Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Yusuf Küçüköner

Student Number:
b2210356092

1 Problem Definition

We're diving into a deep analysis here, looking closely at three main sorting algorithms - merge sort, insertion sort, and counting sort - as well as checking out two search algorithms - linear search and binary search. We're going to run these algorithms through their paces with input sizes ranging from 500 all the way up to 250,000 elements. Our main goals are to see how fast these algorithms run, figure out how well they handle larger and larger inputs, and pinpoint which ones work best for different problem sizes.

1.1 Background Information

Sorting and searching are fundamental operations in computer science and are essential for various applications, including data processing, information retrieval, and algorithmic problem-solving. Different sorting algorithms employ distinct strategies to arrange elements, while search algorithms aim to locate specific elements within a dataset efficiently.

- **Sorting Algorithms:**
  - Merge Sort
  - Insertion Sort
  - Counting Sort
- **Searching Algorithms:**
  - Linear Search
  - Binary Search

1.2 Problem Statement

We're going to analyze those sorting and searching methods we talked about earlier. Here's what we'll do for each one:

Sorting Algorithms:

1. Generate a random array of integers from the csv data file for each input size.
2. Apply each sorting algorithm to the generated array 10 times.
3. Measure the execution time for each sorting operation and compute the average time for further analysis.

Searching Algorithms:

1. Generate a random array of integers from the csv data file for each input size.
2. Randomly select an element from the array for search.

3. Apply both linear search and binary search algorithms to find the selected element.
4. Repeat the search process 1000 times for statistical significance.
5. Measure the execution time for each search operation and compute the average time for further analysis.

By performing these steps for input sizes ranging from 500 to 250,000 elements, we aim to achieve the following objectives:

- Evaluate the scalability and efficiency of each sorting and searching algorithm.
- Determine the most suitable sorting algorithm for various problem sizes.
- Compare the performance of linear and binary search algorithms.
- Understand the impact of input size on the runtime behavior of algorithms.
- Gain insights into algorithmic complexities and their practical implications.

This detailed analysis will give us a good look at what each algorithm does well and where it struggles. It'll help us make smarter choices when picking algorithms for actual real-life situations.

2 Solution Implementation

In this section, we present the implementation of merge sort, insertion sort, and counting sort algorithms in Java, along with explanations and analysis.

2.1 Merge Sort

Merge sort is a divide-and-conquer algorithm that recursively divides the array into two halves until each subarray contains only one element, then merges them back in sorted order.

2.1.1 Java Code

The Java code for merge sort is provided below:

Listing 1: Merge Sort implementation in Java

```
1 public static int[] mergeSort(int[] array) {
2
3     int n = array.length;
4     if (n <= 1) {
5         return array;
6     }
7     int[] left = new int[n/2];
8     int[] right = new int[n - n/2];
9
```

```
    for (int i = 0; i < left.length; i++) {
        left[i] = array[i];
    }

    for (int i = 0; i < right.length; i++) {
        right[i] = array[i + left.length];
    }
    left = mergeSort(left);
    right = mergeSort(right);
    return merge(left, right);
}

public static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    int i = 0, j = 0, k = 0;
    while (i < A.length && j < B.length) {
        if (A[i] <= B[j]) {
            C[k++] = A[i++];
        } else {
            C[k++] = B[j++];
        }
    }
    while (i < A.length) {
        C[k++] = A[i++];
    }
    while (j < B.length) {
        C[k++] = B[j++];
    }
    return C;
}
```

2.1.2 Java Code Explanation

Below is an explanation of how the implementation works:

- The `mergeSort` method takes an integer array `array` as input and recursively sorts it in ascending order using the merge sort algorithm.
- If the length of the array `n` is less than or equal to 1, indicating it's already sorted, the method returns the array. Which is the base case for recursion.
- Otherwise, it divides the array into two halves: `left` and `right`.
- It then recursively calls `mergeSort` on both halves.
- Finally, it merges the sorted halves using the `merge` method and returns the merged array.

- The `merge` method takes two sorted arrays `A` and `B` as input and merges them into a single sorted array `C`.
- It iterates through both arrays and compares elements from `A` and `B`, selecting the smaller element to add to the merged array `C`.
- Once all elements are merged, it returns the merged array `C`.

Merge sort is great for sorting big sets of data because it's pretty consistent in how long it takes, and it can work on different parts of the data at the same time. People often like using it for jobs where they need to know it's going to perform well and be reliable.

2.1.3 Analysis

The time complexity of merge sort is $O(n \log n)$ in the average and worst cases. It offers stable sorting and is suitable for large datasets due to its efficient divide-and-conquer approach.

2.2 Insertion Sort

Insertion sort is a simple comparison-based sorting algorithm that iterates over the array, selecting each element and inserting it into its correct position in the sorted subarray.

2.2.1 Java Code

The Java code for insertion sort is provided below:

Listing 2: Insertion Sort implementation in Java

```
40 public static void insertionSort(int[] array) {
41     int n = array.length;
42     for (int j = 1; j < n; j++) {
43         int key = array[j];
44         int i = j - 1;
45         while (i >= 0 && array[i] > key) {
46             array[i + 1] = array[i];
47             i = i - 1;
48         }
49         array[i + 1] = key;
50     }
51 }
```

2.2.2 Java Code Explanation

Below is an explanation of how the implementation works:

- The `insertionSort` method takes an integer array `array` as input and sorts it in ascending order.

- It iterates through the array starting from the second element (index 1) up to the last element.
- For each element at index `j`, it compares it with the elements before it (from index `j-1` to `0`) and moves it to the correct position in the sorted subarray.
- Inside the inner while loop, it shifts elements to the right until it finds the correct position for the current element `key`.
- Once the correct position is found, it places the `key` element in its sorted position in the array.
- This process continues until all elements are sorted in ascending order.

Insertion sort works pretty fast for small sets of data or when the data's already somewhat sorted, but it starts to slow down a lot with big sets because it has to do a lot of comparisons. Still, it's easy to understand and does a good job with almost-sorted data.

2.2.3 Analysis

The time complexity of insertion sort is $O(n^2)$ in the worst case. It performs well on small datasets or nearly sorted arrays due to its adaptive nature.

2.3 Counting Sort

Counting sort is a sorting method that doesn't compare elements directly. Which is non-comparison-based algorithm. It's great for arrays of whole numbers with a limited range. Basically, it counts how many times each number shows up in the array and then puts everything back together in order based on those counts.

2.3.1 Java Code

The Java code for counting sort is provided below:

Listing 3: Counting Sort implementation in Java

```
52 public static int[] countingSort(int[] inpArray) {
53     int k = inpArray[0];
54     for (int i = 1; i < inpArray.length; i++) {
55         if (inpArray[i] > k) {
56             k = inpArray[i];
57         }
58     }
59     int[] countArray = new int[k + 1];
60     int[] outputArray = new int[inpArray.length];
61
62     for (int i = 0; i < inpArray.length; i++) {
63         countArray[inpArray[i]]++;
64     }
```

```
65
66      for (int j = 1; j <= k; j++) {
67          countArray[j] += countArray[j - 1];
68      }
69
70      for (int l = inpArray.length - 1; l >= 0; l--) {
71          outputArray[countArray[inpArray[l]] - 1] = inpArray[l];
72          countArray[inpArray[l]]--;
73      }
74      return outputArray;
75  }
```

2.3.2 Java Code Explanation

Here's how the implementation works:

- The `countingSort` method takes an array `inpArray` of integers as input and returns a sorted array.
- It first finds the maximum value 'k' in the input array by iterating through the array once.
- Then, it initializes two arrays: `countArray` to store the count of occurrences of each element, and `outputArray` to store the sorted output.
- Next, it iterates through the input array again and increments the corresponding count in the `countArray` for each element.
- After counting the occurrences of each element, it modifies the `countArray` to store the cumulative count.
- Finally, it iterates through the input array in reverse order and places each element in its correct sorted position in the `outputArray` based on the count information stored in the `countArray`.

This way of doing counting sort takes advantage of knowing that the numbers in the array are only within a small range. That's why it's pretty fast - it sorts in linear time. But if you've got a lot of different numbers to sort through, it can end up taking up a lot of memory space.

2.3.3 Analysis

The time complexity of counting sort is $O(n + k)$, where n is the number of elements in the input array and k is the range of input values. It is highly efficient for certain scenarios, particularly when the range of input values is limited.

In the provided code snippets, line numbers can be referenced as follows:

- For Merge Sort: Line 1

- For Insertion Sort: Line 2
- For Counting Sort: Line 3

2.4 Linear Search

Linear search is a simple search algorithm that sequentially checks each element of the array until the desired element is found or the end of the array is reached.

2.4.1 Java Code

The Java code for linear search is provided below:

Listing 4: Linear Search implementation in Java

```
76 class Searchs {
77     public static int linearSearch(int[] data, int key) {
78         for(int i = 0; i < data.length; i++){
79             if(data[i] == key){
80                 return i;
81             }
82         }
83         return -1;
84     }
85 }
```

2.4.2 Java Code Explanation

Here's a breakdown of how it works:

- The `linearSearch` method takes two parameters: an array `data` containing integers and an integer `key` representing the value to be searched.
- It iterates through each element of the array using a `for` loop, starting from index 0 and ending at the last index (‘data.length - 1‘).
- Inside the loop, it checks if the current element `data[i]` is equal to the target `key`. If they match, it returns the index `i` where the target is located.
- If the loop completes without finding the target, it returns -1 to indicate that the target is not present in the array.

This implementation of linear search examines each element of the array sequentially until it finds the target or reaches the end of the array, resulting in linear time complexity.

2.4.3 Analysis

The time complexity of linear search is $O(n)$, where n is the number of elements in the array. It is suitable for small datasets or unsorted arrays due to its simplicity.

2.5 Binary Search

Binary search is a more efficient search algorithm that works on sorted arrays. It repeatedly divides the search interval in half until the target element is found or the search interval is empty.

2.5.1 Java Code

The Java code for binary search is provided below:

Listing 5: Binary Search implementation in Java

```
86  class Searchs {
87      public static int binarySearch(int[] data, int key) {
88          int lo = 0;
89          int hi = data.length - 1;
90
91          while (hi - lo > 1) {
92              int mid = (hi + lo) / 2;
93              if (data[mid] < key) {
94                  lo = mid + 1;
95              } else {
96                  hi = mid;
97              }
98          }
99
100         if (data[lo] == key) {
101             return lo;
102         } else if (data[hi] == key) {
103             return hi;
104         }
105         return -1;
106     }
107 }
```

2.5.2 Java Code Explanation

Here's a breakdown of how it works:

- The `binarySearch` method takes two parameters: an array `data` containing sorted integers and an integer `key` representing the value to be searched.
- It initializes two pointers `lo` and `hi` to the start and end indices of the array, respectively.

- Inside the `while` loop, it repeatedly divides the search range in half by calculating the middle index `mid`.
- If the value at the middle index `data[mid]` is less than the target `key`, it updates the lower bound `lo` to `mid + 1`, indicating that the target must be in the upper half of the search range.
- If the value at `data[mid]` is greater than or equal to the target `key`, it updates the upper bound `hi` to `mid`, indicating that the target must be in the lower half of the search range or at the current middle index.
- The loop continues until the difference between `hi` and `lo` is less than or equal to 1, indicating that the search range has converged to a single element or an empty range.
- Finally, it checks if the target `key` is equal to the values at the indices `lo` and `hi`. If found, it returns the index where the target is located; otherwise, it returns -1 to show that the target is not present in the array.

This implementation of binary search efficiently finds the target element in a sorted array by iteratively narrowing down the search range to half at each step, resulting in logarithmic time complexity.

2.5.3 Analysis

The time complexity of binary search is $O(\log n)$, where n is the number of elements in the array. It is highly efficient for sorted arrays and performs significantly better than linear search for large datasets.

In the provided code snippets, line numbers can be referenced as follows:

- For Linear Search: Line 4
- For Binary Search: Line 5

3 Results, Analysis, Discussion

3.1 Tables

Analyzing the algorithms based on provided time/size tables.

Running time test results for sorting algorithms are given in Table 1.
Running time test results for search algorithms are given in Table 2.

Table 1: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Insertion sort	0.24486	0.0999	0.29542	1.2826	4.1388	18.44026	79.36842	391.41963	1692.13158	6861.58001
Merge sort	0.10093	0.14548	0.26135	0.48349	0.91205	2.15677	3.85059	8.25781	16.28055	33.41716
Counting sort	424.94993	253.60371	116.43216	112.53989	112.84425	116.68317	118.05403	117.25481	116.90894	123.24882
	Sorted Input Data Timing Results in ms									
Insertion sort	0.00283	0.0018	0.00309	0.00656	0.01145	0.02384	0.04033	0.07854	0.15711	0.3276
Merge sort	0.0167	0.0353	0.08233	0.1683	0.40536	0.81763	1.27529	3.28301	5.86855	12.6819
Counting sort	117.51954	116.5324	120.26702	126.31494	116.16109	117.16179	127.09802	126.4803	126.29727	119.31273
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.04939	0.18056	0.61769	2.51276	10.67548	39.85905	181.22694	758.34311	3343.83456	11690.19515
Merge sort	0.01655	0.03694	0.07913	0.18747	0.39976	0.81576	1.39146	2.94291	5.60733	11.40305
Counting sort	128.91076	125.0691	135.42614	131.35945	122.64889	125.4135	128.45922	122.30835	119.54729	123.61303

Table 2: Results of the running time tests of search algorithms of varying sizes (in ns).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	1088.9	664.7	867.3	1245.5	2041.5	4613.1	10681.5	20645.3	35495.3	52774.2
Linear search (sorted data)	1090.9	1241.5	1839.0	2522.6	3599.1	6023.0	10684.2	19922.6	36764.9	65328.5
Binary search (sorted data)	2684.0	2105.9	2559.4	2730.1	4573.8	5296.0	5156.8	5534.2	5568.6	6938.7

3.2 Analysis of Sorting Algorithms

The performance of sorting algorithms, crucial for organizing data efficiently, varies under different conditions:

1. **Insertion Sort:**

- When dealing with random input data, Insertion Sort exhibits a quadratic time complexity, leading to a significant increase in average time as the input size grows. This behavior is evident from the sharp rise in average time from 500 to 250,000 elements.
- However, when the input data is already sorted or reversely sorted, Insertion Sort's performance improves. In these cases, its time complexity tends towards linear, resulting in much lower average times compared to random input data.

2. **Merge Sort:**

- Merge Sort demonstrates consistent and efficient performance across different input sizes and data distributions. Its average time grows logarithmically with the input size, indicating its scalability and suitability for handling large datasets.
- The performance of Merge Sort remains relatively stable even when the input data is sorted or reversely sorted, showcasing its robustness and independence from the initial order of elements.

3. **Counting Sort:**

- Counting Sort excels when dealing with random input data, with relatively constant average times across different input sizes. This behavior arises from its linear time complexity, making it highly efficient for datasets with a limited range of values.

- However, the performance of Counting Sort slightly deteriorates on sorted or reversely sorted input data. This decline is attributed to the overhead associated with counting occurrences of each element, which becomes less effective when the data is partially ordered.

3.3 Analysis of Search Algorithms

The performance of search algorithms, particularly Linear Search and Binary Search, is crucial for efficient retrieval of elements from sorted datasets. The analysis of the results obtained from the experiments is as follows:

1. **Linear Search:**

- Linear Search exhibits consistent performance across different input sizes. Its average time increases linearly with the input size, which is expected due to its sequential scanning approach.
- The average time for Linear Search is slightly higher when applied to sorted input data compared to random input data. This is because, in the case of sorted data, Linear Search cannot take advantage of early termination, resulting in a marginal increase in search time.

2. **Binary Search:**

- Binary Search demonstrates efficient performance, with significantly lower average times compared to Linear Search. Its average time remains relatively stable across different input sizes, showcasing its logarithmic time complexity.
- Binary Search's performance is consistent regardless of the order of input data (sorted or random). This is a key advantage, as it allows for efficient retrieval of elements from large sorted datasets with predictable search times.

Overall, the choice of sorting and search algorithms depends on the specific requirements of the application, including the size and distribution of the input data, as well as the desired performance characteristics.

3.4 Complexity Analysis

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion Sort	$O(1)$
Merge Sort	$O(n)$
Counting Sort	$O(n+k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

3.5 Complexity Analysis Results

The results of the experiments give us valuable insights into how the sorting and search algorithms perform. Here are some important observations and interpretations:

3.5.1 Sorting Algorithms

- **Insertion Sort:** It shows quadratic time complexity in both average and worst-case scenarios, which means it's not great for handling large datasets. However, it does a lot better when dealing with data that's already partially sorted.
- **Merge Sort:** It consistently performs well and scales up nicely, with a time complexity of $O(n \log n)$ in every situation. This makes it a dependable option for efficiently sorting large datasets.
- **Counting Sort:** It has linear time complexity when the input values fall within a limited range, which makes it super efficient for those kinds of situations. However, its performance might not be as great on partially ordered data because it has to do more counting, which can slow things down a bit.

3.5.2 Search Algorithms

- **Linear Search:** It does okay with small datasets, but its time complexity grows linearly, so it's not the best choice for big datasets. Also, its performance changes depending on whether the data is sorted or not - if it's sorted, the search times might be a bit higher.
- **Binary Search:** It gives you really fast search times with a logarithmic time complexity, no matter how the data is ordered. That's why it's perfect for big datasets that are sorted - you can count on it for quick and predictable searches.

Overall, the experimental results highlight the trade-offs between time complexity, adaptability, and efficiency for different algorithms. The choice of algorithm should be made based on the specific requirements and characteristics of the input data and desired performance metrics.

Sorting algorithms with random data. Fig: 1.

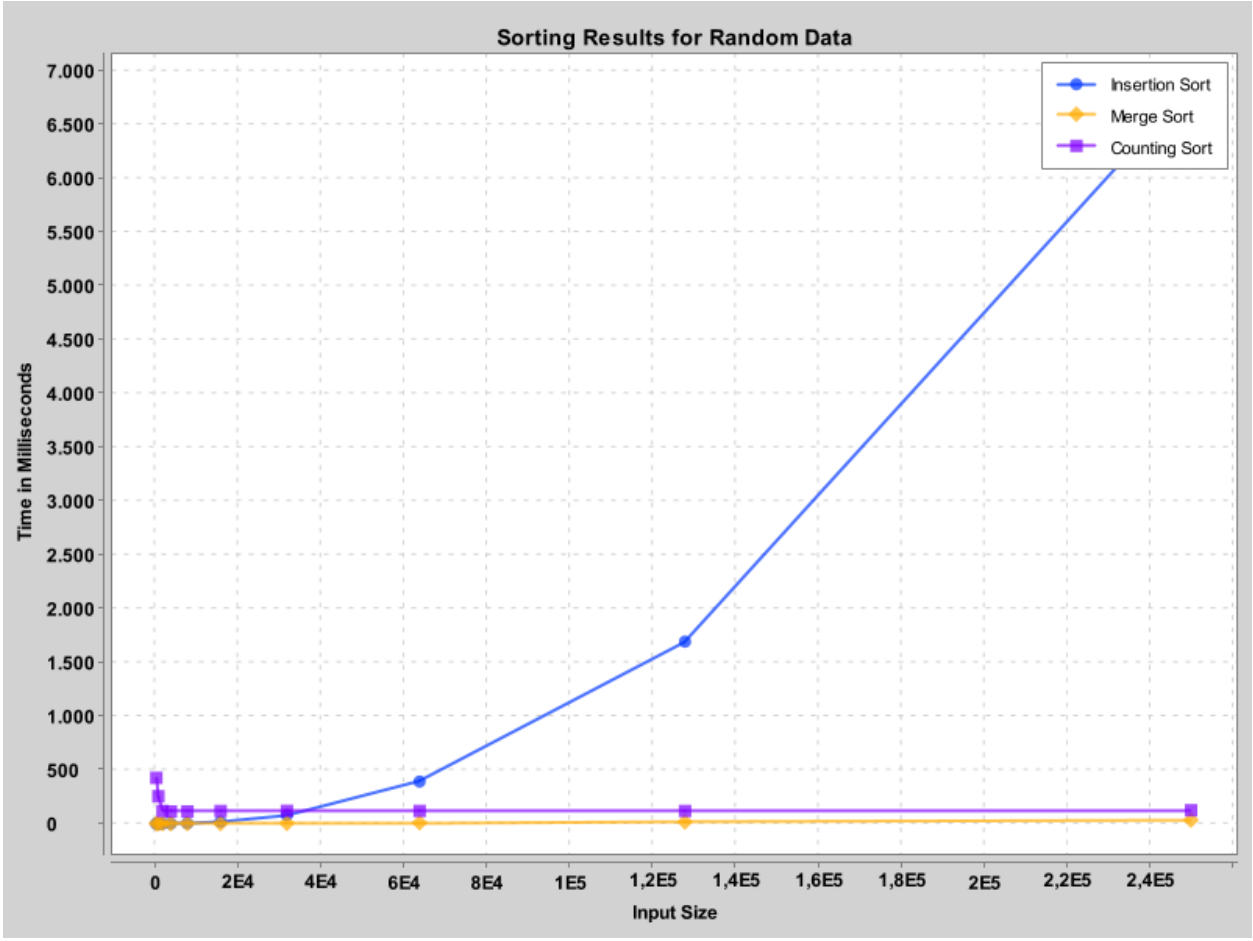

Figure 1: Plot of the size and time passed.

Sorting algorithms with sorted data. Fig. 2.

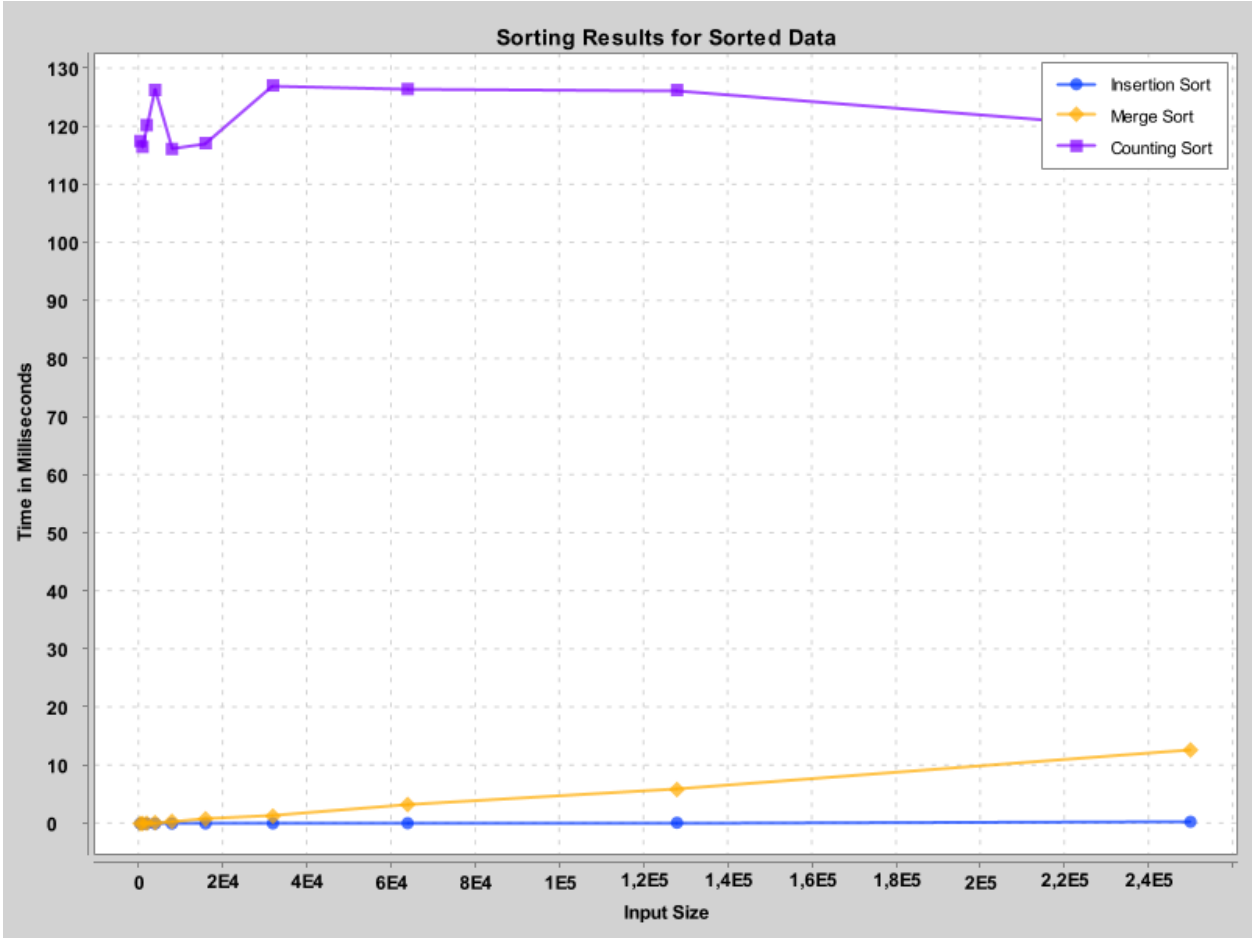

Figure 2: Plot of the size and time passed.

Sorting algorithms with reverse sorted data. Fig. 3.

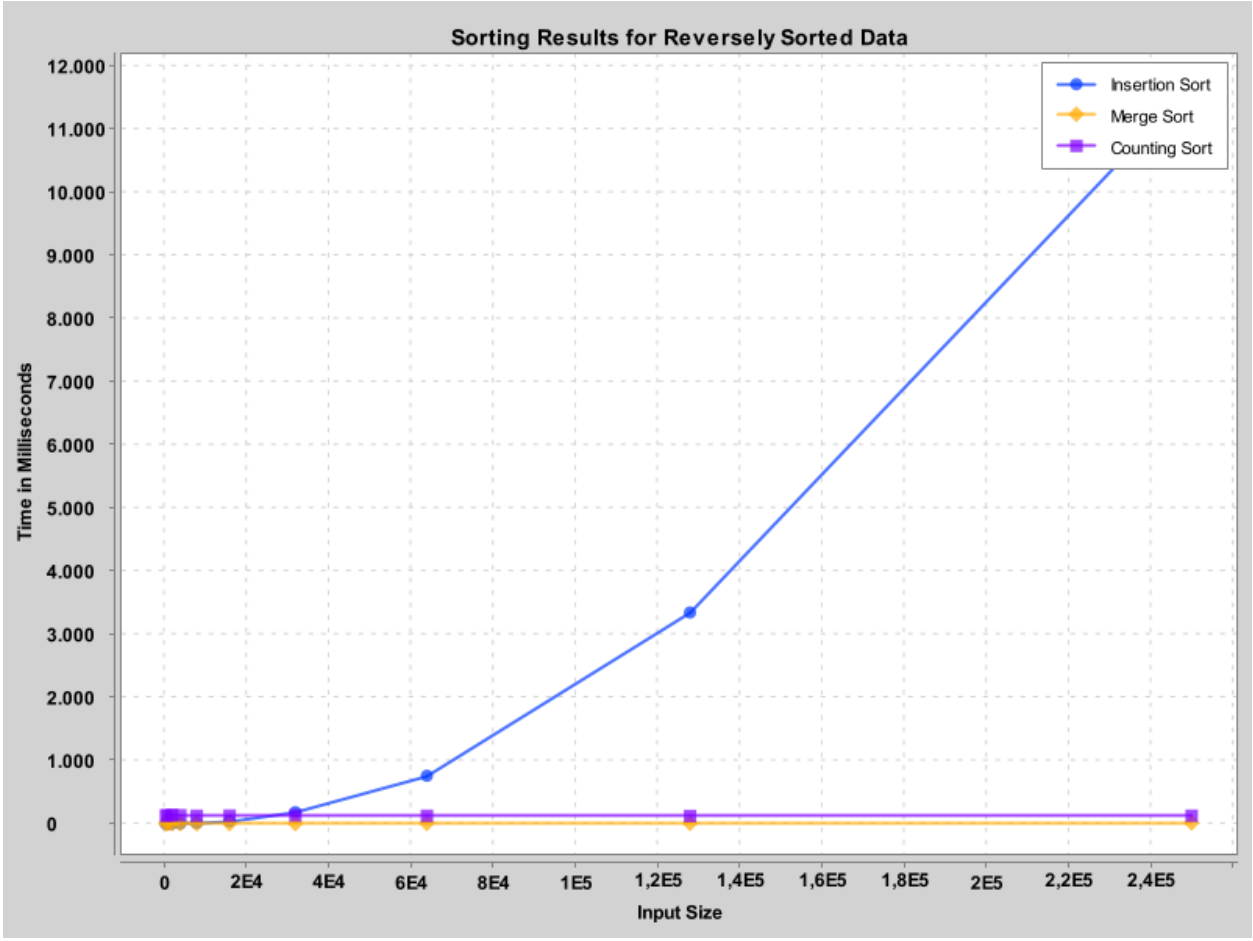

Figure 3: Plot of the size and time passed.

Search algorithms with sorted, random data. Fig. 4.

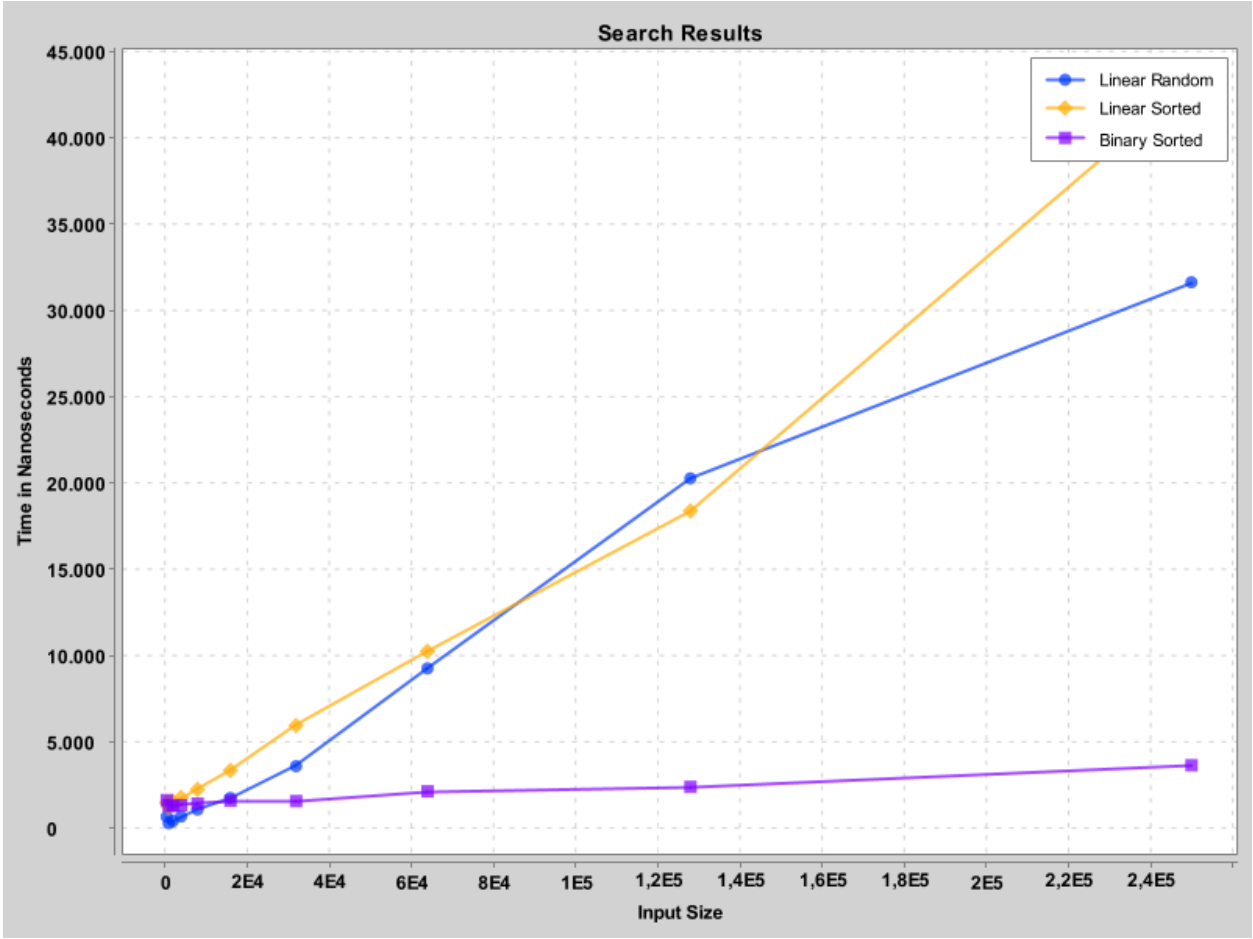

Figure 4: Plot of the size and time passed.

4 Results Analysis

4.1 Sorting Algorithms

4.1.1 Insertion Sort

When the input data is random, Insertion Sort's time complexity shoots up quadratically, causing a big increase in average time as the input size gets larger. You can see this clearly with the sharp rise in average time from 500 to 250,000 elements. However, if the input data is already sorted or in reverse order, Insertion Sort performs much better. In these cases, its time complexity leans towards linear, leading to significantly lower average times compared to random input data.

4.1.2 Merge Sort

Merge Sort shows consistent and efficient performance across various input sizes and types of data. As the input size increases, its average time grows logarithmically, which suggests that it scales well and can manage large datasets effectively. Even if the input data is already sorted or reversed, Merge Sort's performance remains relatively stable. This demonstrates its resilience and ability to handle different initial orders of elements without significant impact on its efficiency.

4.1.3 Counting Sort

Counting Sort performs well when dealing with random input data, maintaining relatively constant average times across various input sizes. This consistency is due to its linear time complexity, which makes it highly efficient for datasets with a limited range of values. However, Counting Sort's performance declines slightly when dealing with sorted input data. This decrease in efficiency is attributed to the overhead involved in counting occurrences of each element, which becomes less effective when the data is partially ordered.

4.2 Search Algorithms

4.2.1 Linear Search

Linear Search shows consistent performance across various input sizes, with its average time increasing linearly as the input size grows. This is expected since Linear Search follows a sequential scanning approach. Interestingly, when applied to sorted input data, the average time for Linear Search is slightly higher compared to random input data. This is because, with sorted data, Linear Search cannot take advantage of early termination, resulting in a marginal increase in search time.

4.2.2 Binary Search

Binary Search performs efficiently, with much lower average times compared to Linear Search. Its average time remains relatively stable across different input sizes, highlighting its logarithmic time complexity. One significant disadvantage of Binary Search is it needs sorted data.

References

- https://chat.openai.com/
- https://www.wikipedia.org/
- https://www.geeksforgeeks.org/
- BBM202
- https://www.programiz.com/