Venue Availability System

Jana Almira Boco
Student, Ateneo de Manila University
Katipunan Ave., Loyola Heights
Quezon City, Philippines, 1108
Jana.boco@student.ateneo.edu

Nathan Luna
Student, Ateneo de Manila University
Katipunan Ave., Loyola Heights
Quezon City, Philippines, 1108
luis.luna@student.ateneo.edu

Juliana Valdez
Student, Ateneo de Manila University
Katipunan Ave., Loyola Heights
Quezon City, Philippines, 1108
juliana.valdez@student.ateneo.edu

ABSTRACT

The Venue Availability System was developed to aid in the functions of the Central Facilities Management Office (CFMO) of Ateneo de Manila University to streamline venue reservations. The system addresses inefficiencies in the manual checking process by consolidating venue availability into a single platform. Using algorithms such as Binary Search and MergeSort, the system ensures efficient search and sorting operations. Arrays and dictionaries manage large datasets effectively, providing real-time information on venue availability, capacities, and equipment. The project enhances transparency, reduces overbooking, and facilitates a seamless user experience for students. This paper outlines the design, methodology, and implementation of the system, including future enhancement opportunities.

Keywords: Venue Availability, Central Facilities Management Office (CFMO), Binary Search, MergeSort, Data Structures, Ateneo de Manila University, Venue Management

1. Problem Statement & Context

This project is developed for the Central Facilities Management Office (CFMO), an office within Ateneo De Manila University that oversees the maintenance, allocation, and scheduling of venues for various events. These venues are usually reserved by students for events ranging from academic functions to social events for different student groups.

For this project we will be focusing on improving the venue reservation system, specifically the checking of the availability of the venues. In the current process for reserving, users must check the availability of different venues manually. They would need to navigate different parts of their website to see the availability.

This could lead to delays and inefficiencies, specifically the manual checking of the availability of the venues. This process could lead to confusion since it requires users to navigate through various pages to find their venues. These inefficiencies have created the need for a better system that could allow users to reserve venues more efficiently.

This update is especially important now since the improvements have been at a standstill. There is an increasing number of events within Ateneo, thus there has been a need for an upgraded system. Multiple students have had issues with booking venues leading to delays in the approval process. For example, students book a venue at the same time because of the lack of transparency, leading to overbooking. By enhancing the visibility of the available venues, students will be able to reserve venues more efficiently.

1.1 OBJECTIVE

The objective of this project is to create a venue availability system that coordinates the availability of multiple venues into a single monitoring system for students seeking to book venues with the Central Facilities Management Office (CFMO). The aim is to create a system that can seamlessly allow users to view all available venues specific to the needs of their planned activities in a consolidated manner. The system would recommend potential venues that will eliminate the need for users to visit each venue's specific page or link individually.

The following are major activities that can be expected throughout the design and development of this project:

1. Problem Identification where the application of algorithms and data structures may address the current gap in an existing information system;
2. Identification of System Requirements—Algorithmic Requirements, Data Structure Needs alongside a proposed solution approach;
3. Extraction of data from the current Venue Reservation System;
4. Implementation of Searching and Sorting Algorithms—incorporated with the use of elementary data structures;
5. Execution of Evaluative Metrics during testing;
6. Analysis of time and space complexity used in the solution; and
7. Overall Assessment of implemented information systems solution.

1.2 SCOPE & FEATURES

The scope of this project covers the venue reservation system—primarily the CFMO Reservation Portal's venue availability pages that is currently in place. However, for this project, the group decided to focus on only one building, Padre Faura Science Hall (FAURA BUILDING represented by the building code F) due to the time constraints in creating the system.

Additionally, the system can only accept limited inputs for its search including a dropdown selection of time slots that adhere to the most common time slots of reservations, and a limited dropdown selection of 'Sort by' and 'Order' columns.

Furthermore, data has been limited to 1000 records where the look-up dates only range from a month—January 15, 2025 to February 14, 2025. Similarly, the time slots only range from the most frequently occurring class schedules from 8:00 AM to 8:00 PM in increments of 1.5 hours.

The project hopes to better inform students about the availability of rooms/venues for extracurricular activities. Concerning venues/rooms. For the testing and implementation, only venues located in Faura Hall will be considered. Due to the special nature of some venues, the scope does not include rooms with special restrictions and/or other requirements based on the department under which the room falls. This system does not cover the reservation process itself.

The following are the inputs that are relevant to the system:

1. Search Function
    a. Date and Time of reservation
    b. Room Capacity requirement
2. Sort Function
    a. ‘Sort by’
    b. ‘Order’

These inputs will allow for the system to conduct the algorithms required with the best precision and will produce the best-fit outcome, which is as follows:

1. Available rooms are listed in tabular format
2. Features of rooms will include basic information (room capacity, etc)

For the users of the system, which will be students, all the inputs that have been mentioned will come from them. The outputs that they will receive are all of the displayed rooms according to their search and all of the information about these rooms.

2. DESIGN DECISIONS & RATIONALE

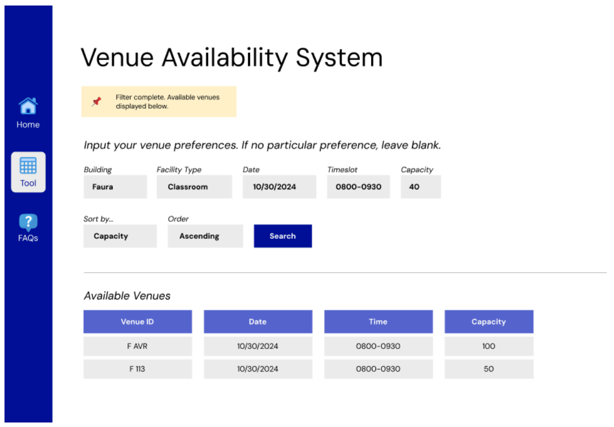
2.1 User Interface

To ensure a simple, intuitive, and familiar interface, the software solution adheres to established design conventions, specifically Nielsen’s 10 Usability Heuristics (Nielsen, 1994), a widely accepted framework for guiding interactive design[1].

Table 1: User Interface Principles

Principle	**Implementation in Software**
Visibility of system status	Displays status messages to keep users informed. Examples: *“Checking availability…”*, *“Retrieving Reservation Data…”*.
Match between system and real-world	Uses concepts familiar to users. Example: *“Classrooms”* instead of technical jargon.
User control and freedom	Allows users to easily modify their venue preferences.
Consistency and standards	Implements uniform terminology, layout, and interaction approaches across the interface.
Error prevention	Limits user input to dropdown menus to reduce errors.
Recognition rather than recall	Includes hover pop-ups for additional information. Example: Details for *venue_id*.
Flexibility and efficiency of use	Provides a default or quick setup for scheduling preferences.
Aesthetic and minimalist design	Maintains an uncluttered interface, prioritizing clear and comprehensible design elements.
Help users recognize, diagnose, and recover from errors	Offers informative notifications and actionable feedback. Examples: *“No venue matches available. Kindly reconsider your preferences.”* or *“Available venues displayed below.”*.
Help and documentation	Provides accessible FAQs and step-by-step tutorials to guide users.

2.2 Mockup for Venue Availability System



Venue Availability System

Filter complete. Available venues displayed below.

Input your venue preferences. If no particular preference, leave blank.

Building	*Facility Type*	*Date*	*Timeslot*	*Capacity*
Faura	Classroom	10/30/2024	0800-0930	40

Sort by...	*Order*
Capacity	Ascending

Available Venues

Venue ID	Date	Time	Capacity
F AVR	10/30/2024	0800-0930	100
F 113	10/30/2024	0800-0930	50

Figure 1. Mockup for Venue Availability System

3. METHODOLOGY

3.1 Sampling and Input Data

3.1.2 Reservations (MOCK_DATA.csv)

This mock dataset includes 1000 records of available rooms across multiple buildings and venues.

Table 3: Reservations Fields

Field Name	**Description**
building	Building where the venue is located (e.g., Faura)
venue_name	Name of the venue being reserved
reservation_date	Date of the reservation. (Data is limited from a span of 1 month only)
target_time	Reservation time slot (e.g., "09:30 - 11:00")
capacity	Capacity of each room

Data was generated using Mockaroo, a synthetic data generation platform. This dataset simulates real-world reservation scenarios to test the functionality of filtering venues based on availability, reservation dates, and time slots.

3.2 Search Algorithm

3.2.1 Search Algorithm: Binary Search

Among the 2 search algorithms(Linear search and Binary search), Binary Search has been identified as the most appropriate sorting algorithm due to its greater efficiency, with the Big-O notation of Binary search at the worst case being that of **O(log n)** while Linear search at the worst case has a time complexity of O(n) implying a linear relationship between the number of venues, and the time required to execute the algorithm.

Table 4: Time Complexity Comparison

Search Algorithm	Time Complexity		
	Best	Average	Worst
Linear	O(1)	O(n)	O(n)
Binary	O(1)	O(log n)	O(log n)

This may cause issues in larger data, such as that of all of ADMU's venues. It is also important to note that the binary search algorithm will only be executed after the data is sorted, and only then, will a Binary search be done.

Using the search algorithm, the system will go through the list of available venues in the CSV file and show which fits the users need the most.

3.3 Sorting Algorithm

3.3.1 Sorting Algorithm: MergeSort

Among the three basic—Bubble, Insertion, and Selection—and two advanced—MergeSort and QuickSort—sorting algorithms discussed, MergeSort has been identified as the most appropriate sorting algorithm evaluated on the following factors: (1) Data Volume and Nature of Data Collection, (2) Algorithm Complexity, and (3) Stability.

The venue reservation system will handle a medium-to-large volume of records. Over a 5-year period, the system will manage reservations for 127 venues across 10 semesters. Each day can have up to 8 full 1.5-hour reservation slots, leading to an estimated 1,259,840 records. This scale requires an efficient sorting algorithm for frequent data access, filtering, and sorting.

Given that the data collection process does not follow a strict chronological order, with venue reservations potentially made at random, stability is essential to preserve the order of records when sorting.

Table 5: Time Complexity Comparison

Search Algorithm	Time Complexity		
	Best	Average	Worst
Mergesort	O(n log n)	O(n log n)	O(n log n)
Quicksort	O(n log n)	O(n log n)	O(n^2)

Table 6: Space Complexity Comparison

Search Algorithm	Space Complexity		
	Best	Average	Worst
Mergesort	O(n)	O(n)	O(n)
Quicksort	O(n log n)	O(n log n)	O(n^2)

Stability in sorting ensures that when two records have the same key or sorting criterion, their relative order is preserved. In the Venue Reservation System, this is important for maintaining the order of venue reservations that occur at the same time.

1. MergeSort is stable, which means that it preserves the relative order of records with identical sorting keys.
2. QuickSort, in its basic implementation, does not guarantee this stability.

Given the data volume, the need for stable sorting, and the consistent performance across all cases, MergeSort is the most suitable algorithm for the Venue Reservation System. Its efficiency and reliability ensure optimal performance even as the system scales over time.

Using the sort algorithm, the system will sort the list of available venues based on the capacity that the user wants.

3.4 Data Structures

3.4.1 Dictionary

The Dictionaries are used to store data in key-value pairs, and provide fast access to the venues. It is used to quickly look up specific venue details, such as capacity and availability, using the different column names as the key [2].

Use in the System:

1. available_slots elements
    a. Used to store each row of the CSV file as a key-value pair.
    b. Allowed easy access to values by column names.
2. slots in filtering
    a. The individual dictionaries in available_slots are accessed and filtered based on keys (reservation_date, target_time, capacity).
3. Form: cleaned_data
    a. Stores the cleaned and validated data from the form
4. Context Dictionary for Rendering
    a. Passes filtered slots and form data to the template

3.4.2 List

Lists store ordered collection of items. Lists are easy to search through by filtering by date and time slot [2].

Use in the system:

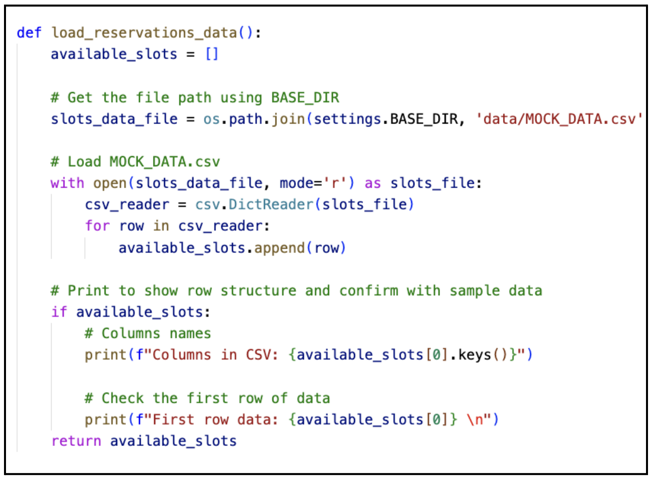
1. available_slots
    a. Used to store and manipulate collections of data, such as available slots loaded from the CSV file and intermediate filtered results

2. TIME_SLOTS
    a. Used to maintain the time slots.
3. left and right in merge_sort
    a. Temporary lists used to divide the data during recursive merge sort
4. merged in merge
    a. Temporary list used to combine sorted sublists into a single sorted list
5. filtered_slots
    a. A list that stores the result of filtering operations based on user inputs (e.g., capacity, date, target_time).
6. choices in Form Fields
    a. Used in fields like target_time and sort_by to limit the input of the user

The implementation of these two data structures is used in conjunction. Each element in the list is a dictionary, which represents a single row.

4. IMPLEMENTATION

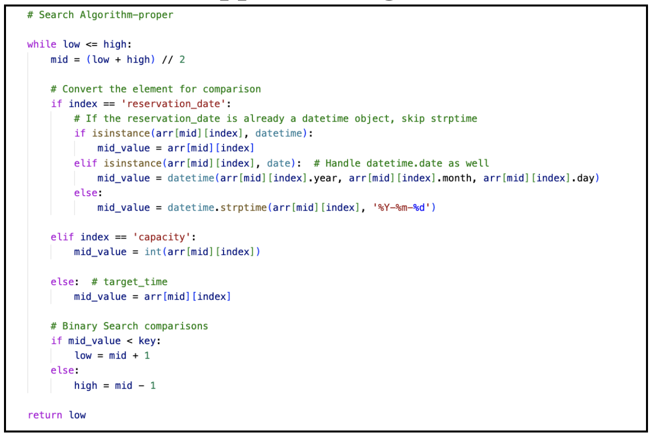
4.1 Code Snippets for Loading Data



```python
def load_reservations_data():
    available_slots = []

    # Get the file path using BASE_DIR
    slots_data_file = os.path.join(settings.BASE_DIR, 'data/MOCK_DATA.csv'

    # Load MOCK_DATA.csv
    with open(slots_data_file, mode='r') as slots_file:
        csv_reader = csv.DictReader(slots_file)
        for row in csv_reader:
            available_slots.append(row)

    # Print to show row structure and confirm with sample data
    if available_slots:
        # Columns names
        print(f"Columns in CSV: {available_slots[0].keys()}")

        # Check the first row of data
        print(f"First row data: {available_slots[0]} \n")
    return available_slots
```

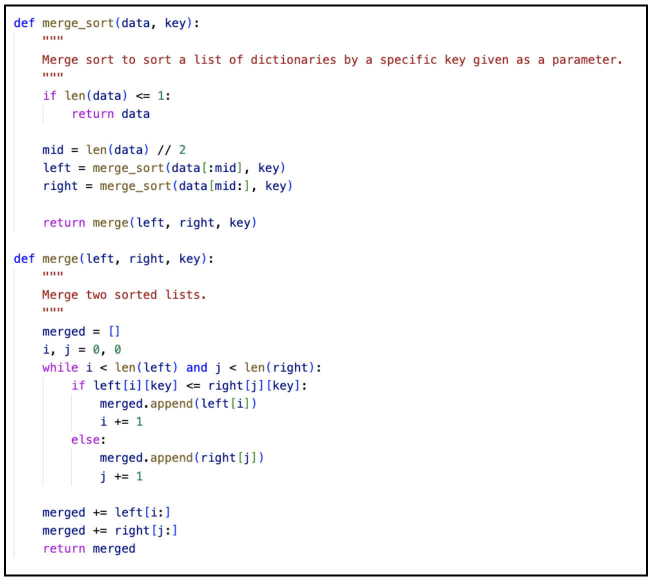
Image 2. Code for Loading Data from the CSV file

4.2 Code Snippets for Algorithms



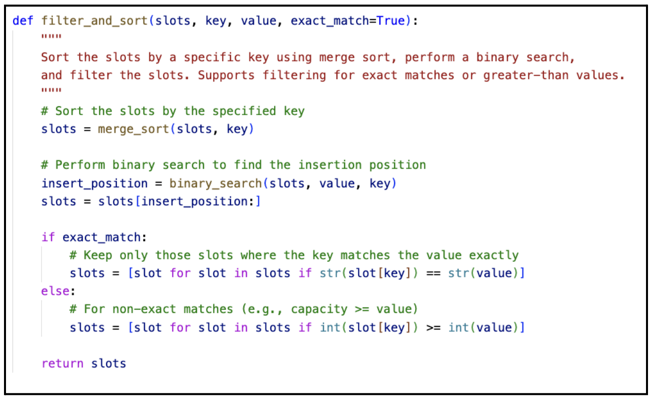
```python
# Search Algorithm-proper

while low <= high:
    mid = (low + high) // 2

    # Convert the element for comparison
    if index == 'reservation_date':
        # If the reservation_date is already a datetime object, skip strptime
        if isinstance(arr[mid][index], datetime):
            mid_value = arr[mid][index]
        elif isinstance(arr[mid][index], date):  # Handle datetime.date as well
            mid_value = datetime(arr[mid][index].year, arr[mid][index].month, arr[mid][index].day)
        else:
            mid_value = datetime.strptime(arr[mid][index], '%Y-%m-%d')

    elif index == 'capacity':
        mid_value = int(arr[mid][index])

    else:  # target_time
        mid_value = arr[mid][index]

    # Binary Search comparisons
    if mid_value < key:
        low = mid + 1
    else:
        high = mid - 1

return low
```

Image 3. Code for Search Algorithm



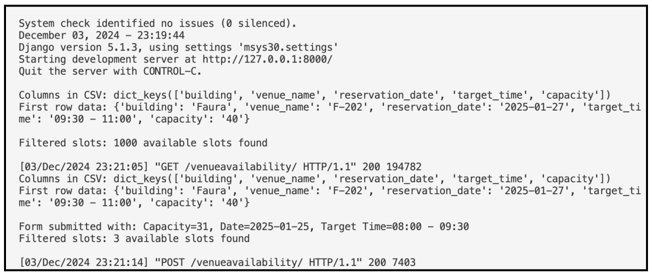
```python
def merge_sort(data, key):
    """
    Merge sort to sort a list of dictionaries by a specific key given as a parameter.
    """
    if len(data) <= 1:
        return data

    mid = len(data) // 2
    left = merge_sort(data[:mid], key)
    right = merge_sort(data[mid:], key)

    return merge(left, right, key)

def merge(left, right, key):
    """
    Merge two sorted lists.
    """
    merged = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i][key] <= right[j][key]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    merged += left[i:]
    merged += right[j:]
    return merged
```

Image 4. Code for Merge sort



```python
def filter_and_sort(slots, key, value, exact_match=True):
    """
    Sort the slots by a specific key using merge sort, perform a binary search,
    and filter the slots. Supports filtering for exact matches or greater-than values.
    """
    # Sort the slots by the specified key
    slots = merge_sort(slots, key)

    # Perform binary search to find the insertion position
    insert_position = binary_search(slots, value, key)
    slots = slots[insert_position:]

    if exact_match:
        # Keep only those slots where the key matches the value exactly
        slots = [slot for slot in slots if str(slot[key]) == str(value)]
    else:
        # For non-exact matches (e.g., capacity >= value)
        slots = [slot for slot in slots if int(slot[key]) >= int(value)]

    return slots
```

Image 5. Code for Merge sort



```
System check identified no issues (0 silenced).
December 03, 2024 - 23:19:44
Django version 5.1.3, using settings 'msys30.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

Columns in CSV: dict_keys(['building', 'venue_name', 'reservation_date', 'target_time', 'capacity'])
First row data: {'building': 'Faura', 'venue_name': 'F-202', 'reservation_date': '2025-01-27', 'target_ti
me': '09:30 - 11:00', 'capacity': '40'}

Filtered slots: 1000 available slots found

[03/Dec/2024 23:21:05] "GET /venueavailability/ HTTP/1.1" 200 194782
Columns in CSV: dict_keys(['building', 'venue_name', 'reservation_date', 'target_time', 'capacity'])
First row data: {'building': 'Faura', 'venue_name': 'F-202', 'reservation_date': '2025-01-27', 'target_ti
me': '09:30 - 11:00', 'capacity': '40'}

Form submitted with: Capacity=31, Date=2025-01-25, Target Time=08:00 - 09:30
Filtered slots: 3 available slots found

[03/Dec/2024 23:21:14] "POST /venueavailability/ HTTP/1.1" 200 7403
```

Image 6. Sample Print Statements from the Terminal

4.3 Testing Plan

4.3.1 Functional Testing

Goal: Ensure that the system correctly filters and sorts venue reservations based on user inputs (e.g., date, time slot, capacity).

Tests:

- Date Filter Test: Test if the system only returns venues with reservation dates greater than or equal to the selected date.
- Time Slot Filter Test: Test if the system filters venues based on the selected time slot.
- Capacity Filter Test: Test if the system only returns venues with a capacity greater than or equal to the specified value.
- Sort by Capacity/Date/Time Slot: Test if the system correctly sorts venues by capacity, reservation date, or time slot in both ascending and descending order.

4.3.2 Performance Testing

Goal: Measure the response time and efficiency of the filtering and sorting operations, especially as the dataset grows.

Tests:

- Load Testing: Simulate large datasets (e.g., 1,000 venue slots) and measure the system's response time for sorting and filtering.
- Time Complexity Validation: Measure the time taken for sorting and binary search operations in datasets of different sizes. The expected time complexity for the sorting algorithm (merge sort) is O (n log n) and for the binary search, it's O(log n).

4.3.3 Edge Case Testing

Goal: Verify the system handles unusual or unexpected inputs.

Tests:

- Empty Data:
    - Test if the system correctly handles cases where no venue data is available.
- Invalid Date/Time Inputs:
    - Ensure that the system properly handles invalid or missing inputs (e.g., non-date values in the date field).
- Boundary Testing:
    - Test edge cases for sorting and filtering, such as the smallest and largest possible values for capacity, date, or time slot.

4.4 Evaluation Metrics

4.4.1 Accuracy

Metric: Percentage of correct results returned for each filter (date, time slot, capacity).

Evaluation: Compare the system's output with the expected output for a variety of test cases. The system should return the correct list of venues that match the search criteria.

4.4.2 Efficiency

Metric: Time taken for filtering and sorting operations.

Evaluation: Measure the time taken for the sorting and binary search operations with different dataset sizes. Ideally, sorting should take O(n log n), and searching should take O(log n).

Performance Goals:

- For small datasets (up to 1,000 records), sorting and filtering should take less than 1 second.

4.4.3 System Robustness:

Metric: System's ability to handle errors and edge cases.

Evaluation: The system should not crash or behave unpredictably when faced with invalid or unexpected input. Proper error handling should be in place.

By testing these factors and evaluating the results based on the described metrics, you can ensure the system meets both functional and performance requirements. This process also helps identify areas for improvement and optimization.

5. DISCUSSION & CONCLUSION

The choice of data structures in this code is primarily driven by performance, simplicity, and the need for efficient algorithms to handle data. Here's a deeper analysis of the data structures used, their time complexities, and how they integrate:

5.1 Lists

Use in Code: Lists are used to store the reservation data after it's loaded from the CSV file, as well as for sorting and filtering operations.

- In the function load_reservations_data, a list (available_slots) stores the rows from the CSV file, where each row is a dictionary representing a reservation.
- Lists are also used in the merge_sort and filtered_slots steps to store data during sorting and filtering.

5.1.1 Time Complexity

Access by index:

- O(1). Lists allow direct access to elements by index, which is efficient when processing specific data points (e.g., available_slots[i]).

Append:

- O(1) on average. Adding an item to the end of a list is efficient.

Sorting:

- O(n log n) with algorithms like merge_sort that work on lists. This is the time complexity for each merge operation, making it suitable for the dataset size we expect.

Filtering:

- O(n). A list comprehension is used to filter based on conditions (like matching target_time), iterating over each element once.

Lists are ordered and maintain the sequence of the CSV rows, which is important for operations like sorting and filtering.

Lists provide efficient traversal and modification, and since we are handling a collection of reservation data, using lists keeps the code simple and readable.

5.2 Dictionaries

Use in Code: Dictionaries are used to represent individual rows of the reservation data. Each row from the CSV is stored as a dictionary, where keys are the column names (e.g., reservation_date, target_time, capacity), and values are the corresponding values for that row.

- For example, in the available_slots list, each element is a dictionary representing a single reservation.

5.2.1 Time Complexity

Access by key:

- O(1). Dictionary lookups by key are efficient, making it easy to access values like slot['reservation_date'] or slot['target_time'].

Iteration:

- O(n). Iterating through a list of dictionaries (for filtering or sorting) requires checking each dictionary's keys.

Insertion/Deletion:

- O(1). While not used here, adding or removing dictionary keys can be done in constant time.

The dictionary format offers semantic clarity, as it pairs column names with corresponding values, making it easier to understand and process the data.

Since we often need to access values by column name (e.g., target_time, reservation_date), dictionaries provide a natural structure that allows constant-time lookups, reducing the need for additional logic to identify values.

5.3 Merge Sort Algorithm

Use in Code: Merge sort is used to sort the reservation data based on different fields (e.g., reservation_date, target_time, capacity).

5.3.1 Time Complexity

Merge Sort:

- O(n log n). Merge sort splits the list in half and recursively sorts each half. It then merges the sorted halves in linear time (O(n)), resulting in an overall time complexity of O(n log n).
- Sorting is the most computationally expensive part of the process. Merge sort's efficiency in handling large datasets makes it well-suited for this task.

Merge sort guarantees stable sorting and handles large datasets efficiently with O(n log n) complexity, making it better than simpler algorithms like bubble sort (O(n^2)).

Stability is important because it maintains the relative order of records with the same sorting key, which is crucial when sorting by multiple fields (e.g., reservation_date then target_time).

5.4 Binary Search Algorithm

Use in Code: Binary search is used to efficiently find the position of the first element that is greater than or equal to the key (e.g., a target reservation_date or capacity) in a sorted list.

5.4.1 Time Complexity

Binary Search:

- O(log n). Binary search works by repeatedly dividing the search interval in half, making it logarithmic in complexity. This significantly reduces the number of comparisons required when searching through a sorted list.

Binary search is highly efficient for searching through sorted data. Since the data is sorted via merge_sort, binary search ensures that we can quickly find the appropriate insertion point or match, significantly improving performance compared to linear search (O(n)).

By reducing the time complexity of searching, binary search optimizes the filtering step where we need to find and isolate slots based on a specific criterion.

5.5 Integration of Data Structures

The integration of lists and dictionaries, along with the sorting and searching algorithms, is highly efficient. Here's how they work together:

5.5.1 Data Loading

Data is loaded from a CSV file into a list of dictionaries, with each dictionary representing a reservation. The list holds all the rows, making it easy to iterate over and manipulate the data.

5.5.2 Sorting:

The list of dictionaries is sorted using merge sort based on specific keys (e.g., reservation_date, target_time, capacity). Merge sort is applied to the list, and the dictionaries are sorted in an ordered fashion. This ensures that subsequent operations (like binary search) will operate on a sorted list.

5.5.3 Filtering:

After sorting, binary search is applied to the sorted list to quickly locate the first slot matching the query criteria (e.g., a reservation on or after a certain date). Filtering is then applied by iterating over the sorted list and keeping only the entries that match the desired values.

5.5.4 Performance Optimization:

The combination of sorting with merge sort (O(n log n)) and searching with binary search (O(log n)) ensures that the solution is scalable and efficient. Instead of performing a linear search, the binary search narrows down the results in logarithmic time, which is critical for handling larger datasets.

This venue reservation system improves upon the current process at CMFO by streamlining the search for available rooms, making it more efficient and user-friendly. Users can easily filter rooms by date, time slot, and capacity, eliminating the need for manual searches through individual room availabilities and capabilities. The system developed by this group simplifies what was once a tedious and time-consuming task.

The integration of data structures like lists for sequential storage, dictionaries for fast lookups, merge sort for efficient sorting, and binary search for quick filtering ensures that the application performs optimally. By leveraging these data structures, the system handles filtering and sorting with high efficiency, even as the dataset grows. The use of these well-established algorithms reduces redundant operations, optimizes performance, and provides scalability, making it an ideal solution for managing venue availability in this context.

6. ENHANCEMENTS

6.1 Integration with Real-Time Data

Currently, the system utilizes mock datasets to simulate reservation scenarios. In future iterations, the system should be integrated with the CFMO's actual database to reflect real-time venue availability and bookings. This will enhance the system's accuracy and usability, allowing users to access up-to-date information and make informed decisions when reserving venues.

6.2 Incorporation of the Booking Process

The current scope focuses on venue availability, leaving the booking process outside the system's scope. A significant enhancement would be to include end-to-end booking capabilities within the system. This would allow users to check the availability of the venues.

6.3 Expansion of Scope to All Ateneo Buildings

At present, the system is limited to venues in Faura Hall. A natural progression would be to expand its scope to cover all buildings and facilities managed by the CFMO across Ateneo de Manila University. This would ensure a unified platform for venue reservations, benefiting a broader range of users and events. The system could be designed to handle unique requirements for specific buildings or venues, such as specialized equipment or restricted access.

By implementing these enhancements, the Venue Availability System could evolve into a comprehensive tool that meets the growing needs of the Ateneo community while maintaining its commitment to efficiency and user satisfaction.

7. REFERENCES

[1] Nielsen, J. 1995. Ten Usability Heuristics for User Interface Design. Nielsen Norman Group. Retrieved from https://www.nngroup.com/articles/ten-usability-heuristics/ (Accessed: December 2, 2024)

[2] 2024.Data Structures and Algorithms. Retrieved from https://ateneo.instructure.com/courses/47823/Modules (Accessed: December 2, 2024).

8. APPENDIX

The final version of the Django application developed for this project will be submitted along with this report. The application, which includes all relevant documentation, will be provided as a complete package.

Ateneo de Manila University Department of Information Systems and Computer Science

CERTIFICATE OF AUTHORSHIP

Instructions

- Download and fill this PDF form completely.
- Each course requirement submission, unless otherwise specified by the Course Instructor, whether in electronic or paper form, must be accompanied by a corresponding properly accomplished Certificate of Authorship.

Description of Submission

Title of Submission:	MSYS30C-FinalProject-Group2
Type of Submission:	☐ Program ☐ Project ☐ Report ☐ Paper
	☐ Other (specify) Program and Paper
Date of Submission:	December 3, 2024

Certification

I hereby certify that the submission described in this document abides by the principles stipulated in the DISCS Academic Integrity Policy document. I further certify that I am the author of this submission and that any assistance I received in its preparation is fully acknowledged and disclosed in the documentation. I have also cited all sources from which I obtained data, ideas, or words that are directly copied or paraphrased in this document. Sources are properly credited according to accepted standards for professional publication.

Modules in Class
https://docs.djangoproject.com/en/5.1/ref/contrib/messages/
https://plainenglish.io/blog/importing-csv-data-into-django-models

Declaration of Use of Generative AI

Tool:

Purpose:

We have reviewed and revised the content as we see fit. We take full responsibility for the content and ownership of the submitted / published work.

Group Information

Full Name	Signature
Jana Almira J. Boco	[illegible]
Nathan Luna	[illegible]
Juliana Ysabelle S. Valdez	[illegible]

Course Code & Section MSYS 30 - C
Course Title Data Structures and Algorithms
Course Instructor Josh Daniel L. Ong