LOYOLA SCHOOLS
School of Science and Engineering
Department of Information Systems and Computer Science
First Semester
2024-2025

Customer Relationship Management (CRM) Implementation for CostKo

MSYS 30
Data Structures and Algorithms
Final Project Report

Submitted by:

Daveigh Kate Abogado
Nathaniel Benedicto
Marco Luis Coros

Submitted on: November 28, 2024

Project Background

Context

CostKo, Inc. is an established retail company specializing in consumer goods sales. It has a number of branches throughout the Visayas-Mindanao region, and the company is planning to expand to Luzon with a new membership shopping business model.

Currently, each individual branch provides a company-wide discount coupon. The discount coupon is only provided on single transactions that reach the specified amount. Cumulative transactions are not tracked, and neither are the individual products being sold. Each branch has an idea of what goods are being sold by tracking inventory, but they don't know who exactly purchases what.

This, however, has not solved the problem of ensuring a steady cash flow of transactions. The company believes that this can be rectified by encouraging customers to make smaller purchases regularly. They also believe that it would be helpful to keep track of who is using the discount coupons. **The company sees the value in storing consumer data and is open to analyzing patterns in purchases in order to optimize their branches' performance.** Certain branches tried to keep track of individual sales by making a customer tab and reading receipts associated with those customers, but **the practice soon fell off due to the sheer amount of work needed to manually track all purchases made by customers**—hence their request for it integrated with their new membership shopping business model.

Another problem encountered was the increase in time needed to process customers at the counter. With more customers using coupons, branch owners found that delays became common as cashiers experienced transaction delays.

Targeted Business Process

With this in mind, the main process to be targeted is the operations management process; specifically, **the company's handling of their customer membership retention and loyalty program**. This case is based on the assumption that the company has already created a membership registration system under their operations management via their new membership shopping business model.

Need

With the company's current plan for membership shopping expansion into the Luzon region, the company urgently requires a solution that can transition the old system of their loyalty program into a modern, scalable system. With the new member-based business model, the coupon method as a loyalty program should be replaced to the point system due to its inability to:

Track Customer Activity in Real Time

There is a need for **a centralized system that can monitor and update customer loyalty points instantly after each transaction**, rather than waiting for all coupons to be punched. A points system simplifies the tracking of customer loyalty as well as simplifying the granting of discounts.

Improve Scalability

As the company makes the transition, the old loyalty system becomes obsolete and does not match the new business model. The old system struggles to track customers and keep up with the influx of customers leading to delays and inefficiencies in data reporting. Thus, **a more scalable solution** is needed to avoid compromising speed or accuracy and to further complement the new membership system

Enhance Customer Experience

The manual printed process often frustrates customers, negatively affecting their perception of the loyalty program and their likelihood to engage with it. A more streamlined, user-friendly digital solution is needed to improve the overall customer experience and encourage continued loyalty—one that they can take with them anywhere without worrying about bringing physical coupons with them.

With these in mind, a modern, algorithm-driven customer loyalty program that can register, update, sort on specific parameters, and search customer information to build a centralized, real time tracking system is primarily needed to address these growing operational inefficiencies.

Value

Implementing this algorithm-driven loyalty program will provide significant value in the following ways:

1. **Reduce the time needed to register and update customer information**
    a. Transactions will be recorded in real-time, eliminating the delays caused by the initial physical coupon system.
        i. Points of members update as a transaction is registered under them.

2. **Enable more efficient management of customer data**
    a. Allowing employees to easily query and retrieve customer data — their personal details and points.

Ultimately, the enhanced customer experience will strengthen brand loyalty and drive long-term business growth that complements the new membership system they are vying for

Design and Implementation

Functional Requirements

The project implemented a system wherein an administrator is able to log in to the system and access the following information:

1. **Member Masterlist**

   The member masterlist contains a centralized database of all members of CostKo, Inc. across all branches in the Philippines. This database tracks the points each member has gained so far–updating the moment a member has made a purchase that would give them points.

   The member masterlist shall contain the following columns as sampled in the table below.

Table 1. Member Masterlist

Branch	*Member ID*	*Last Name*	*First Name*	*Birthdate*	*Date added*	*Points*
Butuan	1	Abogado	Daveigh	2002/01/01	2015/03/15	2133
…	…	…	…	…	…	…

2. **Transaction Masterlist**

   The member masterlist contains a centralized database of all transactions made with CostKo, Inc. across all branches in the Philippines. As this database is populated by transactions with varying total purchase values, the points of the customer who made a transaction is updated in the Member Masterlist. A customer gains 1 point for every Php 10 purchase.

   The transaction masterlist shall contain the following columns as sampled in the table below.

Table 2. Transaction Masterlist

Transaction ID	*Customer ID*	*Date and Time*	*Total Purchase Value*
1	0000001	2024/10/07	5320.26

…	…	…	…

When accessing the masterlists, the administrator must be able to perform the following actions:

- Sort the members based on the following criteria:
    - Number of points
    - Date Added
    - Birthdate
- Sort the transactions based on the following criteria:
    - Date
    - Total purchase value
- Search the Member masterlist for a specific customer's information using the customer's name.
- Search the Transactions masterlist for a specific transaction using the customer's name.

System Requirements

The system is a web tool developed using HTML, CSS, and Django utilizing Bootstrap UI elements.

Proposed Solution Approach

Merge sort was chosen as the sorting algorithm for the system given that its average case time complexity is consistently O(n log n) even in terms of the worst and best case. While utilizing merge sort poses the need for additional memory space from storing a separate array during runtime, the company has already dedicated external memory for their customer information database. Thus, merge sort was chosen as the sorting algorithm for handling the large dataset of customer information given its consistency and efficiency.

Binary search is the fastest way of searching through large datasets which is vital for a company that is expected to have large datasets. It requires a given dataset to be already sorted before its use. Since merge sort is planned to be utilized in sorting the data prior to use, this will not be an additional problem to solve. Its average and worst time complexity is O(log n) which makes it ideal even for the largest of data sets, unlike linear search with a much slower time complexity of O(n). Binary search is therefore the most efficient solution that can be used for the given application.

Project Timeline

Due Date	Task
2024 October 07	Create project proposal ● Scope ● Features

2024 October 12	Begin creation of the Django program by creating: ● UI ○ Basic view ■ Template page ● Models ○ Customer table ○ ~~Point System conversion table~~ ○ Transaction tracking table
2024 October 26	● Search algorithm function ● Sorting algorithm function
2024 November 09	● UI ○ Functional, integrated with functions ○ Sort/filter Customer: ■ Number of points ■ Date added / registration date ■ Birthdate (to get age) ○ Sort/filter Transactions ■ Date ■ Total amount ■ Number of points ■ Receipt number ○ Search ■ Customer Name ■ Receipt ID
2024 November 16	● Create a test dataset ● Test system performance ● Revisions ● Polishing

Actual Implementation

Model Implementation

Django was used to create the models for the customer table and transactions table. These models were named Member and Transaction. Below can be seen the code used to implement them. The columns for the attributes that are included in the models can be seen. These attributes store the specific types of data necessary for the software.

```
class Member(models.Model):
    Branch= models.CharField(max_length=300)
    LastName= models.CharField(max_length=300)
    FirstName= models.CharField(max_length=300)
```

```
    Birthdate= models.DateField()
    DateAdded= models.DateField(auto_now_add=True)
    Points= models.DecimalField(default=0, max_digits=8,decimal_places=2)
    SortID= models.IntegerField(default=0)

    def __str__(self):
         return f"{self.FirstName} {self.LastName}"

class Transaction(models.Model):
    CustomerNumber= models.ForeignKey(Member, on_delete=models.CASCADE)
    DateTime= models.DateField(auto_now_add=True)
    TotalPurchaseValue= models.DecimalField(max_digits=8, decimal_places=2)
    SortID= models.IntegerField(default=0)
```

Search Algorithms

The initial solution approach only included the use of binary search in the implementation. During development we realized that it was important to also be able to search using partial names, and not just full names. This led us to implementing linear search as well. While it was generally slightly slower, it did allow for the software to search for partial matches.

```
## Searching if the input is NOT formatted as FirstName_LastName
def linear_search(input_queryset, query):
    sorted_list = input_queryset.order_by('pk')
    sorted_list = list(sorted_list.values('pk', 'FirstName',
'LastName'))
    matching_records = []

    for customer in sorted_list:
      if query.lower() in f"{customer['FirstName']}
{customer['LastName']}".lower():
        matching_records.append(customer)

    return matching_records
```

Since it was necessary to distinguish between user inputs that could or couldn't be used with binary search, the following code was used to check if the input was following the FirstName (space) LastName format. If a space was found, then that would mean that it was following the format, and thus binary search would be used. Otherwise, linear search would be used.

```
def check_search_type(input_queryset, query):
  if " " in query: ## Checks if there is a space in the middle of the
query like in FirstName_LastName
      return binarysearch(input_queryset, query)
  else:
      return linear_search(input_queryset, query)
```

The binary search code is found below. During implementation, it was found that a single linear search function could not support both Member and Transaction search. A second linear search function was made to accommodate the details necessary for Transaction search. The difference with the two is that instead of searching for a string, an exact numerical match was to be found.

```
## Searching if the input is formatted as FirstName_LastName
def binarysearch(input_queryset, query):
  sorted_list = input_queryset.order_by('FirstName', 'LastName')
  sorted_list = list(sorted_list.values('pk', 'FirstName',
'LastName'))
  query = query.lower()
  return_list = []
  ## Format of sorted_list made to become similar to the DFs used
  ## when we made the binary search in class
  low = 0
  high = len(sorted_list) - 1
  while low <= high:
    mid = (low + high) // 2
    locate_mid = f"{sorted_list[mid]['FirstName']}
{sorted_list[mid]['LastName']}".lower()
    ## Format is FirstName (space) LastName
    if locate_mid == query:
      return_list.append(sorted_list[mid])
      return return_list
      ## This returns the matching record as a dictionary containing
pk, FirstName, LastName
    elif locate_mid < query:
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

Sorting Algorithms

For the sorting, a merge function was made with different switch cases to accommodate different parameters. This same function was flexible enough to be used with both Member and Transaction models. A snippet of the code is shown below, as the full extent is fairly long due to the switch cases.

```
def merge(a, b, switch_case):
    c = []
    attribute_map = { # Map switch_case values to their corresponding
attribute names
        0: 'Birthdate',
        1: 'Points',
        2: 'DateAdded',
        3: 'DateTime',
        4: 'TotalPurchaseValue',
    }

    attr = attribute_map.get(switch_case)
```

The merge sort function itself is fairly standard, with the biggest difference being that a switch case variable is being passed around so that the right attributes are actually used.

```
def mergesort(a, c):
  if len(a) <=1:
    return a
  split = len(a)//2
  switch_case = c

  a1 = a[:split]
  a2 = a[split:]

  a1= mergesort(a1, switch_case)
  a2 = mergesort(a2, switch_case)

  return merge(a1,a2, switch_case)
```

Passing Data from Backend to Frontend

All the sorting and search algorithms won't make any sense if no data is passed to them, or if nothing is sent from them to the frontend. To achieve this, a link needed to be made between the HTML files and the data that was being passed around in the back. This was done through

the *views* file and by manipulating the HTML files so that they could receive or send data to the relevant functions in the *views* file. An example of this can be seen below.

```
      def search_transactions(request):
        results = []
        not_in_record = False
        transactions_objects = Transaction.objects.all()

        ## Get list of all matching Member entries
        if (request.method == "POST"):
          input_name = int(request.POST.get('customer_id'))
          if input_name:
            results = linear_search_transaction(transactions_objects,
input_name)
            if len(results) == 0:
              not_in_record = True
        else:
          not_in_record = True

        return render(request, 'costko/transactions.html',
{'transactions':transactions_objects, 'results': results, 'not_in_record':
not_in_record})
```

Here, a function called *search_transactions* is found. Since it has the parameter *request*, it is able to interact with the front end portion of the software. It does this using *POST*, which is shared with the HTML file that we want to interact with.

```
<form method="POST" action="{% url 'search_transactions' %}" class="row g-3">{%
csrf_token %}
            <div class="form-group">
                <label for="customer"> Customer ID: </label>
                <input required type="number" min="0" class="form-control"
id="customer_id" name="customer_id" oninput="fetchCustomerData()">
            </div>
```

POST and *% url 'search_transactions' %* allow a form on the front end to interact with the back end *views* function. It is *customer_id* that tells the function that the contents of the input form are to be used by the function. The line below is how the backend makes use of it.

```
 input_name = int(request.POST.get('customer_id'))
```

This essentially gets the content of that field in the form and allows it to be used as a value. Note that in this case it was necessary to have it set as an *int* as by default form inputs are in *string*.

For sending data back to the front end, the line below was used for this specific function.

```
return render(request, 'costko/transactions.html',
{'transactions':transactions_objects, 'results': results, 'not_in_record':
not_in_record})
```

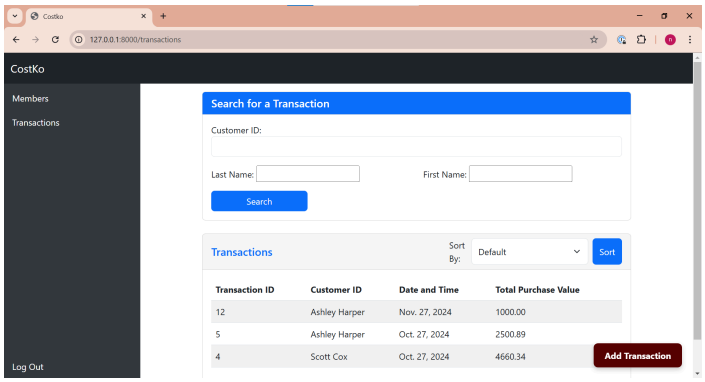
What this part does is that it takes the contents of the variables (in blue) and makes them accessible to the HTML file as long as the contents in quotation marks are instantiated. Below is a snippet of how *results* was used.

```
 {% for result in results %}
            <tr>
                <td>{{ result.pk }}</td>
                <td>{{ result.CustomerNumber }}</td>
                <td>{{ result.DateTime }}</td>
                <td>{{ result.TotalPurchaseValue }}</td>
            </tr>
        {% endfor %}
```

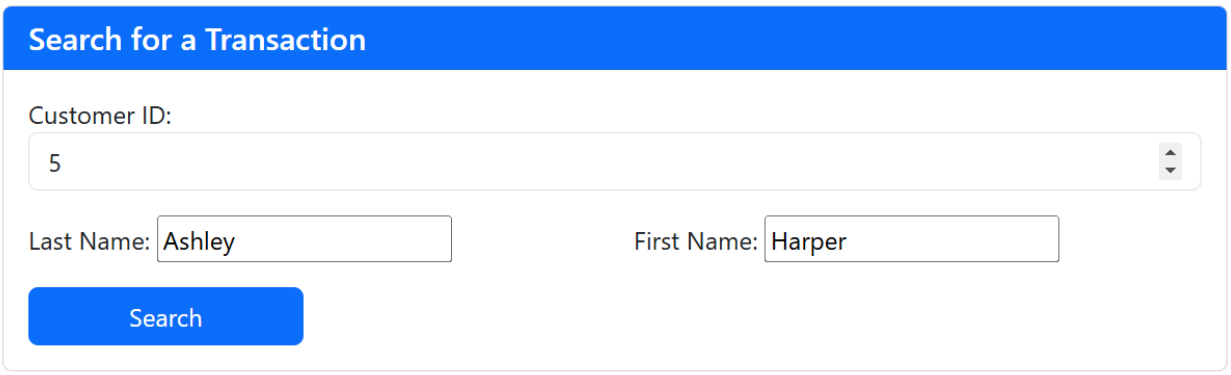
Since *results* is output as a list, a for loop can be run inside the HTML file to display contents of the list. Also, since *result* is an object of Transaction, its properties (the fields specified) can be accessed.

User Interface

For the user, how this would look and feel—after logging in—should be fairly simple. Taking the previous example and seeing it from the user's perspective, the following view can be seen.

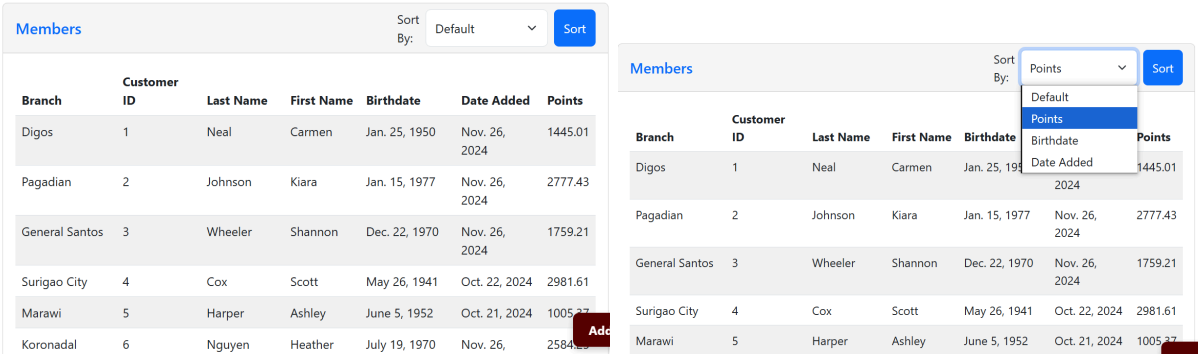


For the user to perform a search of all the transactions a member has made, they must enter the customer ID of that member. The display automatically shows the name of the customer associated with the ID, helping to make sure that the user is searching for the right person.

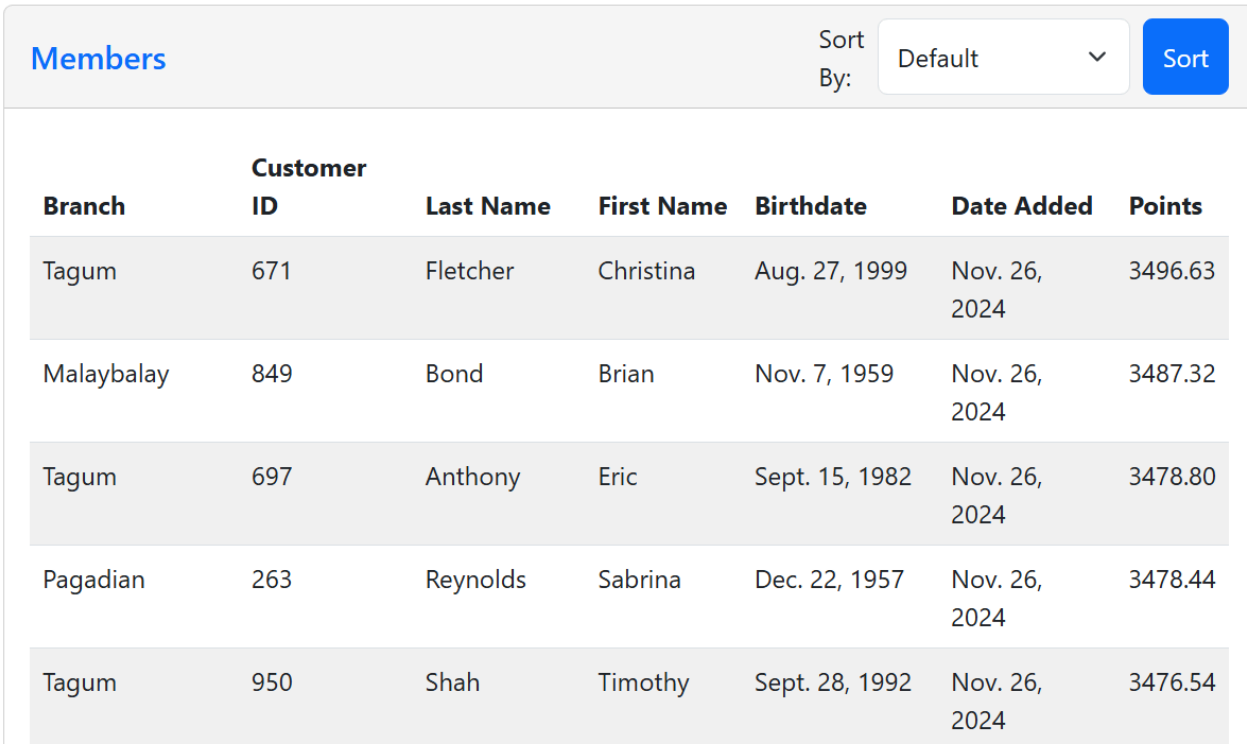


As soon as the user clicks on “Search” the site will display all transactions associated with that person. Or, if no transactions are found, a message will display showing the same. All this happens as the front and back ends interact with each other to pass the relevant input data and resulting output.

The member search works similarly in the front end, with some differences in the back end to accommodate for the differences in data used. The sorting algorithm works exactly the same for both members and transactions. To use it, the user simply has to select the parameter they want to sort by, and then click sort.



The display will automatically update to show the sorted table.



Data Structures Used

The program makes use of the following data structures in its implementation:

1. List
2. Dictionary
3. Tuple

Lists were used throughout the program. In this example, a list is used to store objects from the model so that their attributes can all be accessed.

```
 if sortby=='default':
        for transaction in transaction_objects:
            transaction.SortID = transaction.pk
        Transaction.objects.bulk_update(transaction_objects, ['SortID'])
      else:
        switch_case = sortby_attributes.index(sortby) + 3 # Index position of the
sortby in the tuple +3 is the switch_case equivalent
        update_sortID(switch_case)
```

Dictionaries were used for the switch cases used to distinguish between the different attributes that are used to sort.

```
def merge(a, b, switch_case):
    c = []
    attribute_map = { # Map switch_case values to their corresponding attribute
names
```

```
        0: 'Birthdate',
        1: 'Points',
        2: 'DateAdded',
        3: 'DateTime',
        4: 'TotalPurchaseValue',
    }

    attr = attribute_map.get(switch_case)
```

Tuples were used to indicate the position of the attribute that we want to sort using. This is a faster method of getting the index needed for the switch case number since arrays are faster to access than lists.

```
def sort_transactions(request): # This sorts the transactions in the transactions table
    if request.method == "POST":
      sortby = request.POST.get('transactionSort')
      transaction_objects = Transaction.objects.all()
      sortby_attributes = ('date_time','total_purchase_value')
```

Testing Procedures and Results

Binary and Linear Search

System testing involved verifying the correctness and efficiency of both binary and linear search implementations. In order to evaluate the two search algorithms, 10 test runs were conducted for each search method using a **Timer Function** to record execution times. The primary goals of the test were to confirm:

1. Accuracy: do both methods correctly identify the matching record?
2. Efficiency: Execution times for both methods under similar conditions (searching for the same name)
3. Behavior with varied inputs: the ability of the system to choose which of the two algorithms to utilize to handle full-name and partial name searches effectively.

Procedure:

1. A database with a sample set of 1,000 member records was generated and utilized
2. Binary search was tested with inputs formatted as FirstName LastName.
3. Linear search was tested with partial names (e.g., “Nathaniel” or “Benedicto”)

4. The Timer function tracked the execution time for both search methods across 10 iterations and printed the results into the system console for retrieval.

Results:

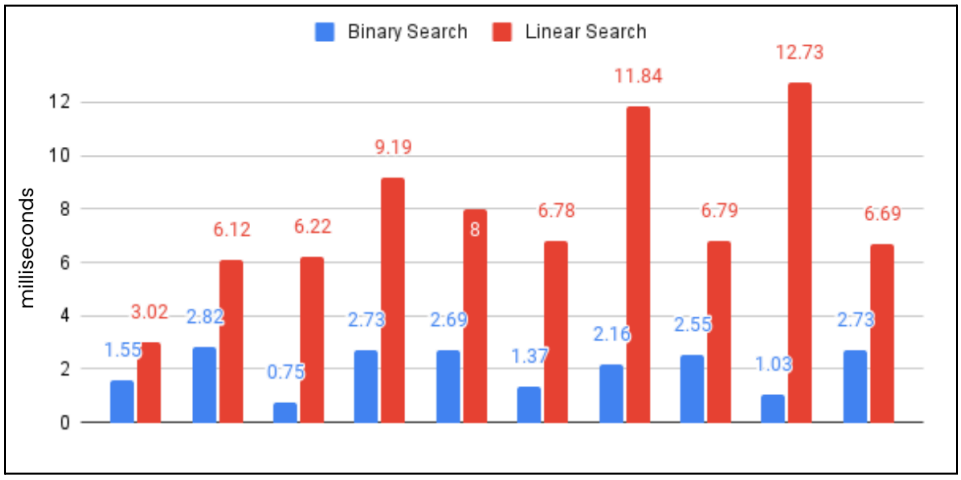


Figure 1. Comparison of Search Algorithm Runs

As shown in Figure 1 above, Binary search consistently outperformed Linear search in terms of execution time. This is as expected with its average runtime being significantly lower due to its logarithmic time complexity o (log n).

In all attempts, the search algorithms were 100% successful in querying the corresponding search parameter — both confirming their correctness and accuracy. Linear search, however, had higher execution times due to its sequential nature, which is expected with its O(n) time complexity. On the other hand, binary search shows lower variability and showed execution times consistently below 7 milliseconds.

These results align with the expected behavior based on the search types: binary search is more efficient, whereas linear search is needed for partial matches but comes with a performance tradeoff.

Mergesort

System testing for **merge sort** and **sort members,** the subsequent database saving process, focused on evaluating the efficiency and performance of both operations. The goal was to determine how well the system handles sorting 1000 members and saving the sorted data to the database. In this case, two key operations were tested: the sorting of data using merge sort and the saving of the sorted data to the database. For this analysis, 10 test runs were conducted, and a Timer function too was employed to record the execution times. The main objectives of the test were:

1. Efficiency of sorting: does merge sort provide optimal performance?

2. Definite the accuracy of the algorithm
3. Differentiate between sorting proper and database performance: which aspect causes the longer execution time?

Procedure:

1. Similar to Linear and Binary search, a database with a sample set of 1,000 member records was generated and utilized
2. Merge sort and sort members are run successively when a sort is called under a certain parameter
3. The Timer function tracked the execution time for both processes across 10 iterations and printed the results into the system console for retrieval.

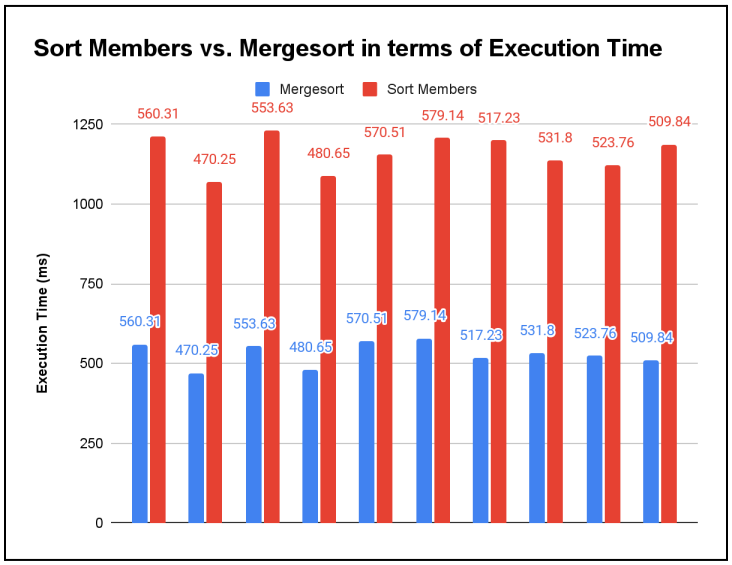


Figure 2. Time Complexity of Mergesort

The data provided shows the execution times for merge sort and sort_members operation across multiple runs. The merge sort times, which reflect the sorting of data (actual implementation of merge sort), consistently fell between relatively stable times between 470.25 ms and 579.14 ms.

On the other hand, the sort_members operation, which involves saving the newly sorted data into the database, takes significantly longer. It's time ranges from 1069.33 ms to 1230.64 ms, indicating that the database-saving process is a bottleneck compared to the sorting step. The disparity between these times suggests that while the merge sort algorithm is performing well, the time-consuming aspect of this process lies in database interactions, likely due to write operations, data insertion, or indexing tasks.

Potential improvements or future enhancements

1. Integration with Inventory Management Systems

A potential improvement to the system would be the feature to sync the CRM with CostKo's inventory systems. This will allow the CostKo management to create dashboards that will help them track purchasing trends and automate inventory restocking based on demand. With this, they are also able to create potential dashboards for real-time decision making. This can also help push for certain items like sustainability products, etc.

2. Mobile App Integration

The current implementation of the web application is created for computers. There can be opportunities to expand the system to cater to more devices — allowing employees/managerial roles to be mobile while on duty (allowing them to aid customers on the spot). Moreover, the current implementation of the application is only for management use. However, there might be an opportunity to also extend this functionality for customers. In this way, they can remotely check their personal current points and even purchasing trends.

3. Branch & Customer Performance Functionalities

One enhancement would be dialing in on generating branch-specific performance reports and customer “loyalty” performance — giving branches and customers the ability to be given tiers and rewards based on their transaction histories. This is an attempt to hit two birds with one stone: customer care and employee motivation.