STUDENT ID: 24128456

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

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

Technical Report: Filtering and Ranking Books by User-Selected Criteria (Genera, Price and Rating)

Author:	**Prashant Acharya (BCU)**
Date:	**June 19, 2024**
Wordcount:	**Approximately 1700 words**
Page count:	**21**
Confidential:	**Yes – Department only**

Acknowledgement:

I am deeply thankful for the incredible opportunity to present my technical report on "Student Study Performance" as part of the "Data Structure and Algorithm" module. Throughout the module, I encountered numerous challenges. However, with the guidance and support of my teacher, I was able to navigate through them successfully. His advice was invaluable, and I am profoundly grateful for his assistance. Additionally, the internet tools I utilized during my research were immensely helpful in completing this project.

Table of Contents

Contents

Lists of Figures and Tables

Executive Summary:

This technical report presents the design and implementation of a book filtering and ranking system aimed at enhancing the efficiency of book selection in bookstores and libraries. The system employs fundamental data structures such as lists and dictionaries for the dynamic storage and categorization of books, facilitating easy manipulation and efficient access. The core algorithm used for sorting is Quick Sort, chosen for its superior average-case time complexity of O(n log n), making it well-suited for handling large datasets. The implementation includes reading and parsing CSV data, organizing books by genre, and allowing user input to determine the ranking method based on price or rating. By integrating these components, the system ensures that users can quickly locate books that match their preferences, thereby improving the overall user experience in large-scale book databases.

Performance evaluation of the system demonstrates its effectiveness in terms of time and space complexity, with Quick Sort providing a good balance between speed and resource usage. The best and average case time complexity is O(n log n), while the worst case is $O(n^2)$, with the total space complexity being O(n + m + k), where n is the total number of books, m is the number of books in selected categories, and k is the number of categories. A comparison with other sorting algorithms like Merge Sort and Heap Sort reveals that while Quick Sort is generally efficient and straightforward, the others might be preferable in scenarios requiring more predictable worst-case performance. The report concludes that the system successfully addresses the need for efficient book filtering and ranking, offering a practical, scalable solution that can be adapted for various user requirements in bookstores and libraries.

Introduction:

Understanding need of Sorting Books in Huge Volume

Book Sorting systems is an important topic to focus on, finding a perfect book based on the user's need. Implementing this system boosts the efficiency of bookstores and libraries in finding books. Filtering books eases the difficulty for readers to quickly find the books they want when choosing books. Thus, the book recommendation system is becoming more and more important (Wang & Hou, 2021).

Thus, the system enables a reader to know which books are of high quality or read most. This can help the reader a lot of time and even make sure that they put their money in the right place investing in books that they will find interesting.

Objective of the report

Making a ranking system of books based on their genera helps us to find books, show books in the store and, showcasing best way to refer the perfect. This works by introducing more nested ranking and filtering system to enhance the system. In this report, we will work on the bookstore dataset, which provides data for ranking the books mainly focusing on genre, prices of book and rating. Thus, the goal of this report is to reach an acceptable level of complexity at computational level and the result of ranking.

2.Theory:

Data Structure and Algorithm used

This section contains a brief description of the algorithms and data structures required for the proposed algorithm.

Data Structure:

- **List**: Used to store books in a specific category and to hold books that are in stock. Operations such as filtering and sorting are done here. As allows dynamic storage and easy iteration over book entries, it is preferred more than other datasets.
- **Dictionary**: Used to categorize books by genre. This allows efficient access and manipulation of book categories. Provides average O (1) time complexity for lookups, making it fast and efficient to categorize and access books by genre. With its key-value property it is comparatively better than other data structures to effortlessly categorize huge volume of data.

Algorithm:

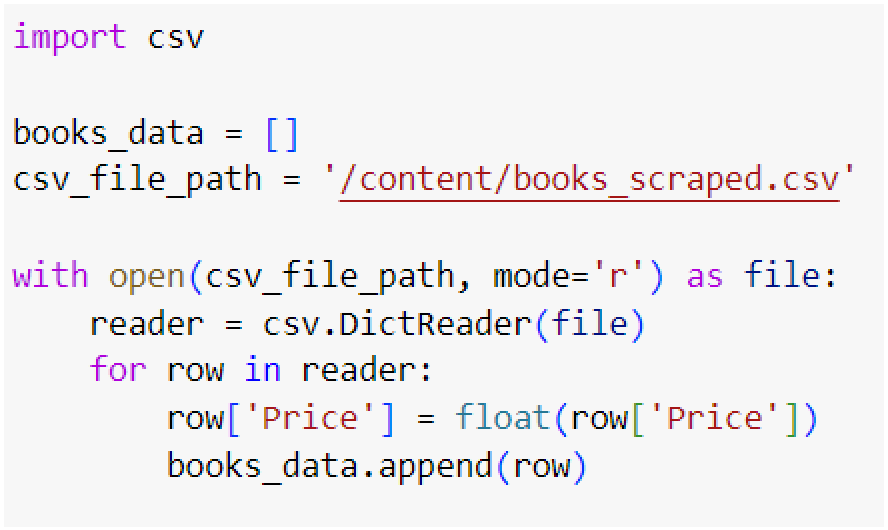
- **Filtering:** It filters all the raw data from the CSV file and filters them based on the genre of the books present in the bookstore. Then stores them into list category for further filtering and ranking of data. It is mainly used to categorize users' interest-based genre books, availability of book or is/not in stock and preparation for ranking algorithms.

- **Sorting:** The quicksort function is employed to sort the books either by price or rating. Quick Sort is a highly efficient sorting algorithm with an average time complexity of O (n log n), making it suitable for sorting lists of data. It was chosen for its efficiency in sorting large datasets. Its divide-and-conquer approach ensures that even in the worst case, it performs reasonably well compared to other O(n^2) sorting algorithms like bubble sort.

3.Implementation:

3.1 Reading and Parsing CSV Data:

Data is being read from CSV file, price field is converted into float and each book's data is stored in a list of dictionaries.

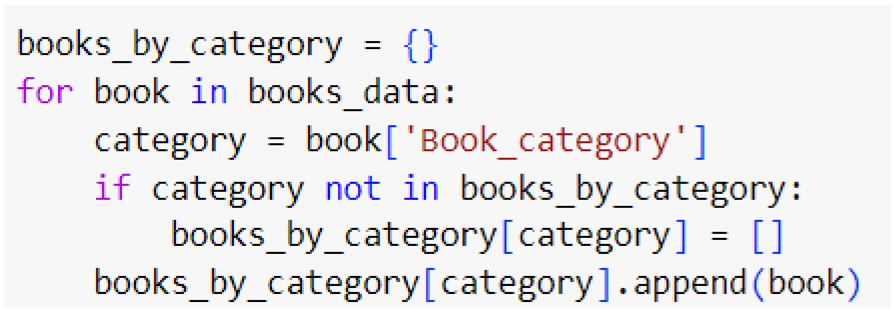


```
import csv

books_data = []
csv_file_path = '/content/books_scraped.csv'

with open(csv_file_path, mode='r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        row['Price'] = float(row['Price'])
        books_data.append(row)
```

Figure1: code snippet of reading CSV Data

3.2 Organizing books by category:

Books are grouped into categories for easy access.

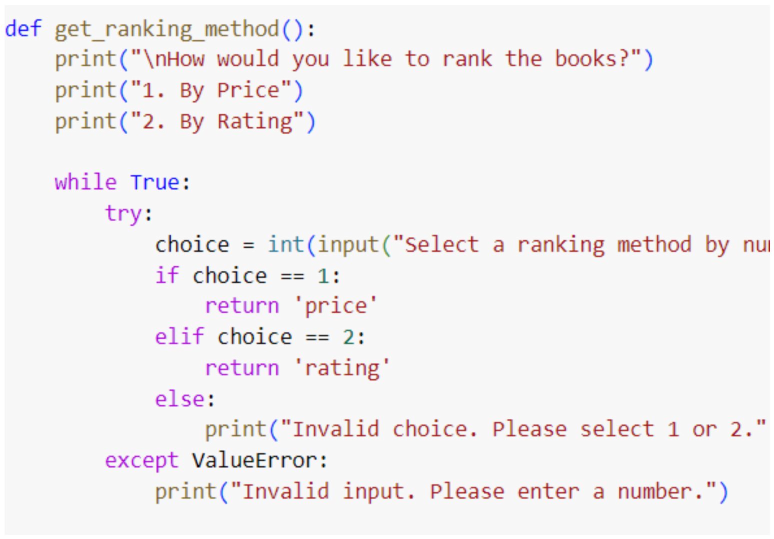


```python
books_by_category = {}
for book in books_data:
    category = book['Book_category']
    if category not in books_by_category:
        books_by_category[category] = []
    books_by_category[category].append(book)
```

Figure 2: code snippet of organizing book by category

3.3 User input for ranking method:

Users are allowed to choose how to rank the books, either by price or rating.

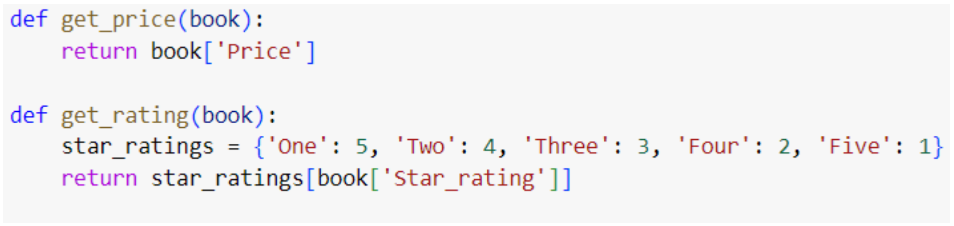


```python
def get_ranking_method():
    print("\nHow would you like to rank the books?")
    print("1. By Price")
    print("2. By Rating")

    while True:
        try:
            choice = int(input("Select a ranking method by nu
            if choice == 1:
                return 'price'
            elif choice == 2:
                return 'rating'
            else:
                print("Invalid choice. Please select 1 or 2."
        except ValueError:
            print("Invalid input. Please enter a number.")
```

Figure3: code snippet for ranking method

3.4 Key function for sorting:

This code snippet defines key functions to retrieve the price or rating of a book.

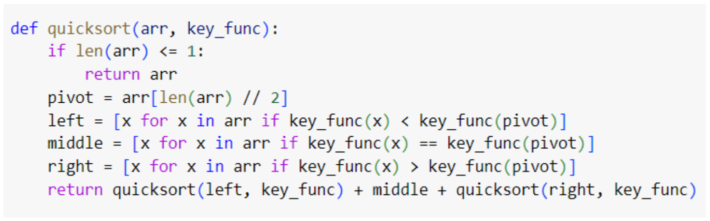


```python
def get_price(book):
    return book['Price']

def get_rating(book):
    star_ratings = {'One': 5, 'Two': 4, 'Three': 3, 'Four': 2, 'Five': 1}
    return star_ratings[book['Star_rating']]
```

Figure4: code *snippets for sorting*

3.5 Sorting using quick sort:

Quick sort is implemented to sort books based on the chosen keys.

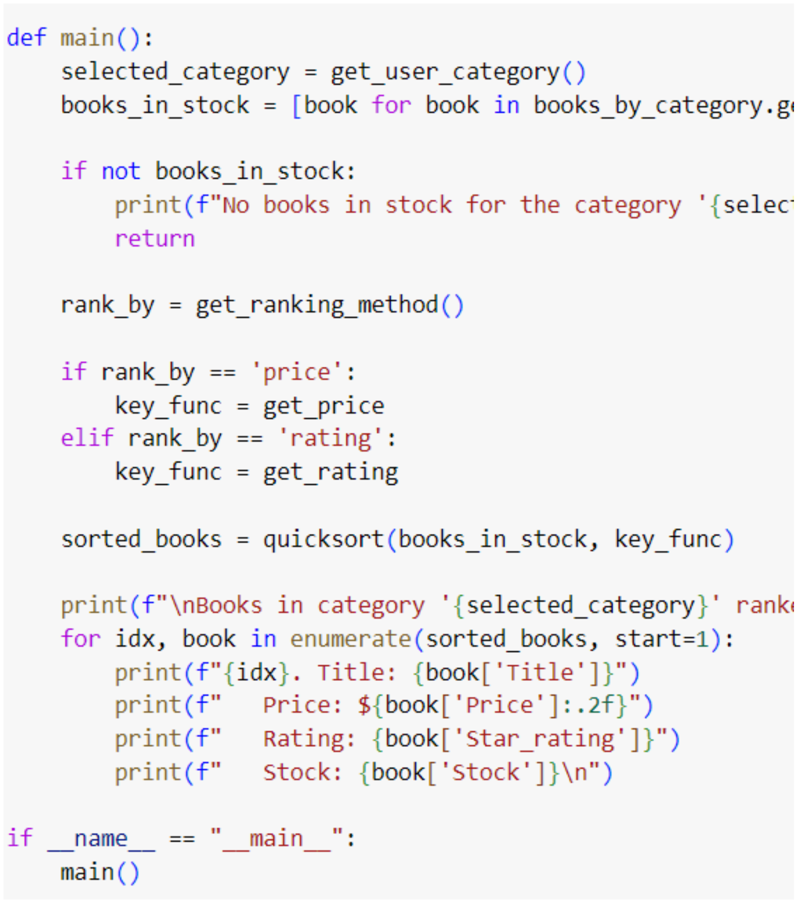


```python
def quicksort(arr, key_func):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if key_func(x) < key_func(pivot)]
    middle = [x for x in arr if key_func(x) == key_func(pivot)]
    right = [x for x in arr if key_func(x) > key_func(pivot)]
    return quicksort(left, key_func) + middle + quicksort(right, key_func)
```

Figure5: code snippet of using quick sort

3.6 Main Functions:

All the functions for filtering, sorting books and displaying sorted list are combined.



```
def main():
    selected_category = get_user_category()
    books_in_stock = [book for book in books_by_category.g

    if not books_in_stock:
        print(f"No books in stock for the category '{selec
        return

    rank_by = get_ranking_method()

    if rank_by == 'price':
        key_func = get_price
    elif rank_by == 'rating':
        key_func = get_rating

    sorted_books = quicksort(books_in_stock, key_func)

    print(f"\nBooks in category '{selected_category}' rank
    for idx, book in enumerate(sorted_books, start=1):
        print(f"{idx}. Title: {book['Title']}")
        print(f"   Price: ${book['Price']:.2f}")
        print(f"   Rating: {book['Star_rating']}")
        print(f"   Stock: {book['Stock']}\n")

if __name__ == "__main__":
    main()
```

Figure6: code snippet of main function

4.Performance Evaluation:

In this section, we will evaluate the performance of the implemented solution in terms of time complexity, space complexity, efficiency, scalability, and compare it with other sorting algorithms.

Time Complexity

Operation	Best Case	Average Case	Worst Case
Reading and Parsing CSV	O(N)	O(N)	O(N)
Categorizing Books	O(N)	O(N)	O(N)
Filtering Books in Stock	O(M)	O(M)	O(M)
Sorting Books with Quicksort	O(M log M)	O(M log M)	O(M.M)

Table1: Time complexity analysis

Where,

N = number of books in the dataset

M = number of books in selected categories

Asymptotic Analysis:

The overall time complexity is dominated by the steps that involve reading the CSV file, categorizing the books, and sorting them i.e.

The Best/average-case time complexity of whole solution is,

$$T(N, M) = O(n) + O(n) + O(m \log m) + O(m)$$

$$= O(m \log m)$$

Since $m \leq n$ (because m is the number of books in a specific category and n is the total number of books), the time complexity simplifies to O (n log n) in the average case.

Likewise for worst case time complexity of whole solution,

$$T(N, M) = O(n) + O(n) + O(m^2) + O(m)$$

$$= O(m^2)$$

Space Complexity

Operation	Best Case	Average Case	Worst Case
Reading and Parsing CSV	O(n)	O(n)	O(n)
Categorizing Books	O(k)	O(k)	O(k)

Filtering Books in Stock	O(m)	O(m)	O(m)
Sorting Books with Quicksort	O(m)	O(m)	O(m)

Table2: Space complexity analysis

Where,

N = number of books in the dataset

M = number of books in selected categories

K = number of categories

Thus, the total Space Complexity = O(n) + O(m) + O(k) = O (n + m + k)

Comparison with Other Sorting Algorithms

Sort Algorithm	Best Case	Worst Case	Space Complexity	Disadvantage in Original Code
Merge Sort	O(m log m)	O(m log m)	O(m) auxiliary	In the context of sorting books by category, uses extra space which may not be ideal, especially for large categories.
Heap Sort	O(m log m)	O(m log m)	O(1) auxiliary	Heap Sort has a slightly higher constant factor in time complexity and in-place sorting may not be advantageous for preserving original order.
Insertion Sort	O(m)	O(m.m)	O(1) auxiliary	Worst-case time complexity of O(m^2) makes it less efficient for larger datasets or categories with many books.

Table3: Comparison with other algorithms

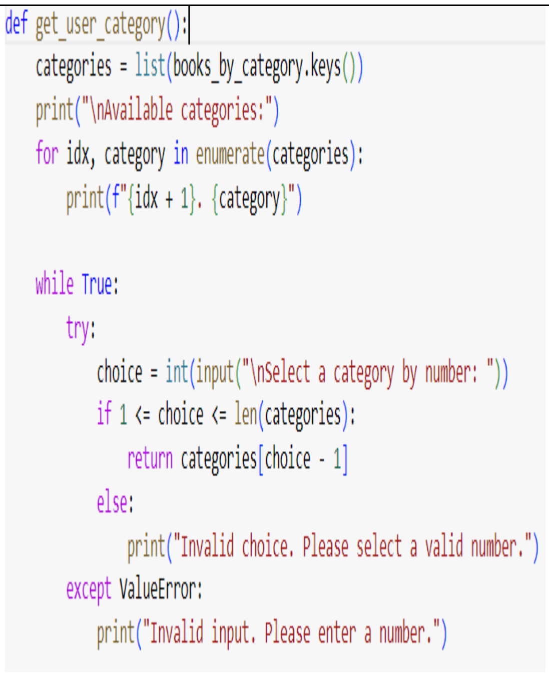
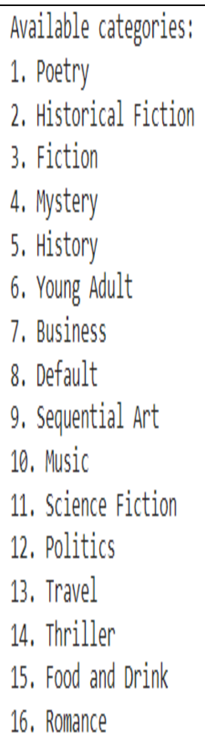
Which Algorithm is Better?

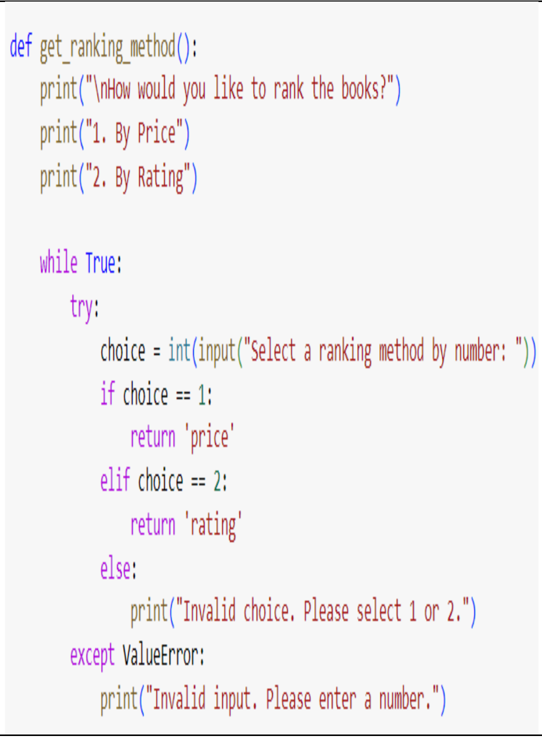
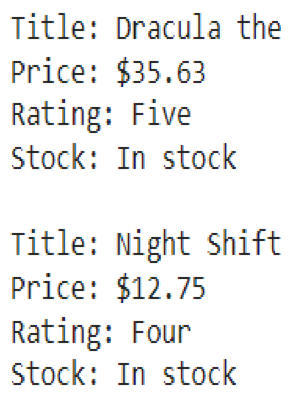
- **Merge Sort vs Quick Sort:** Merge Sort guarantees O(m log m) time complexity in the worst case, making it more predictable than Quick Sort. However, Merge Sort uses O(m) additional space, whereas Quick Sort can be more space-efficient (O(log m) with tail recursion optimization).
- **Heap Sort vs Quick Sort:** Heap Sort also offers O(m log m) time complexity and O(1) space complexity, making it efficient in terms of space. However, it tends to have higher constant factors compared to Quick Sort in practice.
- **Insertion Sort and Bubble Sort:** Both are generally inefficient for large datasets due to their O(m^2) average and worst-case time complexities.

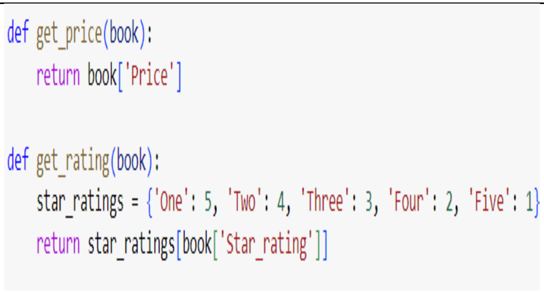
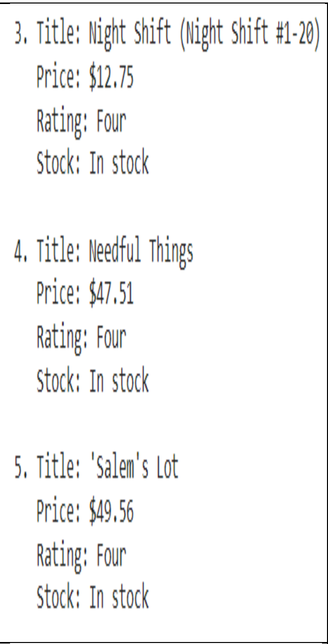
For our use case, Quick Sort is chosen due to its generally efficient performance and simplicity. However, if worst-case scenarios are a major concern, Merge Sort or Heap Sort could be considered for more predictable performance.

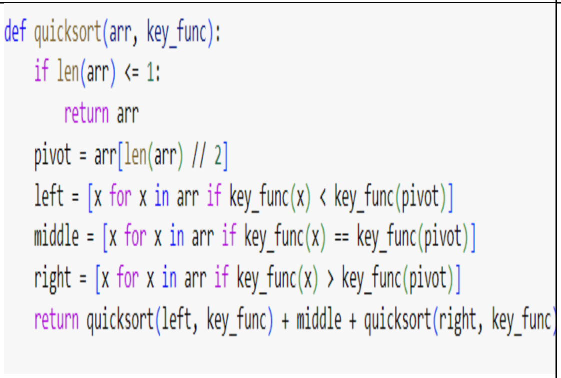
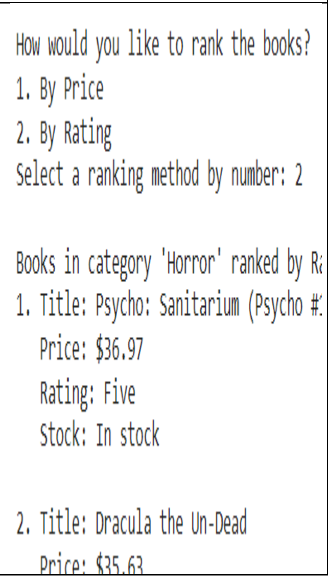
5.Assertion Table

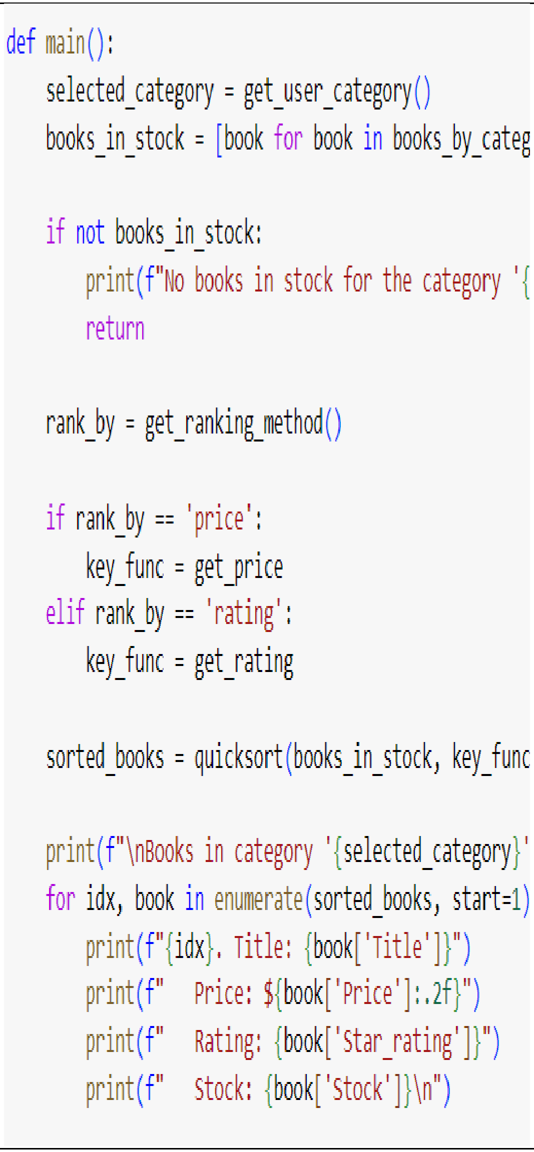
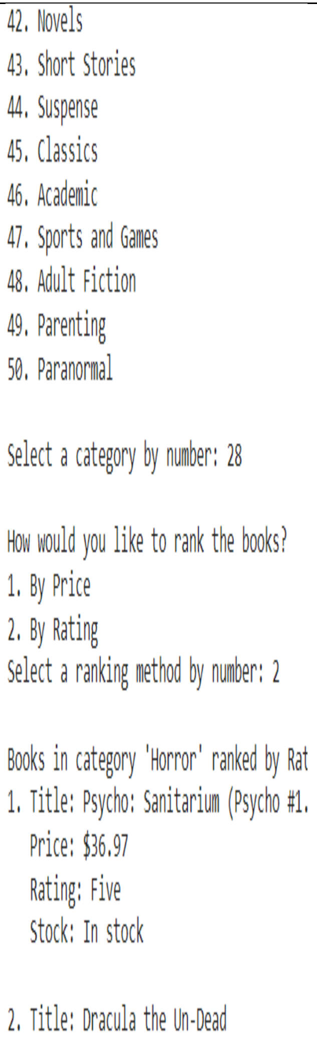
Assertion	Code snippets	Expected Output	Functionality

1. getting category of books	 `def get_user_category():`
	`    categories = list(books_by_category.keys())`
	`    print("\nAvailable categories:")`
	`    for idx, category in enumerate(categories):`
	`        print(f"{idx + 1}. {category}")`
	
	`    while True:`
	`        try:`
	`            choice = int(input("\nSelect a category by number: "))`
	`            if 1 <= choice <= len(categories):`
	`                return categories[choice - 1]`
	`            else:`
	`                print("Invalid choice. Please select a valid number.")`
	`        except ValueError:`
	`            print("Invalid input. Please enter a number.")`
		 `Available categories:`
		`1. Poetry`
		`2. Historical Fiction`
		`3. Fiction`
		`4. Mystery`
		`5. History`
		`6. Young Adult`
		`7. Business`
		`8. Default`
		`9. Sequential Art`
		`10. Music`
		`11. Science Fiction`
		`12. Politics`
		`13. Travel`
		`14. Thriller`
		`15. Food and Drink`
		`16. Romance`
			Working

2. Ranking books by price and rating	 `def get_ranking_method():`
	`    print("\nHow would you like to rank the books?")`
	`    print("1. By Price")`
	`    print("2. By Rating")`
	
	`    while True:`
	`        try:`
	`            choice = int(input("Select a ranking method by number: "))`
	`            if choice == 1:`
	`                return 'price'`
	`            elif choice == 2:`
	`                return 'rating'`
	`            else:`
	`                print("Invalid choice. Please select 1 or 2.")`
	`        except ValueError:`
	`            print("Invalid input. Please enter a number.")`
		 `Title: Dracula the`
		`Price: $35.63`
		`Rating: Five`
		`Stock: In stock`
		
		`Title: Night Shift`
		`Price: $12.75`
		`Rating: Four`
		`Stock: In stock`
			Working

3. getting price and ratings of book	
	`def get_price(book):`
	`    return book['Price']`
	
	`def get_rating(book):`
	`    star_ratings = {'One': 5, 'Two': 4, 'Three': 3, 'Four': 2, 'Five': 1}`
	`    return star_ratings[book['Star_rating']]`
		
		`3. Title: Night Shift (Night Shift #1-20)`
		`   Price: $12.75`
		`   Rating: Four`
		`   Stock: In stock`
		
		`4. Title: Needful Things`
		`   Price: $47.51`
		`   Rating: Four`
		`   Stock: In stock`
		
		`5. Title: 'Salem's Lot`
		`   Price: $49.56`
		`   Rating: Four`
		`   Stock: In stock`
			Working

4.using quicksort for ranking according to ratings or costs	
	`def quicksort(arr, key_func):`
	`    if len(arr) <= 1:`
	`        return arr`
	`    pivot = arr[len(arr) // 2]`
	`    left = [x for x in arr if key_func(x) < key_func(pivot)]`
	`    middle = [x for x in arr if key_func(x) == key_func(pivot)]`
	`    right = [x for x in arr if key_func(x) > key_func(pivot)]`
	`    return quicksort(left, key_func) + middle + quicksort(right, key_func`
		
		`How would you like to rank the books?`
		`1. By Price`
		`2. By Rating`
		`Select a ranking method by number: 2`
		
		`Books in category 'Horror' ranked by Ra`
		`1. Title: Psycho: Sanitarium (Psycho #`
		`   Price: $36.97`
		`   Rating: Five`
		`   Stock: In stock`
		
		`2. Title: Dracula the Un-Dead`
		`   Price: $35.63`
			Working

5.Running main	
	`def main():`
	`    selected_category = get_user_category()`
	`    books_in_stock = [book for book in books_by_categ`
	
	`    if not books_in_stock:`
	`        print(f"No books in stock for the category '{`
	`        return`
	
	`    rank_by = get_ranking_method()`
	
	`    if rank_by == 'price':`
	`        key_func = get_price`
	`    elif rank_by == 'rating':`
	`        key_func = get_rating`
	
	`    sorted_books = quicksort(books_in_stock, key_func`
	
	`    print(f"\nBooks in category '{selected_category}'`
	`    for idx, book in enumerate(sorted_books, start=1)`
	`        print(f"{idx}. Title: {book['Title']}")`
	`        print(f"   Price: ${book['Price']:.2f}")`
	`        print(f"   Rating: {book['Star_rating']}")`
	`        print(f"   Stock: {book['Stock']}\n")`
		
		`42. Novels`
		`43. Short Stories`
		`44. Suspense`
		`45. Classics`
		`46. Academic`
		`47. Sports and Games`
		`48. Adult Fiction`
		`49. Parenting`
		`50. Paranormal`
		
		`Select a category by number: 28`
		
		`How would you like to rank the books?`
		`1. By Price`
		`2. By Rating`
		`Select a ranking method by number: 2`
		
		`Books in category 'Horror' ranked by Rat`
		`1. Title: Psycho: Sanitarium (Psycho #1.`
		`   Price: $36.97`
		`   Rating: Five`
		`   Stock: In stock`
		
		`2. Title: Dracula the Un-Dead`
			Working

6.Conclusion:

The employed solution solves the need of filtering/ranking books by using basic data structures and algorithms given that a user selects the required criteria. The application of dictionaries for categorization as well as their assignment to Quick Sort for sorting ensures efficiency and a more fulfilling response to users' requirements. From the above evaluation, Quick Sort is seen as having overall good efficiency and scalability in this application. Nevertheless, other sorting methods that could be used include the Merge Sort or the Heap sort if there is any use in being certain of the time that would take in the worst case.

References

Wang, Z.H. and Hou, D.Z. (2021) ‘Research on book recommendation algorithm based on collaborative filtering and interest degree’, *Wireless Communications and Mobile Computing*, 2021, pp. 1–7. doi:10.1155/2021/7036357.

Appendix A: Code of the system

```python
import csv
books_data = []
csv_file_path = '/content/books_scraped.csv'
with open(csv_file_path, mode='r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        row['Price'] = float(row['Price'])
        books_data.append(row)
books_by_category = {}
for book in books_data:
    category = book['Book_category']
    if category not in books_by_category:
        books_by_category[category] = []
    books_by_category[category].append(book)
def get_user_category():
    categories = list(books_by_category.keys())
    print("\nAvailable categories:")
    for idx, category in enumerate(categories):
        print(f"{idx + 1}. {category}")
    while True:
        try:
            choice = int(input("\nSelect a category by number: "))
            if 1 <= choice <= len(categories):
                return categories[choice - 1]
            else:
                print("Invalid choice. Please select a valid number.")
        except ValueError:
            print("Invalid input. Please enter a number.")
def get_ranking_method():
    print("\nHow would you like to rank the books?")
    print("1. By Price")
    print("2. By Rating")
    while True:
        try:
            choice = int(input("Select a ranking method by number: "))
            if choice == 1:
                return 'price'
            elif choice == 2:
                return 'rating'
            else:
```

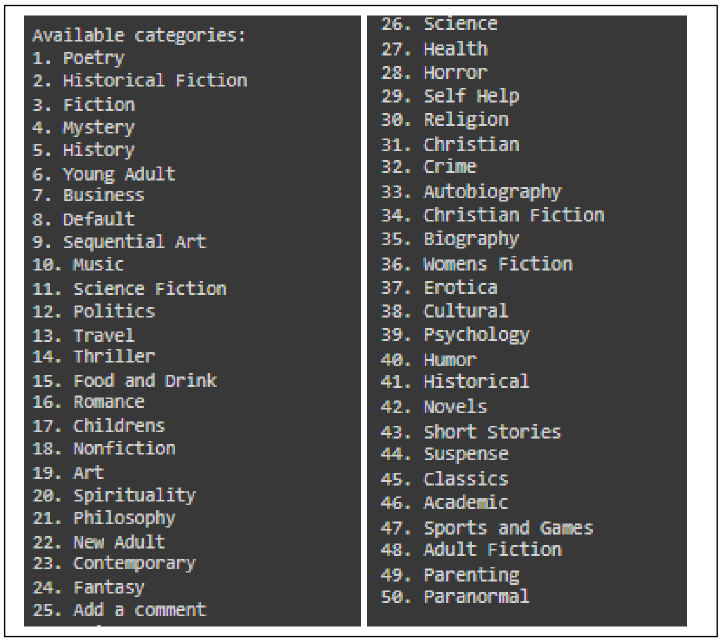
```python
def get_price(book):
    return book['Price']

def get_rating(book):
    star_ratings = {'One': 5, 'Two': 4, 'Three': 3, 'Four': 2, 'Five': 1}
    return star_ratings[book['Star_rating']]

def quicksort(arr, key_func):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if key_func(x) < key_func(pivot)]
    middle = [x for x in arr if key_func(x) == key_func(pivot)]
    right = [x for x in arr if key_func(x) > key_func(pivot)]
    return quicksort(left, key_func) + middle + quicksort(right, key_func)

def main():
    selected_category = get_user_category()
    books_in_stock = [book for book in books_by_category.get(selected_category, []) if book['Stock'] == 'In stock']

    if not books_in_stock:
        print(f"No books in stock for the category '{selected_category}'.")
        return

    rank_by = get_ranking_method()

    if rank_by == 'price':
        key_func = get_price
    elif rank_by == 'rating':
        key_func = get_rating

    sorted_books = quicksort(books_in_stock, key_func)

    print(f"\nBooks in category '{selected_category}' ranked by {rank_by.capitalize()}:")
    for idx, book in enumerate(sorted_books, start=1):
        print(f"{idx}. Title: {book['Title']}")
        print(f"   Price: ${book['Price']:.2f}")
        print(f"   Rating: {book['Star_rating']}")
        print(f"   Stock: {book['Stock']}\n")

if __name__ == "__main__":
    main()
```

Appendix B: Output and Data Sample

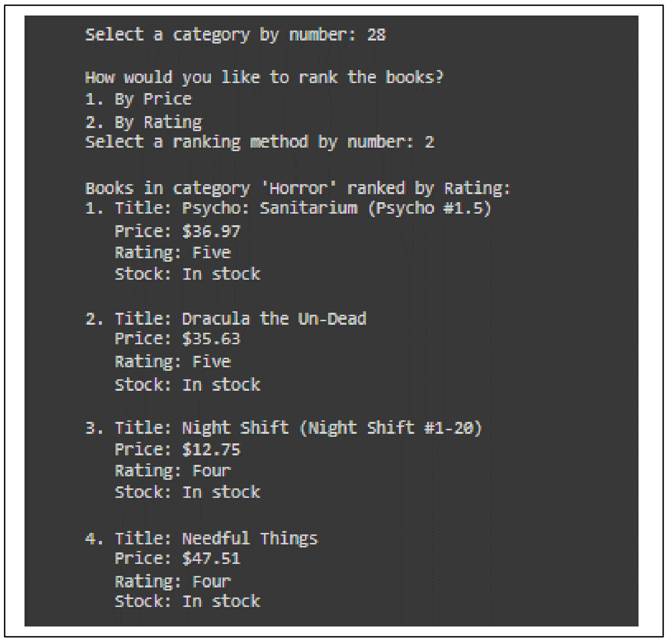
The CSV file (books_scraped.csv) should contain the following columns:

- Title: Title of the book
- Book_category: Genre or category of the book
- Star_rating: Rating of the book (One, Two, Three, Four, Five)
- Price: Price of the book in dollars
- Stock: Stock status (e.g., 'In stock')

Output:



```
Available categories:
1. Poetry
2. Historical Fiction
3. Fiction
4. Mystery
5. History
6. Young Adult
7. Business
8. Default
9. Sequential Art
10. Music
11. Science Fiction
12. Politics
13. Travel
14. Thriller
15. Food and Drink
16. Romance
17. Childrens
18. Nonfiction
19. Art
20. Spirituality
21. Philosophy
22. New Adult
23. Contemporary
24. Fantasy
25. Add a comment
26. Science
27. Health
28. Horror
29. Self Help
30. Religion
31. Christian
32. Crime
33. Autobiography
34. Christian Fiction
35. Biography
36. Womens Fiction
37. Erotica
38. Cultural
39. Psychology
40. Humor
41. Historical
42. Novels
43. Short Stories
44. Suspense
45. Classics
46. Academic
47. Sports and Games
48. Adult Fiction
49. Parenting
50. Paranormal
```



```
Select a category by number: 28

How would you like to rank the books?
1. By Price
2. By Rating
Select a ranking method by number: 2

Books in category 'Horror' ranked by Rating:
1. Title: Psycho: Sanitarium (Psycho #1.5)
   Price: $36.97
   Rating: Five
   Stock: In stock

2. Title: Dracula the Un-Dead
   Price: $35.63
   Rating: Five
   Stock: In stock

3. Title: Night Shift (Night Shift #1-20)
   Price: $12.75
   Rating: Four
   Stock: In stock

4. Title: Needful Things
   Price: $47.51
   Rating: Four
   Stock: In stock
```