Student Number: 7547821

Student Name: Gia Bach Nhu

Here is my pseudo-code:

```
// Define constants
MAX_QUEUE = 500
// Define event types
customer_arrival = 0
service_completion = 1
// Define struct for an event
struct Event {
    eventType type
    float time
    float arrival
    float serviceDuration
    float enteredQueue
    int priority
    float queueDuration
    int assignedServer
}
// Define class for Arrival Queue
class ArrivalQueue {
    Event[] arrivals
    int queueLength
    int queueMaxLength
    // Constructor to initialize the queue
    function ArrivalQueue.initialize() {
        queueLength = 0
        queueMaxLength = 0
    }

    // Add an event to the queue
```

```
function ArrivalQueue.addQueue(ev: Event) {
   if queueLength < MAX_QUEUE {
      arrivals[queueLength] = ev
      queueLength++
      siftUp(queueLength - 1)
      queueMaxLength = max(queueMaxLength, queueLength)
   }
}
// Remove an event from the queue
function ArrivalQueue.removeQueue(): Event {
   if queueLength > 0 {
      ev = arrivals[0]
      arrivals[0] = arrivals[queueLength - 1]
      queueLength--
      siftDown(0)
      return ev
   }
}
// Check if the queue is empty
function ArrivalQueue.isQueueEmpty(): bool {
   return queueLength == 0
}
// Get the maximum queue length
function ArrivalQueue.maxQueueLength(): int {
   return queueMaxLength
}

// Sift up an event in the queue
function ArrivalQueue.siftUp(current: int) {
   // Implementation of siftUp
}
```

```
    // Sift down an event in the queue
    function ArrivalQueue.siftDown(current: int) {
        // Implementation of siftDown
    }
}
// Define class for Event Queue
class EventQueue {
    Event[] events
    int actualE
    int expectedE
    // Constructor to initialize the queue
    function EventQueue.initialize(expected: int) {
        expectedE = expected
        events = new Event[expectedE]
        actualE = 0
    }
    // Add an event to the queue
    function EventQueue.addE(ev: Event) {
        if actualE < expectedE {
            events[actualE] = ev
            actualE++
            siftUp(actualE - 1)
        }
    }
    // Remove an event from the queue
    function EventQueue.removeE(): Event {
        if actualE > 0 {
            ev = events[0]
            events[0] = events[actualE - 1]
            actualE--
            siftDown(0)
```

```
            return ev
        }
    }
    // Check if the queue is empty
    function EventQueue.isEmpty(): bool {
        return actualE == 0
    }
    // Get the next event type
    function EventQueue.getNextEventType(): eventType {
        return events[0].type
    }
    // Sift up an event in the queue
    function EventQueue.siftUp(current: int) {
        // Implementation of siftUp
    }
    // Sift down an event in the queue
    function EventQueue.siftDown(current: int) {
        // Implementation of siftDown
    }
}
// Define class for Teller
class Teller {
    enum status {
        busy
        idle
    }
    struct state {
        status current
        int served
        float idle
        float lastFinish
```

```
}
state[] servers
int serverCount
int serversBusy
int nextServer
// Constructor to initialize teller
function Teller.initialize(serverC: int) {
   serverCount = serverC
   servers = new state[serverCount]
   serversBusy = 0
   nextServer = 0
   // Initialize server states
   for i = 0 to serverCount - 1 {
      servers[i].current = idle
      servers[i].served = 0
      servers[i].idle = 0.0
      servers[i].lastFinish = 0.0
   }
}
// Assign a customer to an available server
function Teller.assign(thisStart: float, thisDuration: float): int {
   assignedServer = -1
   if serversBusy < serverCount {
      for i = 0 to serverCount - 1 {
         if servers[nextServer].current == idle {
            assignedServer = nextServer
            break
         } else {
            nextServer = (nextServer + 1) % serverCount
         }
      }
```

```
            servers[nextServer].served++
            servers[nextServer].idle = servers[nextServer].idle + (thisStart - servers[nextServer].lastFinish)
            servers[nextServer].lastFinish = thisStart + thisDuration
            servers[nextServer].current = busy
            serversBusy++
            nextServer = (nextServer + 1) % serverCount
        }
        return assignedServer
    }
    // Release a server
    function Teller.release(assignedServer: int) {
        if serversBusy > 0 {
            servers[assignedServer].current = idle
            serversBusy--
        }
    }
    // Check if there is an idle server
    function Teller.isIdleServer(): bool {
        return serversBusy != serverCount
    }
    // Get the number of customers served by a server
    function Teller.numServed(serverId: int): int {
        return servers[serverId].served
    }
    // Get the idle time of a server
    function Teller.idleTime(serverId: int): float {
        return servers[serverId].idle
    }
}
// Create an event
function createEvent(arrival: float, duration: float, priority: int, type: eventType): Event {
```

```
    ev = new Event
    ev.time = arrival
    ev.arrival = arrival
    ev.serviceDuration = duration
    ev.priority = priority
    ev.queueDuration = 0.0
    ev.type = type
    return ev
}
// Main function
function main() {
    // Initialize variables and data structures
    // Read inputs (number of tellers and input file name)
    // Initialize arrival queue, event queue, and teller pool
    // Read the first customer's arrival time, service time, and priority
    // Create and add the first customer arrival event to the event queue
    // Main simulation loop
    while moreEvents {
        // Handle service completion events
        if eventQ.getNextEventType() == service_completion {
            // Remove and process the next service completion event
            // Release the server
            // Check if there are customers in the arrival queue and assign one if available
        } // Handle customer arrival events
        else if eventQ.getNextEventType() == customer_arrival {
            // Remove and process the next customer arrival event
            // Add the customer to the arrival queue
            // Check if there is an idle server and assign one if available
            // Read the next customer's arrival time, service time, and priority
        }
        // Update moreEvents based on the simulation progress
```

}

Complexity Analysis:

- Arrival Queue (Max Heap) Operations:
    - When we add or remove items from the arrival queue (which is organized like a stack of books with the biggest on top), it takes time proportional to the logarithm of the number of items (O(log N)).

- Event Queue (Min Heap) Operations:
    - Similarly, when we add or remove items from the event queue (like a stack of books with the smallest on top), it also takes time proportional to the logarithm of the number of items (O(log N)).

- Teller Pool Operations:
    - Operations related to the teller pool, like assigning and releasing tellers, take time proportional to the number of tellers (O(K)). In the worst case, we might have to check all tellers to find an available one.

- Main Simulation Loop:
    - The main loop that keeps the simulation running processes events one by one. In the worst case, it processes every event once, making the complexity O(M), where M is the total number of events.

- Reading Input from File:
    - When reading input data from a file, the time required is inversely correlated to the number of lines (O(N)), where N is the total number of lines.
    - Calculating statistics and printing output involves iterating over the tellers, which takes time proportional to the number of tellers (O(K)).

List of Data Structures Used and Why:

- Arrays: Arrays are like lists that help us store and organize various pieces of information efficiently, like customer arrival events, event queue, and teller states. We use them because they let us quickly find items by their position.

- Enums: Enums are a way to give names to specific values, making the code more readable. We use them to define types of events and states of servers.

- Structs (struct event and struct state): Structs are like containers that allow us to group related pieces of data together. We use them to create custom data types for events and server states, making it easier to work with this data.

- Max Heap and Min Heap: Max and Min Heaps are like specialized piles of items that help us manage events based on their importance. We use Max Heap for the arrival queue and Min

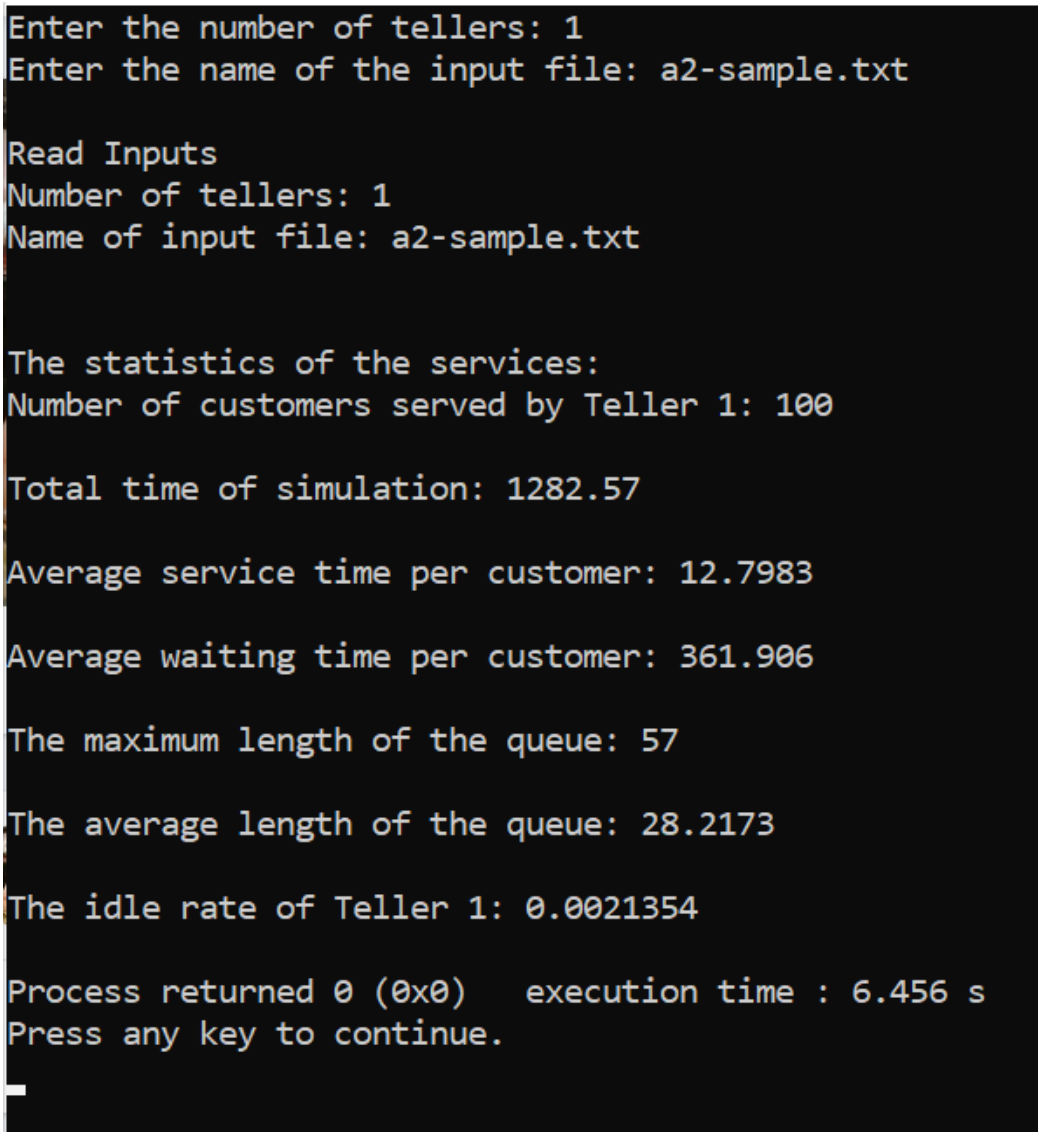
Heap for the event queue because they efficiently handle adding and removing items with high or low priority.

- Dynamic Arrays: Dynamic arrays are flexible lists that can grow or shrink as needed. We use them to store events in the event queue, allowing us to handle varying numbers of events efficiently.

- Classes (ArrivalQueue, EventQueue, Teller): Classes are like blueprints for creating objects with specific properties and behaviors. We use them to organize and manage different parts of the simulation, like queues and tellers, to make our code neat and easier to understand.

- Arrays of Structures: Arrays of structures are like grids that help us store multiple instances of events efficiently. We use them in the arrival and event queues to keep track of many events, each with its own details.

The outputs (three runs in total):



```
Enter the number of tellers: 2
Enter the name of the input file: a2-sample.txt

Read Inputs
Number of tellers: 2
Name of input file: a2-sample.txt


The statistics of the services:
Number of customers served by Teller 1: 52
Number of customers served by Teller 2: 48

Total time of simulation: 653.311

Average service time per customer: 12.7983

Average waiting time per customer: 60.3841

The maximum length of the queue: 22

The average length of the queue: 9.24279

The idle rate of Teller 1: 0.0162805
The idle rate of Teller 2: 0.0164147

Process returned 0 (0x0)   execution time : 6.545 s
Press any key to continue.
```

```
Enter the number of tellers: 4
Enter the name of the input file: a2-sample.txt

Read Inputs
Number of tellers: 4
Name of input file: a2-sample.txt


The statistics of the services:
Number of customers served by Teller 1: 25
Number of customers served by Teller 2: 24
Number of customers served by Teller 3: 25
Number of customers served by Teller 4: 26

Total time of simulation: 531.161

Average service time per customer: 12.7983

Average waiting time per customer: 0.25181

The maximum length of the queue: 1

The average length of the queue: 0.0474074

The idle rate of Teller 1: 0.375467
The idle rate of Teller 2: 0.380956
The idle rate of Teller 3: 0.377213
The idle rate of Teller 4: 0.316222

Process returned 0 (0x0)   execution time : 8.022 s
Press any key to continue.
```

- Scenario 1: One Teller
    - Number of Customers Served: Teller 1 served 100 customers.
    - Average Waiting Time: The average waiting time per customer was high at 361.906 units.
    - Maximum Queue Length: The maximum queue length reached 57 customers.
    - Average Queue Length: The average queue length was 28.2173 customers.
    - Teller 1 Idle Rate: Teller 1 was almost constantly busy with an idle rate of only 0.0021354.
    - Discussion: With just one teller, the service experienced high customer waiting times and long queues. The teller was almost continuously occupied, indicating that they were working at full capacity.
- Scenario 2: Two Tellers

    - Number of Customers Served: Teller 1 served 52 customers, and Teller 2 served 48 customers.
    - Average Waiting Time: The average waiting time per customer improved to 60.3841 units.
    - Maximum Queue Length: The maximum queue length decreased to 22 customers.
    - Average Queue Length: The average queue length was 9.24279 customers.
    - Idle Rates: Both Teller 1 and Teller 2 had low idle rates of around 1.6%.
    - Discussion: Introducing a second teller significantly reduced customer waiting times and queue lengths. Both tellers were more efficiently utilized with a balanced workload.
- Scenario 3: Four Tellers
    - Number of Customers Served: Tellers served approximately 25 customers each.
    - Average Waiting Time: The average waiting time per customer dropped significantly to 0.25181 units.
    - Maximum Queue Length: The maximum queue length was only 1 customer.
    - Average Queue Length: The average queue length was extremely low at 0.0474074 customers.
    - Idle Rates: All four tellers had idle rates ranging from 31.6% to 38.1%.
    - Discussion: With four tellers, waiting times became nearly negligible, and queues were almost non-existent. However, there was notable underutilization of teller resources, with idle rates ranging from 31.6% to 38.1%.

⇨ Summary: Having just one teller led to long waiting times and queues, suggesting low service efficiency. Introducing two tellers significantly improved service efficiency, reducing both waiting times and queues. Having four tellers further reduced waiting times and queues but also resulted in underutilization of resources.