (Needs Improvement):** Significant parts of the project are incomplete or incorrect. Identify the specific criteria and requirements that are not met (especially Completeness, Accuracy, and Algorithm Logic). Focus your efforts on implementing missing parts and fixing fundamental errors.
- **Below 40 (Significant Issues):** Your program has fundamental problems and likely does not produce meaningful results. Revisit the project requirements and core algorithm concepts (shortest path, package selection logic, integration). Break down the problem into smaller parts and build up functionality step-by-step, testing each part thoroughly.

Use this self-evaluation as a roadmap to improve your work before the final submission.

Introduction:

Imagine a company that uses drones to deliver packages to different cities. Your job is to create a computer program that helps plan these deliveries efficiently.

Drones have limits:

- They can only carry a certain maximum weight.
- They can only fly a certain maximum distance before needing to return or recharge.

Packages also have details:

- Each package has a weight.
- Each package has a value or priority (how important it is to deliver).
- Packages need to be delivered to specific cities.

Main Goal:

Your program must create a delivery plan for the company's drones. This plan should decide:

1. **Which packages** each drone should carry on a trip.
2. **Which cities** each drone should visit and in what order (the route).

The plan should try to be "good" by:

- Maximizing the value or priority of the packages delivered.
- Minimizing the total travel distance or time for the drones.
- Making sure drones do not carry too much weight or fly too far.

Information You Will Give to the program (Inputs):

- **Map Data:** A list of cities and the routes between them. For each route, you'll know the distance and maybe the travel time. (This is like a graph where cities are points and routes are lines connecting them).
- **Package Data:** For packages waiting at the warehouse or in cities: their weight, their delivery priority/value, and which city they need to go to.

- **Drone Data:** For each drone: its maximum carrying weight and its maximum travel distance per trip. All drones start at a central location (warehouse).

What Your Program Must Produce (Outputs):
For each planned drone trip, your program should report:

- A list of the packages the drone carried (including their destination city, weight, and value/priority).
- The total value/priority points of the packages delivered on that trip.
- The route the drone took (the sequence of cities visited).
- The total distance traveled or time taken for the trip.

Project Requirements:
To complete this project successfully, your program must:

1. **Read and Understand Input:** Load and store the information about cities, routes (distances/times), packages (weight, value, destination), and drone limits (max weight, max distance).
2. **Find Shortest Paths:** Implement a way to calculate the shortest path (based on distance or time) between any two cities on the map. A common method for this is Dijkstra's algorithm.
3. **Select Packages:** Create a method for deciding which set of packages a drone should carry on a single trip. This decision must consider:
    - The drone's maximum weight capacity.
    - The value or priority of the packages (try to maximize this).
    - *(Advanced: Consider the destinations of the packages in relation to potential routes).* A common method for this type of selection problem is related to the Knapsack problem.
4. **Plan Routes:** Determine the sequence of cities a drone will visit to deliver its selected packages. This route must:
    - Start at the warehouse.
    - Visit the necessary cities for the selected packages.
    - Ensure the total travel distance does not exceed the drone's maximum range.
    - *(Consider efficiency: try to make the route distance/time short using your shortest path calculations).*
5. **Combine Package Selection and Routing:** Integrate the package selection (Requirement 3) and route planning (Requirement 4) so they work together to create a valid and efficient delivery trip plan. This is the core challenge – choosing packages might affect the best route, and the possible routes affect which packages can be delivered within the distance limit.
6. **Generate Output:** Clearly present the results for each planned drone trip, including the list of packages carried, total value/priority, the route taken, and the total distance/time, as specified in the "Outputs" section above.
7. **(Optional Bonus):** Implement extra features like handling dynamic changes (e.g., road closures simulated by removing or increasing the cost of routes) or allowing drones to recharge at specific cities to extend their range.

Project 2: Smart Traffic Management System

Rubric Instructions:

Purpose: This rubric helps you test and evaluate your own program for the Smart Traffic Management System project. By applying these tests and assessing your work against the criteria, you can identify areas for improvement before final submission.

How to Use:

1. Design or obtain input data sets that match the test cases described below.
2. Run your program using these data sets.
3. Carefully observe the output for each test case (traffic light timings, suggested routes).
4. Attach the screenshots against each test case and submit this file carefully along with your project.
5. Evaluate your program against each criterion in the scoring guide below based on its performance on the test cases, the clarity of its output, and the structure of your code.
6. Use the scoring guide to assign yourself points for each criterion.
7. Total your points and refer to the interpretation section.

Test Cases for Self-Evaluation:
Apply your program to at least these types of scenarios:

1. **Basic Functionality Test:**
    - **Input:** A small city map (e.g., 2-3 intersections, simple connections), static (non-changing) traffic data showing light traffic on all roads.
    - **Check For:** Does the program read map and traffic data? Model the city as a graph? Find a basic shortest/fastest path? Produce *any* traffic light timings? Produce *any* suggested routes? Is the output format correct?
    - **Expected Outcome:** The program should produce some valid timings and simple routes.
2. **Dynamic Traffic & Rerouting Test:**
    - **Input:** A slightly larger map (e.g., 4-6 intersections), traffic data that simulates a sudden traffic jam or accident on one key road. Include a scenario where vehicles need to travel through or past this congested area.

- **Check For:** Does the program correctly adjust path costs based on traffic data? Does the rerouting logic suggest an *alternative* path that avoids the congestion? Does the traffic light logic show *some* attempt to adapt timings based on traffic flow changes?
- **Expected Outcome:** Suggested routes should change to avoid the jam. Traffic light timings might favor busier routes entering or leaving the congested area.

3. **Complexity & Optimization Test:**
   - **Input:** A larger or more complex map (e.g., multiple paths between points, potential bottlenecks), traffic data with multiple congested points or complex flow patterns across several intersections. Include a scenario involving multiple vehicles needing routing (stressing Req 6).
   - **Check For:** Does the fastest path algorithm work efficiently on larger graphs with varying costs? Does the traffic light logic attempt to *optimize* flow across multiple intersections simultaneously? Does the rerouting logic handle multiple affected vehicles reasonably? How does your strategy for Req 6 perform or plan? Does the integrated system (Req 7) show coordinated behavior?
   - **Expected Outcome:** The system should attempt to balance traffic flow, reroute vehicles effectively around multiple issues, and the overall plan should aim to reduce total travel time compared to just using static routes.

Interpreting Your Score:

- **80-100 (Excellent/Good):** Your program is largely complete, correct, and meets requirements well. Focus on refining edge cases, improving efficiency, and perhaps enhancing optimization or documentation.
- **60-79 (Satisfactory):** Your program has the core functionality but may have bugs, fail certain tests, or not be well-optimized. Review the criteria where you scored lower. Re-run tests, debug, and refine your algorithms and logic, especially how they interact with live data.
- **40-59 (Needs Improvement):** Significant parts of the project are incomplete or incorrect. Identify the specific criteria and requirements that are not met (especially Completeness, Accuracy, and Algorithm Logic). Focus your efforts on implementing missing parts and fixing fundamental errors in your algorithms and data handling.
- **Below 40 (Significant Issues):** Your program has fundamental problems and likely does not produce meaningful results. Revisit the project requirements and core algorithm concepts (graph modeling, pathfinding with costs, the logic for lights and rerouting, integration). Break down the problem into smaller parts and build up functionality step-by-step, testing each part thoroughly.

Use this self-evaluation as a roadmap to improve your work before the final submission. Good luck!

Introduction:

Imagine a modern city ("smart city") where traffic lights and roads have sensors. These sensors collect information *right now* about how many cars are on the road, where traffic jams are happening, and if there are any accidents. Your job is to create a computer program that uses this live information to make traffic flow better.

Main Goal:

Your program should help manage city traffic by deciding how to:

1. **Control Traffic Lights:** Change how long traffic lights stay green or red at different intersections (road crossings).
2. **Guide Vehicles:** Suggest better routes for cars to take to avoid traffic jams.

The overall aim is to reduce travel time and prevent heavy traffic jams.

How Your Program Should Work (Using Algorithms):

Your program will need to use different types of algorithms (problem-solving methods) learned in class:

- **Represent the City:** Think of the city map as a *graph*, where intersections are points (nodes) and roads are lines (edges).
- **Find Best Routes:** Use methods like *Shortest Path algorithms* to find the quickest way for cars to get from one place to another, considering current traffic. You might also think about how much traffic a road can handle (*Maximum Flow* ideas).
- **Manage Traffic Lights:** Develop a smart way to schedule the traffic lights. This schedule should try to balance traffic flow from all directions. *Dynamic Programming* methods can be useful here.
- **Make Quick Decisions:** Use methods like *Greedy Algorithms* to quickly suggest new routes for cars when traffic conditions change suddenly (like after an accident).
- **Handle Complex Routing:** Planning routes for many vehicles efficiently is complex (like the *Vehicle Routing Problem* or *Traveling Salesman Problem*). Your program should include a strategy to handle this.

Information Your Program Will Use (Inputs):

- **Live Traffic Data:** Information from city sensors (e.g., number of cars passing, average speed, locations of traffic jams or accidents).
- **City Map Data:** Information about where intersections and traffic lights are located, and the roads connecting them.
- **(Optional):** Information about different types of vehicles (e.g., buses, emergency cars).

What Your Program Must Produce (Outputs):

- **Traffic Light Timings:** Updated schedules for traffic lights (e.g., how many seconds green for each direction).
- **Suggested Vehicle Routes:** The best paths for vehicles to take based on current conditions.
- **(Optional):** Predictions about where traffic jams might happen soon.

Project Requirements:

To complete this project successfully, your program must:

1. **Process Input Data:** Read and understand the incoming live traffic data and the city map information.
2. **Model the City Network:** Create a graph representation of the city's roads and intersections.
3. **Calculate Fastest Routes:** Implement an algorithm (like Dijkstra's or A*) to find the shortest/fastest path between locations, considering current traffic data (e.g., treating roads with heavy traffic as having a higher 'cost' or longer travel time).
4. **Develop Traffic Light Control Logic:** Create a system (potentially using Dynamic Programming principles) that adjusts traffic light timings at intersections to help balance traffic flow based on the input data.
5. **Develop Vehicle Rerouting Logic:** Create a system (potentially using Greedy approaches or shortest path updates) that can suggest alternative routes for vehicles when their current path becomes blocked or very slow due to live traffic conditions.
6. **Address Vehicle Routing Complexity:** Implement a strategy to plan or suggest routes for multiple vehicles, considering the complexity involved.
7. **Integrate Components:** Ensure that the traffic light control and vehicle rerouting logic work together using the live data and the graph model.
8. **Generate Clear Outputs:** Produce the required outputs, such as updated traffic light timings and suggested vehicle routes, in a clear and understandable format.

Project 3: Plagiarism Detector

Rubric Instructions

Purpose: This rubric helps you test and evaluate your own program for the Plagiarism Detector project. By applying these tests and assessing your work against the criteria, you can identify areas for improvement before final submission.

How to Use:

1. Prepare sets of student code submissions according to the test cases described below.
2. Run your program using these submission sets as input.
3. Carefully observe the output: the identified clusters of similar submissions and the representatives chosen for each.
4. Attach the screenshots against each test case and submit this file carefully along with your project.
5. Evaluate your program against each criterion in the scoring guide below based on its performance on the test cases, the correctness of the output, and the structure of your code.
6. Use the scoring guide to assign yourself points for each criterion.
7. Total your points and refer to the interpretation section.

Test Cases for Self-Evaluation:

Prepare sample code files (e.g., simple functions or programs) for these scenarios:

1. **Basic Functionality Test:**
    - **Input:**
        - File A: Original simple code (e.g., calculates sum).
        - File B: Exact copy of File A.
        - File C: Slightly modified copy of File A (e.g., variable names changed).
        - File D: Completely different code (e.g., calculates product).
        - Metadata for A, B, C, D.
    - **Check For:** Does the program parse all files? Does Rabin-Karp find *many* matches between A & B, A & C, B & C? Does it find *few* or *no* matches between A & D, B & D, C & D? Is the similarity score high for similar pairs, low for different ones? Do A, B, C form a cluster? Is D in its own cluster (or no cluster if threshold is high)? Are representatives chosen from the A/B/C cluster? Are metadata retrievable via B+ Tree?
    - **Expected Outcome:** A, B, C are clustered. D is separate. Representatives are from A, B, or C.
2. **Constraint & Edge Case Test:**
    - **Input:**
        - File E: Original code (medium size).
        - File F: Copy of E with comments added/removed, whitespace changes.
        - File G: Copy of E with large block of code moved/reordered.
        - File H: Code in a different programming language.
        - File I: Empty file.
        - A large number of submissions (e.g., 50+ entries) to test B+ Tree efficiency.
        - Metadata for all.
    - **Check For:** Does parsing handle comments/whitespace? Does Rabin-Karp handle comments/whitespace changes (ideally yes if tokenized well)? How does RK perform if code blocks are moved (might miss some matches)? Does it correctly identify H as unrelated? Does it handle I gracefully? Does the B+ Tree handle many entries efficiently (e.g., lookup times)? What happens if the similarity threshold is adjusted?
    - **Expected Outcome:** E, F clustered. G might be clustered depending on implementation robustness. H, I are separate. B+ Tree operations are fast.

3. **Algorithm Integration & Complexity Test:**
    - **Input:** A larger set of submissions (e.g., 20+) with multiple small groups of similar files (e.g., Group 1: J, K, L; Group 2: M, N; Others unique), some files with low-level similarity across groups, files specifically designed to test Greedy selection criteria (e.g., one file highly similar to one other, another file moderately similar to several others). Test with varying graph similarity thresholds.
    - **Check For:** Does the graph build correctly with weighted edges based on similarity? Do BFS/DFS correctly identify *all* expected clusters? Does the Greedy algorithm select a representative that makes sense based on your criteria (e.g., highest average similarity to others in the cluster)? Do all components work together smoothly? (Optional: If real-time implemented, add a new file and see if clusters update).
    - **Expected Outcome:** Correct clusters identified. Representative selections appear reasonable based on your Greedy strategy.

Scoring Guide (Maximum Total: 100 Points)

Interpreting Your Score:

- **80-100 (Excellent/Good):** Your program is largely complete, correct, and meets requirements well. Focus on refining edge cases, improving efficiency (especially for RK and B+Tree), and potentially enhancing the Greedy selection logic or documentation.
- **60-79 (Satisfactory):** Your program has the core functionality but may have bugs, fail certain tests, or not implement algorithms optimally. Review the criteria where you scored lower. Re-run tests, debug, and refine your algorithm implementations and how they handle different code variations. Ensure integration is solid.
- **40-59 (Needs Improvement):** Significant parts of the project are incomplete or incorrect. Identify the specific criteria and requirements that are not met, focusing especially on implementing the required algorithms (RK, Graph, BFS/DFS, Greedy, B+Tree) and getting them to work together.
- **Below 40 (Significant Issues):** Your program has fundamental problems and likely does not produce meaningful results. Revisit the project requirements and core algorithm concepts. Break down the problem into smaller parts (parsing, RK, similarity, graph, clustering, greedy, B+Tree), implement and test each part thoroughly before integrating.

Use this self-evaluation as a roadmap to improve your work before the final submission. Good luck!

Introduction:

This project is about building a system to automatically find copied code (plagiarism) in programming assignments. Imagine many students submitting their code online. We need a fast way to compare these submissions and find ones that look too similar.

Main Goal:

Your program should:

1. Quickly compare newly submitted code against other submitted code.
2. Identify groups of student assignments that are very similar to each other, suggesting potential copying.
3. Help teachers focus on the most suspicious cases.

How Your Program Should Work (The Steps):

- **Understand the Code:** First, your program needs to read the student's code files. It should break down the code into smaller, basic pieces or sequences. This makes comparison easier.
- **Find Matching Pieces:** Use a fast algorithm (specifically, the **Rabin-Karp** method) to find identical or very similar small pieces of code between different assignments.
- **Connect Similar Assignments:** Imagine each assignment is a point. Draw connections (lines) between assignments that share similar code pieces. If two assignments share *many* similar pieces, the connection is stronger (has a higher "similarity score"). This creates a *similarity graph*.
- **Find Groups of Similar Code:** Use graph searching methods (like **BFS or DFS**) to explore the connections in your similarity graph. Find groups (clusters) of assignments that are strongly connected, meaning they likely share a lot of code.
- **Pick Examples for Review:** Within each group of similar assignments, use a smart method (a **Greedy algorithm**) to choose one or two assignments that best represent the similarity in that group. This helps teachers quickly see the potential problem without checking every single file in the group.
- **Store Student Information:** Keep track of information about each submission (like student ID, time submitted) using an efficient data storage method (a **B+ Tree**) so you can look it up quickly when needed.

Optional Advanced Feature:

- Make the system work in "real-time": As soon as a student submits code, immediately compare it and update the similarity graph and groups.

What Your Program Must Produce (Outputs):

- A list of groups (clusters) of assignments that are highly similar.
- For each group, identify the representative assignment(s) selected for review.
- (Optional) Similarity scores between pairs of files.

Algorithms Combined:

- String Matching: Rabin-Karp for detecting code similarities.
- Graph Algorithms: BFS/DFS for clustering similar submissions.
- Greedy Algorithms: Selection of representative codes .
- Data Structures: B+ Trees for metadata storage.
- Sorting: Merge sort for ranking similarity scores.

Project Requirements:

To complete this project successfully, your program must:

1. **Parse Code:** Read student code submissions and convert them into a sequence of basic pieces (tokens or similar representation) suitable for comparison.
2. **Implement Rabin-Karp:** Use the Rabin-Karp algorithm to efficiently find and count matching sequences (similar code snippets) between pairs of parsed code submissions.
3. **Calculate Similarity:** Define and calculate a "similarity score" between pairs of submissions based on the matches found by Rabin-Karp.
4. **Build Similarity Graph:** Create a graph where each submission is a node, and an edge exists between two nodes if their similarity score is above a certain threshold (value). The edge weight can represent the similarity score.
5. **Cluster Submissions:** Implement BFS or DFS to traverse the similarity graph and identify connected components or clusters of similar submissions.
6. **Implement Greedy Selection:** Design and implement a Greedy algorithm to select one or more "representative" submissions from each identified cluster based on criteria like highest average similarity within the cluster.
7. **Implement B+ Tree Storage:** Use a B+ Tree (or simulate its functionality if library use is restricted) to store and quickly retrieve metadata associated with each submission (Student ID, timestamp, file name).
8. **Generate Output:** Clearly report the identified clusters (groups of similar submission IDs) and the representative submission(s) selected from each cluster.
9. **(Optional Requirement):** Implement the real-time processing capability, allowing the system to update the graph and clusters incrementally as new submissions arrive.

Project 4: Smart Trash Truck Routing

Rubric Instructions

Purpose: This rubric helps you test and evaluate your own program for the Smart Trash Truck Routing project. By applying these tests and assessing your work against the criteria, you can identify areas for improvement before final submission.

How to Use:

1. Prepare input data sets (bin locations, road network, bin fill levels, truck capacities) corresponding to the test cases described below.
2. Run your program using these data sets.
3. Carefully observe the output: the planned routes for each truck, the sequence of bins, and the calculated metrics (distance/time/fuel).
4. Attach the screenshots against each test case and submit this file carefully along with your project.
5. Evaluate your program against each criterion in the scoring guide below based on its performance on the test cases, the correctness and feasibility of the routes, and the structure of your code.
6. Use the scoring guide to assign yourself points for each criterion.
7. Total your points and refer to the interpretation section.

Test Cases for Self-Evaluation:

Prepare input data sets for at least these types of scenarios:

8. **Basic Functionality Test:**
    1. **Input:** A small city map (e.g., 5-7 bin locations, simple connections), all bins have low-to-medium fill levels, 1 truck with high capacity.
    2. **Check For:** Does the program read input? Model the city as a graph? Implement Kruskal's, Kadane's adaptation/clustering, DP for sequencing, and B+ Tree? Produce *any* routes? Is the output format correct? Are all bins included in routes?
    3. **Expected Outcome:** A single route covering all bins, likely relatively simple. All required algorithms should show some basic function.
9. **Constraint Handling & Prioritization Test:**
    1. **Input:** A map with more bin locations (e.g., 10-15), includes several bins marked as "nearly full" (high priority), 1-2 trucks with limited capacity.
    2. **Check For:** Does the program correctly prioritize the nearly full bins? Are these high-priority bins included in the routes, potentially early? Are truck capacities respected (not overloaded)? Does the routing split bins correctly between trucks if one truck can't handle them all?

3. **Expected Outcome:** Routes should include the high-priority bins. Trucks should not exceed capacity. Multiple routes/trucks might be used.

10. **Algorithm Integration & Optimization Test:**
    1. **Input:** A larger map (e.g., 20+ bin locations) with some clear clusters of bins, varying fill levels, multiple trucks. Include a scenario where Kadane's adaptation/clustering should identify distinct groups, and where optimal sequencing within a group via DP is critical for efficiency.
    2. **Check For:** Is Kadane's adaptation/clustering identifying reasonable clusters? Is the DP algorithm used correctly to sequence visits *within* selected groups or for smaller route segments? How well does the overall route planning integrate these pieces? Does the plan seem reasonably efficient (minimize distance/time) compared to a naive approach? (Optional: If dynamic updates implemented, simulate a bin filling up during a trip and see if the route changes).
    3. **Expected Outcome:** Clusters are identified. Routes show efficient sequencing within parts of the trip. The combined routes cover necessary bins (including priorities) and aim for overall efficiency while respecting constraints.

Scoring Guide (Maximum Total: 100 Points)

Interpreting Your Score:

11. **80-100 (Excellent/Good):** Your program is largely complete, correct, and meets requirements well. Focus on refining edge cases, improving efficiency (especially for DP and B+Tree on larger inputs), and potentially enhancing the optimization or documentation.
12. **60-79 (Satisfactory):** Your program has the core functionality but may have bugs, fail certain tests, or not implement algorithms optimally. Review the criteria where you scored lower. Re-run tests, debug, and refine your algorithm implementations, focusing on how they are integrated to build routes (Req 7). Ensure constraints and priorities are always met.
13. **40-59 (Needs Improvement):** Significant parts of the project are incomplete or incorrect. Identify the specific criteria and requirements that are not met, focusing especially on implementing the required algorithms (Kruskal's, Kadane adaptation/Clustering, DP, B+Tree, Prioritization) and getting them to work together (Integration, Req 7).
14. **Below 40 (Significant Issues):** Your program has fundamental problems and likely does not produce meaningful results or valid routes. Revisit the project requirements and core algorithm concepts. Break down the problem into smaller parts (graph modeling, input, implementing each algorithm, integrating them), implement and test each part thoroughly before combining them.

Use this self-evaluation as a roadmap to improve your work before the final submission. Good luck!

Introduction:

Cities need to collect trash efficiently. This project is about creating a computer program to plan the best routes for trash collection trucks.

The Problem:

- Trash bins in the city fill up at different speeds. Some become full very quickly.
- Trucks need to visit bins before they overflow.
- Traffic conditions change, making travel times unpredictable.
- We want trucks to use less fuel and take less time.

Main Goal:

Your program must create daily route plans for trash trucks. The plans should:

1. Decide which bins a truck should visit on a trip.
2. Decide the best order to visit those bins.
3. Make sure important bins (nearly full ones) are collected on time.
4. Try to minimize the total distance traveled or fuel used by the trucks.
5. Consider truck limits (like how much trash they can hold).

How Your Program Should Work (The Steps & Algorithms):

- **Map the City:** Represent the city as a network (a *graph*). Each bin location is a point (node), and the roads between them are lines (edges). Each road has a cost (distance or estimated travel time).
- **Basic Connection Plan:** Use **Kruskal's algorithm** to find a basic, efficient way to connect all bin locations together. This helps in initial planning.
- **Find Busy Areas:** Identify areas in the city where many bins are close together (clusters). You can use an algorithm like **Kadane's** (adapted for this purpose) to find these dense areas, which might need frequent collection.
- **Order Visits Within Busy Areas:** Once a truck is in a busy cluster, figure out the best order to visit the bins *within that cluster* to minimize travel time. You can use **Dynamic Programming** ideas (similar to a method called *Matrix Chain Multiplication*) for this step.
- **Store Bin Information:** Keep track of information for each bin (like its location, how full it is) using an efficient storage method (like a **B+ Tree**) so you can quickly find and update bin status.
- **(Optional) Use Live Updates:** If sensors on bins provide real-time fullness data, use this information to update the routes dynamically.

Information Your Program Will Use (Inputs):

- A list of all bin locations.
- Map data showing roads connecting the bins and their distances/travel times.

- Information about each bin (e.g., its capacity, current fullness level - which might be updated).
- Information about the trucks (e.g., their capacity).

What Your Program Must Produce (Outputs):

- Optimized collection routes for each truck, showing the sequence of bins to visit.
- Estimated total distance, travel time, or fuel consumption for the planned routes.
- Confirmation that priority bins (nearly full) are included in routes.

Project Requirements:

To complete this project successfully, your program must:

1. **Model the City:** Create a graph representation where nodes are bin locations and edges are roads with associated costs (distance/time).
2. **Process Inputs:** Load data about bin locations, road network, bin status (fill levels, priority), and truck capacities.
3. **Implement Kruskal's Algorithm:** Use Kruskal's to generate a Minimum Spanning Tree (or similar structure) of the bin network for foundational route planning or analysis.
4. **Implement Kadane's Algorithm (Adapted):** Apply an adaptation of Kadane's algorithm (or another suitable method) to identify high-density clusters of bins based on their location or priority.
5. **Implement DP for Sequencing:** Use Dynamic Programming to determine the optimal sequence for visiting a selected set of bins (especially within clusters) to minimize local travel cost.
6. **Implement B+ Tree:** Use a B+ Tree (or simulate its key functionalities) to store and efficiently query bin data (location, fill level, ID).
7. **Integrate Algorithms:** Combine the outputs of the different algorithms (Kruskal's, Kadane's, DP) to generate complete, optimized collection routes for trucks, respecting truck capacity.
8. **Prioritize Bins:** Ensure the route planning prioritizes bins based on urgency (e.g., high fill level). Sorting bin priorities (e.g., using Heap Sort initially) can help.
9. **Generate Clear Outputs:** Produce the final routes for each truck and relevant performance metrics (total distance/time/fuel saved estimate) clearly.
10. **(Optional Requirement):** Add functionality to dynamically adjust routes based on simulated real-time updates of bin fill levels.