Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 20, 2024

Student name:
Emirhan Utku

Student Number:
2210765029

1 Problem Definition

The problem being addressed in this project revolves around the analysis of sorting and searching algorithms. Efficient sorting algorithms are crucial for enhancing the performance of various other algorithms, such as search and merge algorithms, that rely on sorted input data. With the increasing availability of vast amounts of data through modern computing and the internet, efficient searching through this data becomes paramount for computational tasks.

The objective of this assignment is to classify sorting and searching algorithms based on two primary criteria: computational (time) complexity and auxiliary memory (space) complexity. Computational complexity refers to the efficiency of an algorithm in terms of its runtime behavior concerning the size of the dataset. On the other hand, auxiliary memory complexity assesses the additional memory space required by sorting algorithms during their operation.

Through this project, it is expected to gain insight into the performance characteristics of various sorting and search algorithms by implementing them and analyzing their empirical performance against their theoretical asymptotic complexity.This involves experimenting with datasets of different sizes and characteristics to observe how empirical data align with expected growth functions.

2 Solution Implementation

2.1 Insertion Sort

```
1  public static int[] sort(int[] unSortedArray) {
2          for (int j = 1; j < unSortedArray.length; j++) {
3              int key = unSortedArray[j];
4              int i = j - 1;
5              while (i >= 0 && unSortedArray[i] > key) {
6                  unSortedArray[i + 1] = unSortedArray[i];
7                  i = i - 1;
8              }
9              unSortedArray[i + 1] = key;
10         }
11         return unSortedArray;
12     }
```

This Java code implements the insertion sort algorithm to sort an array of integers in-place. It iterates through the array, comparing each element with its predecessors and shifting elements to the right until finding the correct position. Once sorted, the array is returned.

2.2 Merge Sort

```java
public static int[] sort(int[] unSortedArray) {
        int n = unSortedArray.length;
        if (n<=1){return unSortedArray;}

        int mid = n / 2;
        int[] left = Arrays.copyOfRange(unSortedArray, 0, mid);
        int[] right = Arrays.copyOfRange(unSortedArray, mid, n);



        left= sort(left);
        right= sort(right);
        return Merge(left,right);
    }

    public static int [] Merge (int[] leftArray ,int[] rightArray){
        int [] result =new int[leftArray.length + rightArray.length];
        int i = 0, j = 0, k = 0;

        while (i < leftArray.length && j < rightArray.length) {
            if (leftArray[i] <= rightArray[j]) {
                result[k++] = leftArray[i++];
            } else {
                result[k++] = rightArray[j++];
            }
        }

        while (i < leftArray.length) {
            result[k++] = leftArray[i++];
        }

        while (j < rightArray.length) {
            result[k++] = rightArray[j++];
        }
        return result;
    }
```

This Java code implements merge sort to sort an array of integers. It recursively divides the array, sorts each half independently, and merges them. The sorting method calls itself for subarrays, while merging combines sorted arrays. The process repeats until the entire array is sorted, returning the result.

2.3 Counting Sort

```
public static int[] sort(int[] unSortedArray) {
        int k = findMax(unSortedArray);

        int[] count = new int[k + 1];
        for (int i = 0; i < unSortedArray.length; i++) {
            count[unSortedArray[i]]++;
        }

        for (int i = 1; i < count.length; i++) {
            count[i] += count[i - 1];
        }

        int[] output = new int[unSortedArray.length];

        for (int i = unSortedArray.length - 1; i >= 0; i--) {
            output[count[unSortedArray[i]] - 1] = unSortedArray[i];
            count[unSortedArray[i]]--;
        }
        return output;
    }
    private static int findMax(int[] arr) {
        int max = Integer.MIN_VALUE;
        for (int num : arr) {
            if (num > max) {
                max = num;
            }
        }
        return max;
    }
```

This Java code implements counting sort to sort an array of integers. It finds the maximum value, initializes a count array for element occurrences, calculates cumulative frequencies, and fills the output array accordingly.

2.4 Linear Search

```
static int search (int[] array, int desiredValue){
        int size = array.length;
        for (int i = 0; i < size; i++) {
            if (array[i] == desiredValue) {
                return i;
            }
        }
        return -1;
    }
```

This Java code defines a method search to find a desired integer value within an array. It iterates through the array, returning the index if found, else -1.

2.5 Binary Search

```
88  public static int binarySearch(int[] sortedArray, int desiredValue) {
89          int low = 0;
90          int high = sortedArray.length - 1;
91
92          while (high - low > 1) {
93              int mid = (high + low) / 2;
94              if (sortedArray[mid] < desiredValue) {
95                  low = mid + 1;
96              } else {
97                  high = mid;
98              }
99          }
100
101         if (sortedArray[low] == desiredValue) {
102             return low;
103         } else if (sortedArray[high] == desiredValue) {
104             return high;
105         } else {
106             return -1;
107         }
108     }
```

This Java code defines a method binarySearch to find a desired integer value within a sorted array. It uses a binary search algorithm by iteratively narrowing down the search range until the desired value is found or the search range becomes empty. The method returns the index of the desired value if found, otherwise -1 is returned.

3 Results, Analysis, Discussion

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	0	0	0	3	13	50	223	864	3619
Merge sort	0	0	0	0	0	1	2	4	9	20
Counting sort	97	95	95	96	96	94	96	97	98	100
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	0
Merge sort	0	0	0	0	0	0	0	1	3	8
Counting sort	91	95	95	95	95	94	96	96	97	96
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	1	6	25	100	411	1649	6283
Merge sort	0	0	0	0	0	0	0	1	3	8
Counting sort	90	96	95	95	96	94	96	96	98	100

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)	1912	945	195	310	623	813	2184	4086	7917	13223
Linear search (sorted data)	842	855	210	375	726	1460	2898	5603	11850	21968
Binary search (sorted data)	613	129	122	94	86	90	119	110	119	121

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	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

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

Each algorithm's best, average, and worst-case complexities are essential for understanding their performance characteristics. Insertion sort exhibits linear time complexity in the best case, quadratic in the average and worst cases. Merge sort maintains logarithmic time regardless of initial order. Counting sort achieves linear time when the range of elements is small. Linear search operates in constant time for best-case scenarios, but linearly for average and worst cases. Binary search, while constant in the best case, logarithmically searches even in the worst case due to its halving approach. These complexities guide algorithm selection based on dataset size and characteristics.

Insertion Sort:

- No additional space is allocated in this algorithm. It operates in-place, so the auxiliary space complexity is $O(1)$.

Merge Sort:

- Line 6: `left ← A[1]...A[n/2]`
- Line 7: `right ← A[n/2+1]...A[n]`
- Line 13: `C: array`
- These lines indicate the creation of additional arrays (`left`, `right`, `C`) for storing partial results during the merge sort process. Thus, the auxiliary space complexity is $O(n)$.

Counting Sort:

- Line 2: `count ← array of` $k+1$ `zeros`
- Line 3: `output ← array of the same length as A`
- These lines allocate additional space for the `count` array and the `output` array. The size of these arrays depends on the range of elements in the input array, denoted by k. Therefore, the auxiliary space complexity is $O(k+n)$.

Linear Search:

- No additional space is allocated in this algorithm. It operates using only a constant amount of additional space for storing loop counters and temporary variables. Thus, the auxiliary space complexity is $O(1)$.

Binary Search:

- No additional space is allocated in this algorithm. It operates using only a constant amount of additional space for storing loop counters and temporary variables. Thus, the auxiliary space complexity is $O(1)$.

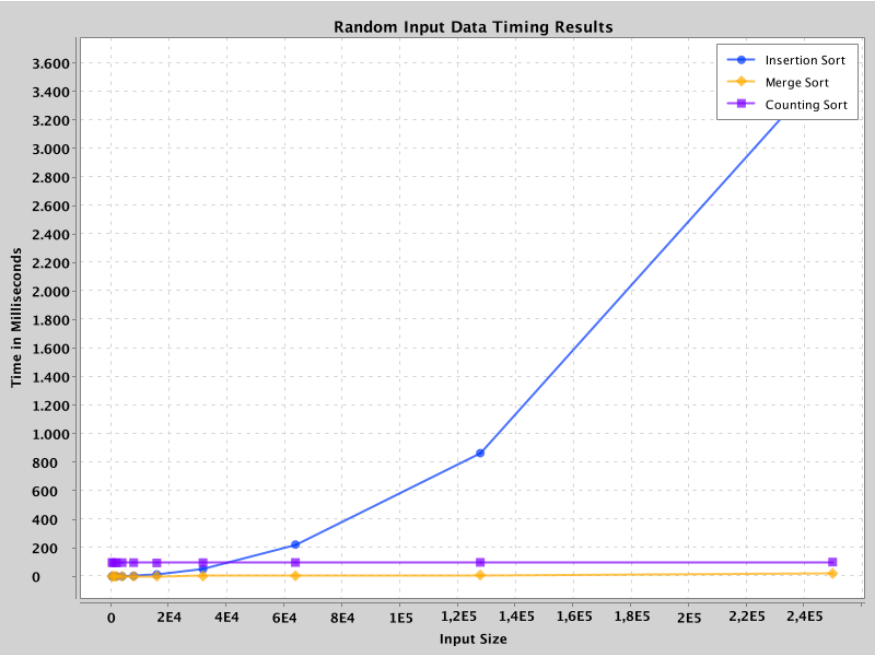

Figure 1: Tests on Random Input Data

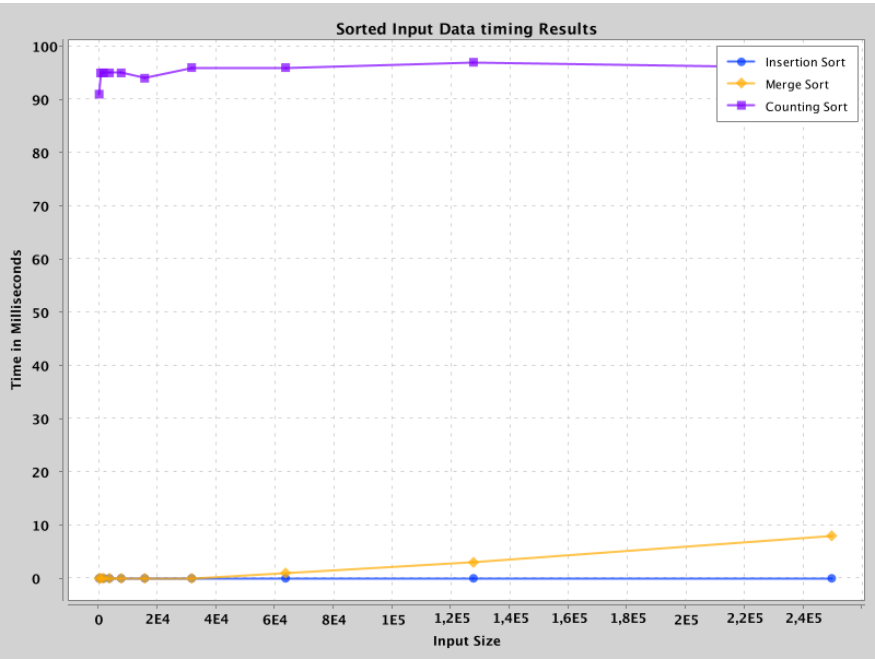

Figure 2: Tests on Sorted Input Data

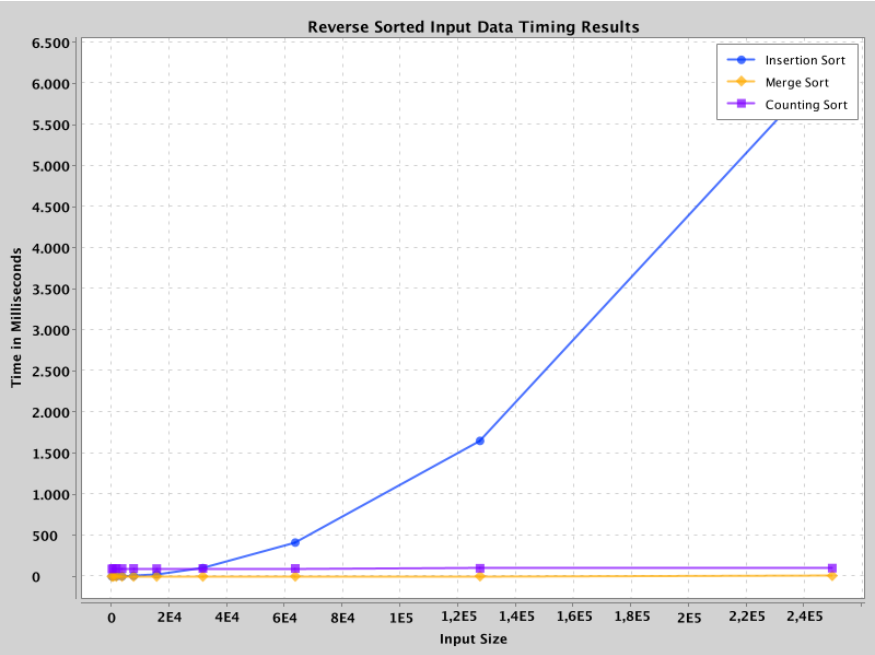

Figure 3: Tests on Reverse Sorted Input Data

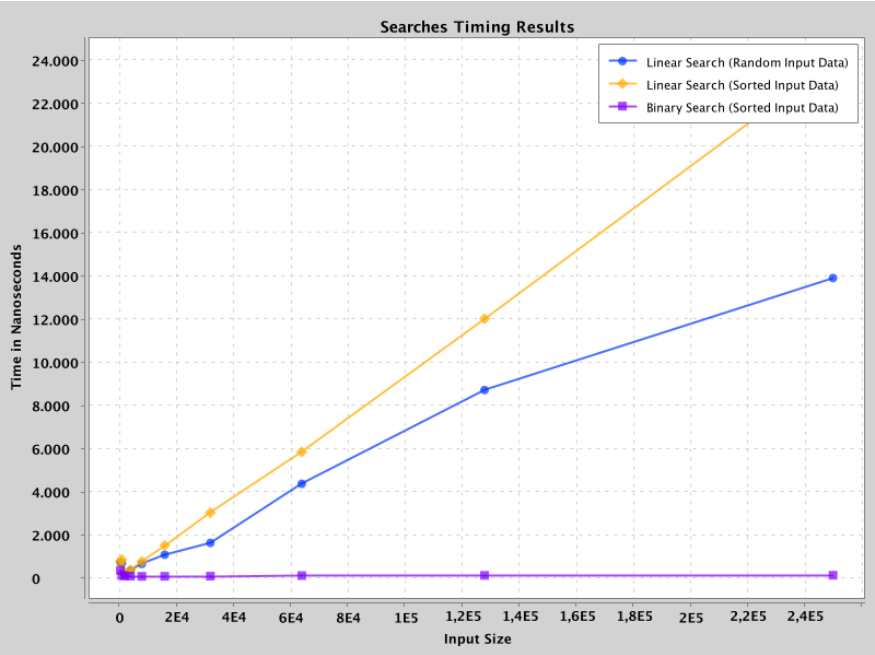

Figure 4: Tests on Searches

4 Notes

Insertion Sort:

- Best Case: $O(n)$ - Occurs when the array is already sorted. In this case, the inner loop will terminate early without any swaps.
- Average Case: $O(n^2)$ - Occurs when elements are randomly ordered.
- Worst Case: $O(n^2)$ - Occurs when the array is in reverse order. In this case, each element has to be compared and moved to the beginning of the array.

Merge Sort:

- Best Case: $O(n \log n)$ - Occurs regardless of the initial order of the elements.
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$

Counting Sort:

- Best Case: $O(n+k)$ - Occurs when the range of input elements (k) is relatively small compared to the number of elements (n).
- Average Case: $O(n + k)$
- Worst Case: $O(n + k)$

Linear Search:

- Best Case: $O(1)$ - Occurs when the desired element is found at the beginning of the array.
- Average Case: $O(n)$ - Occurs when the desired element is randomly located within the array.
- Worst Case: $O(n)$ - Occurs when the desired element is not present in the array or is located at the end of the array.

Binary Search:

- Best Case: $O(1)$ - Occurs when the desired element is located at the middle of the sorted array.
- Average Case: $O(\log n)$ - Occurs when the desired element is randomly located within the sorted array.
- Worst Case: $O(\log n)$ - Occurs when the desired element is not present in the array or is located at the beginning or end of the sorted array.

Conclusion: The choice of sorting and search algorithms significantly impacts data organization and retrieval efficiency. For smaller datasets, insertion sort proves effective, while as dataset size increases, merge sort or specialized algorithms like counting sort become preferable. Binary search stands out for its efficiency across dataset sizes, particularly with sorted data. Additionally, while sorted data generally leads to faster execution times, anomalies like longer runtimes with counting sort may stem from high data ranges. Overall, understanding algorithm characteristics is crucial for optimizing data processing tasks effectively.