

|  |  |  |  |
| --- | --- | --- | --- |
| Course Code | : | CST207 | |
| Course Name | : | Design and Analysis of Algorithms | |
| Lecturer | : | Dr. Mohammed N. M. Ali | |
| Academic Session | : | 2024/09 | |
| Assessment Title | : | Project (Group) | |
| Submission Due Date | : | 30th December | |
| Prepared by |  |  | |
| : | Student ID | Student Name |
|  | CYS2309202 | Sun Zhengwu |
|  | CYS2309205 | Zheng Yucheng |
|  | CYS2309680 | Lim Jun Ming |
|  | CYS2309275 | Mosaddiqur Rahman |
|  | CYS2309193 | Chen Junfeng |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| Date Received | : |  | |
|  |  |  | |
| Feedback from Lecturer:  Mark: | | | |

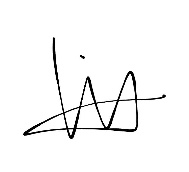
**Own Work Declaration**

I/We hereby understand my/our work would be checked for plagiarism or other misconduct, and the softcopy would be saved for future comparison(s).

I/We hereby confirm that all the references or sources of citations have been correctly listed or presented and I/we clearly understand the serious consequence caused by any intentional or unintentional misconduct.

This work is not made on any work of other students (past or present), and it has not been submitted to any other courses or institutions before.

Signature:

Date:2024.12.27

## Comparison and challenges of implemented search algorithm

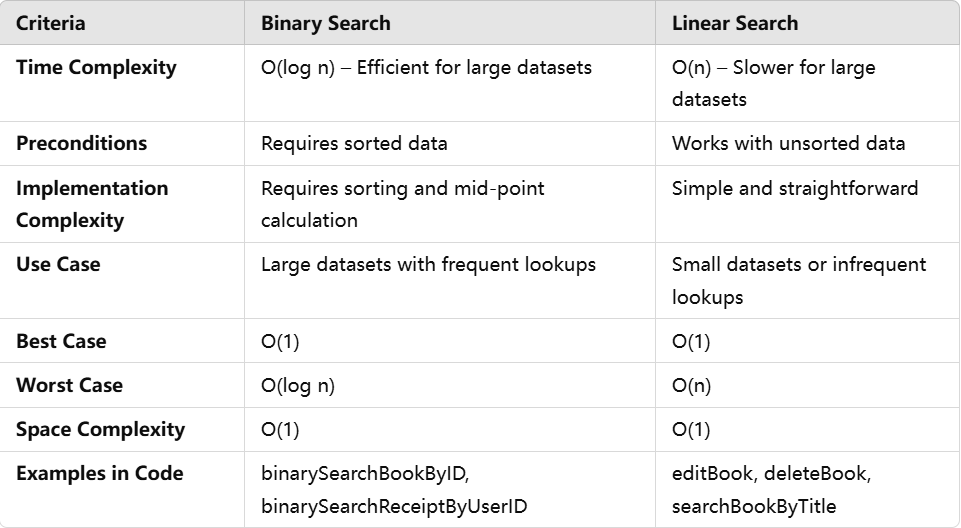
The main search algorithms used in the project are binary search and linear search. Binary search is a search method that continuously divides the search range into two to find the target element. At the same time, the limitation is that the data must be ordered. In the project, we used binary search functions named binarySearchBookByID, binarySearchBookByID and binarySearchReceiptByUserID. The first one is to find books by book ID, the second one is to find books by book title substring, and the last one is to find transaction records by user ID.

Another linear search algorithm is a search method that searches for elements one by one until the target is found or the entire data set is traversed. It is used in the editBook function, deleteBook and some functions in the project. Find\_if function is used to find and edit books by book ID and to find and delete books by book ID. It is also used in the “viewTransactionHistoryByUserID\_LinearSearch” function to find borrowing records by user ID.

Finally, “searchBookByTitle” searches for books based on the substring of the book title. For the comparison of the two-time complexities, the best case of both is O (1). Binary search corresponds to the case when the target element is in the middle, and linear search corresponds to the case when the target element is in the first position. In the average and worst cases, the two are completely different. Binary search can halve the search range in each iteration, so its complexity is O (log \_n), while linear search needs to traverse the entire data set, resulting in its time complexity of O(n). In terms of space complexity, both are O (1). Binary search only needs a few variables to save indexes and intermediate results, and linear search only requires constant space.

Next, considering the characteristics of the two, binary search requires ordered data, while linear search does not, which reflects the limitations of binary search. However, for large data sets, binary search is more efficient, which means that for string search in the corresponding project, binary search is obviously more suitable. In terms of implementation difficulty, linear search logic is intuitive and easier to implement and understand.

The following is a comparison table:



In the implementation of binary search, due to its limitation, data must be ordered, so we have to consider how to keep the data in order after it is updated. In this project, this is reflected in the need to re-sort after each insertion or deletion of a book. After team discussion and reference, we decided to use quick sort to sort the book IDs and bubble sort to sort the book titles. This method ensures that the binarySearchBookByID binary search function can be executed smoothly and efficiently to find books.

When linear search is used in multiple places, the code may be redundant and difficult to maintain. We encapsulate linear search into independent functions, such as viewTransactionHistoryByUserID\_LinearSearch, to improve code reusability and readability.

## Easiest search algorithm

Linear search is the easiest search algorithm as it is a much straightforward option. Linear search operates by traversing each element of a dataset sequentially until the targeted element is found or the list ends.

In our example, we used linear search on the function for viewing transaction history. Our function iterates through the receipts vectors and check each element against the targeted element When a match is found, it prints out the transaction receipt. Else, it prints an error message.

### The reason why linear search is a much easier

Linear search has minimal prerequisites. Linear search requires no preprocessing data, it is applicable for both sorted and unsorted datasets. Compared to binary search, it is mandatory for the dataset to be sorted. Linear search's biggest advantage is its robustness against implementation errors. Linear search does not involve indices, pivot and recursive calls that could result in possible results such as out of bounds and stack overflow. Due to the linear search algorithm conducted in a linear way, testing and debugging is much easier and direct.

## Comparison of sorting algorithms and any difficulties encountered.

To fulfil the need for orderly management of book data and flow records, the assignment required the use of four classic sorting algorithms: Merge Sort, Bubble Sort, Selection Sort, and Quick Sort. These algorithms exhibit certain complexities during implementation and debugging, with potential for coding errors, making it necessary to conduct practical operations to examine their performance. This paper makes a detailed comparison of the usage of these four algorithms and discusses some difficulties during their implementation.

Merge Sort was used for sorting the bibliography list in ascending order according to book IDs. It has a time complexity of O (nlog n) in both average and worst cases, making it suitable for large datasets. Merge Sort is a divide-and-conquer algorithm that recursively divides the data into smaller pieces and then merges them into a sorted sequence. In real implementation, the most important problems were the exact calculation of indices and the use of temporary arrays; any mistake could lead to out-of-bounds or misaligned data. By calculating the index carefully, testing thoroughly, and validating the boundary conditions of recursion, the correctness of Merge Sort was finally ensured.

The Bubble Sort was used for sorting book titles. Because it is simple and intuitive, Bubble Sort repeatedly compares adjacent elements and swaps them if necessary, causing larger elements to "bubble" toward the end of the sequence. Its time complexity is O(n²), but due to the infrequency of sorting book titles and relatively small dataset size, performance is adequate for practical purposes. The major concern was the fact that if the volume of data suddenly increased, the sorting time could also increase substantially. To further improve the performance, an early termination mechanism was implemented that would terminate the algorithm after a traversal if no swaps were performed.

Selection Sort was used to sort the transaction records. Selection Sort is an implementation that repeatedly finds the smallest element in the unsorted portion of the list and swaps it with the current position. It is straightforward to implement and particularly efficient for sorting small lists quickly. The only challenge was accurately identifying and updating the index of the minimum value: any mistake in initialization or updates would lead to wrong sorting. These were resolved by rigorously verifying the initial conditions of each iteration and the correctness of element swaps.

Quick Sort was used to generate user-requested lists sorted by book IDs. It is suitable for dynamic and real-time sorting tasks due to its high average efficiency and in-place sorting capability. The most significant problems were the choice of a suitable pivot and the manipulation of the partitioning phase. A poorly chosen set of pivots could yield O(n²) complexity. No such performance problems are experienced during debugging, yet there are such possible scenarios. To prevent this scenario, a randomized strategy is implemented for the selection of pivots. This randomization prevents bias and yields good reliability and efficiency from the algorithm.

Every one of these above sorting algorithms has their positives and negatives. The appropriate applications include using Merge Sort for large-scale data, Bubble Sort for small-scale and infrequent sorting tasks, Selection Sort for the quick development of a program when the data sets are small, and Quick Sort for its efficiency and flexibility in medium-scale and dynamic sorting tasks. With different optimization strategies adopted for different sorting algorithms according to their characteristics, the reliability and efficiency of the sorting functions could be ensured.

## 4. The Easier Sorting Algorithm to Implement

While developing the university library's LMS (Library Management System), a variety of sorting algorithms were employed, such as Merge Sort, Bubble Sort, Selection Sort, and Quick Sort. Although every algorithm possesses unique strengths and challenges, Bubble Sort was notable for being the simplest to implement. Its straightforwardness, clarity, and low logic demands render it perfect for projects that need fundamental sorting capabilities.

Clarity and Ease of Comprehension

Bubble Sort follows a straightforward concept: examine two neighbouring elements and interchange them if they are not in the correct sequence. This procedure continues until the entire list is sorted. Its gradual approach simplifies understanding for newcomers, as it simulates a natural method of organizing objects, similar to sorting playing cards.

In comparison to other sorting algorithms, Bubble Sort is significantly easier to understand since it only necessitates a fundamental grasp of loops and conditional expressions. Conversely, Merge Sort and Quick Sort encompass more complex ideas, including recursion and divide-and-conquer strategies. Merge Sort, for instance, splits the array into smaller subarrays, sorts them recursively, and then merges them back into one. Likewise, Quick Sort chooses a pivot, divides the array, and sorts the partitions recursively. These steps necessitate a greater comprehension of programming logic, which makes their execution more difficult.

No Requirement for Complex Logic or Recursion

A key benefit of Bubble Sort is its lack of dependence on recursion or intricate logic. Recursion, employed in algorithms such as Merge Sort and Quick Sort, can be challenging for newcomers to comprehend since it demands an understanding of stack memory and termination criteria. In contrast, Bubble Sort employs an iterative method, which simplifies both understanding and execution. This simplicity makes Bubble Sort especially appropriate for tasks that don't demand high efficiency

Simple Troubleshooting and Visualization

Bubble Sort's consistent and orderly nature makes it one of the simplest sorting algorithms to analyse and understand visually. Through gradual comparison and exchange of elements, developers can effectively monitor the algorithm's advancement and spot possible mistakes. Visualization tools frequently utilize Bubble Sort as a demonstration due to its straightforward operation and ease of understanding.

Easier Code and Quicker Execution

In comparison to more intricate algorithms such as Merge Sort and Quick Sort, Bubble Sort demands significantly less code and effort to execute. More complex algorithms necessitate extra procedures, like dividing arrays, handling pivots, or combining subarrays. These actions extend the code's length and intricacy. Bubble Sort, due to its simple method, can be executed quickly with just a few lines of code.

Constraints and Appropriateness

Although Bubble Sort is simple to implement, it has notable drawbacks. The time complexity is O(n^2), rendering it impractical for large data sets. Algorithms such as Merge Sort (O (nlog\_n)) and Quick Sort (average-case O (nlog\_n)) are significantly more effective for sorting larger data sets. Nevertheless, for small datasets where efficiency is not essential, Bubble Sort is a viable option.

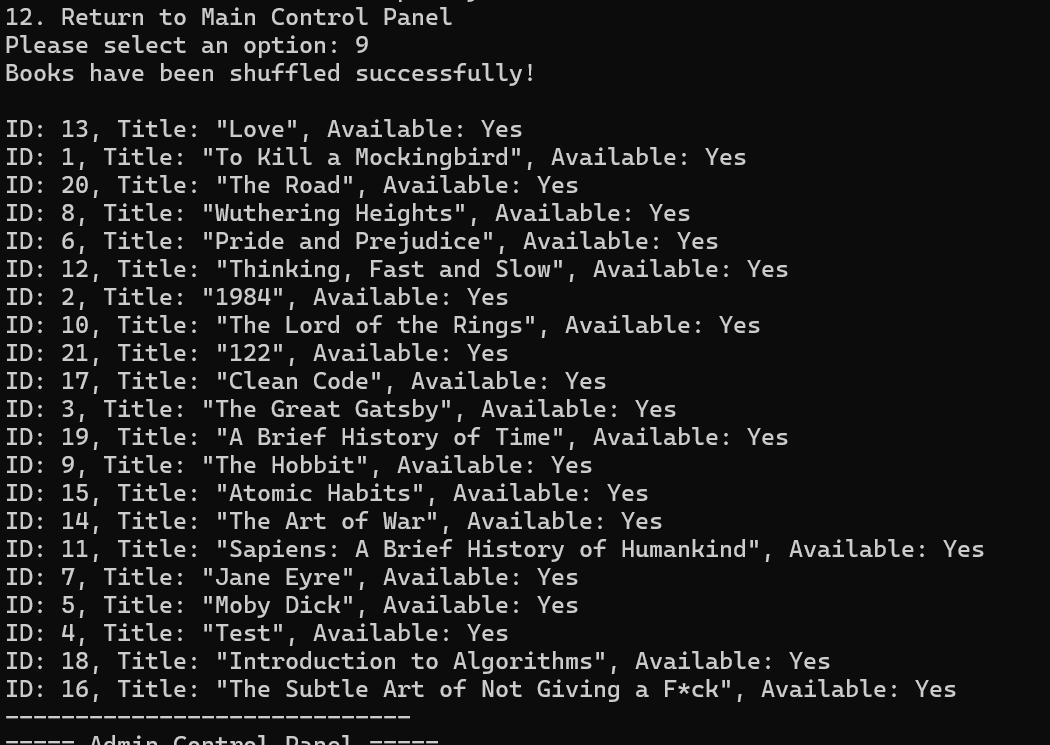
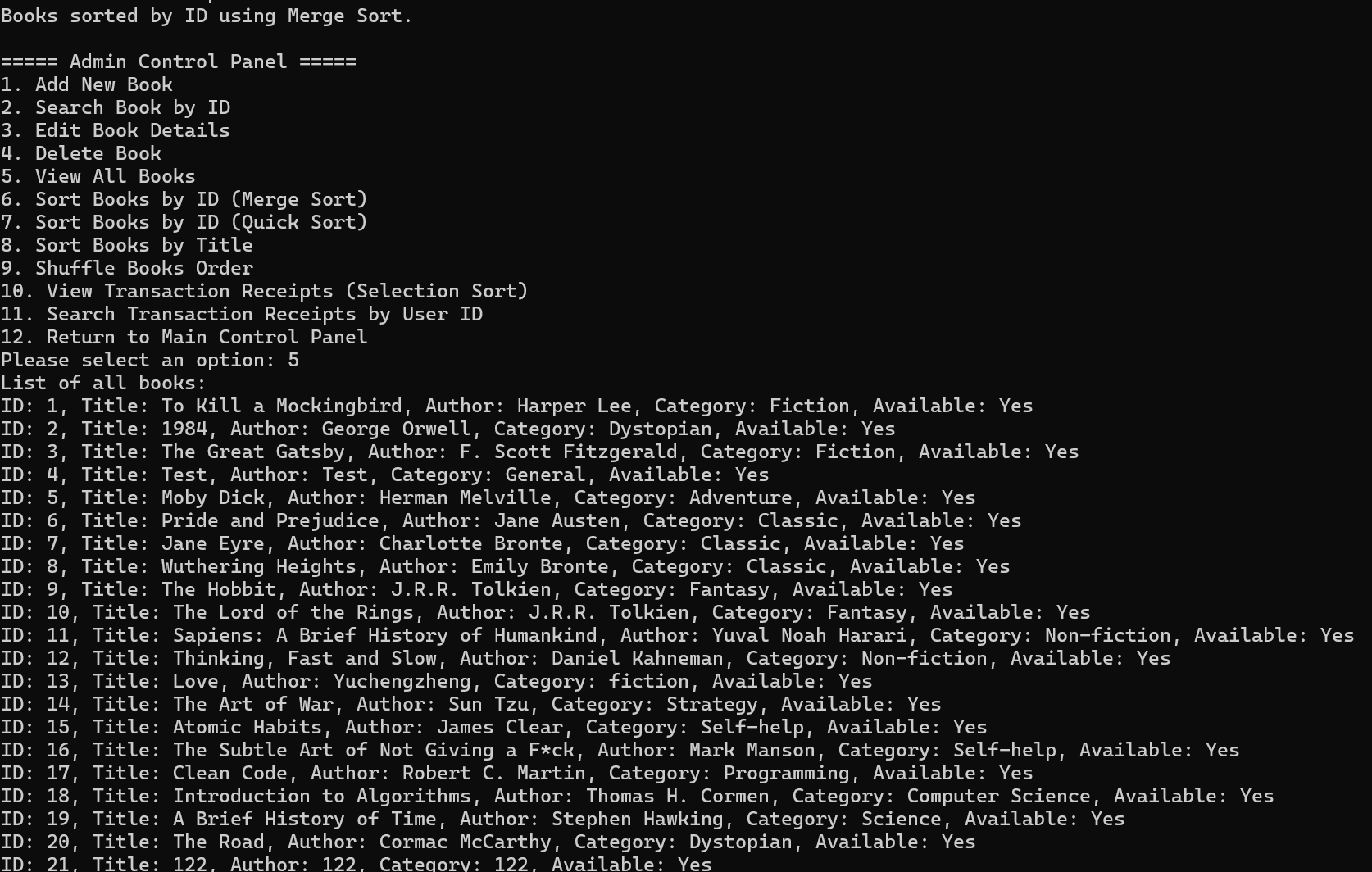
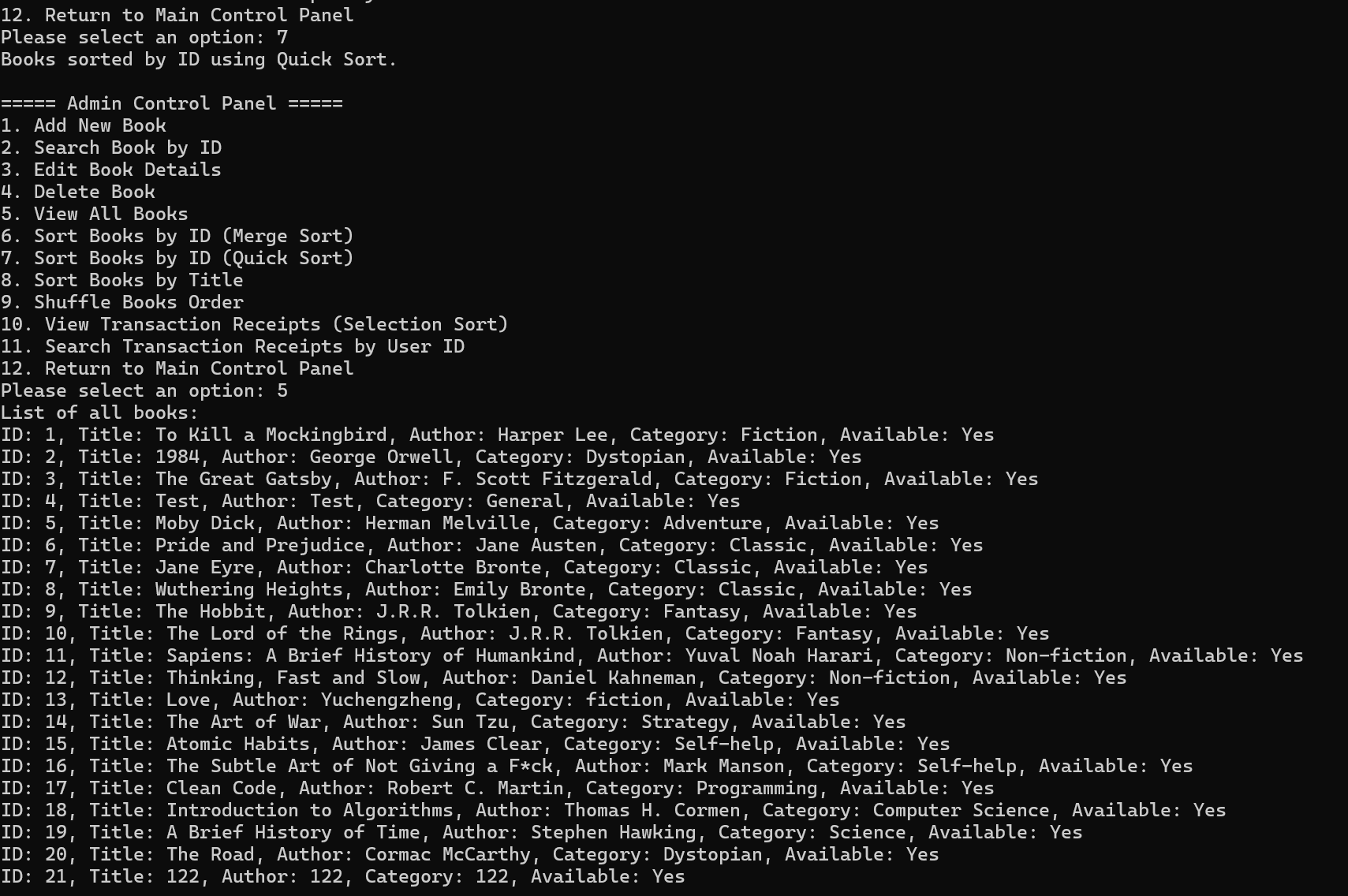
In the LMS project, Bubble Sort was employed to arrange book titles, a task that dealt with a fairly small dataset. Its straightforwardness guaranteed swift and precise execution without needing complex reasoning. For larger sorting assignments, such as organizing books by ID, more effective algorithms like Merge Sort and Quick Sort were employed to enhance performance.

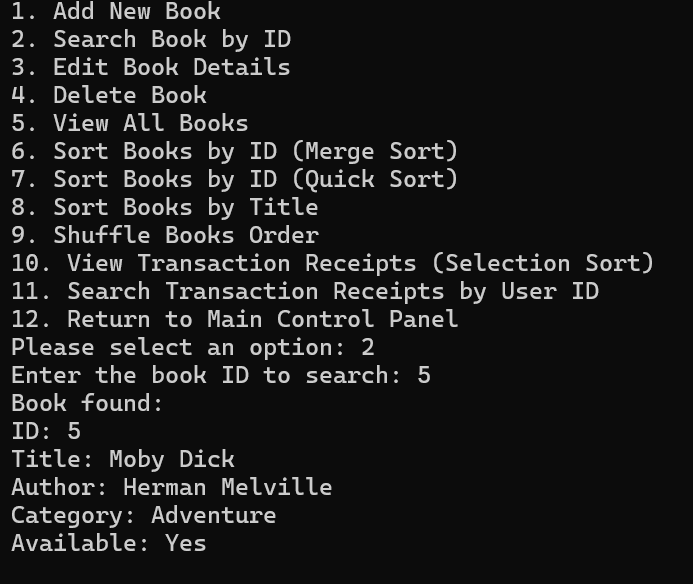
Conclusion

Bubble Sort is known for being the easiest sorting algorithm to use because it has a simple design, easy-to-understand logic, and is straightforward to debug. While it’s not suitable for large datasets due to its inefficiency, its simplicity makes it a handy tool for smaller projects.

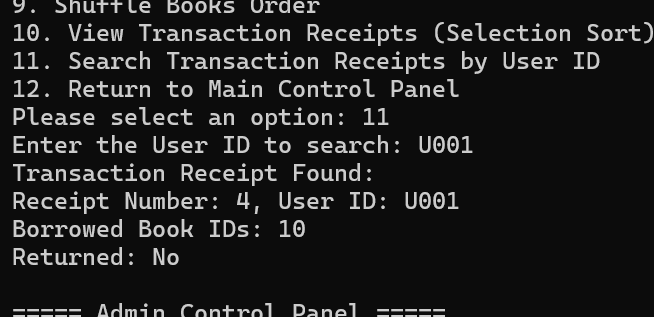
## 5. Brief Explanation of Algorithms Used

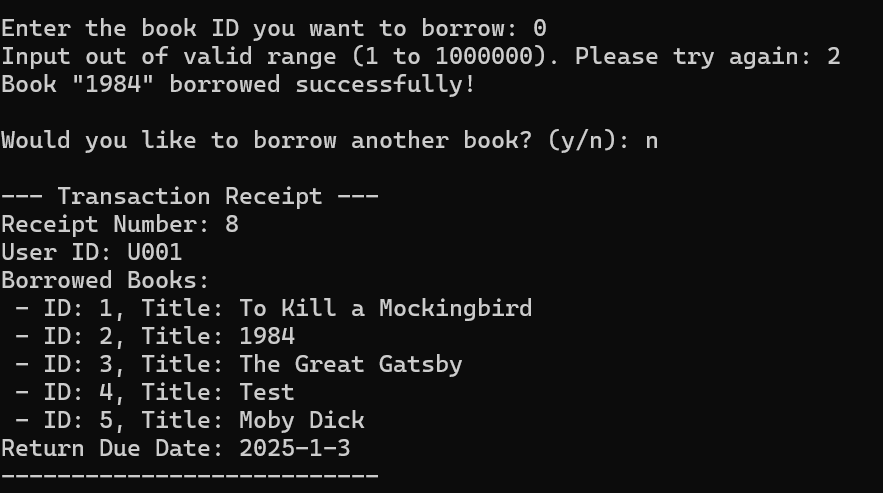
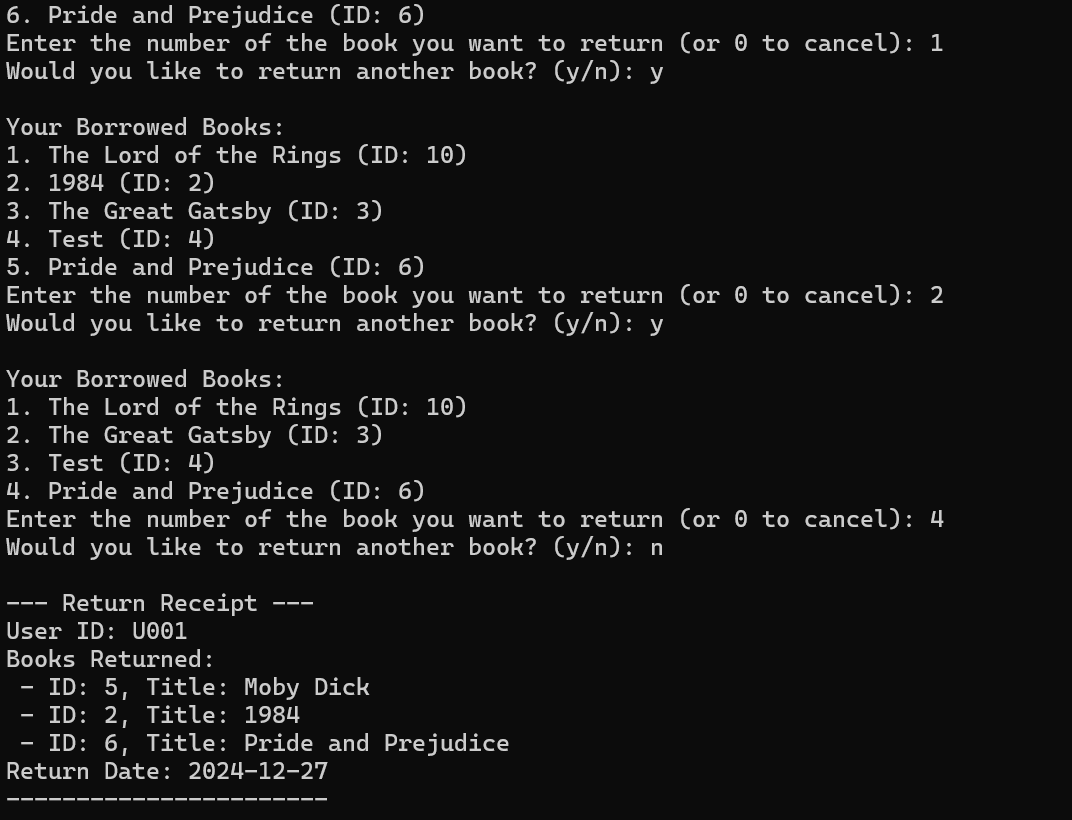
The library management system integrates several fundamental algorithms to optimize data handling and enhance user experience. For instance, **Bubble Sort** is employed to organize books alphabetically by their titles, ensuring that administrators can easily navigate through the collection. **(**Figure 1 ''Before Bubble sort''& Figure 2''After Bubble sort''**)**

Additionally, the system utilizes both **Merge Sort** and **Quick Sort** to efficiently sort books numerically by their IDs, which facilitates rapid retrieval and management of book records. (Figure 3 ''Before sorting'', Figure 4 ''After merge sort'', Figure 5 ''After quick sort'')

When users search for a specific book by its ID, the system leverages **Binary Search** to swiftly locate the desired entry within the sorted list, which significantly reduces search time. (Figure 6 "Binary search")

Similarly, transaction receipts are organized using **Selection Sort** based on receipt numbers, allowing for orderly record-keeping and easy access to transaction histories. ( Figure 7 "Selection Sort")

Furthermore, to display a user's borrowing history, the system implements **Linear Search**, which sequentially scans through transaction records to compile all relevant entries for the user. ( Figure 8 "Linear search")

we also applied quick sort of borrowed book IDs, in order to finally show receipts for borrowed books or returned books.(Figure 10 "Quick sort for borrow receipt", Figure 9 "Quick sort for return receipt")

These algorithmic implementations collectively and ensures that the system operates efficiently, providing quick access to information and maintaining organized data structures for both administrators and users.

6. Error Handling Approach and Challenges

### 6.1 Error Handling Approach

**File Operations**:

The system verifies that files (e.g., books.txt, receipts.txt) are accessible before performing any read or write operations. If a file cannot be opened due to issues like missing files or insufficient permissions, a detailed error message will be displayed on the console and recorded in error\_log.txt.

A retry mechanism is implemented for saving files. This allows the user to fix the issue like provide the correct file path or permissions and attempt the operation again before the system gives up loading.

**User Input Validation**:

We also use input validation to ensure the accuracy of user data. For example, the input format of user IDs must be some strings like “U001”, and book IDs must be valid integers within the specified range. If invalid input is detected, an error message will be displayed, reminding the user to re-enter the data until it meets the required input format.

**Logical Operation Validation**:

We designed logic check when a user borrowing or returning books. For instance, the system ensures users cannot borrow books that are unavailable or return books they have not borrowed.

A rollback mechanism is implemented to maintain data consistency. For example, if an error occurs during a book borrowing transaction, the system reverts all changes (e.g., resetting book availability and removing incomplete receipts).

**Logging**:

A logging system records critical errors in an error\_log.txt and error.txt file. Each entry includes a timestamp and a detailed error description, in that case developers can diagnose issues efficiently.

### 6.2 Challenges Faced

**Limited Understanding of Error Handling**:

As a beginning developer, we initially overlooked many error scenarios, such as file absence or invalid user inputs. This resulted in many compiling errors during early testing.

The use of C++ features like try-catch blocks and error codes was challenging for us, so we need to take extra time to learn the relevant methods by ourselves.

**Handling Complex Scenarios**:

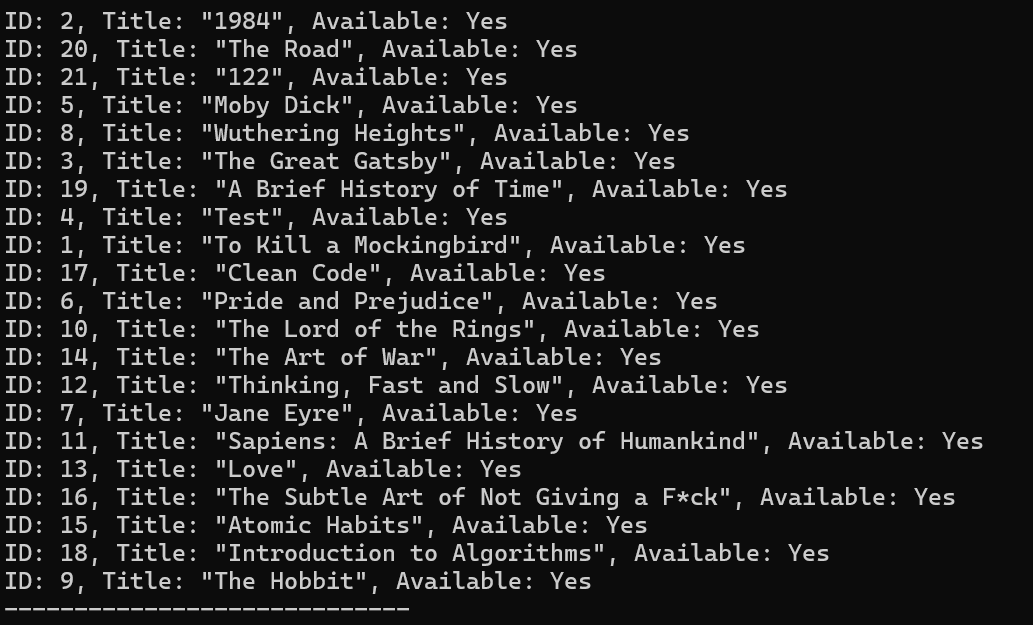
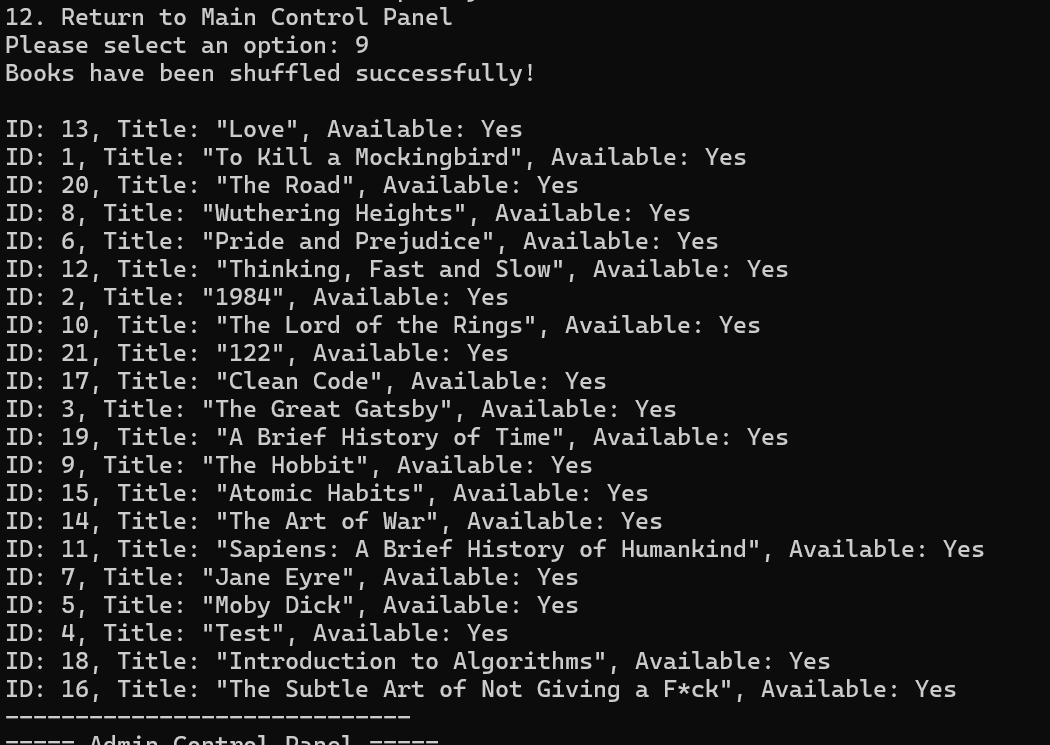
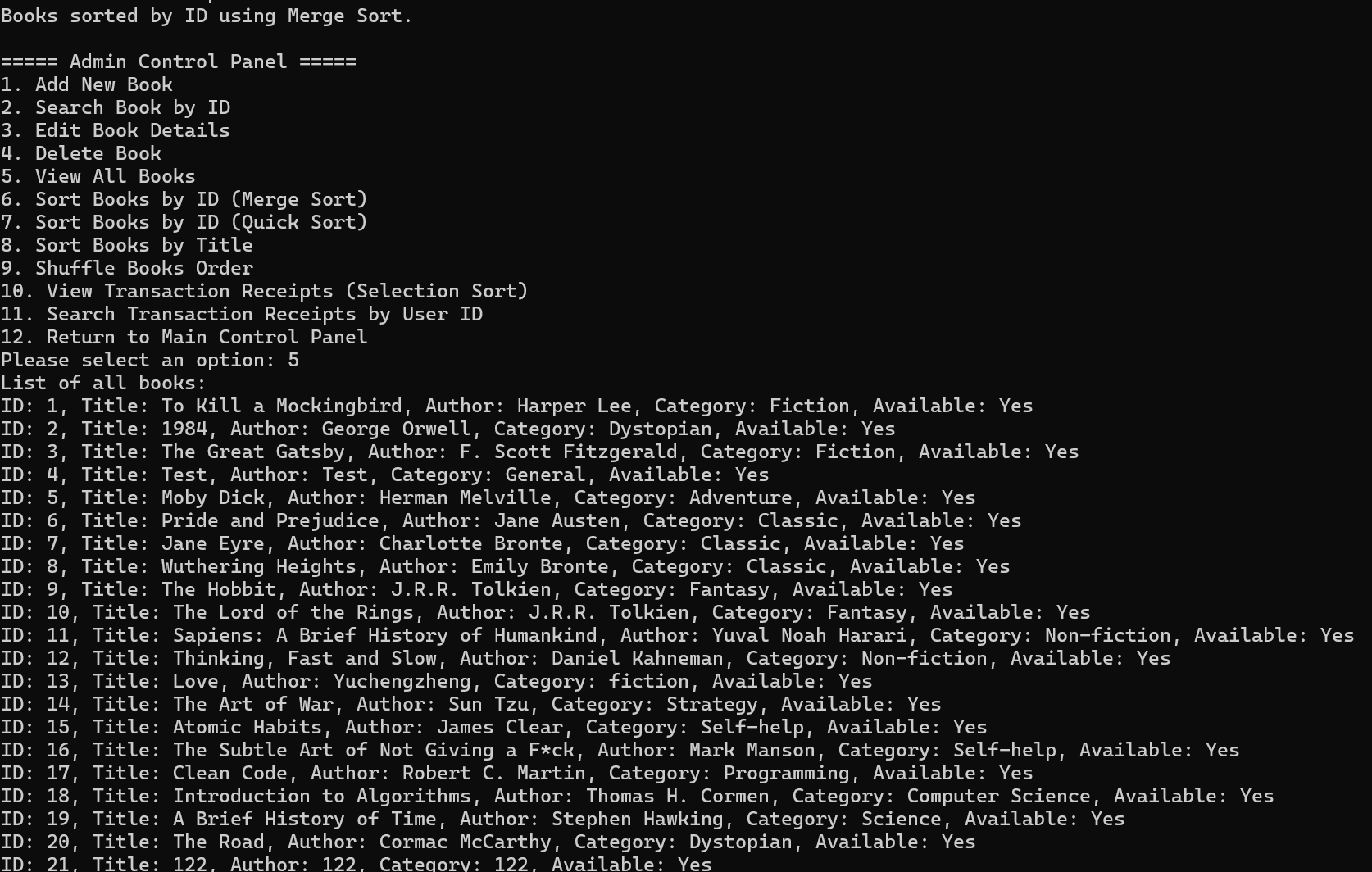
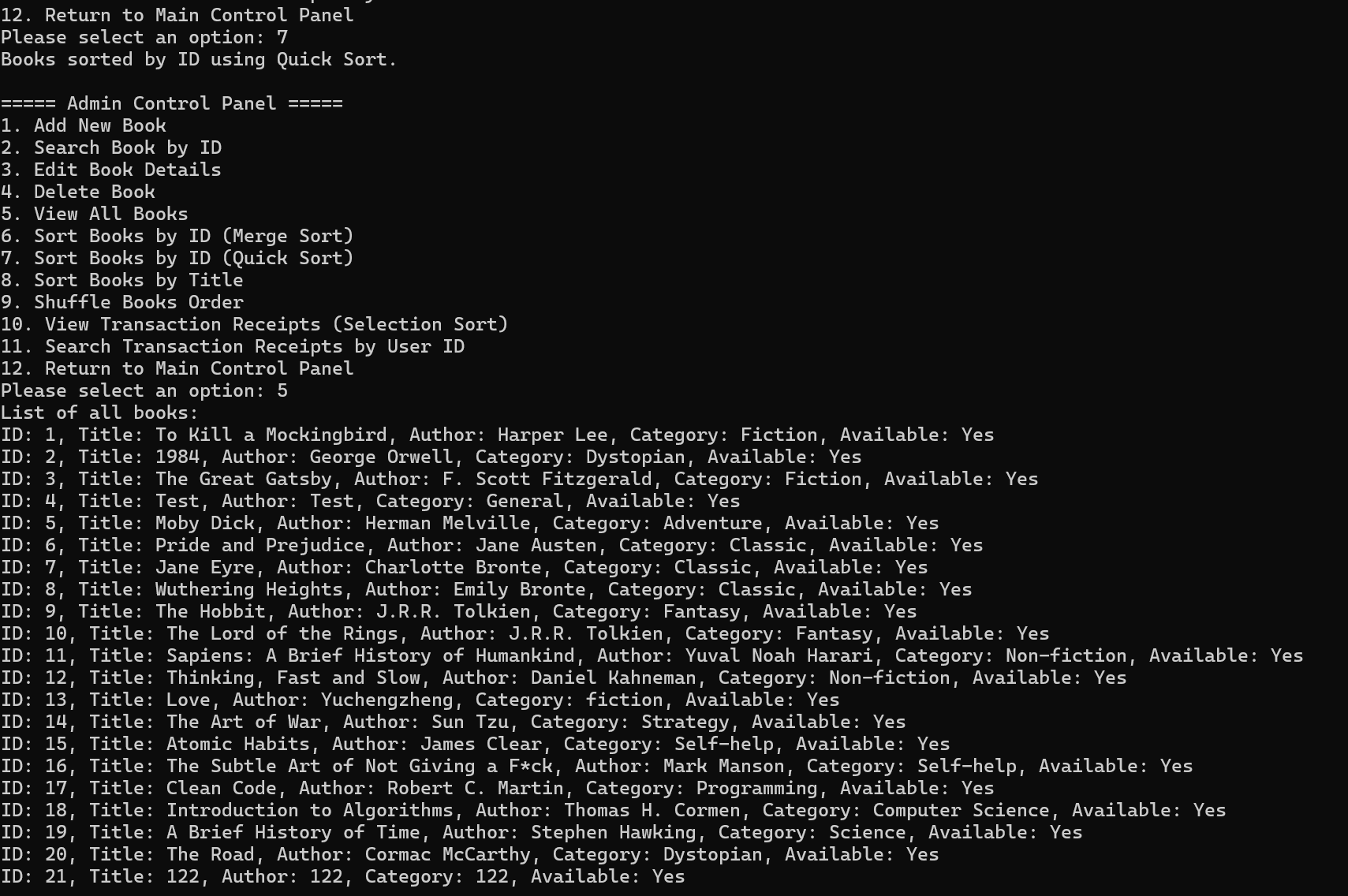
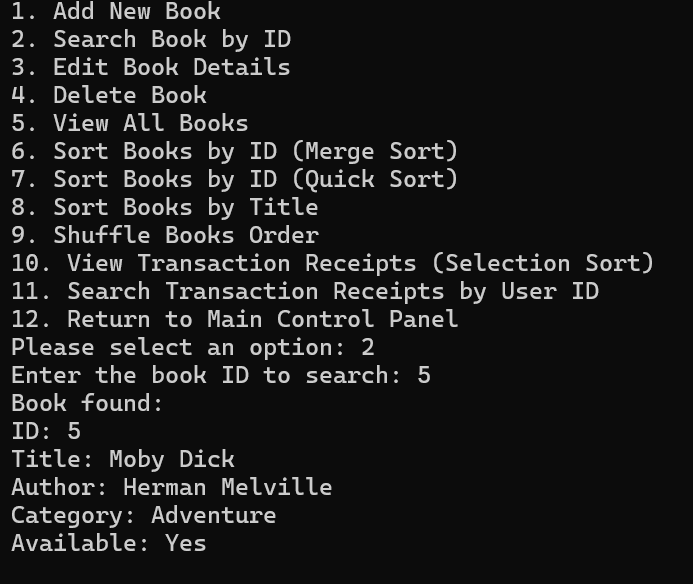
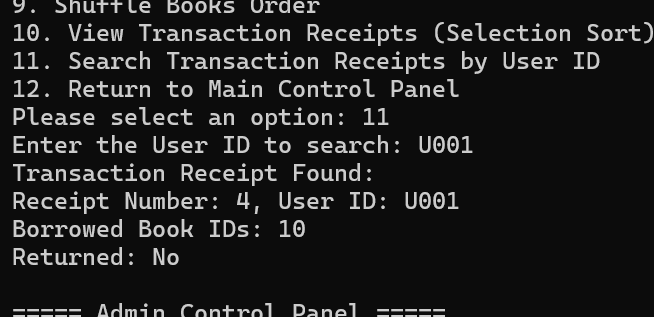
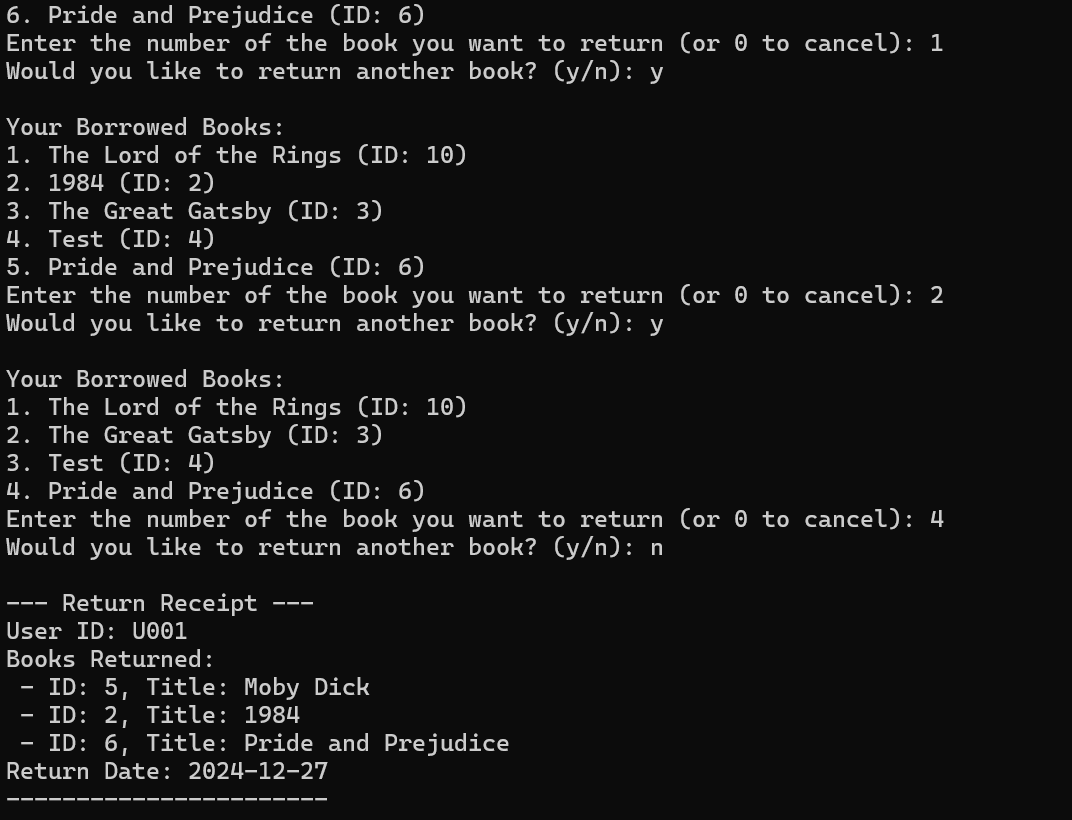
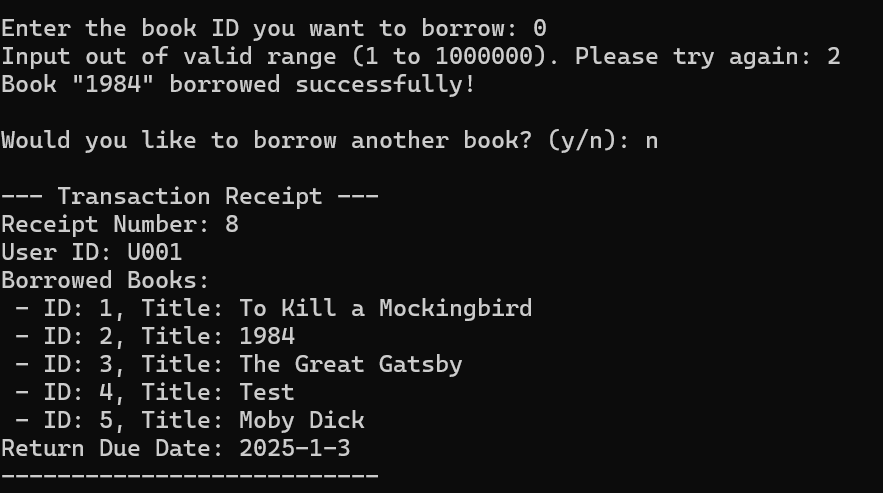
Implementing rollback logic was quite challenging. For example, when something goes wrong, we need to immediately undo the book’s borrowing status and delete the related receipt records. This requires different parts of the system to work closely together to prevent errors. Additionally, adding retry options for file operations means we have to repeatedly check how functions interact and depend on each other, which makes the process even more complicated.

**Debugging Difficulties**:

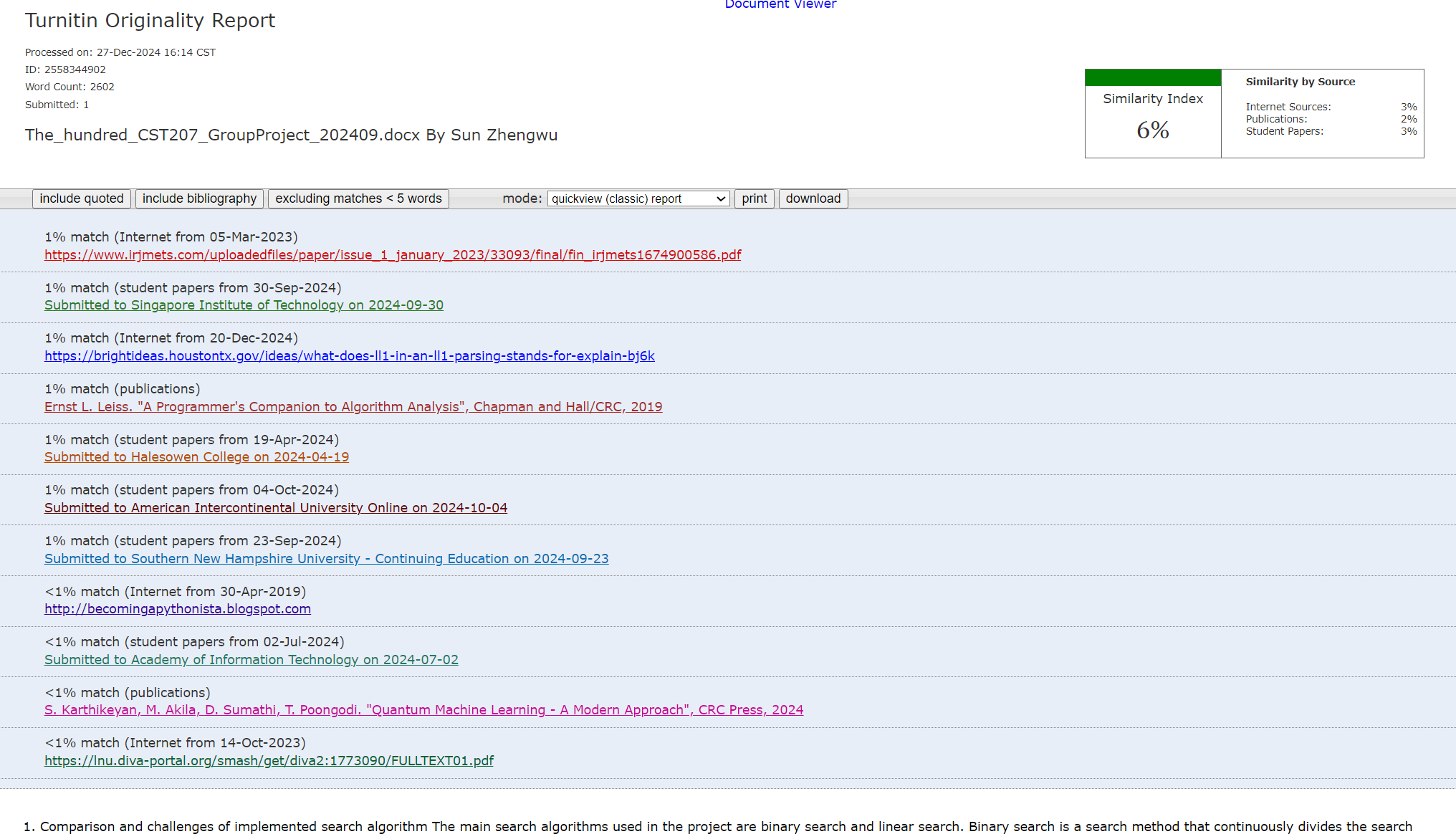
Error handling logic was scattered across multiple .cpp files, which makes it hard to track the flow of errors in the system initially. And some of us cannot use Git to manage the version of our codes, so without a centralized error-reporting and recording mechanism, debugging became time-consuming.

## 7. Appendix

### 7.1 Reference:

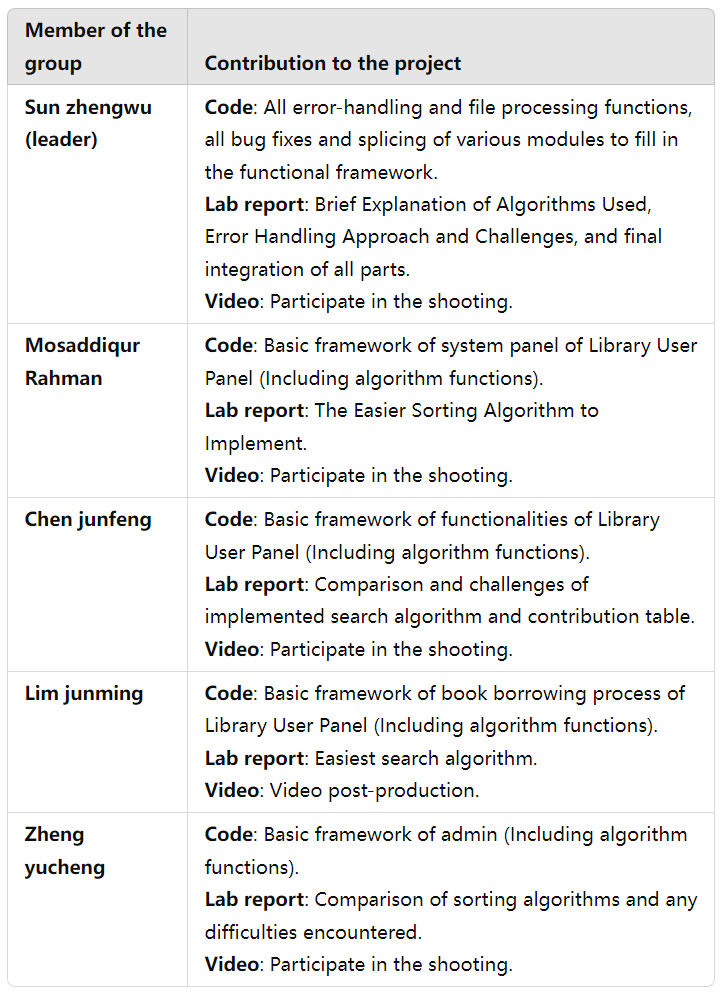
1. Sedgewick, R. (2002). Algorithms in C, Part 5: Graph algorithms. Addison-Wesley. Retrieved from <https://www.pearson.com/us/higher-education/program/Sedgewick-Algorithms-in-C-Parts-1-4-3rd-Edition/PGM199497.html>
2. Weiss, M. A. (2013). Data structures and algorithm analysis in C++ (4th ed.). Pearson. Retrieved from <https://www.pearson.com/us/higher-education/program/Weiss-Data-Structures-and-Algorithm-Analysis-in-C-4th-Edition/PGM1070418.html>
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press. Retrieved from <https://mitpress.mit.edu/9780262033848/introduction-to-algorithms/>
4. Govindswamy, S. (n.d.). Library Management System [Computer software]. GitHub. Retrieved from <https://github.com/govindswamy26/Library-Management-System>
5. GeeksforGeeks. (2024). Bubble sort: How it works and implementation in C++. Retrieved from <https://www.geeksforgeeks.org/bubble-sort/>
6. Patel, J. (2023). Simplified sorting algorithms for beginners. Journal of Computer Science Education, 12(3), 45–50.
7. Wirth, N. (1976). Algorithms + data structures = programs. Prentice Hall.
8. Figure 1 ''Before Bubble sort''
9. Figure 2''After Bubble sort''
10. Figure 3 ''Before sorting''
11. Figure 4 ''After merge sort''
12. Figure 5 ''After quick sort''
13. Figure 6 "Binary search"
14. Figure 7 "Selection Sort"
15. Figure 8 "Linear search"
16. Figure 9 "Quick sort for return receipt"
17. Figure 10 "Quick sort for borrow receipt"

### 7.2 Turnitin Report



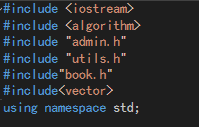


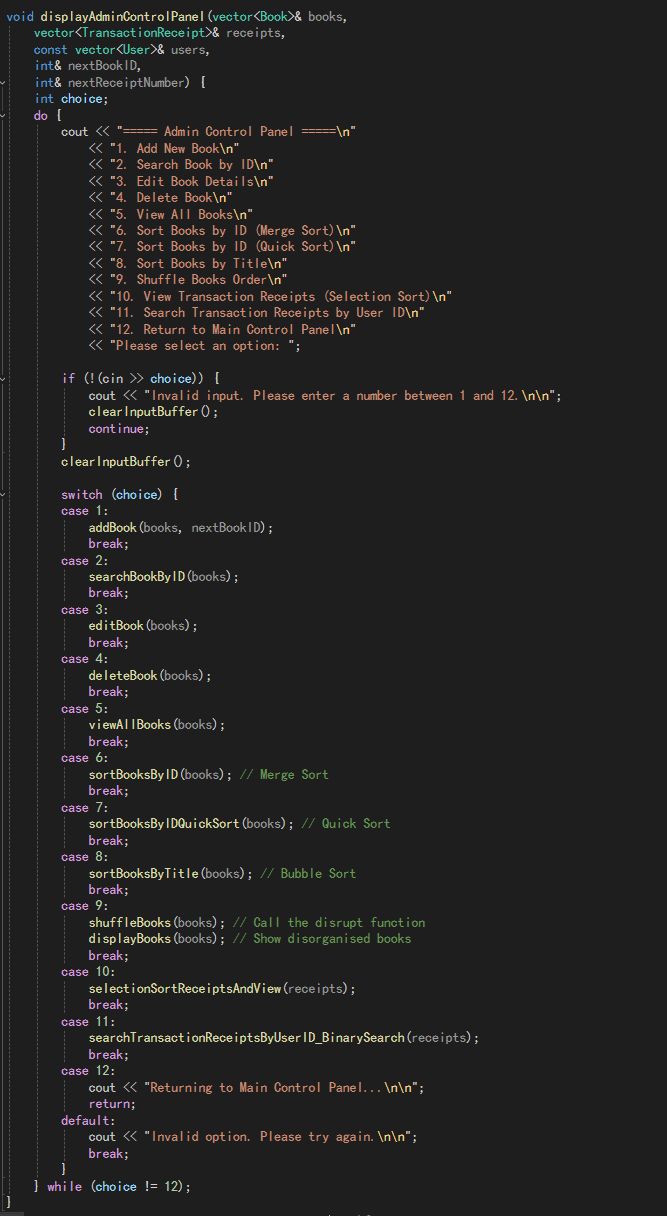
## 8. Contribution Table

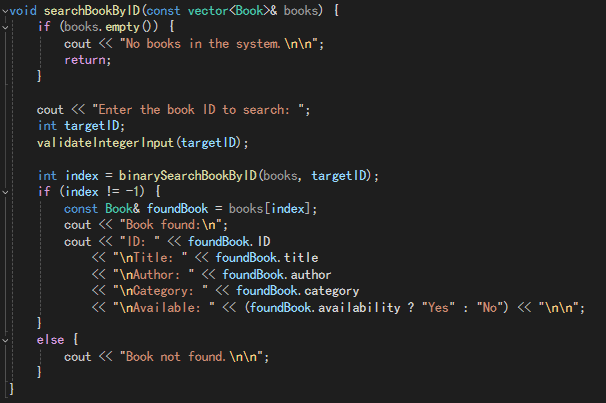
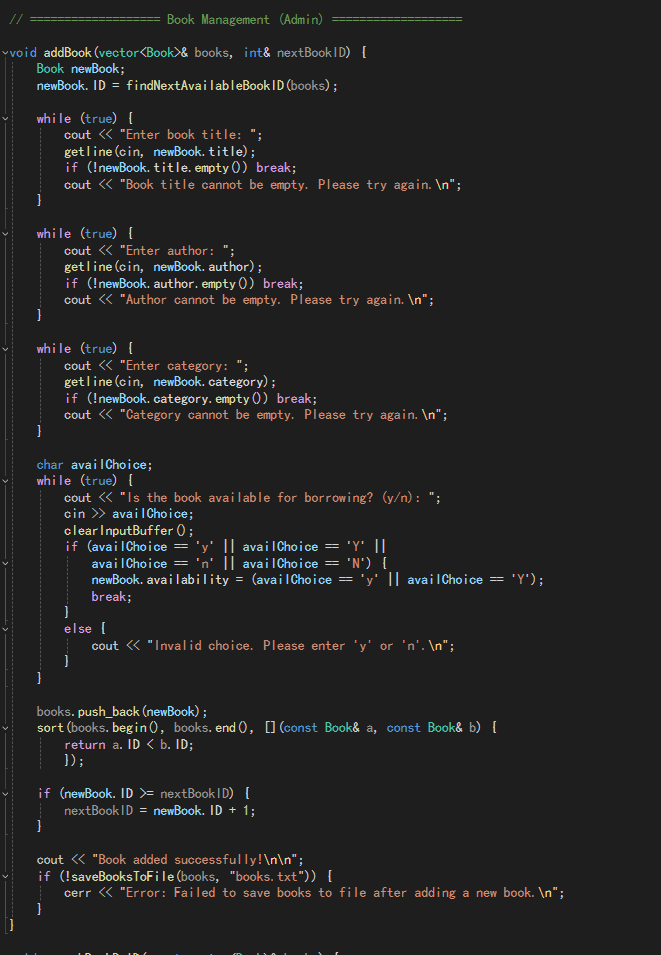


## 9. Appendix:Source Code

### Admin.cpp





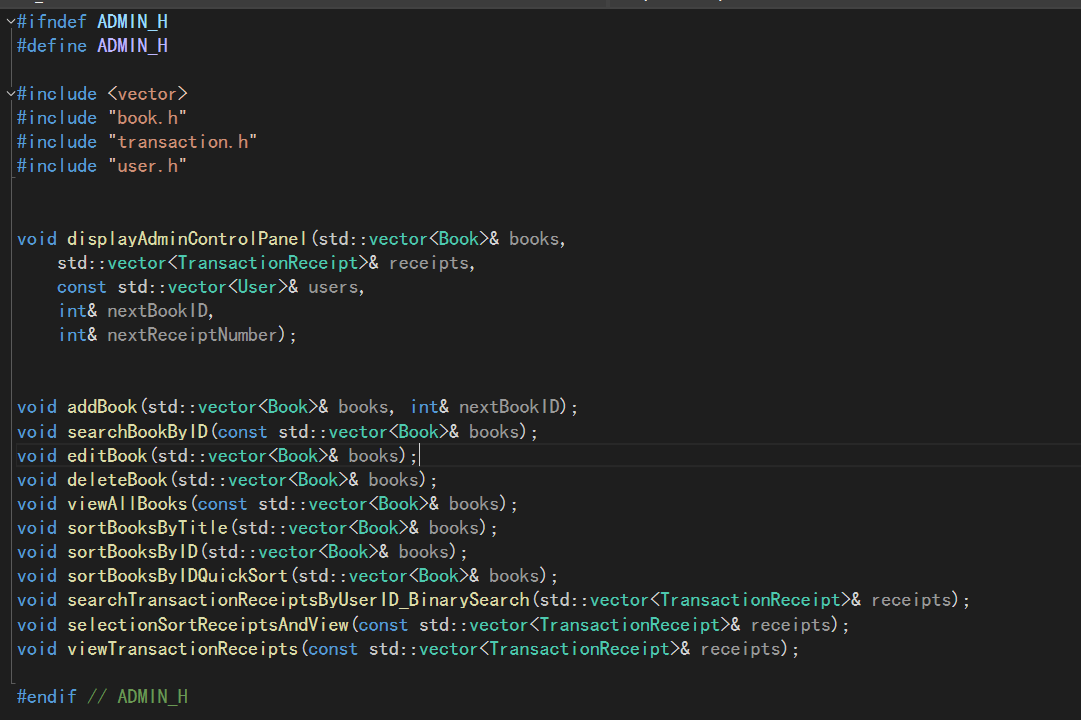




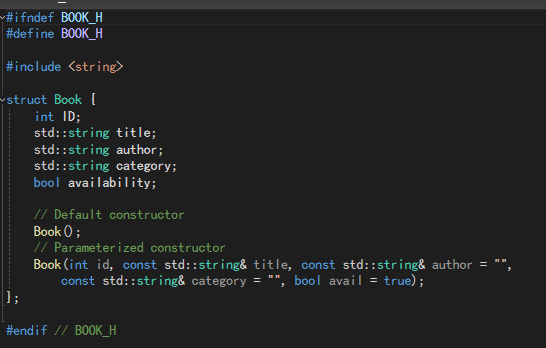




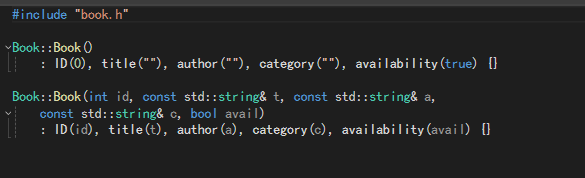
### Admin.h



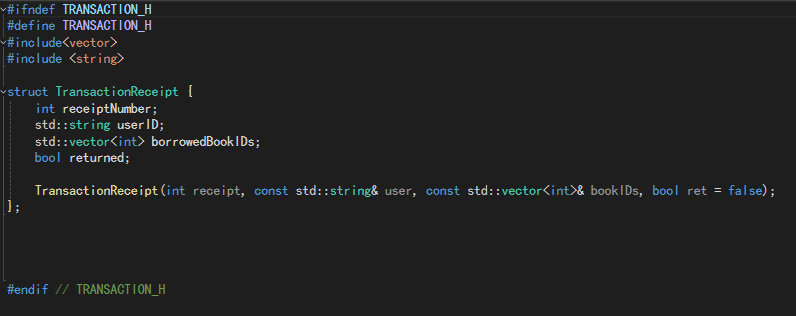
### Book.h



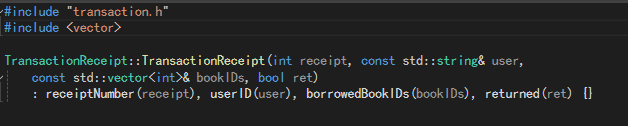
### Book.cpp



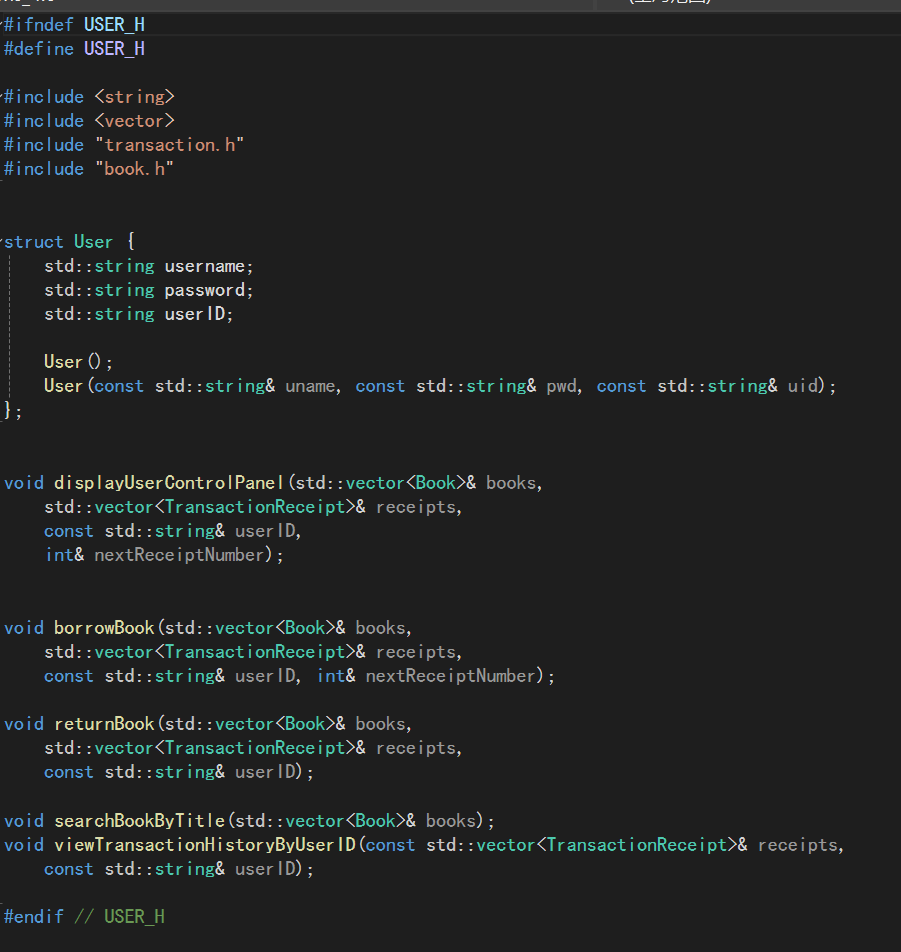
### Transaction.h



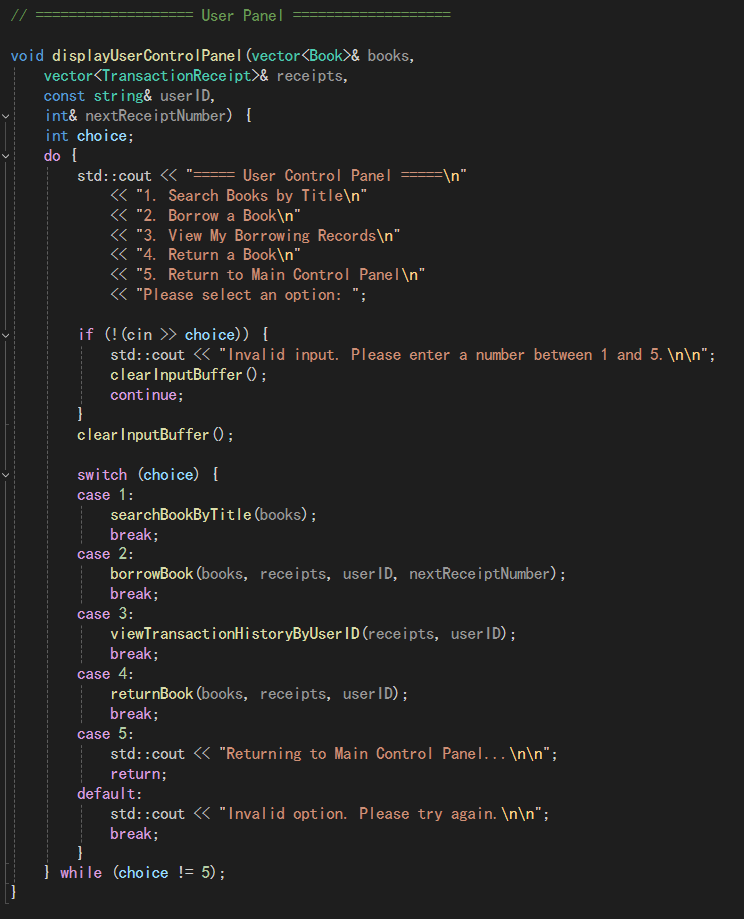
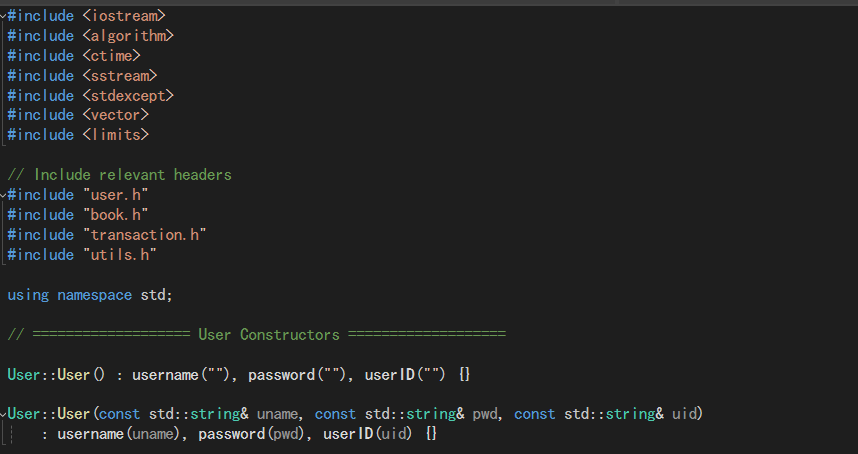
### Transaction.cpp

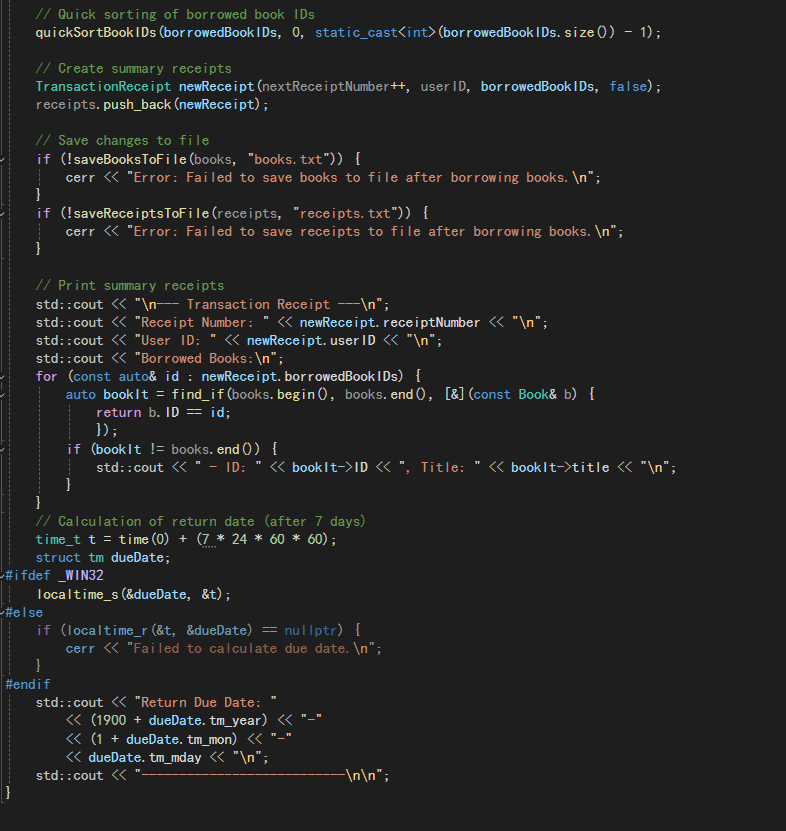


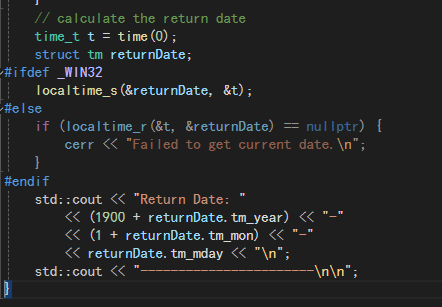
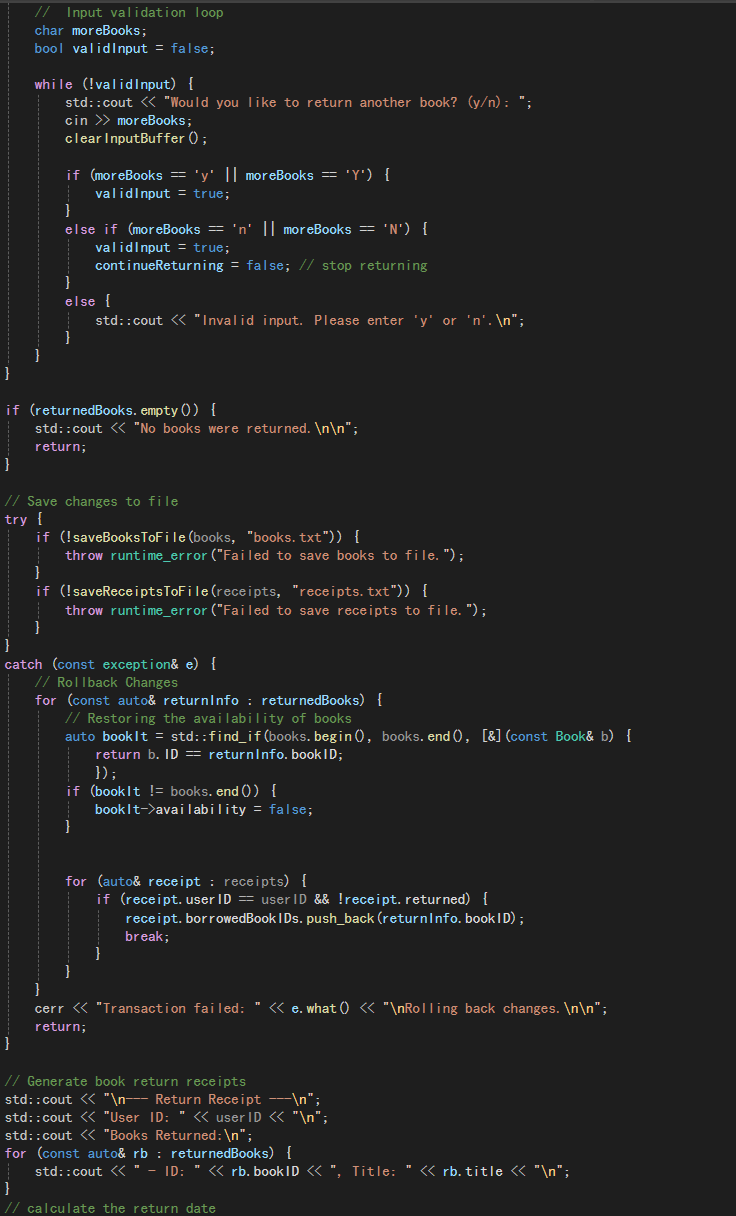
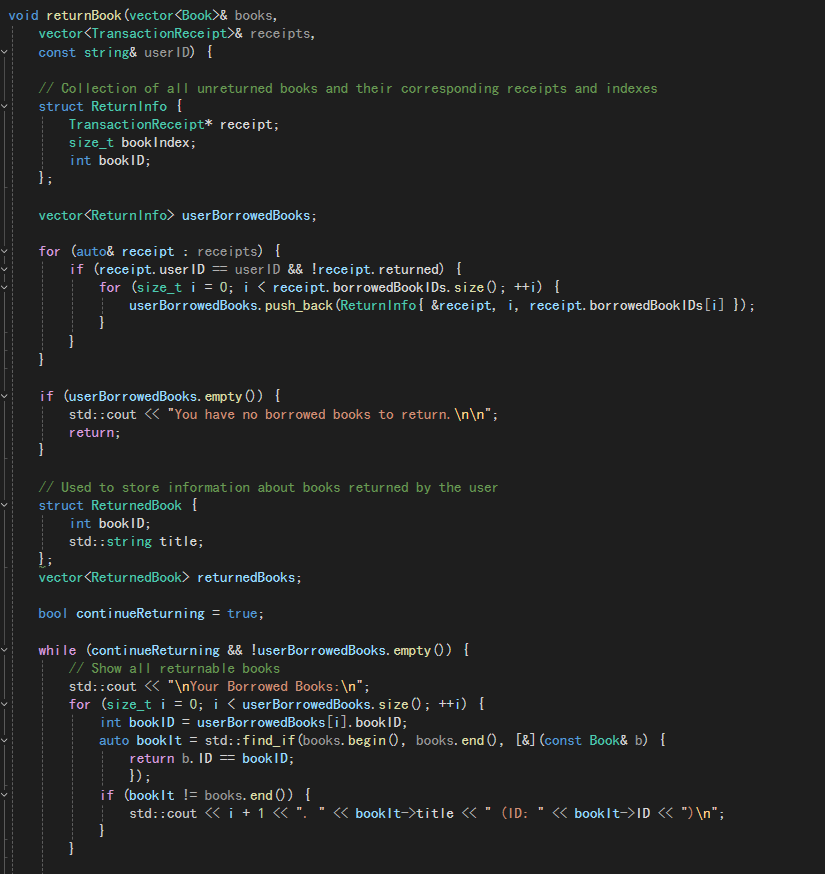
### User.h

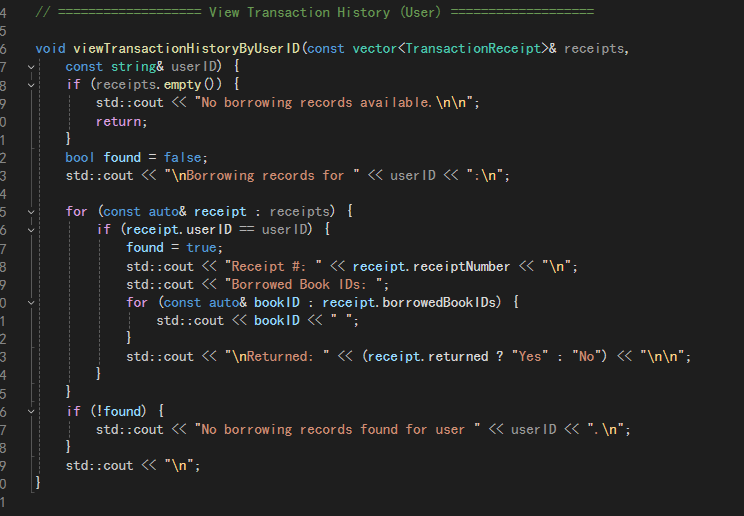


### User.cpp

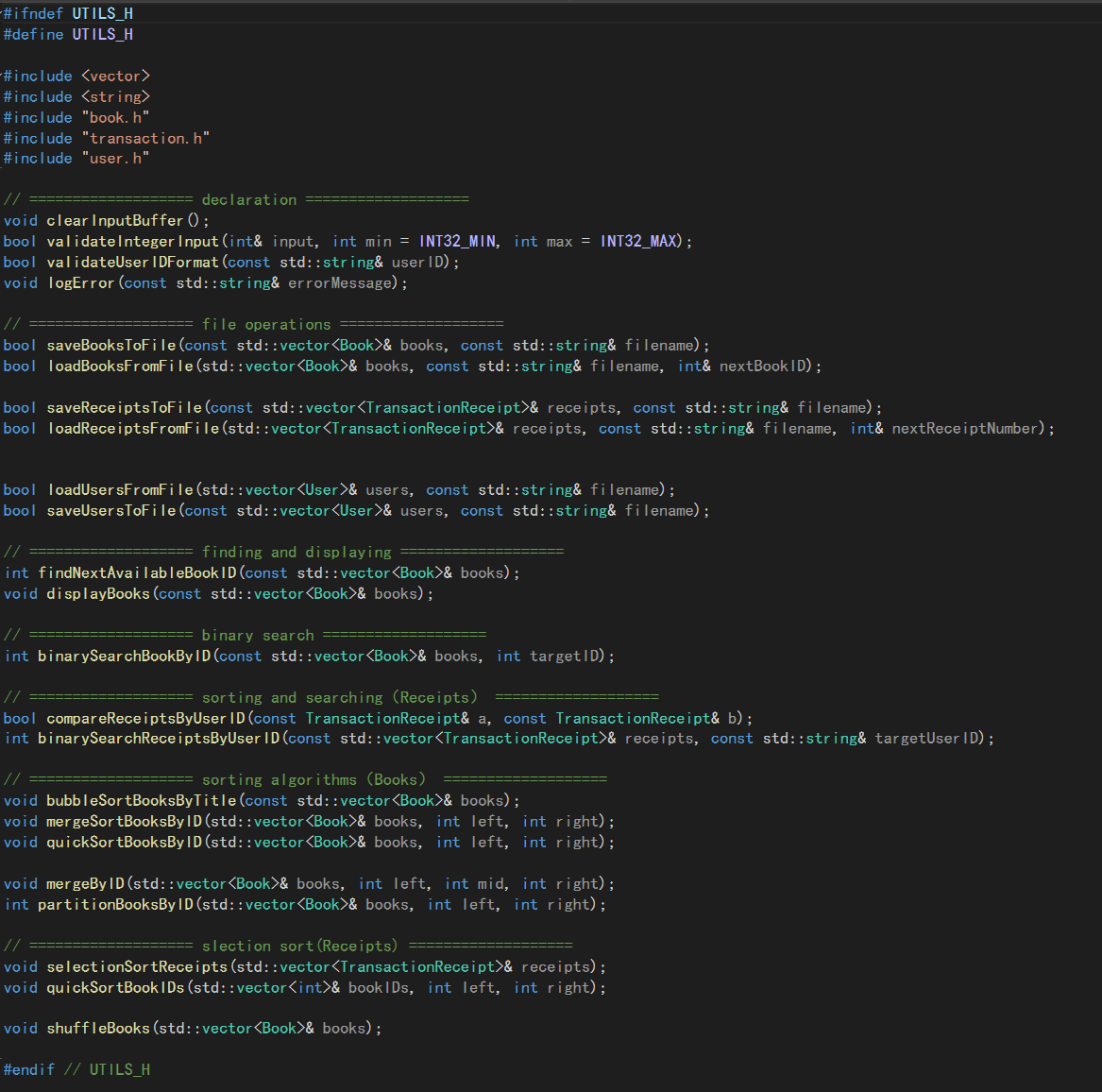




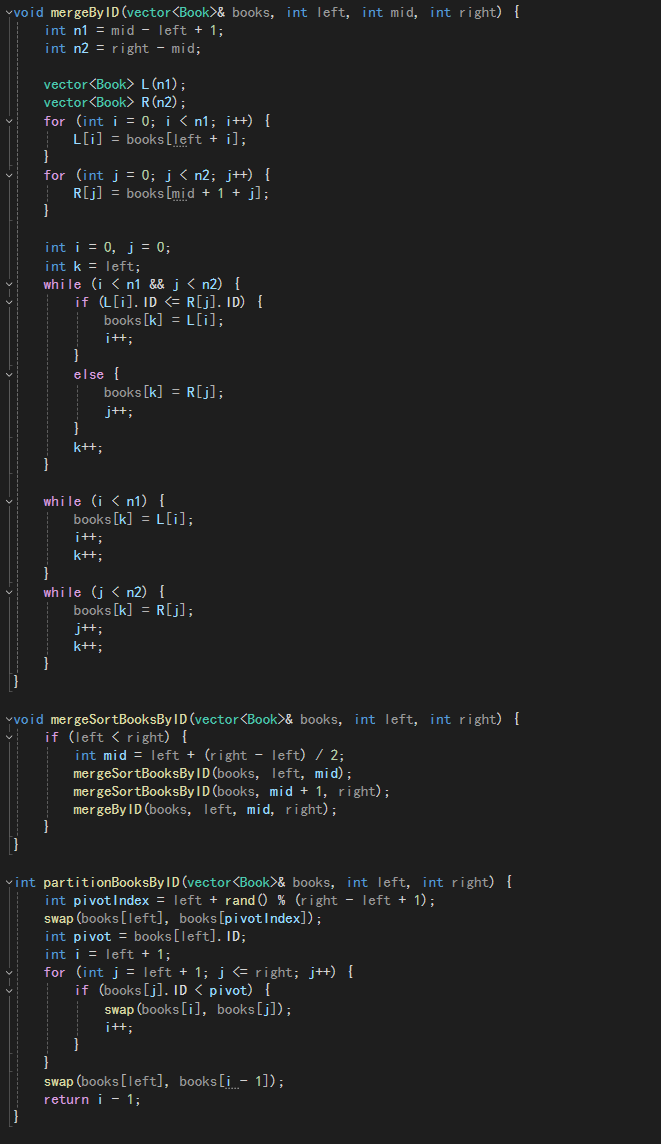
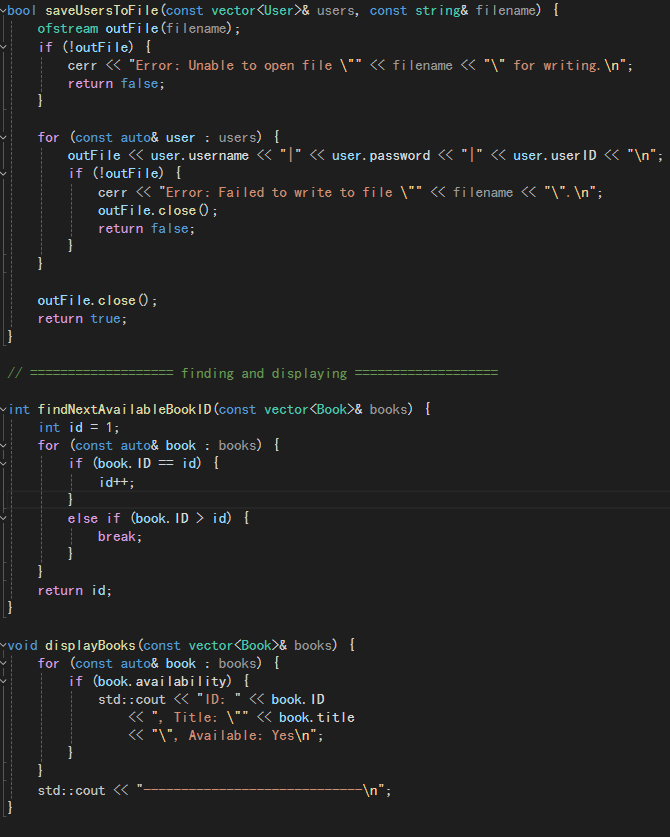
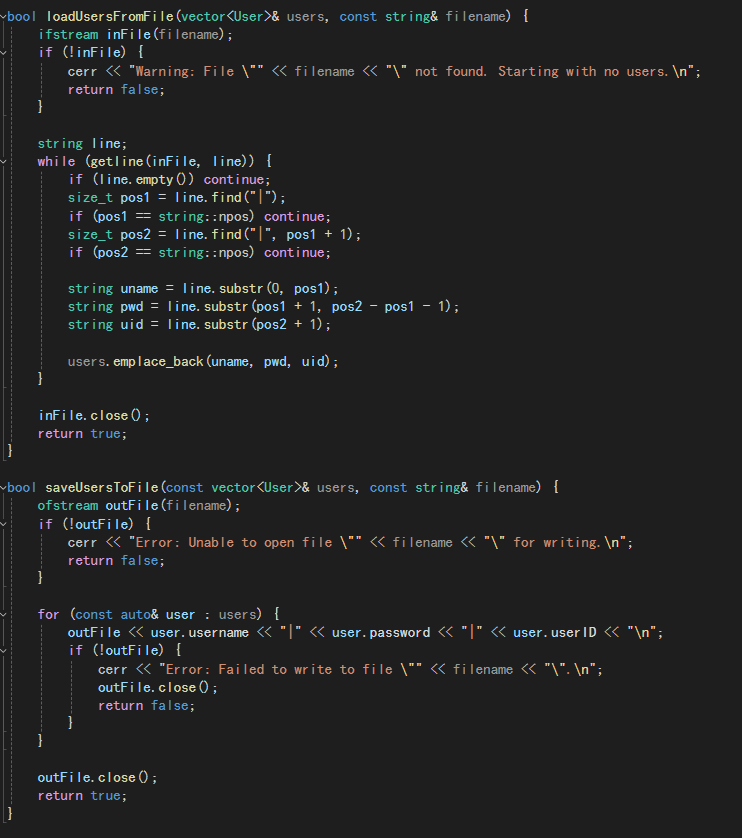
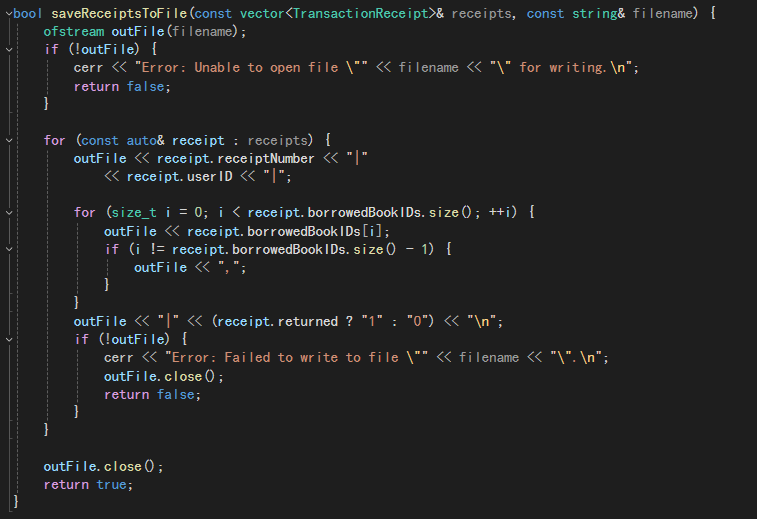
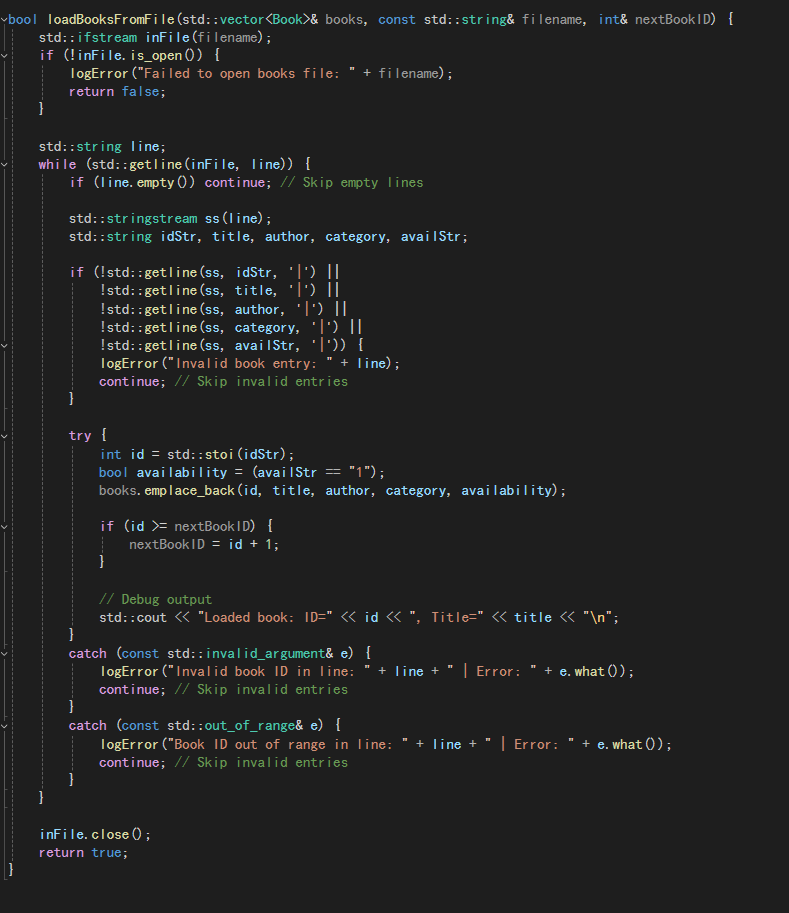




### Utils.h



### Utils.cpp



## 10. Marking Rubric

