Part (i): Asymptotic Upper Bounds (Big-O Notation)

(a) f(n)=4n3+5n2·log(n)

- Dominant term: 4n^3
- Big-O: O(n^3)

(b) g(n)=3n+4·log(n)

- Dominant term: 3n
- Big-O: O(n)

(c) n(n)=1.6n+n5

- Dominant term: 1.6^n (exponential growth dominates polynomial growth)
- Big-O: O(1.6^n)

(d) k(n)=2n2+80·log(n)

- Dominant term: 2n2
- Big-O: O(n2)

Part (i) (continued): Ascending Order of Complexity

O(n) < O(n^2) < O(n^3) < O(1.6^n)

Part (ii): Time Complexity of the Function mystery

```
void mystery (int n) {
    for (int i = 1; i <= n * 4; i = i * 2) {
        for (int j = 0; j < i; j++)
            cout << j;
    }
}
```

Final Time Complexity

- Outer loop: O(log(n)).

- Total iterations of the inner loop: O(n).

Overall: O(n·log(n)).

➢ Question 2

```cpp
#include <iostream>
using namespace std;

int DList::printNodesinOneDirection(Node* curr, char LeftOrRight) {
    if (!curr) { // If curr is NULL, nothing to print
        return 0;
    }

    int count = 0;

    if (LeftOrRight == 'L') {
        // Traverse to the left (previous nodes)
        Node* temp = curr->getPrevious();
        while (temp) {
            cout << temp->getData() << endl; // Print the name
            count++;
            temp = temp->getPrevious(); // Move to the previous node
        }
    } else if (LeftOrRight == 'R') {
        // Traverse to the right (next nodes)
        Node* temp = curr->getNext();
        while (temp) {
            cout << temp->getData() << endl; // Print the name
            count++;
            temp = temp->getNext(); // Move to the next node
        }
    } else {
        cerr << "Invalid direction. Please use 'L' for left or 'R' for right." << endl;
    }

    return count; // Return the number of names printed
}
```

➢ Question 3

```cpp

void Insert(Node* pBefore, Node* pNew) {
    if (!pBefore) {
        // Insert at the beginning
        pNew->setNext(first);
        if (first) {
            first->setPrevious(pNew);
        }
        first = pNew;
    } else {
        // Insert after pBefore
        pNew->setNext(pBefore->getNext());
        pNew->setPrevious(pBefore);
        if (pBefore->getNext()) {
            pBefore->getNext()->setPrevious(pNew);
        }
        pBefore->setNext(pNew);
    }
}
```

```cpp
void Delete(Node* pToBeDeleted) {
    if (!pToBeDeleted) return;

    if (pToBeDeleted == first) {
        // Deleting the first node
        first = pToBeDeleted->getNext();
        if (first) {
            first->setPrevious(nullptr);
        }
    } else {
        Node* prev = pToBeDeleted->getPrevious();
        Node* next = pToBeDeleted->getNext();

        if (prev) {
            prev->setNext(next);
        }
        if (next) {
            next->setPrevious(prev);
        }
    }
    delete pToBeDeleted;
}
```

```cpp
public:
    void push(int val) {
        // Create a new node and insert it at the beginning
        Node* newNode = new Node(val);
        list.Insert(nullptr, newNode); // Insert at the head (nullptr -> before first)
    }

    int pop() {
        Node* topNode = list.getFirst();
        if (!topNode) {
            cerr << "Stack underflow!" << endl;
            return -1; // Indicate stack underflow
        }

        int topValue = topNode->getData();
        list.Delete(topNode); // Delete the first node
        return topValue;
    }
};
```

➢ Compare quicksort and mergesort on the basis of worst case performance, stability and memory usage.

Here’s a comparison between **Quicksort** and **Mergesort** based on the given criteria:

1. Worst-Case Performance

```
1. Quicksort:

2. Time Complexity: O(n2)
3. This occurs when the pivot selection is poor, such as always selecting the smallest or largest
element in a sorted or nearly sorted array.

4. Mergesort:

5. Time Complexity: O(n log n)
6. The division and merging steps are always consistent, making the worst-case performance equal to
the average case.
7. Winner: Mergesort (better worst-case performance).
8.
```

2. Stability

```
1. Quicksort:

2. Stability: Not Stable
3. Elements with equal keys may not maintain their relative order after sorting, as swapping during
partitioning can disrupt their positions.


4. Mergesort:
```

```
Stability: Stable
During merging, elements with equal keys are placed in the same relative order as in the input.
Winner: Mergesort (stable sorting algorithm).

```

---

3. Memory Usage

```
Quicksort:

Memory Usage: O(log n) (in-place)
Quicksort operates in-place, requiring only additional space for the recursion stack. For
iterative implementations, even this overhead is minimized.


Mergesort:
Memory Usage: O(n)
Mergesort requires additional space to hold temporary arrays during the merge step.
Winner: Quicksort (lower memory usage).

```

Summary Table

Criterion	Quicksort	Mergesort	Winner
Worst-Case Performance	$O(n^2)$	$O(n \log n)$	Mergesort
Stability	Not Stable	Stable	Mergesort
Memory Usage	$O(\log n)$ (in-place)	$O(n)$	Quicksort

Final Verdict:

- If worst-case performance and stability are critical, **Mergesort** is better.
- If memory efficiency is the priority and average-case performance suffices, **Quicksort** is preferred.

Specify appropriate data structures for the following situations:

i. Online sales store processing sale requests in the order they are received

Data Structure: Queue

- **Reason**: A queue is ideal for processing requests in a **First-In-First-Out (FIFO)** order, ensuring that the sales requests are handled in the same order they are received.

ii. Word processor with an ↑ key to redisplay preceding commands

Data Structure: Stack

- **Reason**: A stack allows commands to be stored in the order they are entered, and pressing the ↑ key effectively performs a **Last-In-First-Out (LIFO)** traversal. This ensures that the preceding command is redisplayed, and multiple presses retrieve earlier commands.

iii. Administering vaccine doses based on priority

Data Structure: Priority Queue

- **Reason**: A priority queue assigns a priority to each individual. Senior citizens and frontline healthcare workers can have a **higher priority**, ensuring they are administered vaccines before others.

---

iv. Online English to Urdu translation service

Data Structure: Hash Table (or Dictionary)

- **Reason**: A hash table allows for **fast lookups** of English words and their corresponding Urdu translations, typically in $O(1)$ time. This is critical for efficient translation services.

Summary Table

Situation	Data Structure	Reason
Online sales store processing requests	Queue	Ensures First-In-First-Out processing.
Word processor redisplaying preceding commands	Stack	Handles Last-In-First-Out traversal of commands entered.
Administering vaccine doses based on priority	Priority Queue	Allows high-priority individuals to be served first.
Online English to Urdu translation	Hash Table	Enables fast and efficient word lookups.

- **An online delivery app needs to be developed for a local groceries store in Lahore. The idea is to store data about clients who place an order using their mobile phone. The store hires drivers who are handed the grocery items to deliver to the client's address. The priority of the delivery is based on time of order, the oldest order gets delivered first. Also if the client address is within 5km from the store then there are no delivery charges otherwise a delivery charge of Rs 20/km is charged.**

    i. Which data structure(s) and algorithm(s) are most suitable to develop this app and why?

    ii. Please give the overall steps in the form of pseudocode needed to implement the order delivery system, using the data structure(s) and algorithm(s) decided in part(i) above

i. Suitable Data Structure(s) and Algorithm(s)

Data Structure(s):

1. **Queue**:
   - **Why**: Orders need to be processed in the order they were received (**First-In-First-Out**, FIFO). A queue is perfect for this as the oldest order will always be at the front.
2. **Hash Table (or Dictionary)**:
   - **Why**: To efficiently store and retrieve client details (e.g., name, address, phone number) and their associated order details.
3. **Graph (optional)**:
   - **Why**: If the app needs to optimize delivery routes, the client addresses and store location can be represented as a graph, and algorithms like Dijkstra's can be used to find the shortest path.

Algorithm(s):

1. **Order Priority Handling**:
   - Use the **queue** to ensure orders are delivered in the order they are received.
2. **Delivery Charge Calculation**:
   - Use the formula:

$$\text{Delivery Charge} = \begin{cases} 0 & \text{if distance} \leq 5\,\text{km} \\ 20 \times \text{distance (km)} & \text{otherwise} \end{cases}$$

3. **Distance Calculation**:
   - Use the **Haversine Formula** to calculate the distance between the store and the client's address using their latitude and longitude. Alternatively, use a mapping API for precise calculations.

Steps:

1. **Define the Data Structures**:
   - Use a queue for orders.
   - Use a hash table to store client details and order information.
2. **Accept Orders**:
   - Add client order details (e.g., name, address, phone number, and order time) to the queue.
3. **Calculate Delivery Charges**:
   - Compute the distance between the store and the client.
   - Use the delivery charge formula.
4. **Process and Deliver Orders**:
   - Dequeue orders from the queue.
   - Assign the order to a driver.
   - Mark the order as delivered.

Pseudocode:

```
Initialize Queue orders
Initialize HashTable clients

Function placeOrder(clientName, phone, address, orderDetails):
    distance = calculateDistance(storeLocation, address)
    deliveryCharge = 0
    If distance > 5:
        deliveryCharge = 20 * (distance - 5)
    clientID = generateUniqueID()
    clients[clientID] = {name: clientName, phone: phone, address: address, order: orderDetails, charge: deliveryCharge}
    orders.enqueue(clientID)

Function calculateDistance(location1, location2):
    # Use Haversine Formula or Mapping API
    return distance

Function deliverOrders():
    While not orders.isEmpty():
        clientID = orders.dequeue()
        clientDetails = clients[clientID]
        Assign driver to deliver clientDetails.order to clientDetails.address
        Mark order as delivered
        Remove clientID from clients

Function displayOrders():
    For each clientID in orders:
        Print clients[clientID].order
```

Why This Approach Works

- **Efficiency**: The queue ensures $O(1)$ enqueue and dequeue operations for processing orders.
- **Flexibility**: Hash tables provide $O(1)$ lookup for client details and allow easy data management.
- **Scalability**: If route optimization is added, a graph and Dijkstra's algorithm can handle multiple deliveries efficiently.