SDG 9: Logistic Scheduling and Tracking Delivery system

Presented by: The RouteLink!

Final Project/Exam: DSA for Sustainable Development

Member's Name:

Aclan, Trisha Mae

Buenaventura, Hubert

Lamayo, Justine Kurt

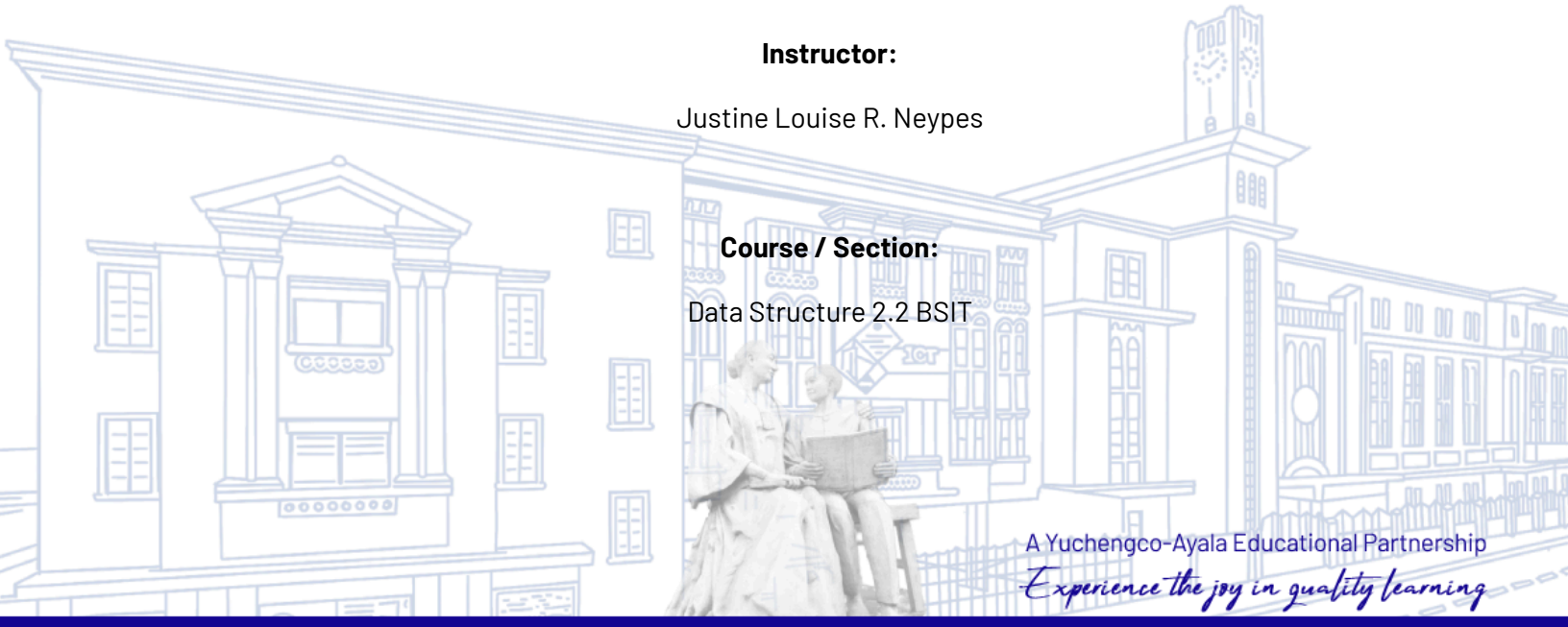
Riza, Christina Alexandra

Instructor:

Justine Louise R. Neypes

Course / Section:

Data Structure 2.2 BSIT



I. Introduction (Kurt, Trisha)

(Trisha) 1.1. Project Overview & UN SDG Target

The aforementioned project is a significant contributor to the UN SDG 9 which is focused on the promotion of innovation and infrastructure development. The digitization of delivery systems is getting more and more important as logistics are significantly associated with business and daily life, particularly when online shopping and same-day delivery services are getting more popular. Our solution aids this transformation by accelerating technological innovation, boosting operational performance, and providing the foundation for more resilient and modern logistics processes. Moreover, it will particularly address the SDG 9.1 goal of enhancing transport-related infrastructure and significantly contribute to the SDG 9.c impact area of increased access to information and communication technologies. The system will eventually make logistical activities significantly more organized, better connected, and less difficult to manage.

(Kurt) 1.2. Problem Statement (What real-world problem does the app solve?)

The current state of logistics, especially for small to medium-scale operations, is often plagued by inefficiency, lack of transparency, and poor organization. Specifically, we see three major real-world problems that our application solves:

1. **Inefficient Manual Scheduling and Dispatch:** Without a structured system, delivery scheduling and vehicle dispatching are often manual, leading to errors, suboptimal routes, wasted time, and increased fuel costs. This results in **delayed deliveries** and an overall **high operational cost** for the business.
2. **Lack of Real-Time Tracking and Transparency:** Customers and management often lack visibility into the current status and location of a parcel. This leads to **frequent inquiries, customer frustration**, and an inability for managers to proactively address bottlenecks or delays in the delivery process.
3. **Difficulty in Data-Driven Management:** Current methods make it difficult to collect, organize, and analyze delivery data (e.g., successful routes, common delays, delivery times). This absence of data prevents businesses from making **informed decisions** to improve their service and scale their operations effectively.

Our Logistic Delivery System solves these issues by providing a digitized, organized, and optimized platform for efficient parcel management, real-time status tracking, and performance reporting.

II. Requirements & Analysis (Riza, Kurt, Hubert)

(Kurt) 2.1. Functional Requirements and Non-Functional Requirements (List of features, e.g., FR1, FR2)

A. Functional Requirements (FR)

Functional requirements define what the system *must do* to support the users' tasks.

ID	Feature Description
FR1	Parcel Registration: The system must allow a user to register a new delivery by providing the unique ID, sender name, receiver name, and destination.
FR2	Delivery Scheduling: The system must automatically add new parcels to the Schedule Queue (FIFO) for dispatch based on arrival time.
FR3	Status Update: The system must allow authorized personnel to update a delivery's status (Pending, Dispatched, Out for Delivery, Delivered).
FR4	Route Mapping (BFS): The system must utilize the Graph structure to find and display the path/route from the warehouse to the final destination for any given parcel.
FR5	Delivery Search (Linear Search): The system must allow users to search for a specific parcel using its unique ID.
FR6	Delivery Filtering: The system must allow users to filter the list of all deliveries based on their current Status (e.g., show only "Pending" or "Delivered" items).

FR7	Report Generation (Sorting): The system must be able to generate reports by sorting all active deliveries by a primary key (e.g., Status) and a secondary key (e.g., Destination) using the Group Sort algorithm.
FR8	Travel Time Lookup (Hashing): The system must use the Travel Times Hash Table to look up the travel duration between any two connected route points in O(1) average time.
FR9	Completed Deliveries: Upon final status update to "Delivered," the system must move the record from the main list to the Completed Deliveries List.

B. Non-Functional Requirements (NFR)

Non-functional requirements specify criteria for judging the operation of the system, rather than specific behaviors.

ID	Requirement Type	Feature Description
NFR1	Performance	All primary operations (Enqueue, Dequeue, Lookups) must be completed in O(1) average time to ensure a highly responsive user experience.
NFR2	Scalability	The system must efficiently handle an increase in the number of deliveries, with core data structures like the Dynamic Array allowing for growth without significant performance degradation (O(1) amortized append).
NFR3	Usability	The user interface must be intuitive, allowing a new user to register a delivery or change a status within three clicks or less.
NFR4	Reliability	The system must ensure that all delivery data is persisted and cannot be lost due to unexpected system failure.
NFR5	Security	Access to modify delivery status and configuration settings must be protected by an authentication and authorization mechanism.

(Hubert) 2.2. Data Requirements (Description of input data structure and size:

The Logistic Delivery System utilizes various organized data collections to maintain, access, and handle the information necessary for logging deliveries, modifying statuses, planning, directing, and producing reports. Below is a summary of the essential input data structures and their anticipated dimensions.

- **Delivery Records (Dynamic Arrays/ List)**

Structure: Each Delivery record is stored as a dictionary with the following fields:

Field	Type	Description
ID	String	Unique Parcel ID entered by the user
Sender	String	Name of the person sending the parcel
Receiver	String	Name of the person receiving the parcel
Destination	String	Delivery location/area
Status	String	Current delivery status (Pending, Dispatched, Out forDelivery,Delivered)

Size: Dynamic — grows as new deliveries are registered.
Expected range: **10–200 delivery records**.
Data Structure Used**:** Python List (`deliveries`)

- **Schedule queue (FIFO Queue)**

Structure: A queue (collections.deque) containing delivery record objects. Used to maintain order in which parcels should be dispatched.

Contents:Same dictionary objects as deliveries list.

Data Structure Used: Deque (FIFO queue)

Size: Dynamic, but usually small — 0 to 20 pending items at any time.

- **Completed Deliveries List**

Structure: List of dictionaries with extended fields for route information**:**

Field	Type	Description
ID	String	Parcel ID
Sender	String	Sender Name
Receiver	String	Receiver Name
Origin	String	Default origin = "Warehouse"
Destination	String	Delivery destination
Route	String	Computed route (e.g., "warehouse → area A → area C)
Time	Integer	Travel time in minutes
Statues	String	Always "Delivered"

Size: Dynamic — depends on how many items have been delivered.

- Expected range: 5–100 completed deliveries.
- Data Structure Used**:** Python List (`completed_deliveries`)

- **Graph Structure for Routes**

Structure: A dictionary mapping each location to its connected nodes (adjacency list).
Size: Static— 7 nodes and fixed edges

- **Travel Time**

Structure: Dictionary mapping tuples of location-pairs to integer travel times.
Size: Static — ~10 route edges.

- **Status Options Table**

Structure: Dictionary mapping number selections to string statuses.

Size: Static — 4 options.

Summary Tablet

Data Structure	Type	Expected Size
Deliveries	Dynamic List	10-200 Items
Schedule Queues	Queue	0-20 Items
Completed Deliveries	List	5-100 Items
Graph Routes	Dictionary	Fixed (7nodes)
Travel Times	Dictionary	Fixed (~10 entries)
Statues Options	Dictionary	4 Entries

2.3. Complexity Analysis: Expected Time/Space complexity of the Core Algorithm (justify using Big O notation).

1. **Dynamic Array**

```
deliveries = [ ]
completed_deliveries = [ ]
```

- **Key Operations:**

a. **Append** (Adding deliveries)

```
deliveries.append(record)
```

➢ Time Complexity: O(1) amortized
➢ Justification: Python lists allocate extra space. When full, they resize (copy all elements to larger array) - this happens rarely, so it averages to O(1)

b. **Access by Index**

```
pivot = arr[len(arr)//2] (In quick_sort)
```

- ➢ Time Complexity: O(1)
- ➢ Justification: Direct memory address calculation: base_address + (index × element_size)

c. **Linear Search** (Finding delivery by ID)

```
for d in deliveries:
 if d["id"] == parcel_id:
```

- ➢ Time Complexity: O(n) where n = number of deliveries
- ➢ Justification: Worst case must check every element

2. Graph (Tree-like structure)

```
routes = {
  "Warehouse": ["Area A", "Area B"],
  "Area A": ["Area C", "Area D"],
  "Area B": ["Area E", "Area F"],
  "Area C": [ ],
  "Area D": [ ],
  "Area E": [ ],
  "Area F": [ ]
}
```

Structure**:**

- ➢ Type: Directed Acyclic Graph (DAG) / Tree
- ➢ Root: Warehouse
- ➢ Representation: Adjacency List (Dictionary → List)
- ➢ Vertices (V): 7 nodes
- ➢ Edges (E): 6 connections

Key Operations:

a. **BFS Traversal** (Find Route)

```
def find_route(destination):
 queue = deque([("Warehouse", ["Warehouse"])])
  visited = set()
  while queue:
    current, path = queue.popleft()
```

(explore neighbors)

- ➢ Time Complexity: O(V + E)
  - Visit each vertex once: O(V)
  - Explore each edge once: O(E)
  - Space Complexity: O(V)
- ➢ Justification**:** BFS explores all reachable nodes level-by-level until destination found

b. Get children

for child in routes.get(current, [ ]):

- ➢ Time Complexity: O(1) for lookup + O(d), where d=degree(number of child)
- ➢ Justification: Dictionary hash lookup O(1), then through child list

3. Queue (Queueing)

schedule_queue + deque()

Key Operations:

a. **Enqueue** (Add to back}

schedule_queue.append(record)

- ➢ Time Complexity: O(1)
- ➢ Justification: dequeues uses doubly-linked, constant-time tail insertion

b. **Dequeue** (Remove from front)

current, path = queue.popleft()

- ➢ Time Complexity: O(1)
- ➢ Justification: Doubly-linked list allows O(1) head removal

c. **Search/Iterate** (Find next pending)

```
for d in schedule_queue:
  if d["status"] == "Pending":
```

- Time Complexity: O(n)
- Justification: Must scan queue to find matching element

4. Sorting Algorithms

Key Operations:

a. **Quick Sort**

```
def quick_sort(arr, key):
  if len(arr) <= 1:
    return arr

pivot = arr[0]
  pivot_val = pivot.get(key)

  left = [x for x in arr if x.get(key) < pivot_val]
  middle = [x for x in arr if x.get(key) == pivot_val]
  right = [x for x in arr if x.get(key) > pivot_val]

return quick_sort(left, key) + middle + quick_sort(right, key)
```

- Time Complexity:
  - Best/Average Case: O(n log n)
  - Worst Case: $O(n^2)$(poor pivot selection)
- Justification: Quick sort efficiently partitions the list and recursively sorts sublists, making it suitable for sorting delivery records by a single attribute such as destination

b. Group Sort

```
def group_sort(arr, primary_key, secondary_key):
  groups = {}

  # Group items by primary key
  for item in arr:
    key = item.get(primary_key)
    if key not in groups:
      groups[key] = [ ]
```

```
    groups[key].append(item)

  # Sort groups by predefined status order
  status_order = {status: i for i, status in enumerate(status_options.values())}
  sorted_group_keys = sorted(
    groups.keys(),
    key=lambda k: status_order.get(k, 99)
  )

  # Sort each group using Quick Sort
  result = []
  for key in sorted_group_keys:
    result.extend(quick_sort(groups[key], secondary_key))

return result
```

- Time Complexity: O(n log n) best, O(n log n) worst
- Justification: Applies Quick Sort within each group to achieve multi-level sorting

5. Hashing(Hashing Table)

```
  # Hash Table 1: Travel Times
  travel_times = {
    ("Warehouse", "Area A"): 3,
    ("Warehouse", "Area B"): 4,
    ("Area A", "Area C"): 3,
    # ...
  }

  # Hash Table 2: Status Options
  status_options = {
    "1": "Pending",
    "2": "Dispatched",
    "3": "Out for Delivery",
    "4": "Delivered"
  }

  # Hash Table 3: Groups (in group_sort)
```

```
groups = {}
for item in arr:
  key = item.get(primary_key)
  if key not in groups:
    groups[key] = []
  groups[key].append(item)

# Hash Table 4: Status Order
status_order = {status: i for i, status in enumerate(status_options.values())}
```

Key Operations:

a. Insert

```
travel_times[("Warehouse", "Area A")] = 3
groups[key] = []
status_order[status] = i
```

- Time Complexity: O(1) average, O(n) worst
- Justification: Hash to index, store value. Collisions rare with good hash function

b. Lookup

```
time = travel_times.get(edge, 0)
status = status_options[choice]
order = status_order.get(k, 99)
```

- Time Complexity: O(1) average, O(n) worst
- Justification: Hash key, retrieve value directly

c. Membership Check

```
if key not in groups:
if choice in status_options:
```

- Time Complexity: O(1) average
- Justification:Hashing provides constant-time access under normal conditions. Worst-case occurs with excessive collisions.

Summary Table:

DSA	Primary Use	Key Operation	Time	Space
Dynamic Array	Store Delivery	Append	O(1)	O(n)
		Search	O(n)	
Queue	FIFO	Enqueue/Deque ue	O(1)	
Graph/Tree	Route mapping	BFS	O(V+E)	O(V)
		Get child	O(1)	
Sorting	Organize deliveries	Quick sort	O(n log n)	O(n log n)
		Group Sort	O(n log n)	O(n log n)
Hashing	Fast lookups and grouping	Insert/Look up	O(1) avg	O(k)

III. Design Specification (Trisha, Riza, Hubert)

(Riza)3.1. Core Data Structures Used (The Five):

- **Justification**: Why was this specific DSA chosen for its role?
  - 1. **Dynamic Array** - Dynamic arrays were chosen to store delivery records because:
    - ➢ Flexible Size: The number of deliveries varies throughout the day, requiring a data structure that can grow dynamically
    - ➢ Fast Access: O(1) time complexity for accessing any delivery record by index
    - ➢ Simple Iteration: Easy to loop through all deliveries for status updates and filtering
    - ➢ Memory Efficient: Python's list implementation automatically manages memory allocation
  - 2. **Queue (Deque)** - A queue (specifically Python's deque) was chosen for delivery scheduling because:
    - ➢ FIFO Ordering: Deliveries should be dispatched in the order they were registered

- ➢ Efficient Operations: O(1) time complexity for both enqueue and dequeue operations
- ➢ Real-World Model: Mirrors how actual delivery systems prioritize orders
- ➢ Automatic Dispatch: Enables automatic dispatching of the next pending delivery when current one completes

- 3. **Sorting Algorithm**
  - Quick Sort was chosen for sorting deliveries by destination because:
- ➢ Efficient Average Case: O(n log n) time complexity on average
- ➢ In-Place Sorting: Can be implemented with minimal extra memory
- ➢ Practical Performance: Generally faster than other O(n log n) algorithms for most real-world data
- ➢ Alphabetical Ordering: Excellent for sorting string-based destinations
- ➢ User Experience: Allows users to view deliveries in organized, alphabetical order
  - Group Sort was implemented to handle complex logistics reporting needs because:
- ➢ Multi-Criteria Organization: Groups deliveries by status first (Pending, Dispatched, etc.), then sorts by destination within each group
- ➢ Logical Status Hierarchy: Maintains defined status order matching workflow priority in logistics operations
- ➢ Enhanced Visibility: Warehouse managers see all pending deliveries together, all dispatched together, etc., with alphabetical organization within each category
- ➢ Operational Clarity: Eliminates confusion from mixed statuses and locations in reports
- ➢ Leverages Quick Sort: Reuses Quick Sort algorithm for efficient secondary sorting within each status group

- 4. **Graph (Tree Structure)** - A directed graph (tree structure) was chosen to represent delivery routes because:
- ➢ Hierarchical Routes: Deliveries follow a tree-like structure from warehouse to destinations
- ➢ Route Visualization: Clearly shows available paths and connections
- ➢ Path Finding: Enables BFS algorithm to find optimal routes
- ➢ Real-World Model: Accurately represents the hub-and-spoke distribution model
- ➢ Travel Time Calculation: Supports weighted edges for time estimation
- 5. **Hashing -** Hash tables were implemented for constant-time lookups because:

- ➢ O(1) Lookup Performance: Instant retrieval of status options and travel times regardless of dataset size
- ➢ Key-Value Mapping: User input ("1", "2", "3") directly maps to status values ("Pending", "Dispatched", "Delivered")
- ➢ Travel Time Retrieval: Location pairs (e.g., ("Warehouse", "Area A")) serve as keys for instant travel time lookup
- ➢ Dynamic Grouping: Used in Group Sort to efficiently create status-based groups without linear searches
- ➢ Scalability: System remains responsive during peak hours with hundreds of simultaneous status updates
- ➢ Memory Efficiency: Stores frequently accessed reference data (status options, travel times) in compact key-value format

● **Implementation Details**: How did you implement it (e.g., adjacency list for Graph, array for Heap)?

1. **Dynamic Array**

```
1    deliveries = []
2    completed_deliveries = []
3
4    def add_delivery (record):
5    deliveries.append (record)
```

- Python lists automatically resize when needed. Start empty and grow as deliveries are added.
- append() adds a new delivery to the end of the array in O(1) amortized time. The list automatically expands if needed.

```
7    def find_delivery_by_id(parcel_id):
8    for d in deliveries:
9    if d["id"] == parcel_id:
10   return d
11   return None
```

- Linear search through the array to find a specific delivery. O(n) time complexity - checks each element until match is found or end is reached.

```
13    def get_active_deliveries():
14    return [d for d in deliveries if d["status"] != "Delivered"]
15
16    def get_deliveries_by_status(status):
17    return [d for d in deliveries if d["status"] == status]
18
```

- ➢ List comprehension filters the array, creating a new list with only non-delivered items. O(n) to check all deliveries.
- ➢ Filters array by specific status (Pending, Dispatched, etc.). Returns a new list with matching deliveries. O(n) time.

```
18
19    def get_all_deliveries():
20    return deliveries
21
```

- Returns reference to the entire array. O(1) operation - no copying occurs.

```
21
22    def add_completed_delivery(record):
23    completed_deliveries.append(record)
24
```

- Separate array for completed deliveries with route info. Keeps historical data organized. O(1) append.

```
24
25    def get_completed_deliveries();
26    return completed_deliveries
```

- Returns all completed deliveries for reporting. O(1) to return reference.

2. Queue (Deque)

```
1    from collection import dequeue
2
3    schedule_queue = dequeue()
```

- deque (double-ended queue) provides O(1) operations at both ends, unlike lists which are O(n) for removing from the front.

```
5     def enqueue_delivery(record):
6     schedule_queue.append(record)
```

- Adds delivery to the back of the queue (FIFO principle). O(1) operation - maintains order of arrival.

```
8     def get_next_pending():
9     for d in schedule_queue:
10    if d["status"] == "Pending":
11    return d
12    return None
```

- Look through the queue to find the next pending delivery without removing it. O(n) worst case if we need to check all items.

```
14    def dispatch_next_pending():
15    for d in schedule_queue:
16            if d["status"] == "Pending":
17                d["status"] = "Dispatched"
18                print(f"\n>>> Next delivery auto-dispatched: ID {d['id']} to {d['destination']}")
19                return
20        print("\n>>> No pending deliveries in queue.")
```

- Finds first pending delivery and changes status to "Dispatched". Automatically triggered when a delivery is completed. O(n) to search, but typically finds pending items quickly.

3. Sorting Algorithm

- Quick Sort

```
2
3     def quick_sort(arr, key):
4         if len(arr) <= 1:
5             return arr
6
```

- Base case for recursion: if the array has 0 or 1 elements, it's already sorted, so return it immediately without doing any work.

```
8
7         pivot = arr[0]
8         pivot_val = pivot.get(key)
9
```

```
10          if pivot_val is None:
11              return arr
```

- Selects the first delivery as the pivot point for comparison, extracts its destination value (or whatever key field was specified), and returns the array unchanged if the key doesn't exist (safety check).

```
13          left = [x for x in arr if x.get(key) < pivot_val]
14          middle = [x for x in arr if x.get(key) == pivot_val]
15          right = [x for x in arr if x.get(key) > pivot_val]
16
17          return quick_sort(left, key) + middle + quick_sort(right, key)
```

- Partitions deliveries into three groups: those with destinations alphabetically before the pivot (left), equal to pivot (middle), and after pivot (right), then recursively sorts left and right partitions and concatenates them as [sorted_left + middle + sorted_right].

```
297
298 def sort_deliveries():
299         print("\n--- Sort Deliveries by Destination ---")
300         sorted_list = quick_sort(deliveries, key="destination")
301
```

- Calls the quick_sort function to sort the entire deliveries array alphabetically by the "destination" field (e.g., "Area A", "Area B"), then stores the sorted result in sorted_list.

- Group Sort

```
21  def group_sort(arr, primary_key, secondary_key):
22      """Sorts deliveries first by primary key, then by secondary key (Group Sort)"""
23      groups = {}
24      for item in arr:
25          key = item.get(primary_key)
26          if key not in groups:
27              groups[key] = []
28          groups[key].append(item)
```

- Creates a hash table (groups) to organize deliveries by their status (primary_key), where each status becomes a key (e.g., "Pending", "Dispatched") and its value is a list of all deliveries with that status.

```
29
30          final_sorted_list = []
31
32          status_order = {status: i for i, status in enumerate(status_options.values())}
33
```

- Creates an empty list for the final result, and builds a hash table that assigns each status a numeric priority (e.g., "Pending":0, "Dispatched":1, "Out for Delivery":2, "Delivered":3) for proper ordering.

```
32
33
34          sorted_groups_keys = sorted(groups.keys(), key=lambda k: status_order.get(k, 99))
35
```

- Sort the status names themselves (the dictionary keys) according to their priority numbers from status_order, so groups will be processed in logical order (Pending → Dispatched → Out for Delivery → Delivered); unknown statuses get priority 99.

```
36          for key in sorted_groups_keys:
37              sorted_group = quick_sort(groups[key], secondary_key)
38              final_sorted_list.extend(sorted_group)
39
40          return final_sorted_list
```

- For each status group in order, sort the deliveries within that group alphabetically by destination (secondary_key) using quick_sort, then append them to the final list, resulting in deliveries grouped by status and sorted by destination within each group.

4. Hashing

```
1   status_options = {
2       "1": "Pending",
3       "2": "Dispatched",
4       "3": "Out for Delivery",
5       "4": "Delivered"
6   }
```

- Hash table that maps user menu choices (keys: "1", "2", "3", "4") to status names (values), allowing O(1) constant-time lookup to convert numerical input into readable status strings.

```
travel_times = {
    ("Warehouse", "Area A"): 3,
    ("Warehouse", "Area B"): 4,
    ("Area A", "Area C"): 3,
    ("Area A", "Area D"): 3,
    ("Area B", "Area E"): 4,
    ("Area B", "Area F"): 4
}
```

- Hash table using tuple pairs of locations as keys (e.g., ("Warehouse", "Area A")) and travel time in minutes as values, enabling O(1) lookup of how long it takes to travel between any two connected areas.

```
groups = {}
for item in arr:
    key = item.get(primary_key)
    if key not in groups:
        groups[key] = []
    groups[key].append(item)
```

- Creates a hash table dynamically during group_sort where each unique status becomes a key, and the value is a list of all deliveries with that status; uses O(1) insertion and lookup to efficiently categorize deliveries.

```
if choice in status_options:
    new_status = status_options[choice]
```

- Checks if the user's choice exists in the hash table (O(1) membership test), then retrieves the corresponding status string (O(1) lookup) to update the delivery status

```
total_time += travel_times.get(edge, 0)
```

- Looks up the travel time for a route edge in O(1) time; if the edge doesn't exist in the hash table, returns a default value of 0 instead of throwing an error.

5. Graph(Tree)

```
from collections import deque

# Graph for mapping delivery routes
routes = {
    "Warehouse": ["Area A", "Area B"],
    "Area A": ["Area C", "Area D"],
    "Area B": ["Area E", "Area F"],
    "Area C": [],
    "Area D": [],
    "Area E": [],
    "Area F": []
```

- Adjacency list representation of a directed graph where each location (key) maps to a list of directly connected destinations (values); forms a tree structure with Warehouse as root and delivery areas as branches/leaves.

```
def find_route(destination):
    """BFS to find route from Warehouse to destination"""
    queue = deque([("Warehouse", ["Warehouse"])])
    visited = set()
```

- Sets up Breadth-First Search (BFS) by creating a queue containing the starting node (Warehouse) with its path, and a set to track visited nodes to avoid cycles and redundant processing.

```
    while queue:
        current, path = queue.popleft()

        if current == destination:
            return path

        if current in visited:
            continue

        visited.add(current)
```

- Dequeues the front node, checks if it's the destination (returns path if found), skips already-visited nodes to prevent infinite loops, then marks the current node as visited.

```
30
31              for child in routes.get(current, []):
32                  if child not in visited:
33                      queue.append((child, path + [child]))
34
35          return []
36
```

- Explores all unvisited children of the current node by looking them up in the graph, adds each child to the queue with its accumulated path, and returns an empty list if no route exists to the destination.

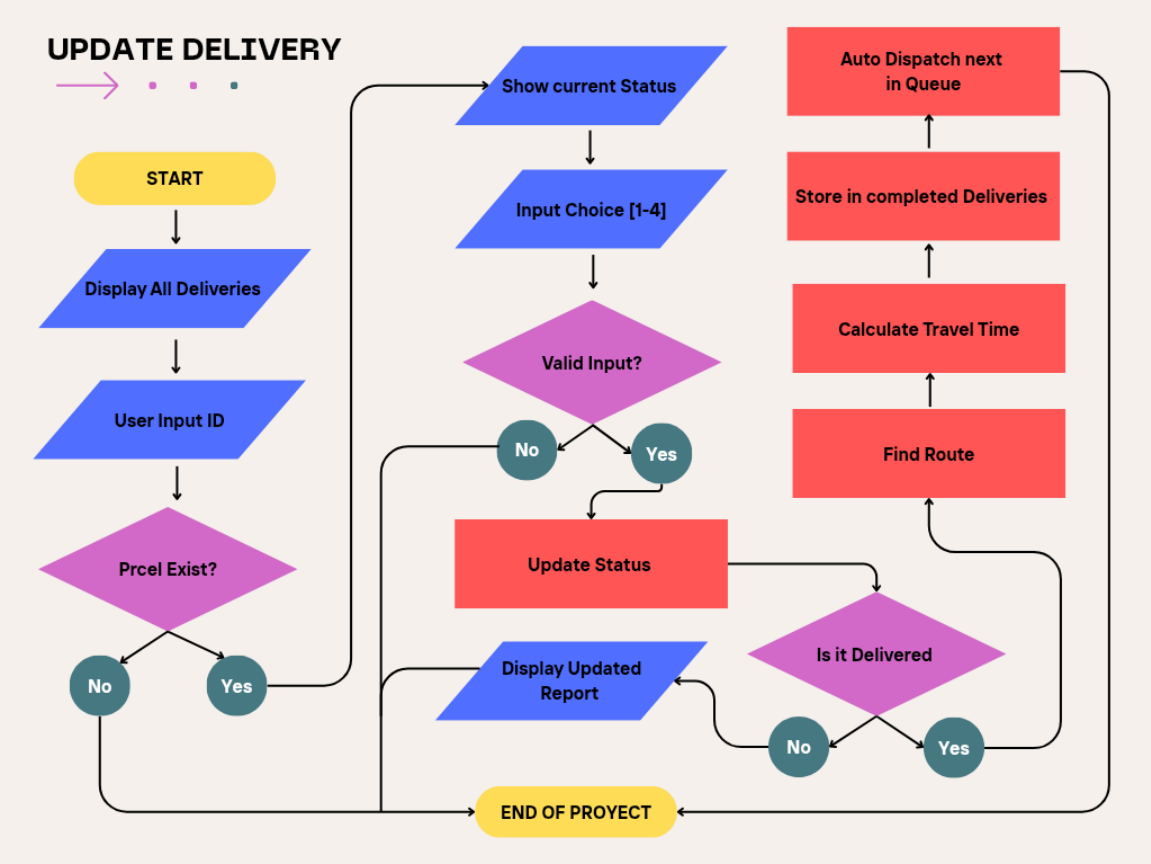
```
37      def calculate_travel_time(route):
38          """Calculate total travel time for a route"""
39          total_time = 0
40          for i in range(len(route) - 1):
41              edge = (route[i], route[i+1])
42              total_time += travel_times.get(edge, 0)
43          return total_time
```

- Takes a complete path (e.g., ["Warehouse", "Area A", "Area C"]) and calculates total travel time by iterating through consecutive location pairs (edges) and summing up their individual travel times from the hash table.

```
45      def display_route_map():
46          """Display the route map"""
47          print("\n--- Route Map (Graph) ---")
48          for place, connected in routes.items():
49              print(f"{place} -> {', '.join(connected) if connected else 'No outgoing routes'}")
```

- Displays the graph structure in human-readable format by iterating through all nodes and showing their outgoing connections (e.g., "Warehouse -> Area A, Area B"), making the delivery network visible to users.

(Trisha) 3.2. Algorithm Flowchart: Include the Flowchart for the system's most complex function (the core algorithm using a Finals concept).
Example below:



(Hubert) 3.3. Module Breakdown: Define the custom C++ classes and how they interact.

1. **Delivery Class (Core Data Module)**
   - Purpose**:** Represents a single delivery record and stores all relevant parcel information.
   - Attributes**:**
     - ➢ id – Unique parcel identifier
     - ➢ sender – Name of sender
     - ➢ receiver – Name of receiver
     - ➢ destination – Delivery location
     - ➢ status – Current delivery status (Pending, Dispatched, Out for Delivery, Delivered)
   - Methods**:**
     - `update_status(new_status)` – Updates the delivery's status

- `__str__()` – Returns a formatted string representation of the delivery

- Interaction:
  The `Delivery` class is used by the DeliveryManager to store, update, and display delivery records.

2. **DeliveryManager Class (Data Management Module)**
   - Purpose: Manages all delivery records and handles registration, updates, and reporting.
   - Attributes:
     - `deliveries` – Dynamic list storing all `Delivery` objects
     - `completed_deliveries` – List of delivered parcels with route details
   - Methods:
     - `register_delivery()` – Creates a new `Delivery` object
     - `update_delivery_status(parcel_id, status)` – Updates delivery status
     - `get_active_delivery()` – Returns the currently active delivery
     - `dispatch_next_pending()` – Automatically dispatches the next pending delivery
     - `generate_status_report()` – Displays delivered deliveries
   - Interaction: Acts as the **central controller**, interacting with:
     - `Scheduler` for queue handling
     - `RouteGraph` for route and travel-time calculation

3. **Scheduler Class (Queue Management Module)**
   - Purpose: Controls delivery scheduling using a queue (FIFO).
   - Attributes:
     - `schedule_queue` – Queue of pending and active deliveries
   - Methods:
     - `add_delivery(delivery)` – Adds a delivery to the queue
     - `get_next_pending()` – Retrieves the next pending delivery
     - `auto_dispatch()` – Dispatches deliveries when a slot is available

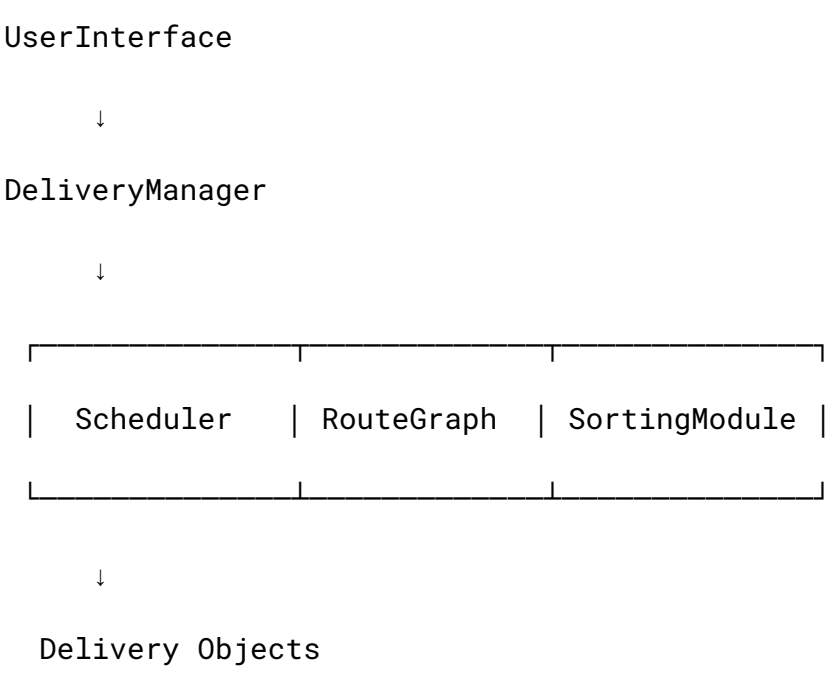
- Interaction: Works closely with the **DeliveryManager** to ensure only one active delivery at a time.

4. **RouteGraph Class**
   - Purpose: Represents delivery locations and routes using a graph.
   - Responsibilities:
     - Find a route from the warehouse to a destination using BFS
     - Calculate total travel time for a route
   - How it is used: When a delivery is completed, this class is used to determine the route taken and the total delivery time.

5. **SortingModule Class**
   - Purpose: Organizes delivery records for reporting and viewing.
   - Responsibilities:
     - Sort deliveries using Quick Sort
     - Group deliveries by status and then by destination
   - How it is used: This module is used when displaying delivery reports to the user.

6. **UserInterface Class (Presentation Module)**
   - Purpose: Handles user interaction through a menu-driven console interface.
   - Methods:
     - `display_menu()` – Displays the main menu
     - `handle_user_choice()` – Redirects user input to appropriate system functions
   - Interaction: Communicates with **DeliveryManager** to execute system operations.

Module Interaction Overview



IV. Conclusion and Contributions

In conclusion, the development of the RouteLink Logistic Scheduling and Tracking Delivery System proves a good example of how the data structure and algorithms can be used to solve practical logistics problems in a real world system and also it supports the goals of UN Sustainable Development Goal 9. The system enhances delivery scheduling, route planning, tracking in transparency, and overall operational efficiency in logistics by making use of dynamic arrays, queues, graphs, sorting algorithms, and hashing. In this project strengthened the team to understand how efficient algorithm, time complexity and proper organization affects the system. More important of this is that the system aligns with the SDG9 by promoting technological innovation and proper management of data structures towards modern and sustainable logistics systems.