Stage 1 Report – Simple Lists and CSV Handling

1. Introduction

This report documents the implementation of Stage 1 of the Melbourne real estate data project. The primary focus of this stage was on using basic Python list structures to manipulate and analyze CSV data. The objective was to design a simple console-driven application capable of reading, displaying, modifying, and sorting real estate records using Python's built-in functionality.

2. Objectives

Stage 1 aimed to:

- Implement functions using simple Python lists for:
  - Reading a CSV file
  - Displaying a subset of the data
  - Adding new entries to the dataset
  - Sorting the data by a specified column
- Design a user interface with basic console interactions
- Analyze the time complexity of the major operations
- Develop unit tests using pytest for core functionalities
  Document all aspects of the design and implementation

3. Implementation Overview

a. CSV Reading

The program uses Python's csv.DictReader to read melb_data.csv into a list of dictionaries. Each dictionary corresponds to one row of real estate data, allowing key-based access to column values.

b. Display Function

A function print_data(data, num_rows) displays the first few rows of the dataset. This enables quick inspection of the data structure and sample contents.

c. Add Row

The add_row(data, row) function allows the user to append a new record to the existing dataset. This was implemented via direct interaction from the console by entering a Python dictionary (e.g., {'Price': '1000000', 'Suburb': 'Melbourne'}).

d. Sort Functionality

The sort_data(data, column) function sorts the dataset using Python's built-in sorted() function. It includes basic error handling for:

- Type conversion errors (e.g., non-numeric strings in numeric columns)
- Invalid column names (e.g., typos or case issues)

e. User Interface

A command-line loop was implemented to let the user choose between viewing data, adding rows, sorting by column, or exiting. Input validation and meaningful prompts were provided to guide the interaction.

4. Testing and Results

a. Automated Testing

Pytest was used to test the following functions:

- read_csv() – verified that data is loaded and key fields are present.
- add_row() – confirmed that a new dictionary is appended to the dataset.

A mock CSV file was used to avoid modifying the original dataset during tests.

b. Big O Time Complexity Analysis

Time complexity was estimated for both reduced (2000 rows) and full dataset sizes:

- read_csv(): O(n), where n is the number of rows
- add_row(): O(1) – single append operation

No significant memory or performance issues were noted during testing.

5. Summary and Reflection

Stage 1 successfully demonstrated foundational data processing using simple Python list structures. The implementation met all core requirements, including reading, modifying, sorting, and interacting with CSV data. The testing process confirmed functional correctness, and time complexity analysis provided insight into performance under scale. While the techniques used were elementary, they laid the groundwork for more advanced stages involving custom data structures and algorithms.

This stage highlighted the importance of good user interface design, robust error handling, and automated testing. The experience gained in handling real-world datasets and designing user-driven workflows directly contributed to deeper understanding in subsequent stages.

Stage 2 Report – Sorting Real Estate Data with Custom Data Structures

1. Introduction

This report outlines the implementation of Stage 2 of the Melbourne real estate data project. The focus of this stage was on sorting large datasets using a custom Abstract Data Type (ADT), specifically a list-based data structure. The main objective was to sort real estate records by a specified column using merge sort and to evaluate the efficiency of these operations. This stage extends the work from Stage 1 by introducing a more advanced sorting algorithm and organizing the data in a custom structure.

2. Objectives

Stage 2 aimed to:

- Implement a custom list data structure (PythonList) that supports operations such as insertion, retrieval, and modification of data.
- Design and implement the merge sort algorithm to efficiently sort the real estate dataset by a user-specified column.
- Enable reading from and writing to CSV files, while preserving data integrity.
- Provide a console-based user interface for interacting with the program, where users can specify the column by which the data should be sorted.
- Analyze the time complexity of the sorting algorithm, especially for larger datasets.
- Develop unit tests using pytest to ensure the functionality of key operations, such as sorting and data handling.

3. Implementation Overview

a. Custom List Data Structure (PythonList)

A custom list class was implemented as an Abstract Data Type (ADT). This class provides several operations, including:

- insert(item): Adds an item to the list.
- get(index): Retrieves the item at the specified index.
- set(index, item): Modifies the item at the specified index.
- length(): Returns the number of items in the list.
- to_list(): Converts the custom list into a standard Python list.

b. Merge Sort Algorithm

The merge sort algorithm was used to efficiently sort the dataset. This divide-and-conquer algorithm splits the dataset into smaller parts, recursively sorts them, and then merges the sorted parts back together. It operates in O(n log n) time complexity, making it suitable for large datasets. The sorting operation was based on a user-specified column, where the column was identified by its header in the CSV file.

c. CSV Reading and Writing

The program used CSV file handling to load and save data:

- read_csv(filename): Reads data from a CSV file and returns the header and rows as lists.
- write_csv(filename, header, data): Writes the sorted data back into a new CSV file.

d. User Interface

A basic console interface was provided for the user to:

- Choose the CSV file to read.
- Specify the column by which to sort the data.
- Trigger the sorting operation and output the results to a new file.

e. Time Complexity Analysis

The merge sort algorithm was the most computationally intensive part of the program. The sorting operation for a list of n elements (rows) has a time complexity of O(n log n). For the reduced dataset (2000 rows), this was sufficient to provide efficient sorting. The insertion and retrieval operations on the PythonList data structure were O(1) on average.

4. Testing and Results

a. Automated Testing

Unit tests were created using pytest to validate key functionalities:

- PythonList methods were tested for correctness, ensuring that insertion, retrieval, and modification worked as expected.

- merge_sort was tested to ensure that it correctly sorted datasets by the specified column.
- CSV reading and writing were tested using StringIO, simulating CSV file input and output without modifying actual files.

Tests confirmed that the program could:

- Read and parse the input data correctly.
- Sort the data based on the user's specified column.
- Write the sorted data back into a CSV file.

b. Big O Time Complexity Analysis

The time complexity for the main operations were as follows:

- read_csv(): O(n), where n is the number of rows in the CSV file.
- merge_sort(): O(n log n), where n is the number of rows.
- write_csv(): O(n), where n is the number of rows.

5. Summary and Reflection

Stage 2 of the project successfully achieved its goals of sorting real estate data using custom data structures and algorithms. The implementation demonstrated how a custom list data structure can be used in conjunction with an efficient sorting algorithm like merge sort to process large datasets. This stage also highlighted the importance of creating modular, testable code and conducting performance analysis to ensure that algorithms scale with dataset size.

The integration of merge sort with the PythonList class allowed for efficient handling of data, and the CSV file handling provided a seamless interface for reading and writing datasets. The unit tests confirmed the correctness of the implemented functionality.

Stage 3 Report – Search Algorithms and Data Structures

1. Introduction

This report covers Stage 3 of the Melbourne real estate data project, where the goal was to improve the application by incorporating more advanced search algorithms and data structures. In this stage, I focused on building tools to perform range searches and optimize how data is queried, using custom data structures like binary search trees (BSTs), sorted lists, and stacks. These improvements allow for faster and more efficient analysis of real estate data, especially when looking for values within specific ranges.

2. Objectives

The main objectives of Stage 3 were to:

- Implement an Abstract Data Type (ADT) for handling value ranges efficiently.
- Develop custom data structures, such as unordered lists, sorted lists, and binary search trees, to improve search speed.
- Create a stack for managing temporary data during search operations.
- Implement range search functions using both linear and binary search methods.
- Allow users to interact with the system via a simple console interface for searching and filtering data.
- Analyze the time complexity of key operations and ensure scalability for large datasets.
- Write unit tests using pytest to ensure everything works as expected.
- Document the design and implementation clearly.

3. Implementation Overview

a. Value Range ADT

I started by designing an ADT to handle searches within specified value ranges. This allows for easy querying of real estate data within certain price, area, or other numerical ranges. I created two classes for range searching:

- UnorderedInRange: This class performs a linear search through the data, checking if each value falls within the given range.
- SortedInRange: For sorted datasets, this class uses Python's bisect module to perform a binary search, which is much faster than linear search for large datasets.

b. Binary Search Tree (BST)

A Binary Search Tree (BST) was implemented to store data efficiently. The insert method adds values to the tree, while the search_range method finds values within a specified range. The tree is traversed in-order, and the results are returned in sorted order, making it easy to get data in the desired range.

c. Stack

I also implemented a basic stack to manage temporary data during the search process. The stack follows a Last In, First Out (LIFO) order, which is useful for operations like tree traversal when looking for values within a range.

d. CSV Reading and Loading

The real estate data is read from a CSV file and inserted into the appropriate data structures (BST or sorted list) for searching. This makes it easy to query the data interactively and test different search methods.

e. Driver Interface

To interact with the data, I set up a simple command-line interface. The user can enter a search range, choose a search method (linear search, binary search, or BST), and see the results. This makes the system flexible and user-friendly.

4. Testing and Results

a. Automated Testing

I used pytest to test the main functionalities of the program, including:

- Range Search: Verifying that the search_range method works correctly, both for sorted and unsorted datasets.
- BST Insertions: Ensuring that values are correctly inserted into the binary search tree.
- Stack Operations: Testing the push and pop methods to confirm the stack behaves as expected.

Mock data was used during testing, so the original dataset was never modified, and I checked the results against expected outputs for different search ranges.

b. Big O Time Complexity Analysis

I took a closer look at the time complexity of the major operations:

- UnorderedInRange.search_range(): O(n), because it uses a linear search.
- SortedInRange.search_range(): O(log n) for the binary search (finding the insertion points), plus O(k) to extract the range, where k is the number of values within the range.
- BinarySearchTree.search_range(): O(log n) for balanced trees, but it can degrade to O(n) in the worst case if the tree is unbalanced.

- Insert Operations in BST: O(log n) for balanced trees and O(n) for unbalanced trees.

- Stack Operations: O(1) for both push and pop, since these are constant-time operations.

5. Summary and Reflection

Stage 3 was a significant step forward in making real estate data analysis more efficient and flexible. By adding support for range searches with custom data structures like sorted lists and binary search trees, I was able to significantly speed up the querying process. The stack was also a nice addition, helping with data management during searches. Testing went smoothly, and I was able to confirm that everything works as expected. The time complexity analysis showed that the system can handle large datasets effectively, even as the data grows.

Stage 4 Report – Hashing Algorithms and Duplicate Removal

1. Introduction

This report documents the implementation of Stage 4 of the Melbourne real estate data project. The goal of this stage was to address data duplication issues in the dataset, specifically records with identical property descriptions. To efficiently detect and eliminate these duplicates, a custom hash map was implemented to support fast lookups. The final solution reads the full CSV file, removes duplicate descriptions using hashing, and writes the cleaned dataset to a new output file.

2. Objectives

Stage 4 aimed to:

- Design and implement a basic hash map data structure from scratch
- Support constant-time (average case) set, get, and contains operations
- Use the hash map to remove duplicate property descriptions from a CSV file
- Ensure correctness through automated unit testing with pytest
- Analyze the time complexity of core operations

3. Implementation Overview

a. Hash Map Interface and Accessors

A custom HashMap class was created with the following methods:

- set(key, value): Inserts a key-value pair into the hash map, using chaining for collision handling
- get(key): Retrieves the value associated with a key, or returns None if not found
- contains(key): Checks if a key exists in the hash map

The hash function was based on Python's built-in hash() function, followed by modulo operation to fit the internal array size.

b. Duplicate Removal Logic

The remove_duplicates function reads a CSV file and writes a new file containing only unique descriptions. The steps are:

1. Open the original dataset using csv.DictReader
2. Initialize an empty HashMap
3. Iterate through each row and check if the description already exists in the hash map
4. If not, add the row to the output and record the description in the hash map
5. Write the deduplicated results to a new CSV file using csv.DictWriter

c. Output File

The final dataset is saved to a new CSV file (e.g., cleaned_data.csv), containing only one instance of each unique property description.

4. Testing and Results

a. Automated Testing

pytest was used to test:

- Hash map operations (set, get, contains)
- Duplicate removal behavior using a sample dataset with repeated descriptions

A fixture was used to simulate a temporary CSV file, ensuring that test data was isolated and did not modify the actual dataset. The tests validated that:

- The hash map correctly handled insertion and retrieval
- Duplicate rows were accurately identified and removed
- The output file contained only the unique property descriptions

b. Big O Time Complexity Analysis

- Inserting or checking a description in the hash map takes O(1) on average, since we're using hashing for fast access.
- Reading through the CSV file is a linear operation, so it's O(n) where *n* is the number of rows in the dataset.
- Checking for duplicates and adding to the hash map also happens once per row, so this step is also O(n) overall.
- Writing the cleaned data to a new file is a simple pass through the filtered list, which again makes it O(n).

Performance remained acceptable during testing, and the hash map significantly improved the efficiency of detecting duplicates compared to linear search.

5. Summary and Reflection

Stage 4 introduced a key optimization to the data cleaning pipeline by addressing duplicate entries using a custom hash map. This allowed for fast lookup times and scalable performance on large datasets. The hash map structure was simple but effective, supporting core operations needed to filter data in real time.

Unit testing played a critical role in validating both the data structure and the end-to-end cleaning function. Overall, this stage emphasized the practical value of hashing in data management, reinforcing algorithmic thinking while solving a real-world problem.