

**Self-Assessment:** Please **highlight** where you think your report grade should be. Example below.

Criteria	Write simple algorithms using appropriate discrete data structures to solve computational problems (LO3)	Use appropriate methods to analyse the efficiency and correctness of algorithms (LO4)
Weight	25%	25%
<b>0 – 29%</b>	<p>The algorithm does not solve an appropriate problem, or has serious errors. There is little or no discussion of how the algorithm works.</p> <p>No discrete data structure has been used, or the choice of data structure is inappropriate.</p>	The analysis is limited and seriously flawed.
<b>30 – 39%</b>	<p>The algorithm solves part of an appropriate problem. There may be substantial aspects of the problem which are not attempted or explained, or errors in the solution.</p> <p>The explanation is unclear or missing important details about how the algorithm works.</p>	An attempt has been made to analyse an algorithm, but appropriate methods of analysis were not used, and the results of the analysis may be incorrect or meaningless.
<b>40 – 49%</b>	<p>A rudimentary algorithm solving a basic problem. There may be some errors which could be corrected with further work.</p> <p>There is a limited discussion of how the algorithm works. The choice of data structure is inappropriate, or unjustified.</p>	<p>An attempt has been made to measure the running time of the algorithm for some inputs, but the methodology is unclear or the measurement may be inaccurate. There is a limited discussion of some other issues relating to efficiency.</p> <p>Analysis of the algorithm's correctness is vague, or not attempted.</p>
<b>50 – 59%</b>	<p>The algorithm solves an appropriate problem, though it may have minor errors or fail to account for special cases. There is an explanation of how the algorithm works.</p> <p>The choice of data structure may be inappropriate or poorly justified.</p>	<p>The running time of the algorithm has been measured accurately for an appropriate range of inputs, and the methodology has been explained. There is some discussion of other issues relating to efficiency.</p> <p>There is a basic or informal analysis of the algorithm's correctness.</p>
<b>60 – 69%</b>	<p>The algorithm correctly solves an appropriate problem. There is a clear explanation of how the algorithm works.</p> <p>At least one appropriate data structure has been used, and the choice has been adequately justified.</p>	<p>The efficiency of the algorithm has been accurately measured using an appropriate methodology, which has been explained. The measurements may include more than one metric.</p> <p>There is an analysis of the algorithm's correctness, which may specify pre- and post-conditions for part of the algorithm.</p>
<b>70 – 79%</b>	<p>The algorithm correctly solves a challenging problem. There is a clear explanation of how the algorithm works, and the explanation makes clear references to the relevant parts of the source code.</p> <p>Appropriate data structures have been used, and justification is given for each with reference to the specific problem.</p>	<p>The efficiency of the algorithm has been accurately measured using an appropriate methodology, with multiple metrics and a clear explanation. The asymptotic complexity of the algorithm is given. The efficiency may be compared with appropriate alternative algorithm(s).</p> <p>There is a formal analysis of the correctness of at least part of the algorithm.</p>
<b>80 – 90%</b>	<p>A well-designed algorithm which correctly solves a challenging problem. There is a clear, detailed explanation of how the algorithm works, with clear references to the relevant parts of the source code.</p> <p>Appropriate data structures have been used, and justification is given for each with reference to the specific problem.</p>	<p>The efficiency of the algorithm has been accurately measured using an appropriate methodology, with multiple metrics and a clear, detailed explanation. The asymptotic complexity of the algorithm is given. The efficiency has been compared with appropriate alternative algorithm(s).</p> <p>There is a detailed formal analysis of the correctness of the algorithm.</p>
<b>90 – 100%</b>	An excellent algorithm written, explained and evaluated to the highest standards.	An excellent analysis of the efficiency, complexity and correctness of an algorithm, conducted and explained to the highest standards.



**BIRMINGHAM CITY**  
**University**

**Technical Report: Supermarket Sweep**  
**(Better lanes for new shoppers)**

**Authors: Dikshant Madai (BCU)**

**Date: june 18,2023**

**Word count: 1996**

**Pagecount: 17**

**Confidential: NO / YES – INTERNAL  
ONLY/YES – X**

## Executive Summary

This technical report analyzes a simulation system designed to optimize queuing management in a supermarket environment, addressing the issue of long waiting times and customer dissatisfaction. It emphasizes the need for an efficient queuing system that enhances customer experience and improves operational efficiency. The study introduces data structures and algorithms for the simulation system, such as standard lanes, express lane, Greedy algorithm, and Load Balancing algorithm. The system's structure, classes, and methods are described, with the CheckoutLane, SupermarketQueue, and Customer classes introduced. The user interface is also discussed for an intuitive user interface.

To validate the accuracy and effectiveness of the algorithms, an assertion table is presented. This table allows for checks against expected results, allowing for the identification and correction of deviations or errors. The report concludes that optimizing queuing management in supermarkets is crucial for enhancing customer satisfaction and operational efficiency. The simulation system demonstrates its ability to minimize waiting times, distribute customers evenly among checkout lanes, and maintain a balanced queuing system. Further testing, validation, and real-world implementation are recommended to gain insights into the system's performance and identify potential areas for improvement. Implementing an efficient queuing system can significantly improve customer satisfaction, increase revenue, and establish a competitive edge in the market.

## Contents

1. Introduction.....	1
2. Theory .....	1
2.1 Data structure .....	2
A. List .....	2
B. Deque or Double Ended Queue .....	2
2.2 Algorithm.....	4
A. Greedy Algorithm for Customer Assignment .....	4
B. Load Balancing Algorithm: .....	4
3. Comparison with Alternative Algorithms:.....	5
3. Implementation .....	5
4. Assertion Table .....	7
5. Conclusions.....	9
6. References.....	9
Appendix A.....	10
Appendix B .....	12

## 1. Introduction

For supermarkets to improve customer happiness and cut down on wait times, effective queue management is essential (Koeswara et al., 2018). Customer discontent and abandoned purchases may result from lengthy lines at busy times (Luo & Shi, 2020). This technical study presents a simulation system created for Tesco Express in Quigley to solve these issues. By integrating an adaptive queueing method that prioritizes consumers depending on the quantity of things in their trolleys, the system seeks to enhance queue management and customer satisfaction.

The objective of this study is to minimize waiting times and expedite the payment process by simulating and optimizing the supermarket queue management process. The algorithm developed will be discussed in the theory section of this report, focusing on the data structures and algorithms used. The implementation part will evaluate the efficiency of the algorithm, while the assertion table will assess its correctness.

## 2. Theory

This technical report's theoretical portion focuses on the modelling of a supermarket queuing system. The goal is to simulate customer behaviour and improve the queuing system to ensure efficient customer flow and satisfaction. For performance and efficacy, data formats and algorithms must be carefully chosen. We can correctly simulate consumer behaviour and examine many elements of the system to improve queue management by employing appropriate algorithms and data structures.

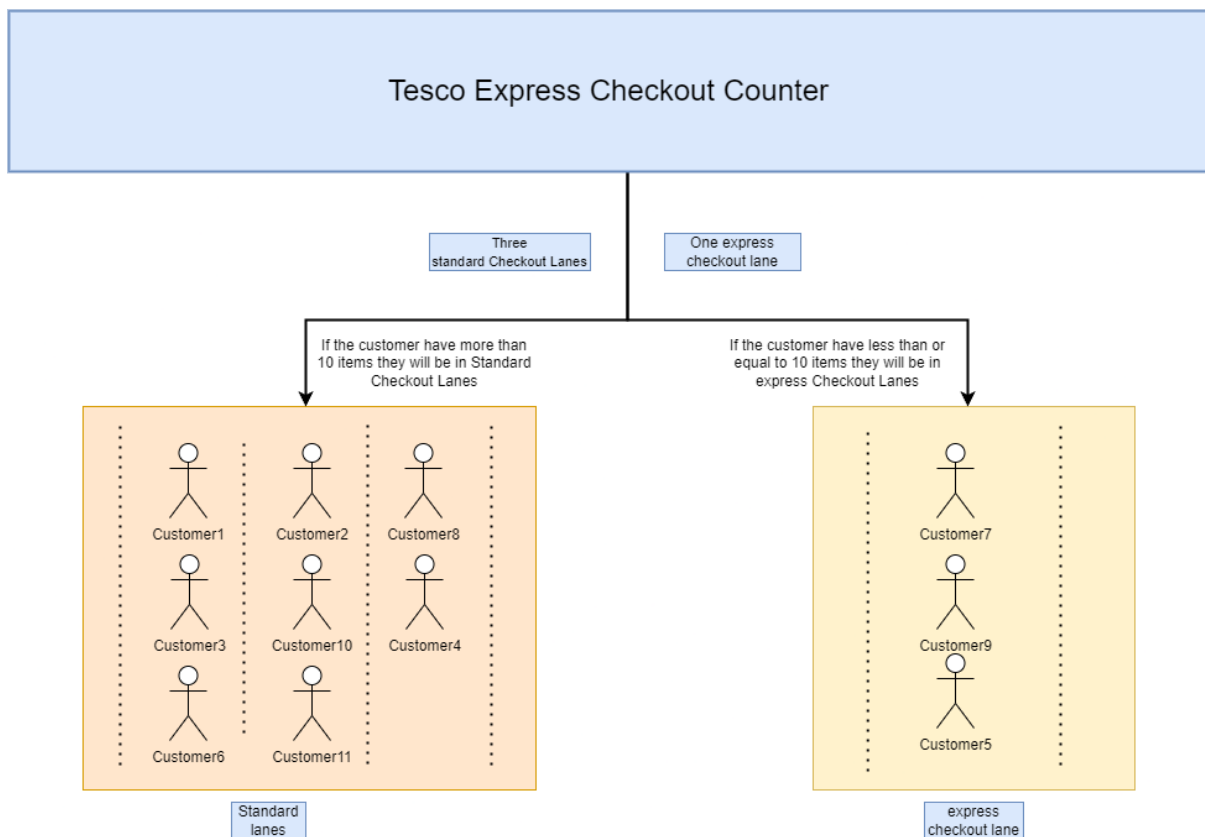


Figure 1: Assumptions Picture scenario of Tesco express checkout counter

The diagram above depicts the Tesco Express checkout counter scenario and related assumptions. The diagram illustrates four lanes, three of which are conventional checkout lanes and one of which is an express checkout lane. According to the scenario, if the customer has more than ten items in their cart, they should use one of the conventional checkout lanes. If the customer has fewer than 10 items or an equal amount of goods in their cart, they should use the express checkout counter.


## 2.1 Data structure

### A. List

The list is an abstract data type in which the components are kept in an ordered way for quicker and more efficient retrieval. Because the List Data Structure allows for repetition, a single item of data can appear more than once in a list ([javatpoint](#)). Using a list for the standard checkout lanes, as demonstrated in the below code, is one of a valid approach. Lists are well-suited when the number of lanes is fixed, and their order is significant. Some benefits are Sequential Access, Simplicity and more.

Lists in Python offer advantages over other data structures for simulating a supermarket queue system. They provide dynamic resizing, fast random access, and traversal, making them an efficient choice for simulating and managing the customer queue. Stacks, which operate on a LIFO principle, are not well-suited for simulating a FIFO approach. Traditional queues, implemented using arrays or linked lists, can handle customer arrivals efficiently but may lack the versatility and flexibility provided by lists.

- Standard lanes: It is a list that represents the standard checkout lanes in the supermarket. Each element in the list is an instance of the Checkout Lane class. The list allows for multiple standard lanes to be created and managed.

```
 self.standard_lanes = [CheckoutLane() for _ in range(num_standard_lanes)]
regular_items = [[] for _ in range(len(self.standard_lanes))]
```

`self.standard_lanes` is a list that stores instances of the `CheckoutLane` class, representing the standard checkout lanes. `regular_items` is a list of lists, where each inner list represents the names of customers being served in each regular checkout lane.

Figure 2: Code of list data structure

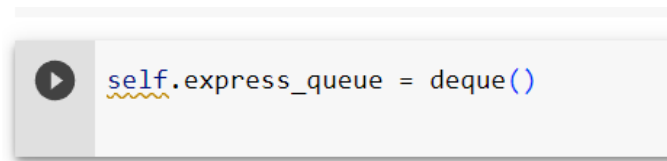
### B. Deque or Double Ended Queue

A queue is a data structure that follows the FIFO (First-In-First-Out) strategy. Insertion in the queue is done from one end known as the tail, whilst deletion is done from the other end known as the front end or the head of the queue ([javatpoint](#)). Deque is an abbreviation for Double Ended Queue. Deque is a linear data structure that performs insertion and deletion operations from both ends. Deque may

be thought of as a generalized variant of the queue. Although insertion and deletion in a deque can be performed on both ends, the FIFO rule is not followed.

Double-ended queue outperforms conventional data structures. It excels at both ends in efficient insertion, removal, and traversal, accommodating dynamic client arrivals and service. It provides for the smooth addition and removal of customers, unlike arrays/lists. It allows efficient access and traversal in both directions as compared to linked lists. Deques, as opposed to stacks and typical queues, provide flexibility for managing consumers in express and normal lanes while responding to variable item counts. Overall, the deque data format is ideally suited to efficiently modelling the dynamic character of a supermarket waiting system.

- Supermarket Queue: The supermarket queue consists of multiple Checkout Lane instances for standard checkouts (stored in a list) and an express queue (stored in a deque).



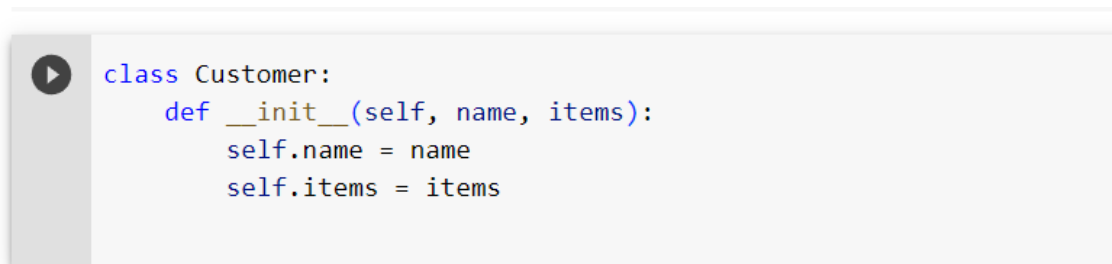
```
self.express_queue = deque()
```

Figure 3: Code of Deque data structure

`self.express_queue` is a deque that is used to store customers with fewer than or equal to 10 items in the express checkout line. The deque provides efficient addition and removal of elements from both ends, making it suitable for managing the queue-like behavior of customers in the express checkout line.

### Customer Data Structure

A single client in the supermarket is represented by the customer data structure. It has two attributes: name and items. The name element holds the customer's name as a string, whereas the items attribute records the customer's item count as an integer. This data structure enables effective organization and access to client information.



```
class Customer:
    def __init__(self, name, items):
        self.name = name
        self.items = items
```

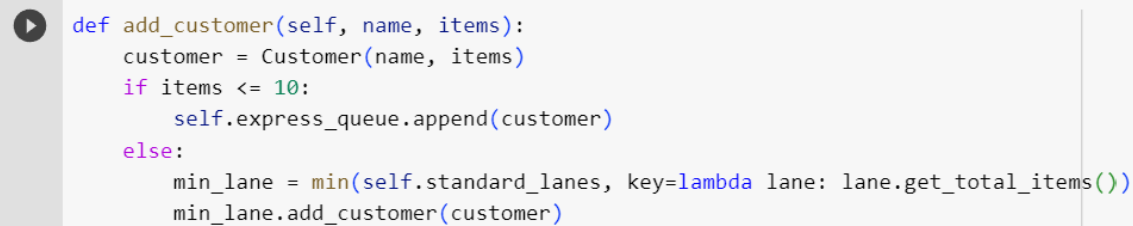
Figure 4: Code of Customer Data Structure

The Customer class is defined in the code with the attributes name and items. When clients come, instances of this class are created, and their names and the quantity of things they have are saved using these properties. In the supermarket simulation, this data structure is critical for managing and servicing consumers.

## 2.2 Algorithm

### A. Greedy Algorithm for Customer Assignment

Greedy is an algorithmic paradigm that assembles a solution piece by piece, constantly selecting the next component that provides the greatest evident and immediate advantage. So Greedy is best suited to cases when picking locally optimal also leads to a global solution ([GeeksforGeeks, 2023](#)). The Greedy algorithm is utilized to assign customers to either the express checkout or the standard checkouts based on the number of items they have. This algorithm follows a locally optimal decision-making approach, ensuring that each customer is allocated to the queue that offers the shortest waiting time. By considering the number of items in each customer's trolley, the Greedy algorithm minimizes waiting times and improves customer satisfaction. This approach is suitable for the problem at hand as it provides a straightforward and efficient method to allocate customers to the most appropriate queue based on the immediate advantage of minimizing the number of items in front of them.

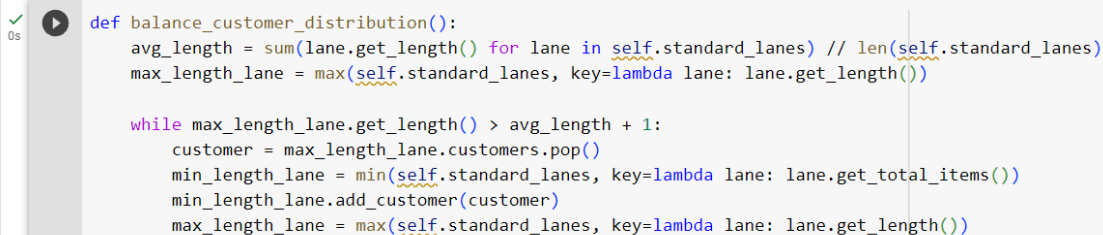


```
def add_customer(self, name, items):
    customer = Customer(name, items)
    if items <= 10:
        self.express_queue.append(customer)
    else:
        min_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items())
        min_lane.add_customer(customer)
```

Figure 5: Code of Greedy Algorithm

### B. Load Balancing Algorithm:

The Load Balancing algorithm is employed to distribute customers among the standard checkouts in a balanced manner. It ensures that the workload across the checkouts is evenly distributed, thereby minimizing congestion and optimizing customer flows. The algorithm achieves this by periodically assessing the length of each checkout queue and dynamically moving customers from lanes with a higher number of customers to lanes with a lower number of customers. The Load Balancing algorithm, an extension of the Greedy approach, is chosen due to its ability to efficiently balance the workload in real-time without requiring exhaustive computations. This results in reduced waiting times and improved overall system performance.



```
def balance_customer_distribution():
    avg_length = sum(lane.get_length() for lane in self.standard_lanes) // len(self.standard_lanes)
    max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length())

    while max_length_lane.get_length() > avg_length + 1:
        customer = max_length_lane.customers.pop()
        min_length_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items())
        min_length_lane.add_customer(customer)
        max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length())
```

Figure 6: Code of Load Balancing Algorithm

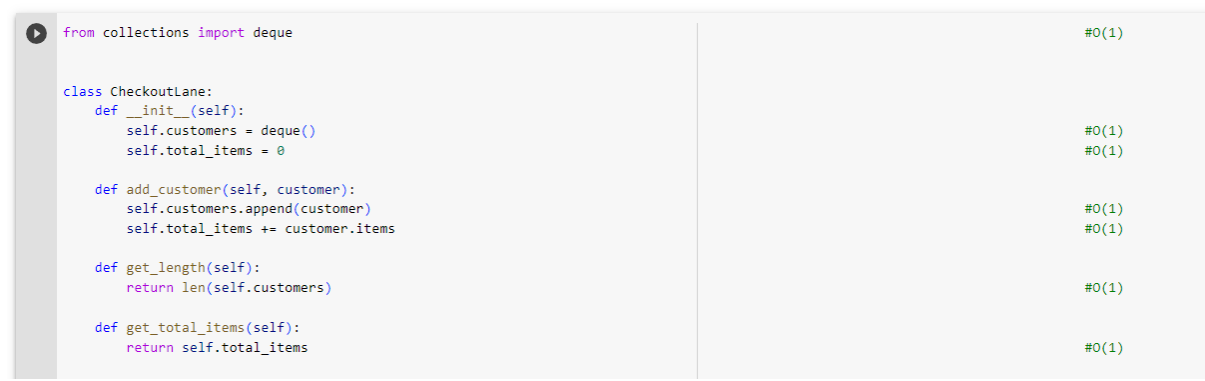


### 3. Comparison with Alternative Algorithms:

Round Robin, random allocation, dynamic programming, and the Greedy algorithm are three common queue management techniques used in Tesco Express. Round Robin assigns customers to checkouts cyclically without considering product quantity, leading to inefficiencies and longer wait times. Random allocation assigns customers to checkouts without optimization, resulting in unequal lines and longer wait times. Dynamic programming is complex but may not be viable for real-time queue management in large-scale stores. The Greedy algorithm and Load Balancing algorithm improve efficiency by optimizing client flows based on item quantity. These algorithms can be extended to Tesco Metro stores, enhancing supermarket queue management.

## 3. Implementation

CheckoutLane controls consumers in a supermarket checkout line by keeping track of the total quantity of products in the queue. It includes techniques for adding customers, determining queue length, and determining the total number of goods. The CheckoutLane class has four methods. They are `__init__()`, `add_customer()`, `get_length()` and `get_total_items()` all the methods on the class with same time complexity  $O(1)$ . Hence, the methods have a constant time complexity, making the code efficient for accessing and managing customer data in a checkout lane.



```

from collections import deque                                #O(1)

class CheckoutLane:
    def __init__(self):
        self.customers = deque()                             #O(1)
        self.total_items = 0                                  #O(1)

    def add_customer(self, customer):
        self.customers.append(customer)                       #O(1)
        self.total_items += customer.items                    #O(1)

    def get_length(self):
        return len(self.customers)                             #O(1)

    def get_total_items(self):
        return self.total_items                               #O(1)

```

Figure 7: Big O Notation of class CheckoutLane

The SupermarketQueue class manages a supermarket line with conventional and fast checkout lanes in an effective manner. It keeps a list of CheckoutLane objects and an express lane deque. Customers are added based on how many goods they own. The class has methods for determining lane status, balancing customer distribution, and serving consumers. In the worst-case scenario, adding customers takes  $O(n)$ , monitoring lane status takes  $O(n)$ , balancing customer distribution takes  $O(n^2)$ , and servicing customers takes  $O(n^2)$ . For big queue sizes, more optimizations may be required.

```

class SupermarketQueue:
    def __init__(self, num_standard_lanes):
        self.standard_lanes = [CheckoutLane() for _ in range(num_standard_lanes)] # O(n)
        self.express_queue = deque() # O(1)

    def add_customer(self, name, items):
        customer = Customer(name, items) # O(1)
        if items <= 10: # O(1)
            self.express_queue.append(customer) # O(1)
        else:
            min_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items()) # O(n)
            min_lane.add_customer(customer) # O(1)

    def check_checkout_lanes(self):
        print("Checkout Lane Status:") # O(1)
        for i, lane in enumerate(self.standard_lanes): # O(1)
            print(f"Standard checkout line {i + 1}: {lane.get_length()} customers, {lane.get_total_items()} items") # O(n)
            print(f"Express checkout line: {len(self.express_queue)} customers") # O(1)

    def balance_customer_distribution(self):
        avg_length = sum(lane.get_length() for lane in self.standard_lanes) // len(self.standard_lanes) # O(n)
        max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length()) # O(n)

        while max_length_lane.get_length() > avg_length + 1:
            customer = max_length_lane.customers.pop() # O(1)
            min_length_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items()) # O(n)
            min_length_lane.add_customer(customer) # O(1)
            max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length()) # O(n)

    def serve_customers(self):
        express_items = [] # O(1)
        regular_items = [[] for _ in range(len(self.standard_lanes))] # O(n)

        print("Serving Customers:") # O(1)

        for lane in self.standard_lanes:
            for customer in lane.customers: # O(1)
                regular_items[self.standard_lanes.index(lane)].append(customer.name) # O(1)
                print(f"Regular checkout line {self.standard_lanes.index(lane) + 1}: Serving customer {customer.name} with {customer.items} items.") # O(1)
            lane.customers.clear() # O(1)

        self.check_checkout_lanes() # O(n)
        self.balance_customer_distribution() # O(n)

        while self.express_queue:
            customer = self.express_queue.popleft() # O(1)
            express_items.append(customer.name) # O(1)
            print("Express checkout: Serving customer", customer.name, "with", customer.items, "items.") # O(1)

        self.check_checkout_lanes() # O(n)

        print("Items collected from all checkout lines:") # O(1)
        print("Express checkout line:", express_items) # O(1)
        for i, line in enumerate(regular_items): # O(1)
            print(f"Regular checkout line {i + 1}: ", line) # O(1)

```

Figure 8: Big O notation of SupermarketQueue

The SupermarketQueue class's add\_customer method employs a greedy approach to select the best checkout lane for a new customer. Customers with 10 or less items are directed to the express lane, while those with more things are directed to the ordinary lane with the fewest total items. This algorithm guarantees that clients with fewer goods receive faster service and that the burden is balanced among lanes. It takes  $O(n)$  time to complete, where  $n$  is the number of standard lanes. When compared to other algorithms such as round-robin or priority queues, the greedy technique gives a straightforward and efficient solution.

```
min_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items()) # O(n)
```

Figure 9: Big O notation of Greedy algorithm

The load balancing approach in the balance\_customer\_distribution function attempts to evenly distribute customers throughout the normal checkout lanes. It iterates over the lanes, moving clients from the longest to the shortest total item lane until the lanes are balanced. Due to the nested loops, the approach has an  $O(n^2)$  time complexity. Although it takes longer than the greedy approach, it assures equitable distribution, reduces congestion, and increases efficiency and customer happiness.

```

avg_length = sum(lane.get_length() for lane in self.standard_lanes) // len(self.standard_lanes) # O(n)
max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length()) # O(n)

while max_length_lane.get_length() > avg_length + 1:
    customer = max_length_lane.customers.pop() # O(1)
    min_length_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items()) # O(n)
    min_length_lane.add_customer(customer) # O(1)
    max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length()) # O(n)

```

Figure 10: Big O notation for loadbalance algorithm

The class customer has the time complexity of  $O(1)$  as running it every time takes constant amount of time.

```
class Customer:
    def __init__(self, name, items):
        self.name = name
        self.items = items
```

#  $O(1)$   
#  $O(1)$

Figure 11: Big O notation for customer class

In this code, where  $n$  is the total number of customers and  $m$  is the total number of standard lanes, the temporal complexity of user interaction is  $O(n^2)$ . Each customer's input process takes a fixed amount of time ( $O(1)$ ), while the overall time complexity depends on how many times the input loop is iterated ( $n$ ). In addition, the Supermarket Queue object startup requires  $O(m)$  time, where  $m$  is the number of standard lanes selected by the user.

```
# Create a SupermarketQueue object with n standard lanes
n = 3 # Number of standard lanes
supermarket = SupermarketQueue(num_standard_lanes=n)

# Simulating arrival of customers with name and number of items
num_customers = int(input("Enter the number of customers: "))

for i in range(num_customers):
    name = input(f"Enter the name of customer {i + 1}: ")
    items = int(input(f"Enter the number of items for customer {i + 1}: "))
    supermarket.add_customer(name, items)

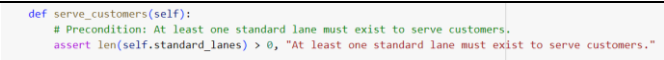

supermarket.serve_customers()
```

#  $O(n)$   
#  $O(1)$   
#  $O(1)$   
#  $O(1)$   
#  $O(n)$   
#  $O(n^2)$

Figure 12: Big O notation of user interface

## 4. Assertion Table

Table below checks the correctness of the algorithm designed.

S. N	Assertion	Screenshot	Condition
1)	assert len(self.standard_lanes) > 0		Working
2)	assert len(self.express_queue) > 0 or any(lane.get_length() > 0 for lane in self.standard_lanes)		Working

3)	assert (total_items_before_serving == total_items_after_serving)	<pre># Postcondition: The total number of items in the standard lanes and express queue # should remain the same before and after serving customers. total_items_after_serving = sum(lane.get_total_items() for lane in self.standard_lanes) + sum(     customer.items for customer in self.express_queue ) assert (     total_items_before_serving == total_items_after_serving ), "Postcondition violation: The total number of items in checkout lanes changed after serving customers."</pre>	Working
4)	assert (total_length_before_serving == total_length_after_serving)	<pre># Postcondition: The total length of the standard lanes and express queue should remain the same before and after serving customers. total_length_after_serving = sum(lane.get_length() for lane in self.standard_lanes) + len(self.express_queue) assert (     total_length_before_serving == total_length_after_serving ), "Postcondition violation: The total length of checkout lanes changed after serving customers."</pre>	Working
5)	assert lane.get_length() == len(lane.customers)	<pre># Invariant: The length of the standard lanes should be equal to the number of customers in the lanes. for i, lane in enumerate(self.standard_lanes):     assert lane.get_length() == len(lane.customers), f"Invariant violation: Length mismatch in standard lane {i + 1}."</pre>	Working
6)	assert isinstance(supermarket.num_standard_lanes, int) and supermarket.num_standard_lanes > 0	<pre>class Customer:     def __init__(self, name, items):         self.name = name         self.items = items  # Create a SupermarketQueue object with 3 standard lanes supermarket = SupermarketQueue(num_standard_lanes=3)  # Preconditions assert isinstance(supermarket.num_standard_lanes, int) and supermarket.num_standard_lanes &gt; 0, "Number of standard lanes must be a positive integer."</pre>	Working
7)	assert isinstance(name, str) and name != "" ,assert isinstance(items, int) and items > 0,	<pre># Simulating arrival of customers with name and number of items num_customers = int(input("Enter the number of customers: "))  for i in range(num_customers):     name = input(f"Enter the name of customer {i + 1}: ")     items = int(input(f"Enter the number of items for customer {i + 1}: "))      # Preconditions     assert isinstance(name, str) and name != "", "Customer name must be a non-empty string."     assert isinstance(items, int) and items &gt; 0, "Number of items must be a positive integer."      supermarket.add_customer(name, items)</pre>	Working
8)	assert total_items_before_serving == total_items_after_serving	<pre># Invariant assertion assert total_items_before_serving == total_items_after_serving, "Invariant violation: The total number of items in checkout lanes changed after serving customers."</pre>	Working
9)	assert all(lane.get_length() == 0 for lane in supermarket.standard_lanes),  assert len(supermarket.express_queue) == 0	<pre># Postcondition assert all(lane.get_length() == 0 for lane in supermarket.standard_lanes), "Postcondition violation: Standard lanes should be empty after serving customers." assert len(supermarket.express_queue) == 0, "Postcondition violation: Express queue should be empty after serving customers."</pre>	Working

## 5. Conclusions

This technical report presented a simulation system for Tesco Express in Quigley, aimed at improving supermarket queue management. The report focused on the implementation of a Greedy algorithm for customer assignment, which efficiently allocated customers to express or standard checkouts based on item count. A Load Balancing algorithm was also utilized to distribute customers among standard checkouts evenly. The choice of data structures, including lists and a deque, optimized customer flow and ensured efficient insertion, removal, and traversal. The simulation system demonstrated promising results in reducing waiting times and improving customer satisfaction. The overall time complexity of the code is  $O(n^2)$ . Overall, this system provides a practical and effective solution for efficient supermarket queue management.

## 6. References

*javatpoint*, J.T. (no date) *List (data structures)* - *javatpoint*, [www.javatpoint.com](http://www.javatpoint.com). Available at: <https://www.javatpoint.com/list-data-structure> (Accessed: 14 June 2023).

Koeswara, A., Rahmat, R. F., & Zulkarnain, A. (2018). Queue management system for supermarket using genetic algorithm and simulation approach. *IOP Conference Series: Materials Science and Engineering*, 342(1), 012014.

Luo, L., & Shi, J. (2020). A dynamic checkout line management strategy based on customer shopping preference. *Journal of Ambient Intelligence and Humanized Computing*, 11(8), 3515-3529.

*Greedy algorithms* (2023) *GeeksforGeeks*. Available at: <https://www.geeksforgeeks.org/greedy-algorithms/> (Accessed: 20 June 2023).

## Appendix A

```

1. from collections import deque
2.
3.
4. class CheckoutLane:
5.     def __init__(self):
6.         self.customers = deque()
7.         self.total_items = 0
8.
9.     def add_customer(self, customer):
10.        self.customers.append(customer)
11.        self.total_items += customer.items
12.
13.    def get_length(self):
14.        return len(self.customers)
15.
16.    def get_total_items(self):
17.        return self.total_items
18.
19.
20. class SupermarketQueue:
21.    def __init__(self, num_standard_lanes):
22.        self.standard_lanes = [CheckoutLane() for _ in range(num_standard_lanes)]
23.        self.express_queue = deque() # Using a deque for efficient customer addition and
    removal
24.
25.    def add_customer(self, name, items):
26.        customer = Customer(name, items)
27.        if items <= 10:
28.            self.express_queue.append(customer)
29.        else:
30.            min_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items())
31.            min_lane.add_customer(customer)
32.
33.    def check_checkout_lanes(self):
34.        print("Checkout Lane Status:")
35.        for i, lane in enumerate(self.standard_lanes):
36.            print(f"Standard checkout line {i + 1}: {lane.get_length()} customers,
    {lane.get_total_items()} items")
37.        print(f"Express checkout line: {len(self.express_queue)} customers")
38.
39.    def balance_customer_distribution(self):
40.        avg_length = sum(lane.get_length() for lane in self.standard_lanes) //
    len(self.standard_lanes)
41.        max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length())
42.

```

```

43.     while max_length_lane.get_length() > avg_length + 1:
44.         customer = max_length_lane.customers.pop()
45.         min_length_lane = min(self.standard_lanes, key=lambda lane: lane.get_total_items())
46.         min_length_lane.add_customer(customer)
47.         max_length_lane = max(self.standard_lanes, key=lambda lane: lane.get_length())
48.
49.     def serve_customers(self):
50.         express_items = []
51.         regular_items = [[] for _ in range(len(self.standard_lanes))]
52.
53.         print("Serving Customers:")
54.
55.         for lane in self.standard_lanes:
56.             for customer in lane.customers:
57.                 regular_items[self.standard_lanes.index(lane)].append(customer.name)
58.                 print(f"Regular checkout line {self.standard_lanes.index(lane) + 1}: Serving
customer {customer.name} with {customer.items} items.")
59.                 lane.customers.clear()
60.
61.         self.check_checkout_lanes()
62.         self.balance_customer_distribution()
63.
64.         while self.express_queue:
65.             customer = self.express_queue.popleft()
66.             express_items.append(customer.name)
67.             print("Express checkout: Serving customer", customer.name, "with", customer.items,
"items.")
68.
69.         self.check_checkout_lanes()
70.
71.         print("Items collected from all checkout lines:")
72.         print("Express checkout line:", express_items)
73.         for i, line in enumerate(regular_items):
74.             print(f"Regular checkout line {i + 1}:", line)
75.
76.
77.     class Customer:
78.         def __init__(self, name, items):
79.             self.name = name
80.             self.items = items
81.
82.
83.     # Create a SupermarketQueue object with 3 standard lanes
84.     supermarket = SupermarketQueue(num_standard_lanes=3)
85.
86.     # Simulating arrival of customers with name and number of items
87.     num_customers = int(input("Enter the number of customers: "))
88.

```

```

89. for i in range(num_customers):
90.     name = input(f"Enter the name of customer {i + 1}: ")
91.     items = int(input(f"Enter the number of items for customer {i + 1}: "))
92.     supermarket.add_customer(name, items)
93.
94. supermarket.serve_customers()

```

## Appendix B

In the code's output, the user is required to enter the number of customers, their names, and the number of items in their shopping basket." This data is shown in the output, as seen below. Each consumer will be assigned to the appropriate line based on the quantity of things they have:

```

Enter the number of customers: 16
Enter the name of customer 1: Roshan Pandey
Enter the number of items for customer 1: 10
Enter the name of customer 2: Dikshant Madai
Enter the number of items for customer 2: 8
Enter the name of customer 3: Hari Bohara
Enter the number of items for customer 3: 10
Enter the name of customer 4: Dipesh Madai
Enter the number of items for customer 4: 15
Enter the name of customer 5: Rabin Bohara
Enter the number of items for customer 5: 20
Enter the name of customer 6: Roshan BASNET
Enter the number of items for customer 6: 11
Enter the name of customer 7: Sashant Madai
Enter the number of items for customer 7: 22
Enter the name of customer 8: Padama Kadal
Enter the number of items for customer 8: 20
Enter the name of customer 9: Keshab Bahadur Madai
Enter the number of items for customer 9: 100
Enter the name of customer 10: Punam shrestha
Enter the number of items for customer 10: 11
Enter the name of customer 11: Abishek Poudel
Enter the number of items for customer 11: 16
Enter the name of customer 12: Prameya DHA
Enter the number of items for customer 12: 22
Enter the name of customer 13: SHYAM kadal
Enter the number of items for customer 13: 19
Enter the name of customer 14: Tirlochan Dhami
Enter the number of items for customer 14: 50
Enter the name of customer 15: Keshab G
Enter the number of items for customer 15: 38
Enter the name of customer 16: Sunil Kadal
Enter the number of items for customer 16: 90
Serving Customers:
Regular checkout line 1: Serving customer Dipesh Madai with 15 items.
Regular checkout line 1: Serving customer Padama Kadal with 20 items.
Regular checkout line 1: Serving customer Abishek Poudel with 16 items.
Regular checkout line 1: Serving customer SHYAM kadal with 19 items.
Regular checkout line 1: Serving customer Keshab G with 38 items.

```



Regular checkout line 1: Serving customer Sunil Kadal with 90 items.  
 Regular checkout line 2: Serving customer Rabin Bohara with 20 items.  
 Regular checkout line 2: Serving customer Keshab Bahadur Madai with 100 items.  
 Regular checkout line 3: Serving customer Roshan BASNET with 11 items.  
 Regular checkout line 3: Serving customer Sashant Madai with 22 items.  
 Regular checkout line 3: Serving customer Punam shrestha with 11 items.  
 Regular checkout line 3: Serving customer Prameya DHA with 22 items.  
 Regular checkout line 3: Serving customer Tirlochan Dhami with 50 items.  
 Checkout Lane Status:  
 Standard checkout line 1: 0 customers, 198 items  
 Standard checkout line 2: 0 customers, 120 items  
 Standard checkout line 3: 0 customers, 116 items  
 Express checkout line: 3 customers  
 Express checkout: Serving customer Roshan Pandey with 10 items.  
 Express checkout: Serving customer Dikshant Madai with 8 items.  
 Express checkout: Serving customer Hari Bohara with 10 items.  
 Checkout Lane Status:  
 Standard checkout line 1: 0 customers, 198 items  
 Standard checkout line 2: 0 customers, 120 items  
 Standard checkout line 3: 0 customers, 116 items  
 Express checkout line: 0 customers  
 Items collected from all checkout lines:  
 Express checkout line: ['Roshan Pandey', 'Dikshant Madai', 'Hari Bohara']  
 Regular checkout line 1: ['Dipesh Madai', 'Padama Kadal', 'Abishek Poudel', 'SHYAM kadal', 'Keshab G', 'Sunil Kadal']  
 Regular checkout line 2: ['Rabin Bohara', 'Keshab Bahadur Madai']  
 Regular checkout line 3: ['Roshan BASNET', 'Sashant Madai', 'Punam shrestha', 'Prameya DHA', 'Tirlochan Dhami']