Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 20, 2024

Student name:
Zaina Abushaban

Student Number:
b2210765053

1 Problem Definition

Complexity analysis in Computer Science is a vital process to understand the efficiency of algorithms. Taking a deeper look with this analysis, we acquire a quantitative understanding of the time and space requirements for our operations, helping us make smarter decisions and choose the most suitable algorithm for our goals and resources. To do this, Here's the general structure that was followed:

- All the algorithms were written, separated in class by the type of algorithm.
- Class GenerateFromCSV was created, in there we have a function that reads the input file, and produces an array of integers from the specified column, the length of this array is determined by an input parameter.
- Class Tester was created, in there we have multiple functions, the most important 2 functions are the functions that run our tests on the search and sort algorithms and return multiple dimensional arrays that include all the info we need.
- Lastly, in main, we call the test functions, and call the plotting function on the results, to get the final charts.

2 Solution Implementation

2.1 Sorting Algorithms

2.1.1 Insertion Sort

Insertion sort is one of the simplest sort algorithms that is based on comparisons, in the implementation, we have a nested loop, which creates the quadratic complexity, the basic idea is to compare each element with the previous elements, the element will keep switching until it reaches an element that is smaller than it, which created a sorted array from the front.

```
1  public static int[] inserstionSort(int[] array) {
2          int arrayLen = array.length;
3          for (int j = 1; j < arrayLen; j++) {
4              int val= array[j];
5              int i = j-1;
6              //We add the equal sign in case the 0th element is not in its
                   right place:
7              while (i>=0 && array[i]>val){
8                  array[i+1]=array[i];
9                  i=i-1;
10             }
11             array[i+1]= val;
12         }
13         return array;}
14 }
```

2.1.2 Merge Sort

Merge Sort is a more efficient, also comparison based sorting algorithm. It is based on a recursive function that uses a divide and conquer approach, by dividing the array into smaller and smaller partitions, and then merging them together again, but in order. The idea here is that since the algorithm is stable, we won't have to do as many comparisons while we make our way up.

```
15      public static int[] merge(int[] left, int[] right) {
16          int[] result = new int[left.length + right.length];
17          int leftIndex = 0, rightIndex = 0, resultIndex = 0;
18
19          while (leftIndex < left.length && rightIndex < right.length) {
20              if (left[leftIndex] <= right[rightIndex]) {
21                  result[resultIndex++] = left[leftIndex++];
22              } else {
23                  result[resultIndex++] = right[rightIndex++];
24              }
25          }
26
27          while (leftIndex < left.length) {
28              result[resultIndex++] = left[leftIndex++];
29          }
30
31          while (rightIndex < right.length) {
32              result[resultIndex++] = right[rightIndex++];
33          }
34
35          return result;
36      }
```

2.1.3 Counting Sort

Counting sort is a non-comparison based algorithm, it tackles the sorting problem in a different way, first we count the number of occurrences for each number in our array, then it preforms an accumulative sum on this list of occurrences, each element at index i of count will hold the number of elements less than or equal to i. Now we start iterating over the original list, putting each index in its correct spot according to how many entries should precede it, every time we add one we delete one of its counts from the count list, until we've gone through the whole list and placed everything correctly.

```
38      public static int[] countingSort(int[] array, int max) {
39
40          //This way we'll have an index for each element, and also an index for
                zero.
41          int[] count = new int[max+1];
42          int size = array.length;
43          int[] result = new int[size];
```

```
44
45          //Initializing our count array
46          for (int i = 0; i < size; i++) {
47              int j = array[i];
48              count[j]++;
49          }
50
51          //Now we do the accumulative sum for the count array:
52          for (int i = 1; i < max+1; i++) {
53              count[i]=count[i]+count[i-1];
54              };
55
56          //Now we shift our count list to the right by one, we do this by
                starting from the end:
57          //This reverse iteration helps maintain the stability of the sort,
                meaning that equal elements retain their original order relative
                to each other.
58          for (int i = size-1; i >= 0; i--) {
59              int num = array[i];
60              //decrement how many we have left:
61              count[num]= count[num]-1;
62              //Then we just set that place in the result list to have that
                    number
63              result[count[num]]=array[i];
64          }
65          return result;
66          };
```

2.2 Search Algorithms

2.2.1 Linear Search

Linear search is the simplest search algorithm, it sequentially checks each element in the array comparing it to the target element, it stops once it has found a match.

```
67  public static int linear(int[] array,int x) {
68          int size = array.length;
69          for (int i=0; i < size; i++){
70              if (array[i]== x){
71              return i;}
72          }
73          return -1;
74      };
```

2.2.2 Binary Search

Binary search is a more efficient sorting algorithm, it follows an approach of splitting our search field in half each iteration, which results in a log(n) complexity. There is a prerequisite for this search, that is the array be sorted.

```
75  public static int binary(int[] array, int x) {
76          int low = 0;
77          int high = array.length - 1;
78          while (low <= high) {
79              int mid = low + (high - low) / 2;
80              if (array[mid] == x) {
81                  return mid;
82              } else if (array[mid] < x) {
83                  low = mid + 1;
84              } else {
85                  high = mid - 1;
86              }
87          }
88          return -1;
89      }
```

3 Hypothesis

3.1 Sorting Algorithms:

3.1.1 Insertion Sort

Time Complexity In the best case, since the data is already sorted, the algorithm will just pass by each number, making one comparison every step, this is why we expect it to be $Omega(n)$

complexity. As for average case and worst case, we have an outer loop and inner loop, in the outer loop we're running n times (for each number), as for the inner loop, on average case we need to re-sort n/2 elements in the array, and up to n-1 element in worse case, in both scenarios the multiplication of the outer loop for all the element and the inner loop for each leads to $O(n^2)$.

$$\text{Total comparisons (worst case)} = 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

$$\text{Total comparisons (average case)} \approx \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$$

Space Complexity Since insertion is an in place sorting algorithm, we did not use any extra arrays during our implementation, which means its space complexity is $O(1)$.

3.1.2 Merge Sort

Time Complexity In merge sort, we are dealing with two steps, first dividing the array, this division results in multiple layers (recursion layers), these layers are all the same array, but with different distributions, so each of the layers has n elements that we need to iterate through once to sort them correctly. Which leads us to log(n) layers * n layers, resulting in an $\Theta(n \log n)$. Since this doesn't change according to how the data is sorted, we expect all cases to be $\Theta(n \log n)$.
Space Complexity As seen in the implementation above, we need extra space for an array, this space for temporary arrays is reused across different parts of the algorithm resulting in a $O(n)$ space complexity.

3.1.3 Counting Sort

Time Complexity Since counting sort is not comparison based, we don't expect the complexity change according to how the data is sorted, it's just a simple set of rules being followed, first we iterate over the array to count the elements which takes $O(n)$ time, then we iterate over the count list to start arranging the final array, let's assume the max number in our array is k, so this step would take $O(k)$ time, adding these together results in k+n time, which means we can expect $\Theta(n)$ time complexity over all cases.
Space Complexity In the above implementation, we are creating two extra arrays, one to store the result and one to store the counts array, which means in total we need $O(n + k)$ space complexity.

3.2 Search Algorithms:

3.2.1 Linear Search

Time Complexity Here, we can easily detect, best, worst, and average cases, best is if the first element is what we're looking for which is $O(1)$, average is if it's in the middle which is n/2 so $O(n)$, and worse is if it's at the end $O(n)$. In our random trials we should expect to see average case, so $O(n)$.
Space Complexity No extra space is used to store anything, so our space complexity is $O(1)$.

3.2.2 Binary Search

Time Complexity Binary search also has a best case scenario of $O(1)$, where the middle of the list is exactly what we're looking for, then average and worst case are the same, of around $O(log(n))$ time. We expect to see that in our results.
Space Complexity Also, no extra space is used, so space complexity is $O(1)$.

4 Results Analysis

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

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	0.40	0.10	0.20	0.90	3.50	14.00	54.60	236.30	946.60	3540.00
Merge sort	0.00	0.00	0.10	0.30	0.40	1.00	2.40	4.70	9.60	20.50
Counting sort	93.60	90.40	90.60	90.40	89.80	90.00	90.80	98.40	93.20	97.90
	Sorted Input Data Timing Results in ms									
Insertion sort	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.10	0.30
Merge sort	0.00	0.00	0.00	0.10	0.10	0.60	0.60	2.00	3.80	7.80
Counting sort	91.70	89.80	90.00	89.70	89.50	89.50	89.30	90.70	93.30	92.90
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.00	0.00	0.40	1.60	6.20	24.30	97.70	408.10	1632.10	6211.20
Merge sort	0.00	0.00	0.00	0.00	0.30	0.20	1.00	1.70	5.90	7.10
Counting sort	94.70	89.70	89.80	89.90	89.30	89.40	89.70	90.60	90.60	91.80

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

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)	1717.80	555.92	726.36	853.28	1153.53	1745.10	2685.66	4382.44	7720.73	14219.10
Linear search (sorted data)	335.82	192.32	286.40	453.53	799.16	1515.22	3003.20	5748.57	11126.79	22047.37
Binary search (sorted data)	466.55	321.72	259.51	176.39	176.54	183.97	200.96	209.25	233.46	254.06

Complexity analysis tables (Table 3 and Table 4):

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

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	$O(1)$

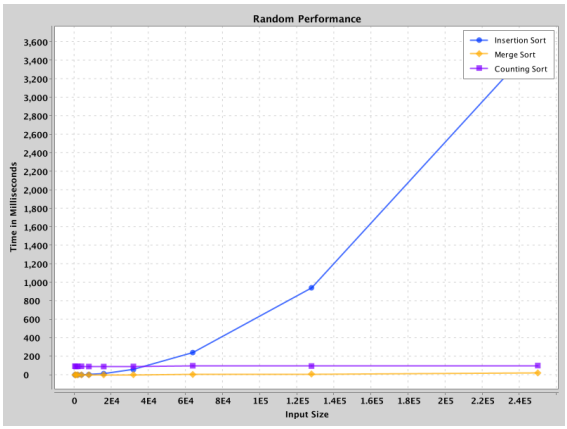

Figure 1: Random Performance

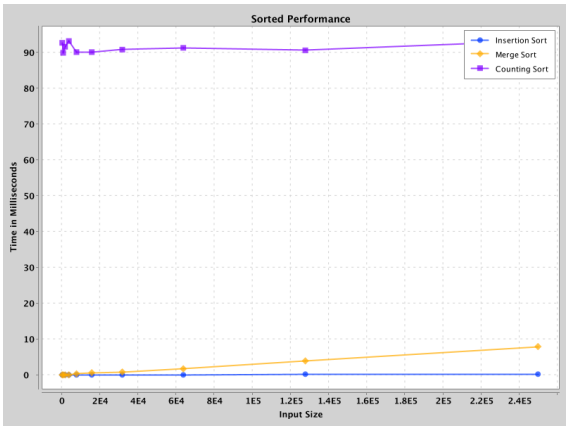

Figure 2: Sorted Performance

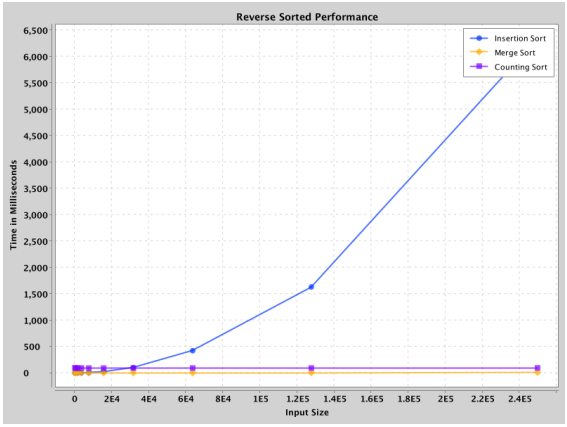

Figure 3: Reverse Sorted Performance

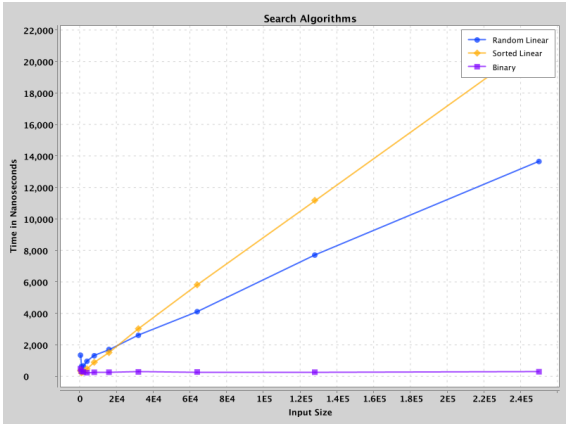

Figure 4: Search Algorithms

4.1 Random data results

As we expected, we got nlog(n) for merge sort, linear time for counting sort, and quadratic time for insertion sort.

4.2 Sorted data results

We also notice the same complexities here, except insertion sort, where we got linear time.

4.3 Reverse data results

Also, as expected, almost identical to random performance.

4.4 Search Algorithms

Here we got linear time for linear search in both random and sorted data, since it doesn't make a difference, the average on both scenarios is the middle of the list, as for binary, we notice a significantly smaller complexity with logarithmic time as we expected.

5 Notes

- It's important to note that while we analyzed the best, average, and worst cases in terms of input data order, for certain algorithms, the performance can be influenced by other factors, such as the distribution of the data. In such cases, the best and worst cases may not align with the traditional definitions based on data order.
- There is a noticeable difference between the graphs and numbers shown in this report and model graphs. These performance variations can stem from hardware capabilities, code optimizations, and the programming environment. Input data characteristics, system load, and non-considered constants in complexity analysis also play a role.

References

- https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-mergesort/
- https://www.shiksha.com/online-courses/articles/time-and-space-complexity-of-sorting-algorithms-blogId-152755