

|  |  |  |  |
| --- | --- | --- | --- |
| Course Code | : | CST207 | |
| Course Name | : | Design and Analysis of Algorithms | |
| Lecturer | : | Najla’a Ateeq Mohammed Draib | |
| Academic Session | : | 2024/04 | |
| Assessment Title | : | Group Project – CST207 | |
| Answered by |  |  | |
| : | Student ID | Student Name |
|  | AIT2209934 | Ang Yi Xun |
|  |  | AIT2209947 | Poh Jia Sheng |
|  |  | AIT2209935 | Choo Qie Sheng |
|  |  | AIT2209874 | Khoo Song Hao |
|  |  | AIT2209097 | Zhang Yun |
|  |  | |  | | --- | | **Total Mark** | |  | | |
| Feedback from Lecturer: | | | |

**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:

|  |  |  |
| --- | --- | --- |
| Student ID | Name | Signature |
| AIT2209934 | Ang Yi Xun | yixun |
| AIT2209947 | Poh Jia Sheng | poh |
| AIT2209935 | Choo Qie Sheng | choo |
| AIT2209874 | Khoo Song Hao | songhao |
| AIT2209097 | Zhang Yun | zhangyun |

Date:7/7/2024

**Search algorithms implemented**

Linear Search

Linear search appears as a simple search method that often referred to as sequential search. It starts the search by visiting all the elements one by one from the beginning of a collection of elements. The search will go through the whole collection until the desired value is found.

Figure 1.1 is a visualization of how Linear Search functions with the provided array and target value:

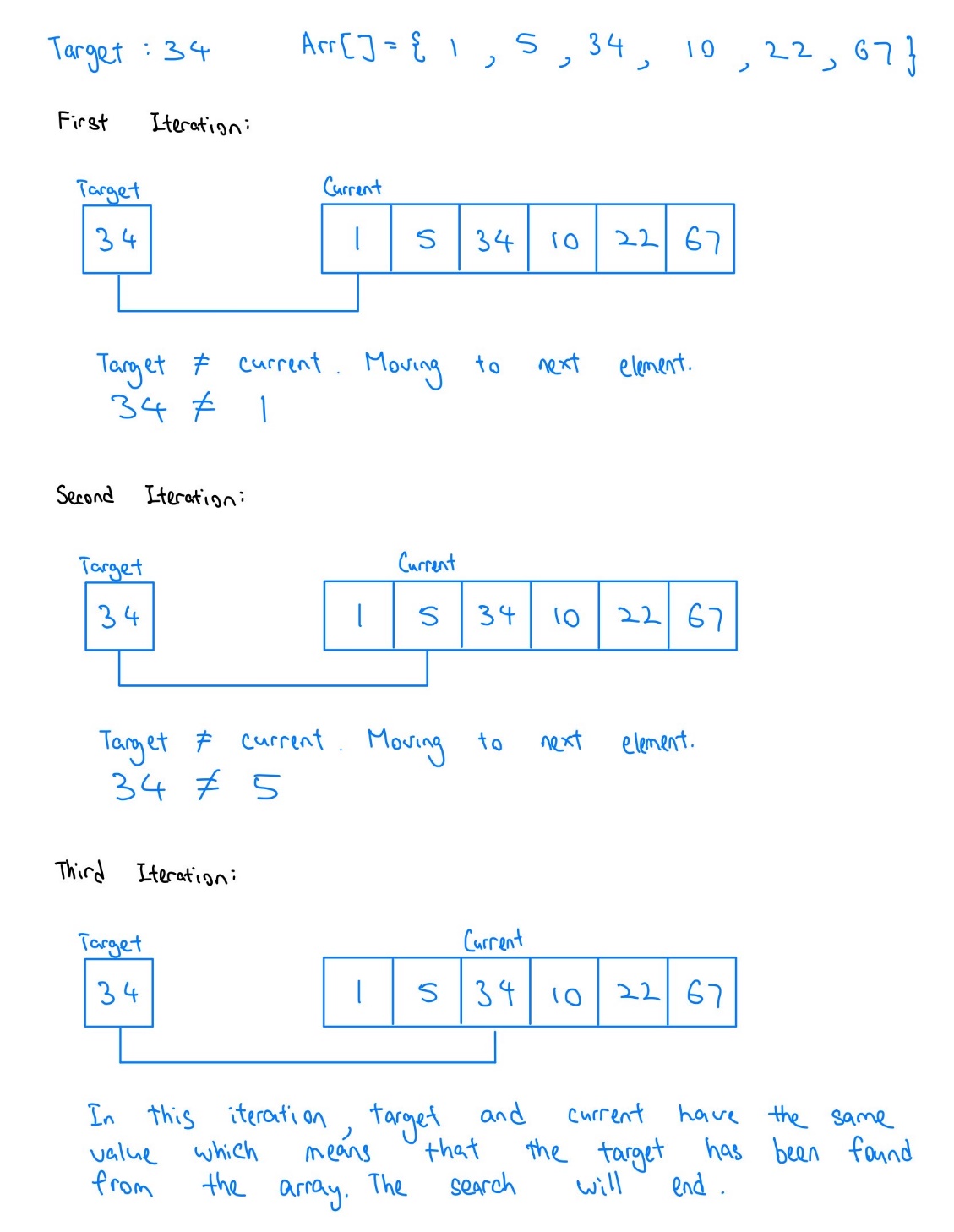
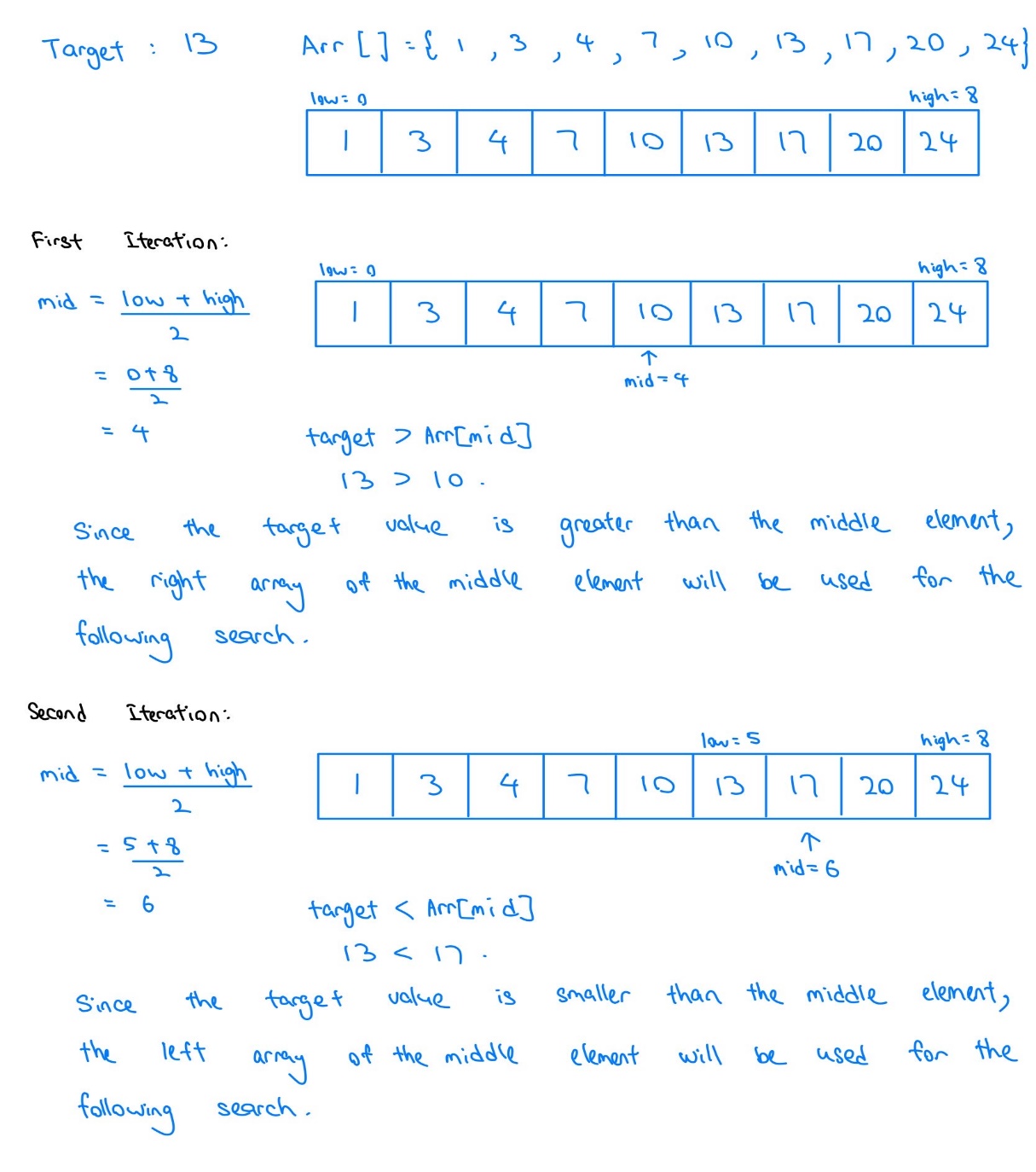


Figure 1.1 – Example Linear Search

Binary Search

Binary Search algorithm looks for a certain value in a sorted array. It will divide the search array in half to find the middle index (GeeksforGeeks, 2024). It will then compare this value with the target value if they have the same value means that the target is found and the process will terminate. If not, the algorithm will decide between the divided arrays to continue the search. The left array will be used for the next search if the target has a smaller value than the middle element, and vice versa. The search will continue to work until the target value is found.

Figure 1.2 is a visualization of how Binary Search functions with the provided sorted array and target value:

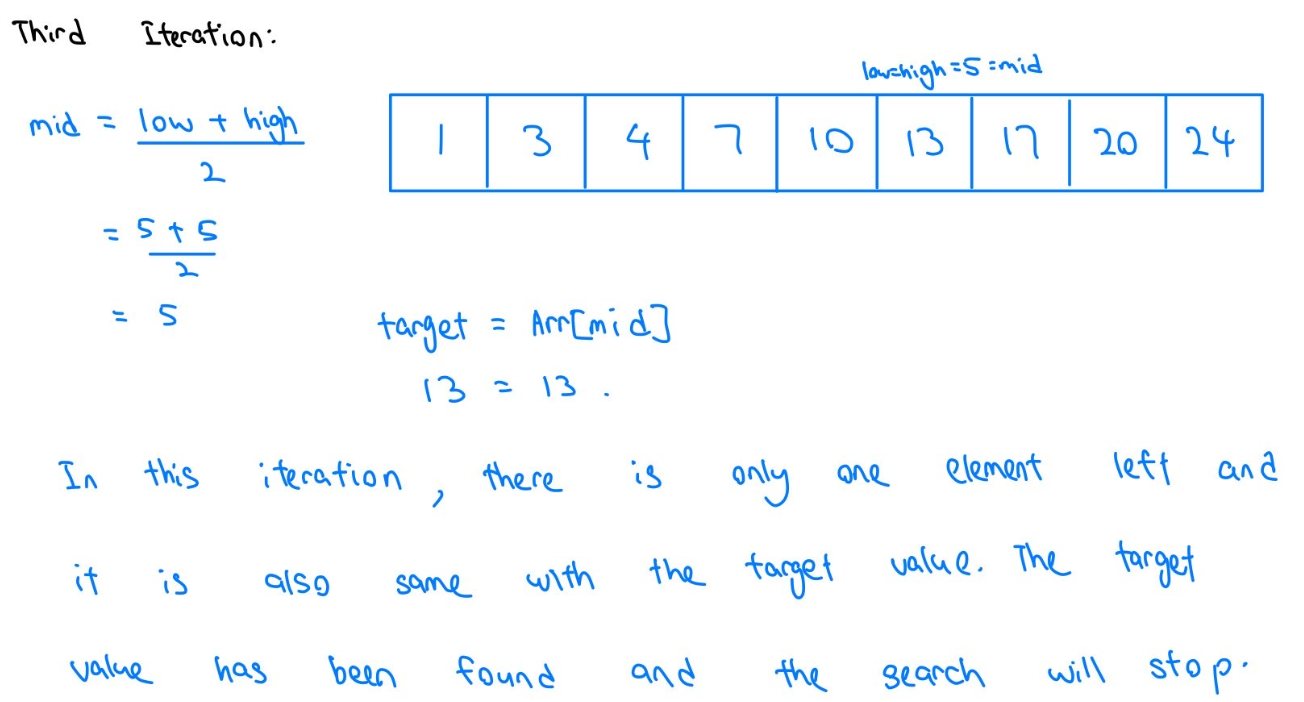


Figure 1.2 – Binary Search

**Sort algorithms implemented**

Merge Sort

Merge sort is a divide and conquer method that works by iteratively dividing the array into smaller subarrays until each subarray has only 1 element. The sub-arrays are then combined back together in sorted order.

Figure 1.3 is a visualization of how Merge Sort functions with the provided array:

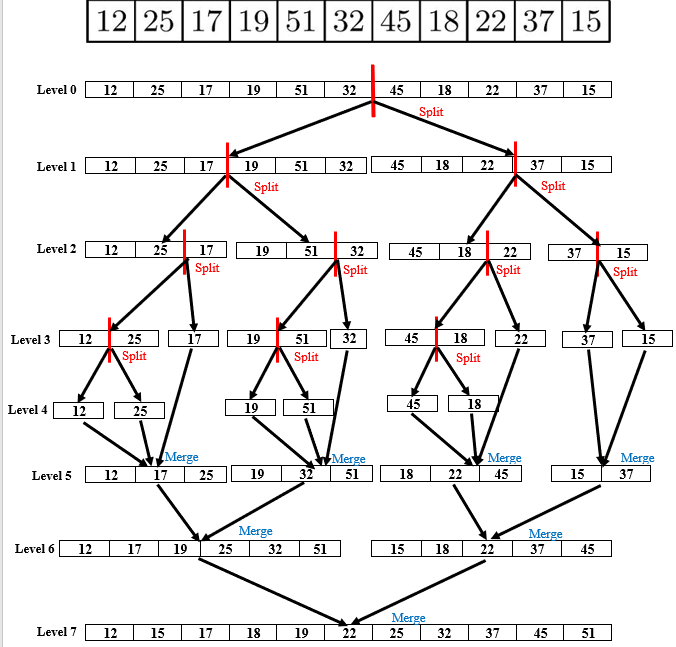


Figure 1.3 – Merge Sort

At Level 0, an array of 11 elements is split in half using the middle of the array found by applying the following formula:

Following the formula, we found that the middle lies at a [5], then we split the array into two halves:

Left half: [12, 25, 17, 19, 51, 32]

Right half: [45, 18, 22, 37, 15]

Subsequently, every array is split in half recursively until it reaches just one element.

At Level 4, every element has been divided into a single element. Then at level 5, they are all merged together in a sorted order. Three rounds of merging will occur until level 7, at which point each element in the array is sorted.

Quicksort

Quicksort is also a divide-and-conquer algorithm by dividing the array into sections centered on the pivot element. Multiple varieties of quick sort apply different algorithms to identify the pivot.

* The first element as pivot
* The last element as pivot
* A random element is chosen as pivot
* The median is chosen as pivot

Figure 1.4 is a visualization of how Quicksort performs by selecting the first element as a pivot:

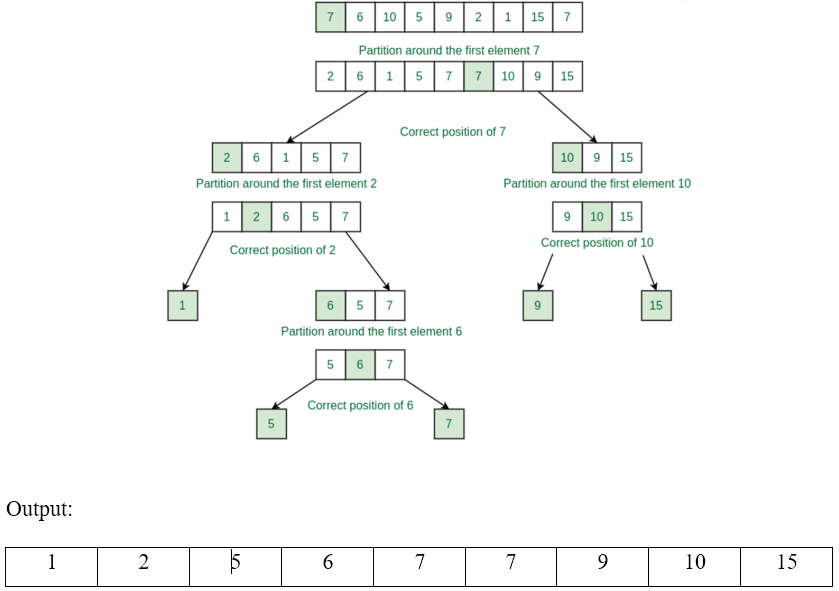


Figure 1.4 – Quick Sort

A partition serves as a Quicksort’s main function. If the array is sorted, partitioning aims to place the pivot in the correct place; place elements that have a smaller value or equal to the pivot to the left of the array, place elements that have a greater value than the pivot to the right array and accomplish this all-in linear time.

The example above shows that the first element is selected as the pivot. After that, each element is compared to the pivot; if it is smaller than the pivot, it is positioned to the left of the array’s partition, and if it is greater than the pivot, it is positioned to the right array’s partition. The process will end until every element in the array has been sorted (GeeksforGeeks, 2024).

Bubble Sort

Bubbles sort compares the current elements to the nearby elements in the array. In the first iteration, it will compare the first element of the array to the next element. If the initial element is greater than the adjacent elements, it will swap their position. This process will keep repeating for each pair of adjacent entries until it reaches the end of the list. After first iteration, the largest element will be move to the end which is the correct position.

In the second iteration, the last element will not be compared since it is in the correct position. This process will be repeated until no more swaps are needed which means it is sorted (GeeksforGeeks, 2024).

Below is a visualization of how Bubble Sort functions with the provided array:

The unsorted array:

[45, 21, 48, 67, 32]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 45 | 21 | 48 | 67 | 32 |
| Index | 0 | 1 | 2 | 3 | 4 |

In 1st iteration, it will start from index 0 and compare to its adjacent index. If index 𝑖 is greater than 𝑖+1 then it will swap. Since 45 is greater than 21 then it will swap.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 21 | 45 | 48 | 67 | 32 |
| Index | 0 | 1 | 2 | 3 | 4 |

Now it will compare between 45 and 48. Since 48 is greater, it will not carry out the swapping operation. Then it will continue to compare between 48 and 67. Since 67 is still greater than 48 then it will move to the next index to do the comparison. Now it will compare between 67 and 32. 67 is greater than 32 then it will swap.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 21 | 45 | 48 | 32 | 67 |
| Index | 0 | 1 | 2 | 3 | 4 |

In the second iteration, the last elements in the array will not be compared because it was already in the correct position. It will now start from index 0 again and compare. The swapping process in the second iteration is on the comparison between 48 and 32.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 21 | 45 | 32 | 48 | 67 |
| Index | 0 | 1 | 2 | 3 | 4 |

48 and 67 are in the right position so both will not be sorted again. The swapping process in the third iteration is between 45 and 32.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 21 | 32 | 45 | 48 | 67 |
| Index | 0 | 1 | 2 | 3 | 4 |

Lastly, our array is in the sorted form where the swapping process in that iteration is not carried out. Therefore, it comes out with the final sorted result.

[21, 32, 45, 48, 67]

**Best search algorithms implemented**

Between these two search methods, I think Binary Search is better than Linear Search. There are several reasons for me to prove that Binary Search is better.

First of all, in terms of time complexity, Binary Search achieves time complexity of O() and Linear Search has a time complexity of O(n). We can see that this is quite obvious that Binary Search is faster than Linear Search, especially for the huge datasets. It can save a lot of time and be quite effective. Whereas Linear Search requests that every element be checked, Binary Search remains a somewhat steady efficiency even in the worst of circumstances.

In terms of pre-processing and indexing, Binary Search can be used in conjunction with pre-processing and indexing procedures to further improve search efficiency. For instance, multi-level index structures such as Binary trees can be utilized to increase search performance on large-scale data sets. However, it is difficult for Linear Search to achieve significant performance improvements through pre-processing or indexing in unordered data.

Therefore, I believe that Binary Search is the best search algorithm due to its effectiveness.

**Best sorting algorithms implemented**

In my opinion, Merge sort is the best sorting algorithm among all the sorting algorithms we’ve created. This is because it applies the concept of divide and conquer by recursively dividing an array into half and then sorting the divided array. After that, it will combine all the sorted arrays.

Besides, I choose merge sort as the best sorting algorithm as it is more stable due to its ability to preserve the relative order of records with equal keys. Moreover, its performance does not degrade with specific input patterns. Therefore, it is more suitable for handling large datasets because of its consistent time complexity of O (n log n) for all cases.

Moving on, merge sort is inherently easier to parallelize. Since each half array is sorted independently, it is an excellent candidate for parallelize processing. This can be beneficial in dealing with large files that do not fit entirely into memory. In my opinion, it can sort various data that come from different files which is a common requirement in external sorting scenarios.

I believe that merge sort is a valuable tool for many practical applications due to its robustness, reliability and consistent performance despite the additional space required by the temporary arrays. Its stability, consistency and ability to be parallelized enhance its effectiveness in various scenarios. That’s all the reasons why I think that merge sort is a much superior sorting algorithm compared to the other two sorting algorithms.

**Time & Space Complexity of Search Algorithms**

Linear Search

Best case: It happens when the target value is the first element in the datasets. Therefore, there will only one comparison is required, andthe time complexity is *O (1)*.

Worst case: The target value is at the last position in the collections of datasets or doesn’t exist in the dataset, hence all elements need to be checked. The time complexity is *O (n)*.

Average case: Assuming that the target element is equally likely to be at any position in the dataset, on average elements need to be checked. Therefore, the average time complexity is *O (n)*.

A linear search has a very small space complexity since it is only used for the variable to go through the whole list (GeeksforGeeks, 2024). So, the space complexity of the Linear Search is *O (1)*. This indicates that the size of the input dataset does not affect the space complexity of the Linear Search.

Binary Search

Best case: When the target is in the middle of the sorted array, the time complexity in the best case is O(1) which means that it can find the target value by doing one comparison.

In average case,

Initial range: n- the size of datasets***.***

First search: The dataset is divided into two parts, each of which is  ***.***

Second search: Divided into two parts again, and the search range becomes  **.**

And so on: After 𝑘 divisions, the search range’s size is **.** When the search range is narrowed down to only one element, that is = 1, we get*k =*from the equation**.**

Therefore, the time complexity of binary search is *O (log n****)***.

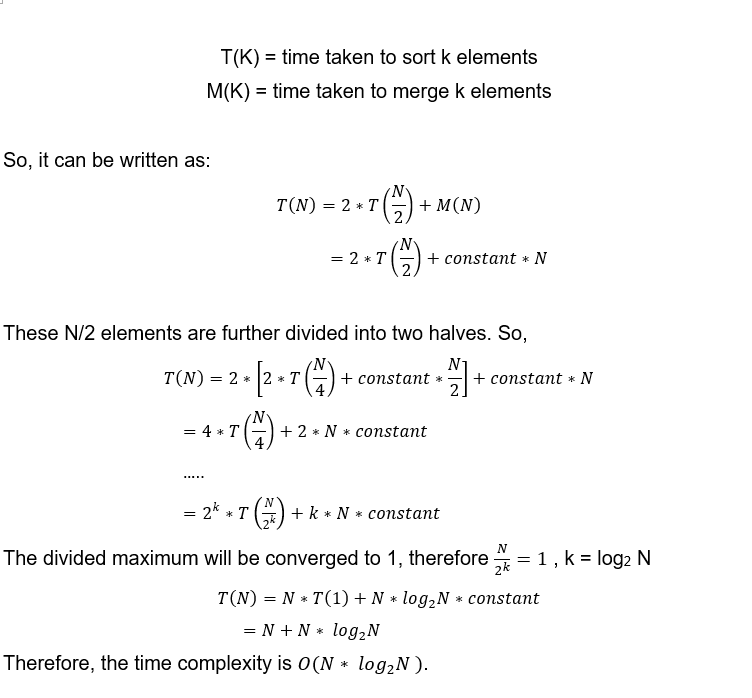
Worst case: The worst case happens when the target is the first element of the sorted array (GeeksforGeeks, 2024). The time complexity is *O (log n****).***

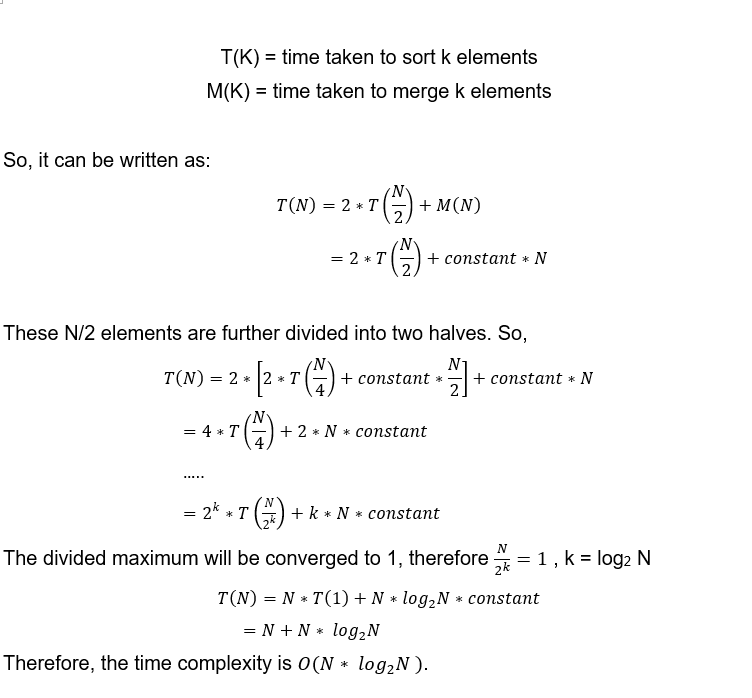
Binary Search only uses several additional variables to store indexes and bounds such as low, high and mid, and not require additional space other than the dataset itself (GeeksforGeeks, 2024). The size of the input array didn’t affect that. Therefore, the space Complexity of Binary Search is O*(1).*

**Time & Space Complexity of Sorting Algorithms**

Merge Sort

The Merge Sort achieves time complexity of in both average and worst cases. In addition, it’s space complexity of . Let’s prove it.

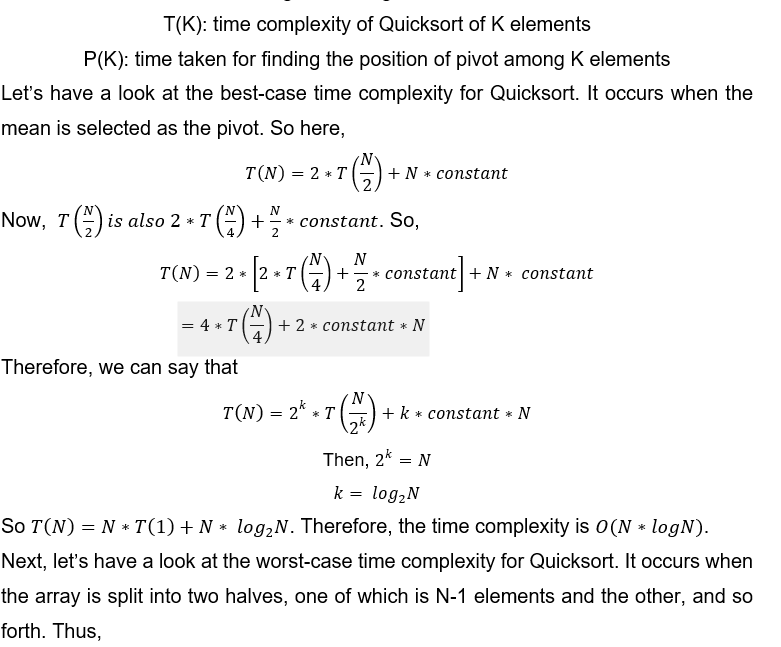


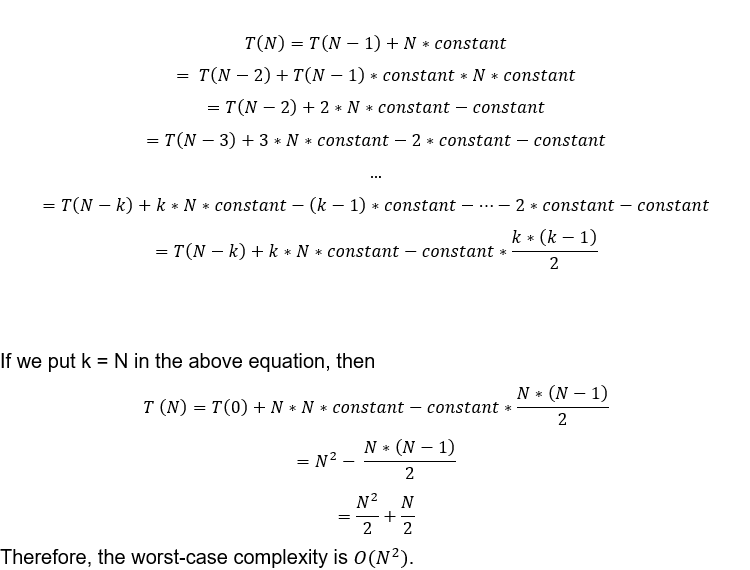


The space complexity of Merge Sort is O(n). This is because it combines the sorted halves of the input array into an auxiliary array of size n. The combined result is saved in the auxiliary array and the sorted result is reprinted in the input array (GeeksforGeeks, 2024).

Quicksort

The Quicksort has the time complexity of on the average case and best case, but when it comes to the worst case, the time complexity will be . For the space complexity, Quicksort has for the best case and average case, but it becomes due to unbalanced partitioning. Let’s prove it.





The best case for space complexity for Quicksort is , due to balanced partitioning, which produces a balanced recursion tree with an call stack. Furthermore, for the worst case for space complexity for Quicksort is , due to uneven partitioning that produced a skewed recursion tree and an call stack (GeeksforGeeks, 2024).

Bubble Sort

|  |  |
| --- | --- |
| Best case |  |
| Worst case |  |
| Average case |  |

The best case is when the input is sorted. So, it will only pass through the array only once. The worst case scenario occurs when the input array is sorted in decreasing order. So, it will go through the array and make n comparison for each element. This results in quadratic time complexity (Alake, 2023).

Space complexity is O(1) , which requires just a constant amount of extra space for the switching procedure. Bubble sorts require minimal space requirements where it only uses a single additional space for swapping the elements. This will benefit those systems that have very limited memory.

**Additional search algorithm and comparison**

Jump Search (VIDHVAN, 2021)

Jump Search is an improved linear search algorithm that can function within sorted arrays. This search has combined the ideas of Linear Search and Binary Search, quickly narrowing the search range by jumping fixed-size blocks and then performing a Linear Search within the found blocks. The performance of the Jump search is better than the Linear Search but worse than the Binary Search (GeeksforGeeks, 2024).

Below is a visualization of how Jump Sort functions with the provided sorted array and the target value,

Arr[] = [ 1, 3, 4, 5, 7, 8, 12, 14, 16, 18, 21, 24, 27, 35, 40, 44, 48]

target=18

Step 1: The block size m is usually close to be , where size of the array is n. In this example, the array size is 17, so the block size is

m = = = 4.1231 ≈ 4

Step 2**:** This step is also the jumping phase where it will start from the first element of the array, compare the target value with the first element, jump 4 elements at a time and compare the element in the jump position with the target value.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 8 | 12 | 14 | 16 | 18 | 21 | 24 | 27 | 35 | 40 | 44 | 48 |

Arr[0] = 1 < 18. Since it is not a match, then jump to the next block to do the following comparison.

First jump: Jump to Arr[4]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 8 | 12 | 14 | 16 | 18 | 21 | 24 | 27 | 35 | 40 | 44 | 48 |

Arr[4] = 7 < 18. Since it is not a match, then jump to the next block to do the following comparison.

Second jump: Jump to Arr[8]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 8 | 12 | 14 | 16 | 18 | 21 | 24 | 27 | 35 | 40 | 44 | 48 |

Arr[8] = 16 < 18. Since it is not a match, then jump to the next block to do the following comparison.

Third jump: Jump to Arr[12]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 8 | 12 | 14 | 16 | 18 | 21 | 24 | 27 | 35 | 40 | 44 | 48 |

Arr[12] = 27 > 18. However, the next jump position has a larger value than the target value. Therefore, this jump will not be taken and the Linear Search will be applied on the current block.

Step 3**:** In this step, the Linear Search is applied to compare the current block with the target value.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 4 | 5 | 7 | 8 | 12 | 14 | 16 | 18 | 21 | 24 | 27 | 35 | 40 | 44 | 48 |

After that, the pointer increments from 8th to 9th. Arr[9] = 18 where it is equal to the target. Returns the 9th index as the output and the search stops.

Comparison of Linear Search, Binary Search and Jump Search

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linear Search | Binary Search | Jump Search |
| Algorithm type | Sequential | Divide and conquer | Block Search |
| Time complexity | Best Case:  Average Case:  Worst Case: | Best Case:  Average Case:  Worst Case: | Best Case:  Average Case:  Worst Case: |
| Sorted Array needed | No | Yes | Yes |
| Search Approach | Compare one by one | Divide the search range repeatedly | Jump few blocks, then Linear Search |
| Datasets suitable | Small datasets | Large and sorted datasets | Large and sorted datasets |

Each of them has their advantages. The search approach of Linear Search makes it inefficient when dealing with large datasets, however, it is very easy to implement. Binary Search and Jump Search have better performance when handling large datasets but they can only work with sorted arrays.

Interpolation Search (G Bhat, 2020)

Interpolation Search is an improved search algorithm that is suitable for uniformly distributed sorted arrays. This search combines the idea of Binary Search and linear interpolation. Unlike Binary Search, Interpolation Search uses interpolation formulas to estimate the target element’s position, thereby quickly narrowing the search range.

Below is a visualization of how Interpolation functions with the provided sorted array,

Arr[] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] target=80

Initialization:

= 0: The first index of the array.

= 9: The last index of the array.

= 80: The target value we want to find.

First Iteration:

Step 1: Calculate the estimated position

Use the interpolation formula to estimate the position of target:

Substitute the initialized values ​​into the formula:

= 0 + = 0 + 7 = 7

The estimated position is 7.

Step2: Compare and adjust

Compare Arr[7] and target value:

Arr[7] = 80, target = 80

Arr[7] is equal to the target, we found the target value, return index 7 and the algorithm exit.

Now, one more example that the target value does not exist in the sorted array, hence, we can further adjust the search range.

Second example

Arr[] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100] target=65

Initialization:

= 0: The first index of the array.

= 9: The last index of the array.

= 65: The target value we want to find.

First Iteration:

Step1: Calculate the estimated position

= 0 + = 0 + 5.5 ≈ 6

The estimated position is 6.

Step2: Compare and adjust

Arr[6] = 70, target = 65

Since Arr[6] > target, adjust the search range:

= - 1 = 6 - 1 = 5

Second Iteration:

Step1: Calculate the estimated position

= 0 + = 0 + 5.5 ≈ 5

The estimated position is 5.

Step2:Compare and adjust

Arr[5] = 60, target = 65

Since Arr[5] < target, adjust the search range:

= + 1 = 5 + 1 = 6

At this time, the position of > , the search ends and the target value 65 is not in this array.

Comparison of Linear Search, Binary Search and Interpolation Search

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linear Search | Binary Search | Interpolation Search |
| Algorithm type | Sequential | Divide and conquer | Based on the position |
| Time complexity | Best Case:  Average Case:  Worst Case: | Best Case:  Average Case:  Worst Case: | Best Case:  Average Case:  Worst Case: |
| Sorted Array needed | No | Yes | Yes |
| Search Approach | Compare one by one | Divide the search range repeatedly | Estimate the position |
| Datasets suitable | Small datasets | Large and sorted datasets | Large and uniformly distributed sorted datasets |

Interpolation Search is a search algorithm that has high requirements on the datasets. It is more efficient than Binary Search when searching the uniformly distributed datasets (GeeksforGeeks, 2023). However, it performs poorly which similar to linear search when dealing with non-uniformly distributed data.

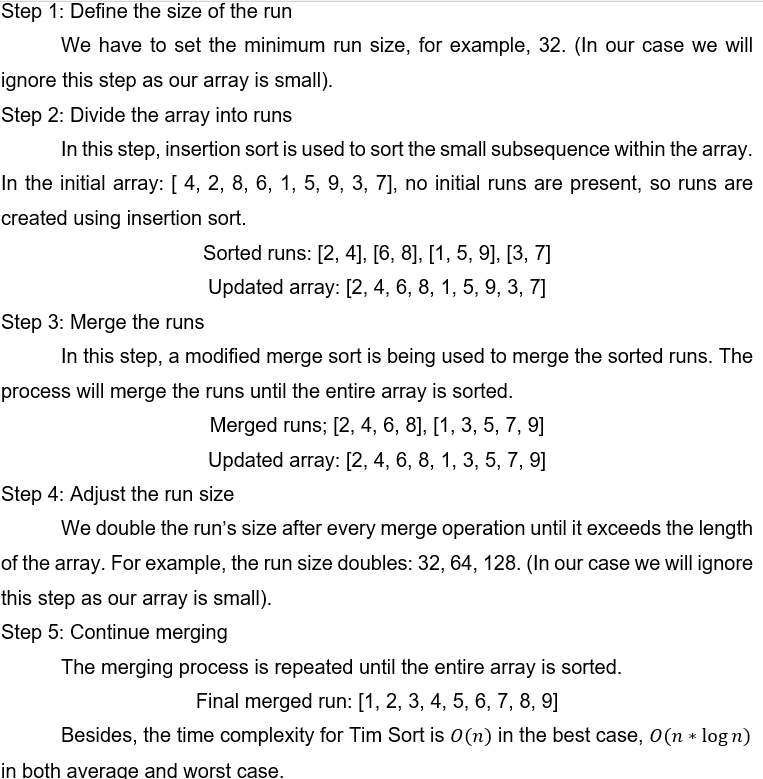
**Additional sorting algorithm and comparison**

Tim Sort (GeeksforGeeks, 2023)

Tim Sort combines the design of Merge Sort with Insertion Sort. It was designed to work well with a wide range of real-world data formats. The major purpose of it is to minimize the amount of comparisons and swaps by uing the data current order. This is performed by first separating the array into small and pre-sorted subarrays called runs. Then it will then combine using modified merge sort algorithms (GeeksforGeeks, 2023).

Below is a visualization of how Tim Sort functions with the provided array,

arr[ ] = {4, 2, 8, 6, 1, 5, 9, 3, 7}:



Comparison of Tim Sort, Merge Sort and Quicksort

|  |  |  |  |
| --- | --- | --- | --- |
|  | Tim Sort | Merge Sort | Quicksort |
| Algorithm type | Hybrid (Combination of Merge Sort and Insertion Sort) | Divide and conquer | Divide and conquer |
| Time complexity | Best Case:  Average Case:  Worst Case: | Best Case:  Average Case:  Worst Case: | Best Case:  Average Case:  Worst Case: |
| Space complexity |  |  |  |
| Stability | Stable | Stable | Not stable |
| Typical use cases | Utilized in standard libraries like Python and Java for sorting and in practice for real-world data. It also works well with large datasets that have a certain degree of order. | Utilized in situations where space complexity is unimportant and stable sorting is necessary. | Frequently used in general-purpose sorting because of its in-place sorting and average-case efficiency. Utilized frequently in embedded systems and system programming. |

Heap Sort (GeeksforGeeks, 2024)

The Heap Sort algorithm is a comparison-based sorting methos based on the binary heap. It is an upgraded version of the selection sort that makes use of the heap data structure to efficiently identify the maximum or minimum element (GeeksforGeeks, 2024).

There are a few steps in the heap sort algorithm. For example, we use an unsorted array with a size of 6.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 8 | 5 | 3 | 9 | 1 |

Step 1: Build a complete binary tree

First, we will convert an array into a complete binary tree. A complete binary tree is a binary tree in which all levels are fully filled and from left to right except for the last levels.

The nodes will follow the sequence of the array where it is unsorted.

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 8 | 5 | 3 | 9 | 1 |

Tree:

A drawing of a tree

Description automatically generated

Step 2: Max-Heapify Constructed Binary Tree

It will then convert the entire binary tree into the maximum heap. A Max heap is a tree in which the parent node is greater than or equal to its offspring. The heapification process begins with the last non leaf node and progresses upwards. It uses the heapify procedure to verify that the subtree rooted at each node meets the maximum heap property

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 8 | 5 | 3 | 2 | 1 |

Tree:

A white grid with black text

Description automatically generated

 Step 3: Remove Maximum from the root and max-heapify

After the max heap tree is constructed, the root node will be the largest element among all the elements. The root node will be removed and swapped with the last element in the heap.  Then the heap’s size will be reduced by 1 and call the max heapify on the root node. This is to make sure the tree satisfies the max heap property. Process repeatedly working to sort all the elements.

From the example above, the 9 is the maximum among all the elements. So, 9 will be swapped with the last element in the tree then only remove 9 from the tree.

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 8 | 5 | 3 | 2 | 1 |

Swap the position of 9 and 1in the array

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 8 | 5 | 3 | 2 | 9 |

A grid with black lines and numbers

Description automatically generated with medium confidence

Now, we need to make sure that the tree satisfies the max heap property. So, we will heapify the tree by swapping the largest child of the parent node (1) to be the parent.

A grid with black letters and numbers

Description automatically generated

Before we carry on to the next iteration, the array and tree will be used for the next iteration will be:

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 8 | 3 | 5 | 1 | 2 | 9 |

Tree:

A diagram of numbers and letters

Description automatically generated with medium confidence

Step 3 will be repeated until only 1 element is in the tree.

**2nd iteration**

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 8 | 3 | 5 | 1 | 2 | 9 |

Tree:

A diagram of numbers and letters

Description automatically generated with medium confidence

Swap 2 and 8.

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 1 | 8 | 9 |

Then carry out heapify

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 | 3 | 2 | 1 | 8 | 9 |

Tree:

A graph paper with numbers and symbols

Description automatically generated

**3rd iteration**

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 | 3 | 2 | 1 | 8 | 9 |

Tree:

A grid with numbers and lines

Description automatically generated

Swap 5 and 1 then carry heapify

A graph paper with numbers and arrows

Description automatically generated

**4th iteration**

Array:

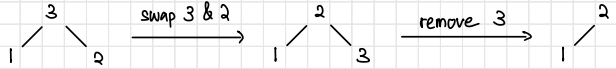
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 1 | 2 | 5 | 8 | 9 |

Tree:

A number and a number on a grid

Description automatically generated with medium confidence

Swap 3 and 2



A close-up of a sign

Description automatically generated

**5th iteration**

Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 5 | 8 | 9 |

Tree:

A black line on a white grid

Description automatically generated

Swap 2 and 1 then heapify

A grid with a number

Description automatically generated with medium confidence

Step 4: Remove the last elements and return the sorted array

Since 1 is the last element in the tree, it will just return where it is the smallest number. Then it will return the sorted array.

Sorted Array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 8 | 9 |

Comparison of Heap Sort, Merge Sort and Quicksort

Heap Sort, Merge Sort and Quick Sort are 3 different sorting algorithms each with unique advantages that suit different use cases. Heap sort has a space complexity of only O (1) because it relies on binary heap data structure. It makes it extremely efficient in terms of space. This is because quick sort uses a recursion stack that requires only O (log n) additional space. It can lead to good cache performance and faster than merge sort and heap sort. Merge sort is a powerful divide and conquer algorithm that reliably separates the array into halves sorts each half then combines the sorted halves back together.

  Heap Sort performs very unstable as it does not preserve the relative order of equal elements, which tends to make it slower for most practical applications; whereas Quick Sort is also an unstable algorithm since its sorting algorithm’s performances are highly dependent on pivot position selections; consequently, Merge Sort establishes stable relationships between equals which come very handy when order matters in some applications.

In a nutshell, Heap Sort is advantageous due to its space efficiency, merge sort is advantageous because it is stable and long-lasting when working with large datasets, and quick sort is advantageous because it is practical and efficient.

**Reference List**

* Alake, R. (2023, November 16). *Bubble sort time complexity and algorithm explained*. Built In. <https://builtin.com/data-science/bubble-sort-time-complexity#:~:text=The%20bubble%20sort%20algorithm%27s%20average,complexity%3A%20O(n%C2%B2)>
* G Bhat. (2020, December 13). *Visualization of interpolation search* [Video]. YouTube. <https://www.youtube.com/watch?v=DlCPTPQD6Mw>
* GeeksforGeeks. (2023, May 15). *Interpolation search*. GeeksforGeeks. <https://www.geeksforgeeks.org/interpolation-search/>
* GeeksforGeeks. (2023, November 20). *TimSort Data Structures and Algorithms Tutorials*. GeeksforGeeks. <https://www.geeksforgeeks.org/timsort/>
* GeeksforGeeks. (2024, January 18). *Jump search*. GeeksforGeeks. <https://www.geeksforgeeks.org/jump-search/>
* GeeksforGeeks. (2024, March 14). *Time and space complexity analysis of Merge sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/>
* GeeksforGeeks. (2024, March 14). *Time and space complexity analysis of quick sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/>
* GeeksforGeeks. (2024, March 18). *Time and space complexity analysis of binary search algorithm*. GeeksforGeeks. <https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/>
* GeeksforGeeks. (2024, May 27). *Implement Quicksort with first element as pivot*. GeeksforGeeks. <https://www.geeksforgeeks.org/implement-quicksort-with-first-element-as-pivot/>
* GeeksforGeeks. (2024, June 24). *Bubble Sort Algorithm*. GeeksforGeeks. <https://www.geeksforgeeks.org/bubble-sort-algorithm/>
* GeeksforGeeks. (2024, June 27). *Heap Sort Data Structures and Algorithms tutorials*. GeeksforGeeks. <https://www.geeksforgeeks.org/heap-sort/>
* GeeksforGeeks. (2024, June 27). *Binary Search Algorithm iterative and Recursive Implementation*. GeeksforGeeks. <https://www.geeksforgeeks.org/binary-search/>
* GeeksforGeeks. (2024, July 5). *Introduction to linear search Algorithm*. GeeksforGeeks. <https://www.geeksforgeeks.org/linear-search/>
* VIDHVAN. (2021, April 20). *Jump Search Algorithm explanation | Implementation of Jump Search Algorithm | Jump Search Made Easy* [Video]. YouTube. <https://www.youtube.com/watch?v=CNP1ARmq2rY>

**Appendix Section**

// Class to store and manage order details

class OrderDetails{

private:

int orderNumber;

vector<string> VIPList = {"ADM123", "NAJ123"};

string buyerName;

string buyerId;

string carId;

string carBrand;

string modelName;

string carColor;

int productionYear;

double carPrice;

double totalPrice;

double discount;

time\_t timeValue; // Time value for the current date

static vector<int> numbers; // List to store generated order numbers to ensure uniqueness

struct tm ltm = {0}; // Local time structure

struct tm currentDate = {0}; // Current date structure

// Generate unique order number

void generateOrderNumber(){

orderNumber = rand()%(1001);

for(auto& n : numbers){

if(n == orderNumber){

generateOrderNumber();

return;

}

}

numbers.push\_back(orderNumber);

}

// Check if the buyer is eligible for a discount

void isDiscount(){

for(auto& vip : VIPList){

if(vip == buyerId){

discount = 0.1;

totalPrice = carPrice - (carPrice \* discount);

}

else{

totalPrice = carPrice; // No discount for non-VIP buyers

}

}

}

public:

// Constructor to initialize order details

OrderDetails(string name, string id, string cid, string brand,

string model, string color, int py, double price){

time\_t now = time(0);

buyerName = name;

buyerId = id;

carId = cid;

carBrand = brand;

modelName = model;

carColor = color;

productionYear = py;

carPrice = price;

generateOrderNumber();

isDiscount();

ltm = \*localtime(&now);

currentDate.tm\_year = ltm.tm\_year; // Year since 1900

currentDate.tm\_mon = ltm.tm\_mon; // Month (0-11)

currentDate.tm\_mday = ltm.tm\_mday; // Day of the month

timeValue = mktime(&currentDate);

}

// Get the customer ID

string getCustomerId(){

return buyerId;

}

// Get the car ID

string getCarId(){

return carId;

}

// Get the final price after discount

double getFinalPrice(){

return totalPrice;

}

// Get the local time structure

struct tm getTime(){

return ltm;

}

// Get the current date as time\_t

time\_t getCurrentDate(){

return timeValue;

}

// Display detailed order information

void displayDetails(){

cout << endl;

cout << "Bill ID: " << orderNumber << endl;

cout << "Customer ID: " << buyerId << endl;

cout << "Customer Name: " << buyerName << endl;

cout << "Car ID: " << carId << endl;

cout << "Car Brand: " << carBrand << endl;

cout << "Car Model: " << modelName << endl;

cout << "Car Color: " << carColor << endl;

cout << "Production Year: " << productionYear << endl;

cout << "Price: RM" << fixed << setprecision(2) << carPrice << endl;

cout << "Discount : " << discount \* 100 << "%" << endl;

cout << "Total Price: RM" << fixed << setprecision(2) << totalPrice << endl;

cout << "Date:" << ltm.tm\_mday << "/" <<

1 + ltm.tm\_mon << "/" <<

1900 + ltm.tm\_year << endl;

cout << "Time (00:00, 24-hour time): "<< ltm.tm\_hour << ":";

cout << ltm.tm\_min;

cout << endl << endl;

}

// Display a summary report for the order

void displayReport(){

cout << endl;

cout << "Car ID: " << carId << endl;

cout << "Car Brand: " << carBrand << endl;

cout << "Car Color: " << carColor << endl;

cout << "Customer Name: " << buyerName << endl;

cout << "Date:" << ltm.tm\_mday << "/" <<

1 + ltm.tm\_mon << "/" <<

1900 + ltm.tm\_year << endl;

cout << "Price: RM" << fixed << setprecision(2) << totalPrice << endl;

cout << endl << endl;

}

};

// Class to store and manage car details

class Car{

private:

string carID;

string brand;

string model;

string color;

string manufactureCountry;

int manufactureYear;

double price;

int Sales;

bool bestSeller;

// Generates a random digit

string generateNumber(){

int num;

num = rand()%10;

return to\_string(num);

}

// Generate a random car ID

void randomID(){

carID = brand.substr(0,3);

for(int i = 0; i<9; i++){

carID += generateNumber();

}

}

public:

Car() = default; // Default constructor

Car(string b, string m, string c, string country, int year, double p, int s = 0){

brand = b;

model = m;

color = c;

manufactureCountry = country;

manufactureYear = year;

price = p;

Sales = s;

randomID();

}

// Get methods

string getCarId(){

return carID;

}

string getBrand(){

return brand;

}

string getModel(){

return model;

}

string getColor(){

return color;

}

string getManufactureCountry(){

return manufactureCountry;

}

int getManufactureYear(){

return manufactureYear;

}

double getPrice(){

return price;

}

int getSales(){

return Sales;

}

// Check if the car is a best seller

bool isBestSelling(){

if (Sales > 8){

bestSeller = 1;

}

else{

bestSeller = 0;

}

return bestSeller;

}

// Set methods

string setBrand(string b){

if(b!="-1")

brand = b;

}

string setModel(string m){

if(m!="-1")

model = m;

}

string setColor(string c){

if(c!="-1")

color = c;

}

string setManufactureCountry(string country){

if(country!="-1")

manufactureCountry = country;

}

int setManufactureYear(int year){

if(year!=-1)

manufactureYear = year;

}

double setPrice(double p){

if(p!=-1)

price = p;

}

void increaseSales(){

Sales += 1; // Increment sales count

}

};

// Class to manage user information

class User{

private:

string userID;

string userName;

string userPassword;

string status;

public:

User() = default; // Default Constructor

User(string id, string name, string password){

userID = id;

userName = name;

userPassword = password;

}

// Get methods

string getUserId(){

return userID;

}

string getUserName(){

return userName;

}

string getUserPassword(){

return userPassword;

}

string getUserStatus(){

return status;

}

// Set methods

void setUserStatus(string s){

status = s;

}

};

// Class to manage the system, including users, cars, orders

class System{

private:

vector<User> userList;

vector<User> adminList;

vector<Car> carList;

vector<Car> temp; // temp vector to stored sorted cars

vector<OrderDetails> orderList;

vector<OrderDetails> reportList;

User currentUser;

int taskOption;

double totalPrice;

// Function to register a new user

void registerFunc(){

string userId;

string name;

bool valid;

string password;

cin.ignore();

cout << endl;

cout << "Registration" << endl;

do{

valid = 1;

cout << "Enter unique Id >";

getline(cin, userId);

for(auto& user : userList){

if (userId == user.getUserId()){

cout << "ID existed, please try another id" << endl;

valid = 0;

}

}

if(userId == "-1"){

cout << "Invalid ID" << endl;

valid = 0;

}

}while(!valid);

cout << "Enter your name >";

getline(cin, name);

cout << "Enter your password >" ;

getline(cin, password);

cout << "Register successfully!" << endl << endl;

registeredUser(User(userId, name, password));

}

// Add an order to the order list and increases car sales

void addOrder (OrderDetails order){

for(auto& car : carList){

if (car.getCarId() == order.getCarId()){

car.increaseSales();

}

}

orderList.push\_back(order);

}

// Handles the purchases of a car

void purchase(int x){

int choice;

cout << "(type in '-1' to exit selection)> ";

cin >> choice;

bool status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "(type in '-1' to exit selection)> ";

cin >> choice;

}

}

if (x == 1){ // If purchasing from the main car list

if(choice == -1){

cout << endl;

cout << "Return to Main Page" << endl;

}

else if(choice < 1 || choice > carList.size()){

cout << endl;

cout << "Option out of range" << endl;

purchase(x);

}

else{

OrderDetails order(currentUser.getUserName(), currentUser.getUserId(), carList[choice-1].getCarId(),

carList[choice-1].getBrand(),carList[choice-1].getModel(), carList[choice-1].getColor(),

carList[choice-1].getManufactureYear(), carList[choice-1].getPrice());

addOrder(order);

order.displayDetails();

}

}

else if(x == 2){ // If purchasing from the temporary list

if(choice == -1){

cout << endl;

cout << "Return to Main Page" << endl;

}

else if(choice < 1 || choice > temp.size()){

cout << endl;

cout << "Option out of range" << endl;

purchase(x);

}

else{

OrderDetails order(currentUser.getUserName(), currentUser.getUserId(), temp[choice-1].getCarId(),

temp[choice-1].getBrand(),temp[choice-1].getModel(), temp[choice-1].getColor(),

temp[choice-1].getManufactureYear(), temp[choice-1].getPrice());

addOrder(order);

order.displayDetails();

}

}

}

// Search Car id by using binary search

int Binary\_Search\_ID(vector<Car> carList, int low\_arr, int high\_arr, string id){

while(low\_arr <= high\_arr){

int middle\_arr = ( ( high\_arr - low\_arr ) / 2 ) + low\_arr ;

if(carList[middle\_arr].getCarId() == id)

return middle\_arr;

if(carList[middle\_arr].getCarId() <= id)

low\_arr = middle\_arr + 1;

else

high\_arr = middle\_arr - 1;

}

return -1;

}

// Search best selling cars using binary search

int Binary\_Search\_bestSelling(vector<Car> carList, int low\_arr, int high\_arr, bool best){

while(low\_arr <= high\_arr){

int middle\_arr = ( ( high\_arr - low\_arr ) / 2 ) + low\_arr ;

if(carList[middle\_arr].isBestSelling() == best)

return middle\_arr;

if(carList[middle\_arr].isBestSelling() <= best)

low\_arr = middle\_arr + 1;

else

high\_arr = middle\_arr - 1;

}

return -1;

}

// Search report within specific period using binary search

int Binary\_Search\_report(vector<OrderDetails> orderList, int low\_arr,

int high\_arr, struct tm startTime, struct tm endTime){

time\_t start\_time\_t = mktime(&startTime);

time\_t end\_time\_t = mktime(&endTime);

while(low\_arr <= high\_arr){

int middle\_arr = ( ( high\_arr - low\_arr ) / 2 ) + low\_arr ;

if(orderList[middle\_arr].getCurrentDate() >= start\_time\_t && orderList[middle\_arr].getCurrentDate() <= end\_time\_t)

return middle\_arr;

if(orderList[middle\_arr].getCurrentDate() < start\_time\_t)

low\_arr = middle\_arr + 1;

else

high\_arr = middle\_arr - 1;

}

return -1;

}

// Sort best selling car according to brand using bubbles sort

void bubbleSort(vector<Car>& arr){

for(int i =0; i < arr.size(); i++){

for(int j =0; j<arr.size()-i-1; j++){

if(arr[j].getBrand() > arr[j+1].getBrand()){

swap(arr[j], arr[j+1]);

}

}

}

}

// Linear search for bills by customer ID

void linearSearch\_Bills(){

for (int i = 0; i < orderList.size(); i++){

if(orderList[i].getCustomerId() == currentUser.getUserId()){

orderList[i].displayDetails();

}

}

}

// Linear search for bills by date

void linearSearch\_Bills\_Date(tm date){

for (int i = 0; i < orderList.size(); i++){

if (orderList[i].getTime().tm\_mday == date.tm\_mday &&

orderList[i].getTime().tm\_mon == date.tm\_mon &&

orderList[i].getTime().tm\_year == date.tm\_year){

orderList[i].displayDetails(); // Display bill details if date matches

}

else{

cout << endl;

cout << "No bill issued that day." << endl;

cout << endl;

}

}

}

// Partition function for quick sort

int partition(vector<OrderDetails> &v, int low, int high){

double pivot = v[high].getFinalPrice(); // Pivot element is the final price of the high element

int i = low - 1;

for(int j = low; j < high; j++){

if(v[j].getFinalPrice() < pivot){

i++;

swap(v[i], v[j]);

}

}

swap(v[i+1], v[high]);

return (i+1);

}

// Quick sort function

void quickSort(vector<OrderDetails> &v, int low, int high){

if (low < high){

int pi = partition(v, low, high); // Partitioning index

quickSort(v, low, pi - 1); // Recursively sort elements before partition

quickSort(v, pi+1, high); // Recursively sort elements after partition

}

}

// merge two halves by brand

void mergeBrand(vector<Car>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

vector<Car> LeftArray(n1);

vector<Car> RightArray(n2);

// The Mid element is saved in LeftArray

for (int i = 0; i < n1; i++)

LeftArray[i] = arr[left + i];

for (int i = 0; i < n2; i++)

RightArray[i] = arr[mid + 1 + i];

int i = 0; // Initial index of first subarray

int j = 0; // Initial index of second subarray

int k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (LeftArray[i].getBrand() <= RightArray[j].getBrand()) {

arr[k] = LeftArray[i];

i++;

} else {

arr[k] = RightArray[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = LeftArray[i];

i++;

k++;

}

// Copy the remaining elements of Right array

while (j < n2) {

arr[k] = RightArray[j];

j++;

k++;

}

}

// merge two halves for merge sort by price

void mergePrice(vector<Car>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

vector<Car> LeftArray(n1);

vector<Car> RightArray(n2);

// The Mid element is saved in LeftArray

for (int i = 0; i < n1; i++)

LeftArray[i] = arr[left + i];

for (int i = 0; i < n2; i++)

RightArray[i] = arr[mid + 1 + i];

int i = 0; // Initial index of first subarray

int j = 0; // Initial index of second subarray

int k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (LeftArray[i].getPrice() <= RightArray[j].getPrice()) {

arr[k] = LeftArray[i];

i++;

} else {

arr[k] = RightArray[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = LeftArray[i];

i++;

k++;

}

while (j < n2) {

arr[k] = RightArray[j];

j++;

k++;

}

}

// merge two halves for merge sort by ID

void mergeId(vector<Car>& arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

vector<Car> LeftArray(n1);

vector<Car> RightArray(n2);

// The Mid element is saved in LeftArray

for (int i = 0; i < n1; i++)

LeftArray[i] = arr[left + i];

for (int i = 0; i < n2; i++)

RightArray[i] = arr[mid + 1 + i];

int i = 0; // Initial index of first subarray

int j = 0; // Initial index of second subarray

int k = left; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (LeftArray[i].getCarId() <= RightArray[j].getCarId()) {

arr[k] = LeftArray[i];

i++;

} else {

arr[k] = RightArray[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = LeftArray[i];

i++;

k++;

}

while (j < n2) {

arr[k] = RightArray[j];

j++;

k++;

}

}

// Function to implement merge sort

void mergeSort(vector<Car>& arr, int left, int right) {

if (left >= right)

return;

int mid = left + (right - left) / 2;

// Sort first and second halves

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

// Merge the sorted halves based on taskOption

if(taskOption == 2){

mergePrice(arr, left, mid, right); // Merge by price

}

else if(taskOption == 3){

mergeBrand(arr, left, mid, right); // Merge by brand

}

else if(taskOption == 4){

mergeId(arr, left, mid, right); // Merge by ID

}

}

// Validation for digit input

void digitInputValidation(){

if(cin.fail()){

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(),'\n');

string msg = "Please enter digits.";

throw msg;

}

}

// Validation for date input

void dateInputValidation(){

if(cin.fail()){

cin.clear();

cin.ignore(numeric\_limits<streamsize>::max(),'\n');

string msg = "Please enter proper date.";

throw msg;

}

}

public:

//constructor for system class

System(){

cout << "Welcome to Car Dealership" << endl;

}

// Function to register a user

void registeredUser(User user){

userList.push\_back(user);

}

// Function to register an admin

void registeredAdmin(User admin){

adminList.push\_back(admin);

}

// Function to register a car

void registeredCar(Car car){

carList.push\_back(car);

}

// Function to get the current user's status

string getStatus(){

return currentUser.getUserStatus();

}

// Function to add a new car

void addNewCar(){

string brand;

string model;

string color;

string country;

int year;

double price;

cout << endl;

cout << "Please enter new car information." << endl;

cout << "Car Brand: ";

getline(cin, brand);

cout << "Car Model Name: ";

getline(cin, model);

cout << "Car Color: ";

getline(cin, color);

cout << "Manufacture country: ";

getline(cin, country);

cout << "Manufacture Year: ";

cin >> year;

// Validate digit input for year

bool status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Manufacture Year: ";

cin >> year;

}

}

cout << "Price: ";

cin >> price;

// Validate digit input for price

status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Price: ";

cin >> price;

}

}

// Register the new car

registeredCar(Car(brand, model, color, country, year, price));

cout << "New car added successfully." << endl << endl;

cin.ignore();

}

// Function to remove a car

void removeCar(){

if (!carList.empty()){

displayAllCars();

int option;

cout << endl;

cout << "Choose a car to delete: ";

cin >> option;

// Validate digit input for option

bool status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Choose a car to delete: ";

cin >> option;

}

}

// Check if the option is within range

if (option > 0 && option <= carList.size()){

carList.erase(carList.begin()+(option -1));

cout << "Car removed successfully." << endl << endl;

}

else{

cout << endl;

cout << "Option out of range" << endl;

cout << endl;

}

}

}

// Function to modify a car's details

void modifyCar(){

string id;

cout << endl;

cout << "Enter the car id to modify: ";

cin >> id;

string brand;

string model;

string color;

string country;

int year;

double price;

// Search for the car by id and modify its details

for (auto& car : carList){

if (car.getCarId() == id){

cout << "Enter -1 if you do not wish to modify this section." << endl;

cout << "Enter new brand: ";

cin >> brand;

car.setBrand(brand);

cout << "Enter new model name: ";

cin >> model;

car.setModel(model);

cout << "Enter new color: ";

cin >> color;

car.setColor(color);

cout << "Enter new Manufacture Country: ";

cin >> country;

car.setManufactureCountry(country);

cout << "Enter new Manufacture Year: ";

cin >> year;

bool status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Enter new Manufacture Year: ";

cin >> year;

}

}

car.setManufactureYear(year);

cout << "Enter new price: ";

cin >> price;

status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Enter new price: ";

cin >> price;

}

}

car.setPrice(price);

cout << "Changes made." << endl << endl;

return;

}

}

cout << "No such car exist." << endl << endl;

}

// Function to handle user login

bool loginFunc(){

string userId;

string password;

char x;

while(true){

cout << "Please Login (-1 to quit)" << endl;

cout << "Enter User Id >" ;

getline(cin, userId);

if (userId == "-1"){

return 0;

}

cout << "Enter Password >" ;

getline(cin, password);

// Check user credentials

for(auto& user : userList){

if(userId == user.getUserId() && password == user.getUserPassword()){

currentUser = User(user.getUserId(),user.getUserName(), user.getUserPassword());

currentUser.setUserStatus("user");

cout << endl;

cout << "Login Successful" << endl;

cout << "Welcome, " << currentUser.getUserName() << endl;

cout << endl;

return 1;

}

}

// Check admin credentials

for(auto& admin : adminList){

if(userId == admin.getUserId() && password == admin.getUserPassword()){

currentUser = User(admin.getUserId(),admin.getUserName(), admin.getUserPassword());

currentUser.setUserStatus("admin");

cout << endl;

cout << "Login Successful" << endl;

cout << "Welcome, " << currentUser.getUserName() << "(Admin)" << endl;

cout << endl;

return 1;

}

}

// Prompt user for registration if login fails

cout << endl;

cout << "Invalid login" << endl;

cout << "Do you wish to register? (Y/N)" << endl;

cin >> x;

if (x == 'Y' || x == 'y'){

registerFunc();

}

else{

cout << endl;

cin.ignore();

}

}

}

// Display available tasks

int availableTasks(){

cout << endl;

cout << "Please choose a task to continue: " << endl;

cout << "1) Display all cars" << endl;

cout << "2) Display cars sorted by price" << endl;

cout << "3) Display cars sorted by brand" << endl;

cout << "4) Display cars sorted by ID" << endl;

cout << "5) Display best selling cars" << endl;

cout << "6) Search car by ID" << endl;

cout << "7) Search issued bills" << endl;

if (currentUser.getUserStatus() == "admin"){

cout << "8) Add new car" << endl;

cout << "9) Modify car" << endl;

cout << "10) Remove car" << endl;

cout << "11) Generate Report" << endl;

cout << "12) Exit" << endl;

}

else{

cout << "8) Exit" << endl;

}

cout << "> ";

cin;

cin >> taskOption;

bool status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "> ";

cin >> taskOption;

}

}

// Validate task option based on user or admin status

if (((taskOption < 1 || taskOption > 8) && currentUser.getUserStatus() != "admin")

|| ((taskOption < 1 || taskOption > 12) && currentUser.getUserStatus() == "admin")){

availableTasks();

}

else{

cin.ignore();

return taskOption;

}

}

// Display all cars or sorted cars based on task option

void displayAllCars(){

int i = 1;

cout << endl << "Car options" << endl;

if (taskOption == 1){

if (carList.empty()){

cout << endl;

cout << "No car available for now." << endl;

cout << endl;

}

else{

cout << endl;

for(auto& car : carList){

cout << i << ")";

cout << car.getBrand() << endl;

cout << "Car ID: " << car.getCarId() << endl;

cout << "Car Model Name: " << car.getModel() << endl;

cout << "Color: " << car.getColor() << endl;

cout << "Manufacture Country: " << car.getManufactureCountry() << endl;

cout << "Manufacture Year: " << car.getManufactureYear() << endl;

cout << "Price: RM" << fixed << setprecision(2) << car.getPrice() << endl;

cout << "Sales: " << car.getSales() << endl << endl;

++i;

}

purchase(1);

}

}

else if(taskOption == 10){

cout << endl;

for(auto& car : carList){

cout << i << ")";

cout << car.getBrand() << endl;

cout << "Car ID: " << car.getCarId() << endl;

cout << "Car Model Name: " << car.getModel() << endl;

cout << "Color: " << car.getColor() << endl;

cout << "Manufacture Country: " << car.getManufactureCountry() << endl;

cout << "Manufacture Year: " << car.getManufactureYear() << endl;

cout << "Price: RM" << fixed << setprecision(2) << car.getPrice() << endl;

cout << "Sales: " << car.getSales() << endl << endl;

++i;

}

}

else{

if (temp.empty()){

cout << endl;

cout << "No car available for now." << endl;

cout << endl;

}

else{

cout << endl;

for(auto& car : temp){

cout << i << ")";

cout << car.getBrand() << endl;

cout << "Car ID: " << car.getCarId() << endl;

cout << "Car Model Name: " << car.getModel() << endl;

cout << "Color: " << car.getColor() << endl;

cout << "Manufacture Country: " << car.getManufactureCountry() << endl;

cout << "Manufacture Year: " << car.getManufactureYear() << endl;

cout << "Price: RM" << fixed << setprecision(2) << car.getPrice() << endl;

cout << "Sales: " << car.getSales() << endl << endl;

++i;

}

purchase(2);

}

}

}

// Sort cars using merge sort and display

void sortCar\_MergeSort(){

temp = carList;

mergeSort(temp, 0, temp.size()-1);

displayAllCars();

}

// Search for a car by its ID

void searchCarByID(){

int found;

string id;

if (!carList.empty()){

temp = carList;

cout << endl;

cout << "Enter Car ID: ";

cin >> id;

// Sort for binary search algorithm

sort(temp.begin(), temp.end(), [](Car& a, Car& b) {

return a.getCarId() < b.getCarId();

});

found = Binary\_Search\_ID(temp, 0, temp.size()-1, id);

if (found != -1){

Car elementToKeep = temp[found];

temp.clear();

temp.push\_back(elementToKeep);

displayAllCars();

}

else{

cout << endl;

cout << "Car with this ID does not exist." << endl << endl;

}

}

else {

cout << endl;

cout << "No car available." << endl << endl;

}

}

// Search for best selling cars

void searchBestSelling(){

int found;

temp.clear();

if (!carList.empty()){

vector<Car> searchList = carList;

// sort for binary search algorithm

sort(searchList.begin(), searchList.end(), [](Car& a, Car& b) {

return a.isBestSelling() < b.isBestSelling();

});

found = Binary\_Search\_bestSelling(searchList, 0, searchList.size()-1, 1);

if (found == -1){

cout << endl;

cout << "No Best Selling Cars for now." << endl << endl;

}

else{

while (found != -1){

temp.push\_back(searchList[found]);

searchList.erase(searchList.begin()+found);

found = Binary\_Search\_bestSelling(searchList, 0, searchList.size()-1, 1);

}

bubbleSort(temp); // bubbleSort function is used to sort best selling cars

displayAllCars();

}

}

else {

cout << endl;

cout << "No car available." << endl << endl;

}

}

// Search issued bills by customer ID or date

void searchIssuedBills(){

int choice;

cout << endl << "Issued bills by: " << endl;

cout << "1)Customer ID " << endl;

cout << "2)Date of bill issued " << endl;

cout << "> ";

cin >> choice;

bool status = true;

while(status){

try{

digitInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "> ";

cin >> choice;

}

}

if (choice == 1){

if (orderList.empty()){

cout << endl << "You have no previous purchase.";

cout << endl << endl;

}

else{

linearSearch\_Bills(); // linearSearch\_Bills function is used for searching bills by customer ID

}

}

else if (choice == 2){

if (orderList.empty()){

cout << endl << "You have no previous purchase.";

cout << endl << endl;

}

else{

struct tm date;

char dash;

cout << endl;

cout << "Enter bill data (day/month/year): ";

cin >> date.tm\_mday >> dash >> date.tm\_mon >> dash >> date.tm\_year;

bool status = true;

while(status){

try{

dateInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Enter bill data (day/month/year): ";

cin >> date.tm\_mday >> dash >> date.tm\_mon >> dash >> date.tm\_year;

}

}

date.tm\_mon -= 1;

date.tm\_year -= 1900;

cout << endl;

linearSearch\_Bills\_Date(date); // linearSearch\_Bills\_Date function is used for searching bills by date

}

}

}

// Generate a report of orders within a specific period

void generateReport(){

struct tm startTime = {0}; // Struct to hold starting time

struct tm endTime = {0}; // Struct to hold ending time

char dash;

int found;

totalPrice = 0;

vector<OrderDetails> tempList = orderList; // Copy of orderList for temporary operations

// Input and validation for starting date

cout << "Enter specific period range" << endl;

cout << "Please type according to the format and include '/' " << endl;

cout << "Starting Date (day/month/year): ";

cin >> startTime.tm\_mday >> dash >> startTime.tm\_mon >> dash >> startTime.tm\_year;

bool status = true;

while(status){

try{

dateInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Starting Date (day/month/year): ";

cin >> startTime.tm\_mday >> dash >> startTime.tm\_mon >> dash >> startTime.tm\_year;

}

}

startTime.tm\_mon -= 1; // Adjust month for tm struct (0-based)

startTime.tm\_year -= 1900; // Adjust year for tm struct (years since 1900)

// Input and validation for ending date

cout << "Ending Date (day/month/year): ";

cin >> endTime.tm\_mday >> dash >> endTime.tm\_mon >> dash >> endTime.tm\_year;

status = true;

while(status){

try{

dateInputValidation();

status = false;

}

catch(string errorMsg){

cout << endl;

cout << "Error: " << errorMsg << endl;

cout << endl;

cout << endl;

cout << "Ending Date (day/month/year): ";

cin >> endTime.tm\_mday >> dash >> endTime.tm\_mon >> dash >> endTime.tm\_year;

}

}

endTime.tm\_mon -= 1;

endTime.tm\_year -= 1900;

// Convert struct tm to time\_t for comparison

time\_t start\_time\_t = mktime(&startTime);

time\_t end\_time\_t = mktime(&endTime);

// Validate and process the entered period

if (end\_time\_t >= start\_time\_t) {

reportList.clear(); // Clear previous report entries

// Search for orders within the specified period

found = Binary\_Search\_report(tempList, 0, tempList.size()-1, startTime, endTime);

if (found == -1){

cout << endl;

cout << "No purchase within this period." << endl << endl;

}

else{

// Gather all matching orders into reportList

while (found != -1){

reportList.push\_back(tempList[found]);

tempList.erase(tempList.begin()+found);

found = Binary\_Search\_report(tempList, 0, tempList.size()-1, startTime, endTime);

}

// Sort reportList for chronological order

quickSort(reportList, 0, reportList.size()-1);

// Display each order's details in the report

for (auto& order : reportList){

order.displayReport();

totalPrice += order.getFinalPrice(); // Accumulate total price

}

cout << "Total price of all sold cars: RM" << fixed << setprecision(2) << totalPrice;

cout << endl << endl;

}

} else {

cout << endl;

cout << "Period is invalid" << endl;

cout << endl;

}

}

};

//static member initialization

vector<int> OrderDetails :: numbers;

int main()

{

srand(time(0)); // for generate random car ID

int optionTask;

char running;

System system;// Initialize the car dealership system

// Initialize existing users and admins

system.registeredAdmin(User("ADM123", "Ang Yi Xun", "Cho030909$"));

system.registeredUser(User("NAJ123", "Dr Najla'a Ateeq M. Draib", "najla2024"));

system.registeredUser(User("AIT935", "Choo Qie Sheng", "QS030208"));

system.registeredUser(User("AIT974", "Khoo Song Hao", "SH050503"));

// Initialize cars

system.registeredCar(Car("Audi", "R8", "black", "Germany", 2021, 1200000, 5));

system.registeredCar(Car("Toyota", "Prius", "Purple", "Japan", 2020, 200000, 10));

system.registeredCar(Car("Peroduo", "MyVi", "Red", "Malaysia", 2022, 80000, 12));

system.registeredCar(Car("Proton", "X70", "White", "China", 2023, 400000, 6));

system.registeredCar(Car("BYD", "Seal", "black", "China", 2024, 400000, 11));

system.registeredCar(Car("Honda", "Civic", "black", "Japan", 2022, 600000, 7));

// Main program loop

while(system.loginFunc()){

running = 'y';

while(running == 'y'){

switch(system.availableTasks()){

case 1:

system.displayAllCars();

break;

case 2:

system.sortCar\_MergeSort();

break;

case 3:

system.sortCar\_MergeSort();

break;

case 4:

system.sortCar\_MergeSort();

break;

case 5:

system.searchBestSelling();

break;

case 6:

system.searchCarByID();

break;

case 7:

system.searchIssuedBills();

break;

case 8:

if(system.getStatus() == "admin")

system.addNewCar();

else{

cout << endl;

running = 'n';

}

break;

case 9:

system.modifyCar();

break;

case 10:

system.removeCar();

break;

case 11:

system.generateReport();

break;

case 12:

cout << endl;

running = 'n';

break;

default:

cout << "Invalid Input" << endl;

}

}

}

cout << endl;

cout << "Thank you for visiting." << endl;

cout << "Hope to see you again." << endl;

return 0;

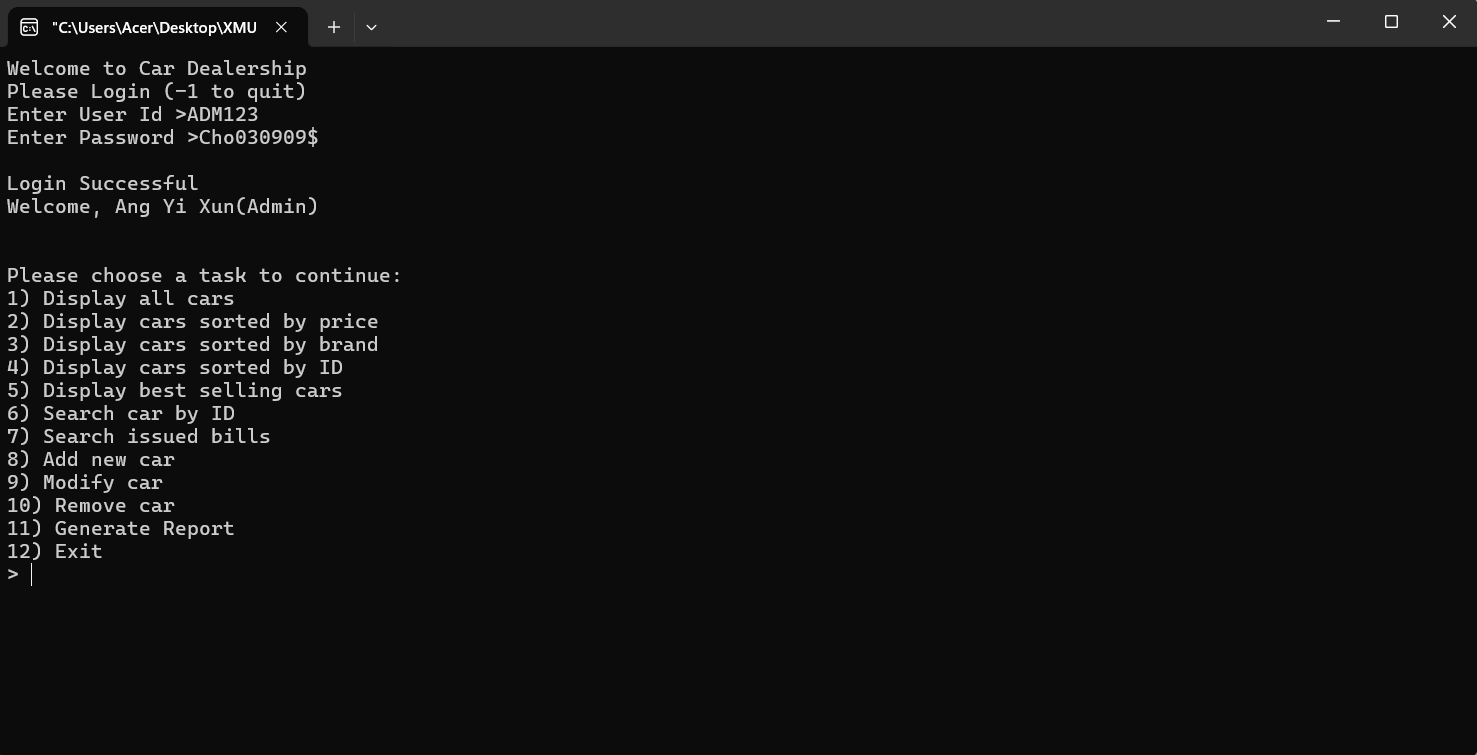
}

Other resource:

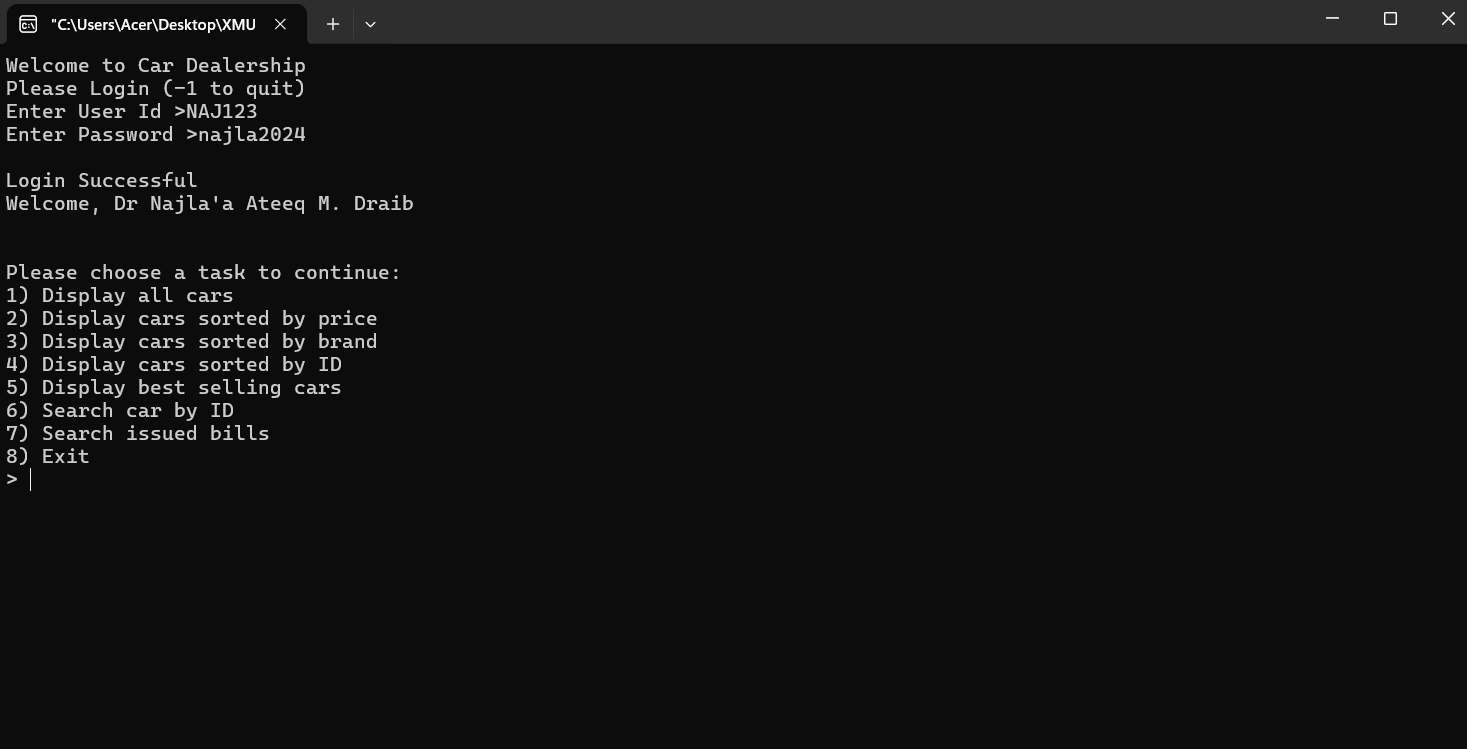
<https://www.hackerearth.com/practice/notes/validating-user-input-in-c/>

**Running results**

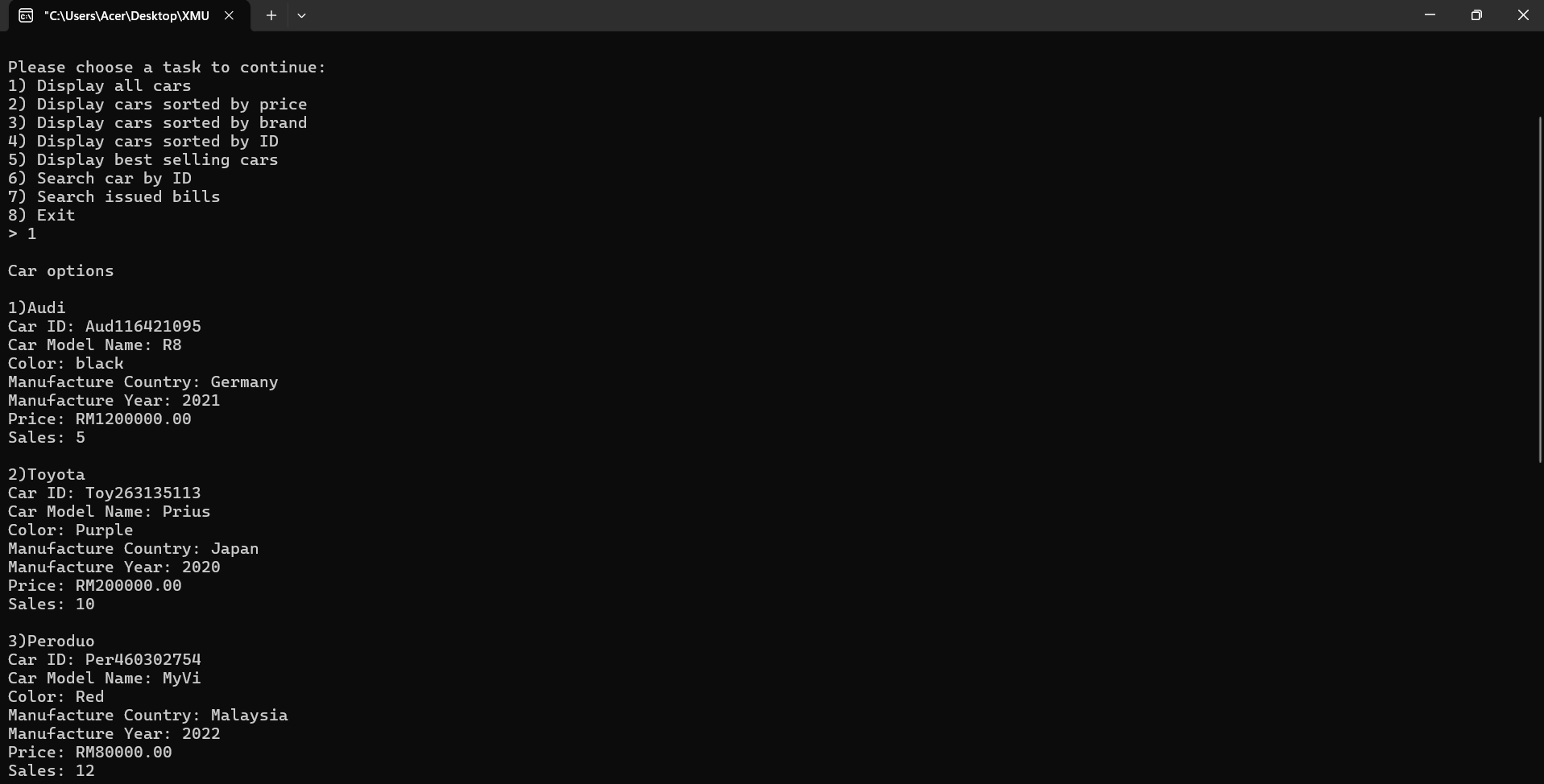
Login for admin



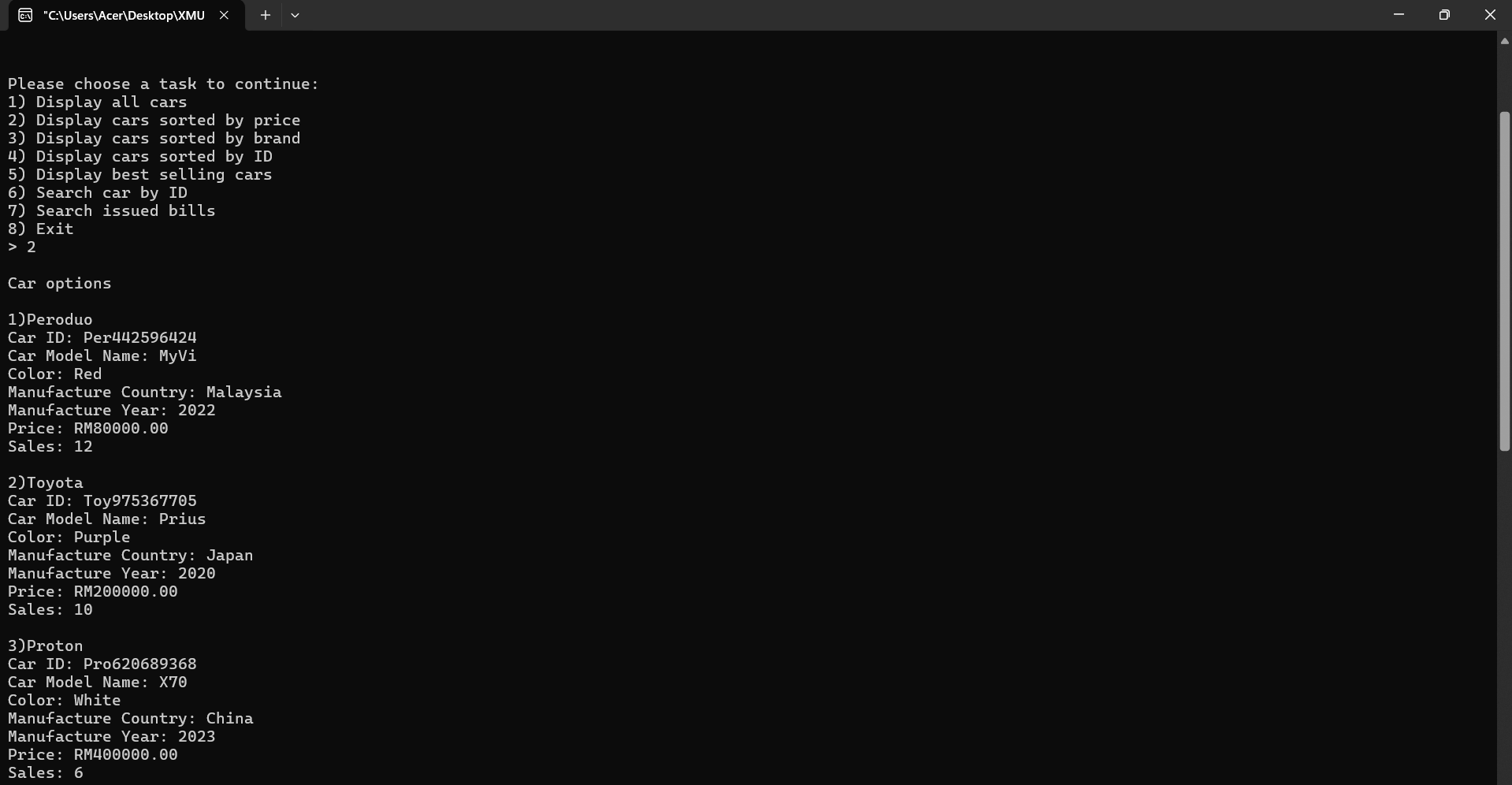
Login for user



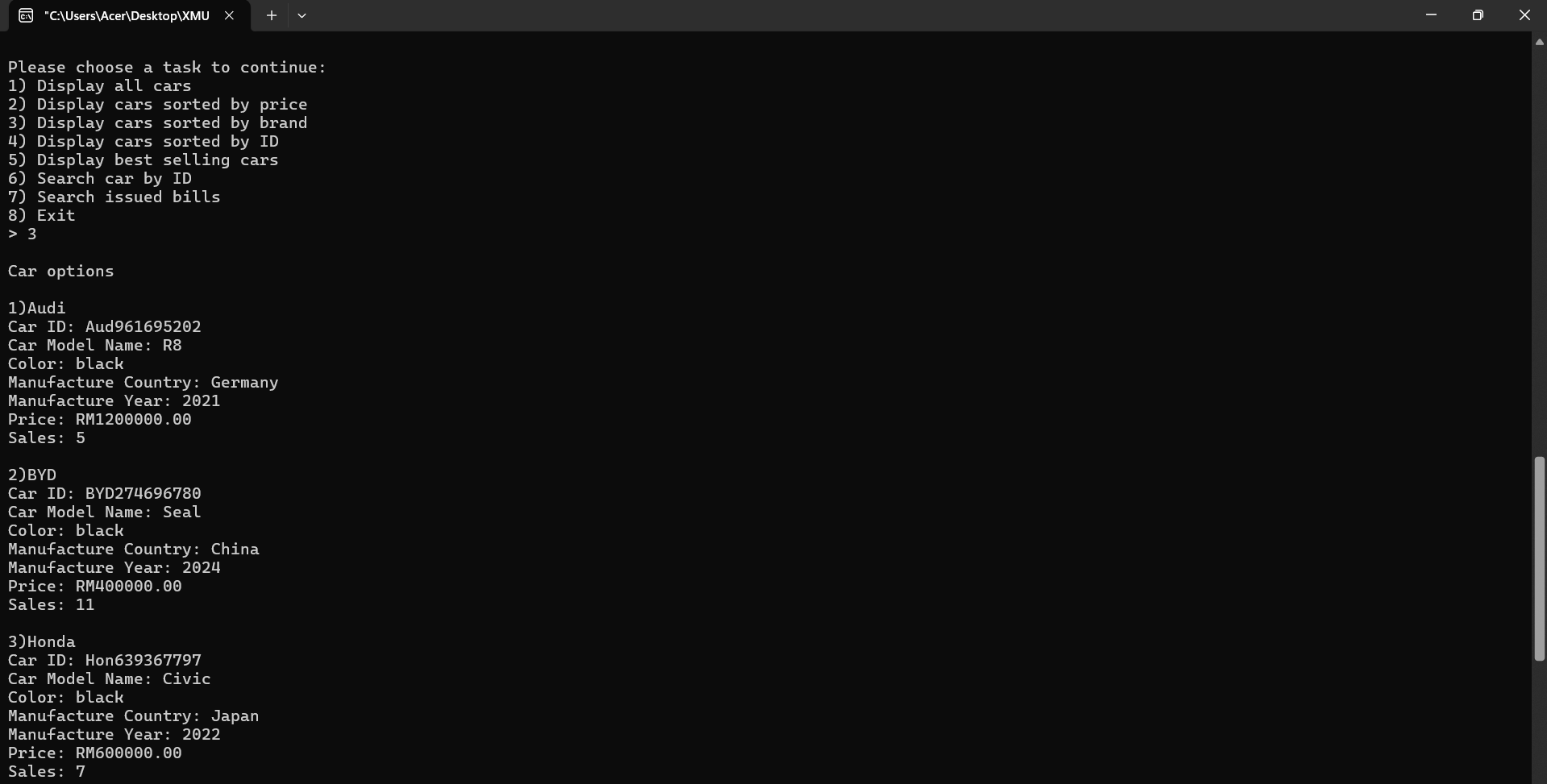
Task 1

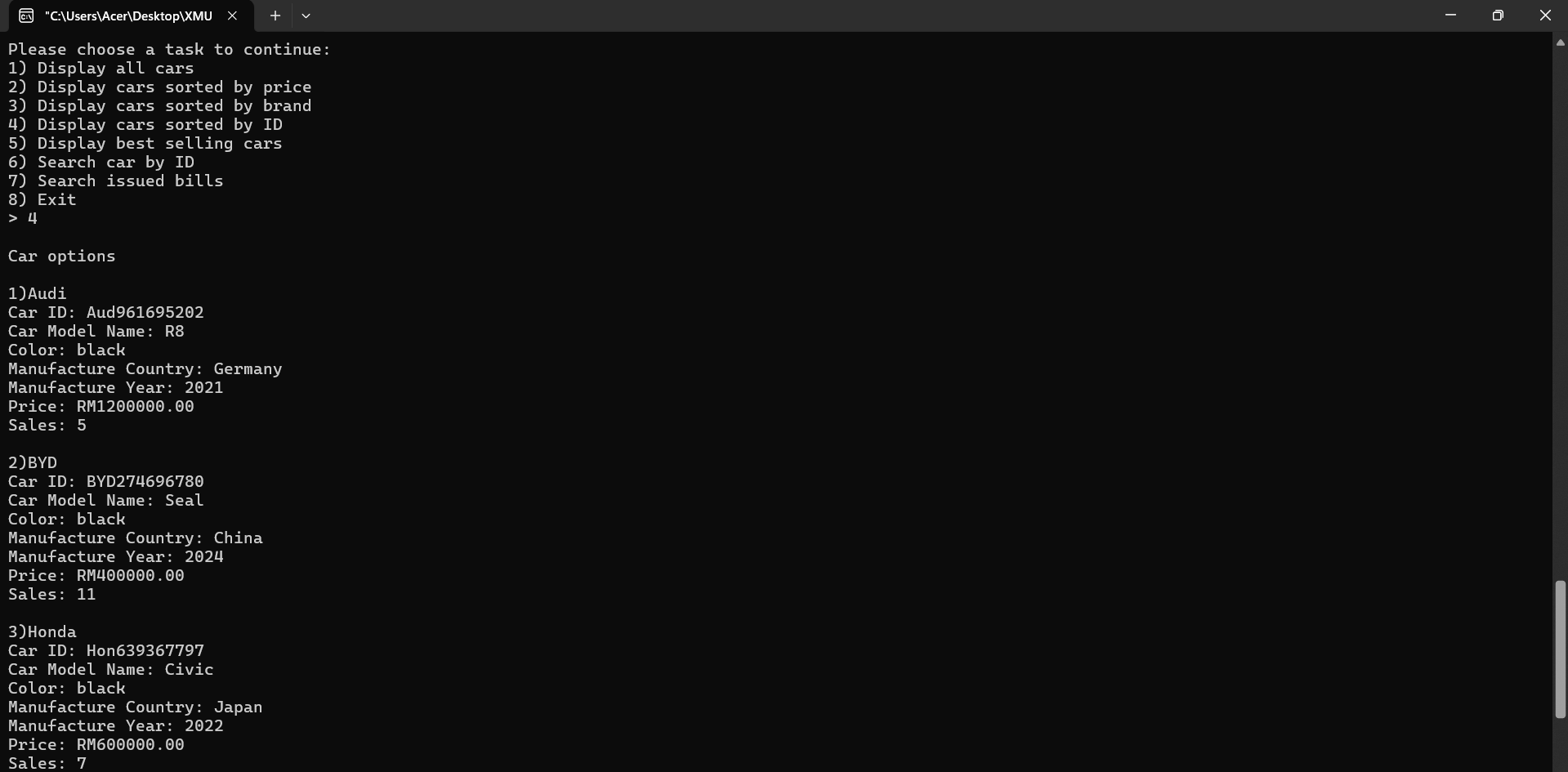
Task 2



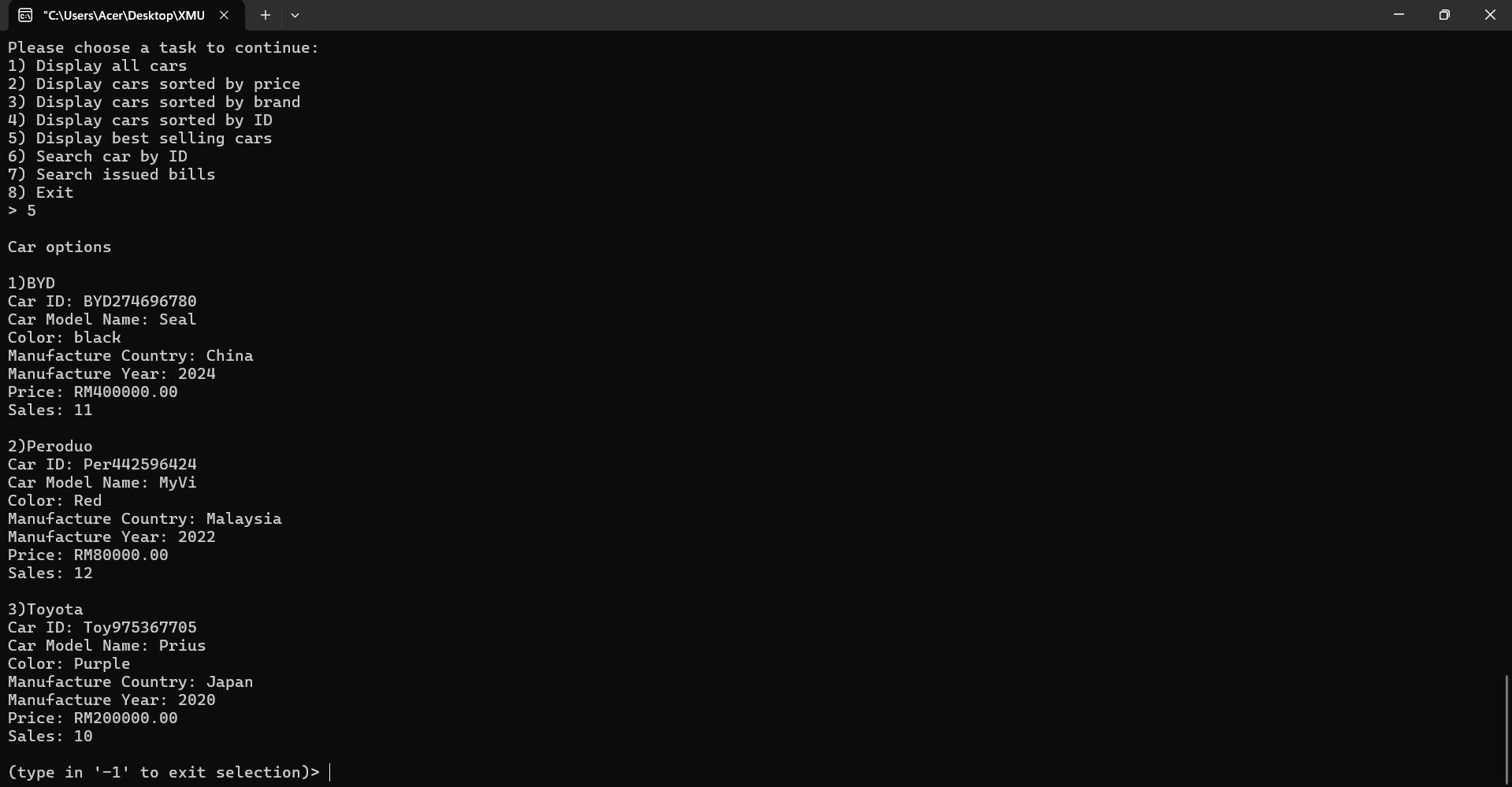
Task 3

Task 4

Task 5



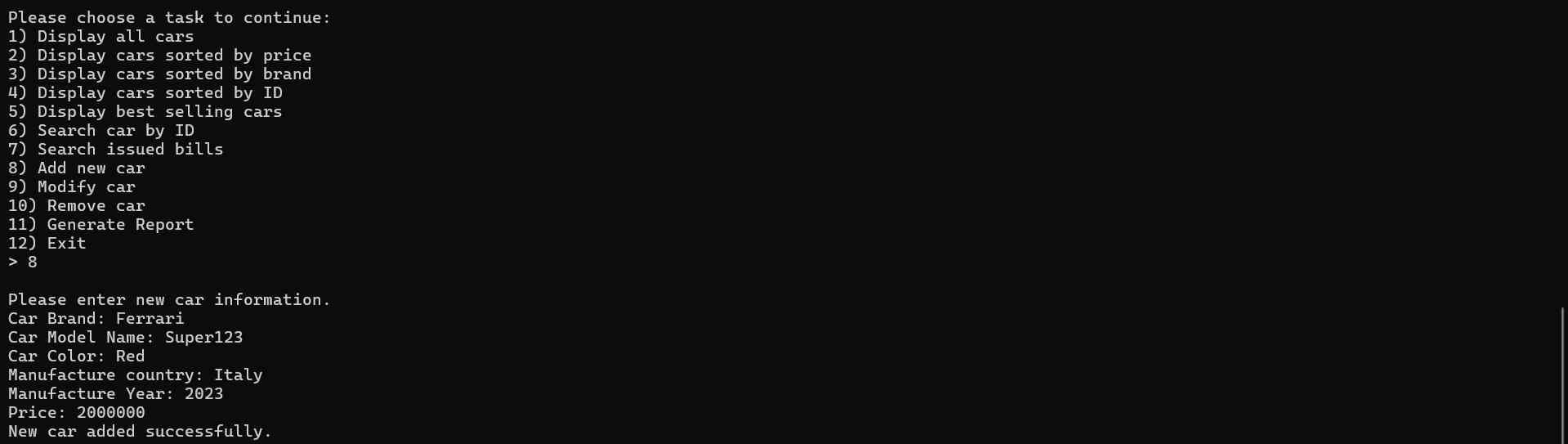
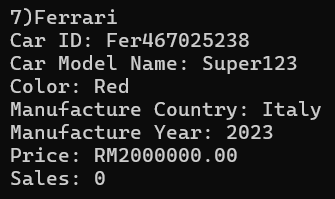
Task 6



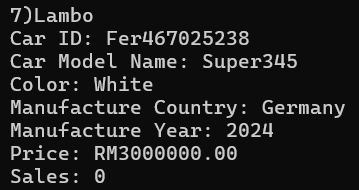
Task 7

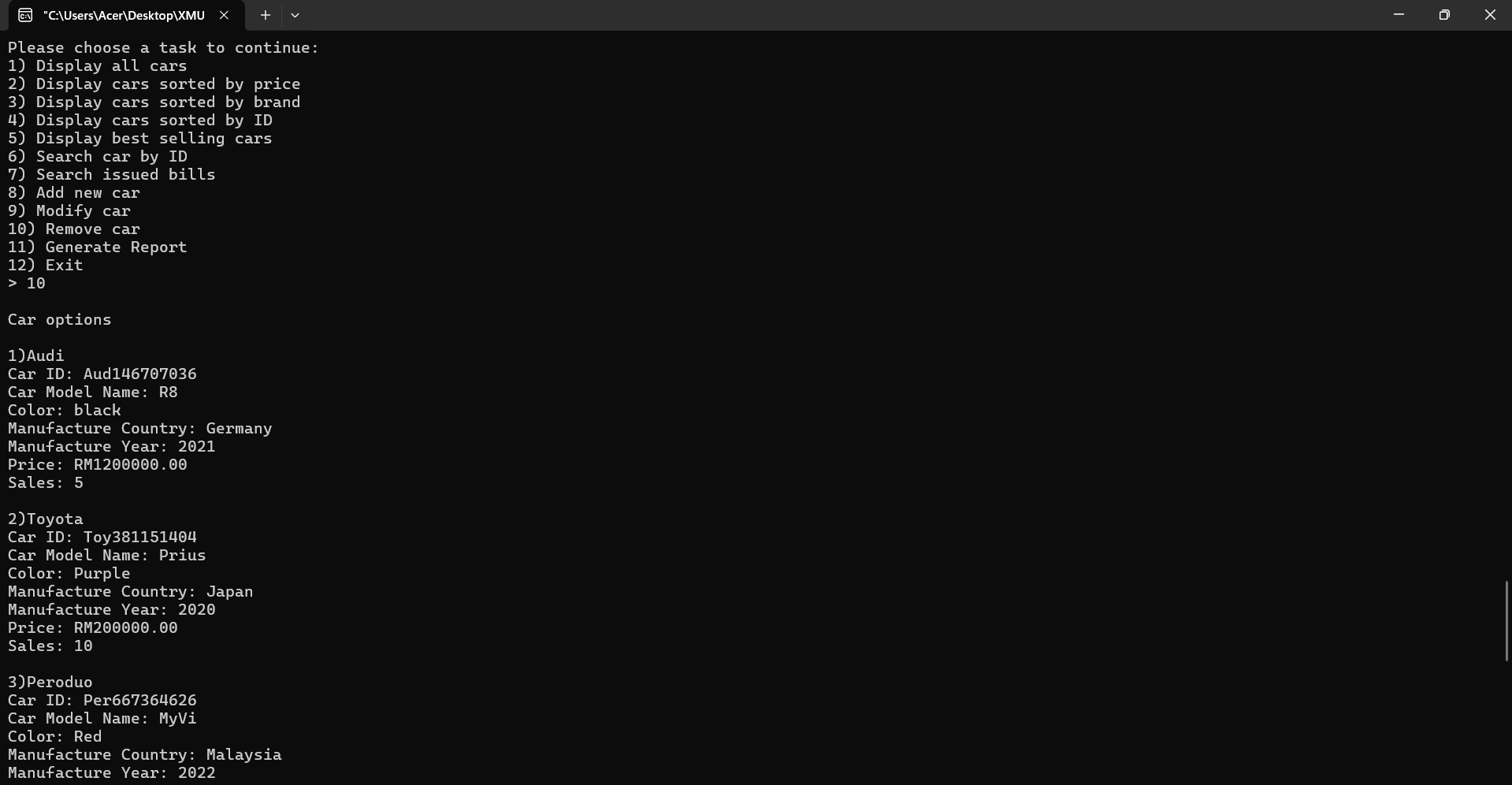
Task 8 (admin)

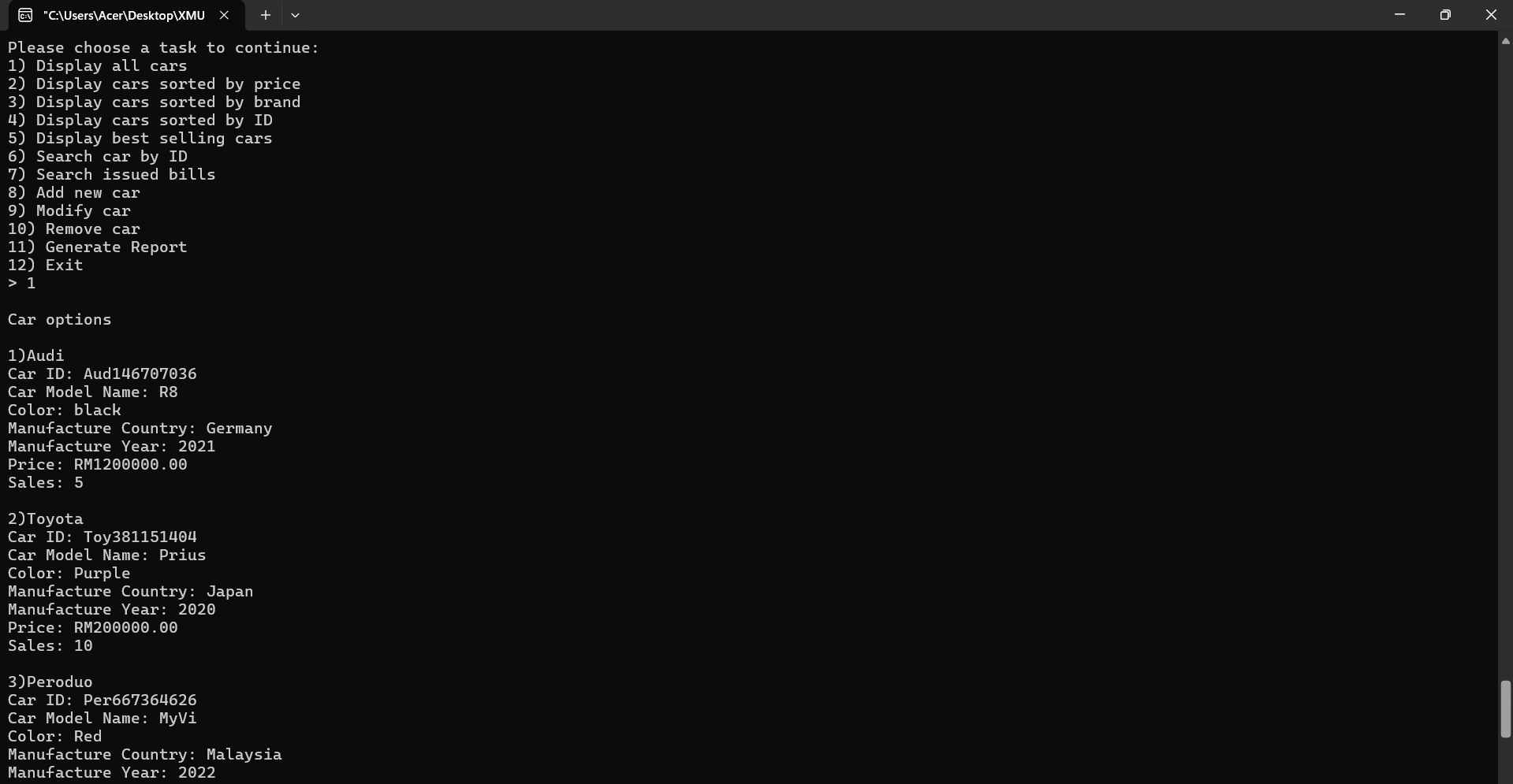
 

Task 9 (admin)

Task 10 (admin)

Task 11 (admin)

