Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Adam Sattout

Student Number:
b2220765061

1 Problem Definition

In this assignment we are trying to compare how different sorting/searching algorithms work while looking at their time and space complexities using different dataset variations (sorted, reversely sorted or randomly sorted).

2 Solution Implementation

For sorting, we will use insertion, merge and counting sort. While for searching, we will use both linear and binary search

2.1 Insertion Sort

```java
1  public static void insertionSort(List<Integer> list)
2      {
3          for(int i = 1; i < list.size(); i++){
4              int key = list.get(i);
5              for(int j = i - 1; j >= 0; j--){
6                  if(key < list.get(j)){
7                      list.set(j + 1, list.get(j));
8                      list.set(j, key);
9                  }
10                 else break;
11             }
12         }
13     }
```

For the sorting algorithms, since we need to remove elements from arrays later on in merge sort 32 and to unify the input with all sorting algorithms, lists were chosen without an impact on algorithm's complexities.

2.2 Merge Sort

```java
15 public static List<Integer> mergeSort(List<Integer> list){
16         int n = list.size();
17         if(n <= 1){
18             return list;
19         }
20         List<Integer> left = new ArrayList<>(list.subList(0, n/2));
21         List<Integer> right = new ArrayList<>(list.subList(n/2, n));
22         left = mergeSort(left);
23         right = mergeSort(right);
24         return merge(left,right);
25     }
26
```

```
    public static List<Integer> merge(List<Integer> left, List<Integer> right)
        {
        List<Integer> merged = new ArrayList<>();
        while(left.size() != 0 && right.size() != 0){
            if(left.get(0) <= right.get(0)){
                merged.add(left.get(0));
                left.remove(0);
            }
            else{
                merged.add(right.get(0));
                right.remove(0);
            }
        }

        while(left.size() > 0){
            merged.add(left.get(0));
            left.remove(0);
        }
        while(right.size() > 0){
            merged.add(right.get(0));
            right.remove(0);
        }
        return merged;
    }
```

2.3 Counting Sort

```
        int k = getMax(list);
        int[] count = new int[k + 1];
        Integer[] output = new Integer[list.size()];
        int size = list.size();
        for(int i = 0; i < size; i++){
            int j = (int) list.get(i);
            count[j] =  count[j] + 1;
        }
        for(int i = 2; i < k+1; i++){
            count[i] = count[i] + count[i -1];
       }
        for(int i = size - 1; i >= 0; i--){
            int j = (int) list.get(i);
            count[j] = count[j] - 1;
            output[count[j]] = (int) list.get(i);
        }

       List<Integer> returnable = new ArrayList<>(Arrays.asList(output));
       return returnable;
   }
```

```java

    public static int getMax(List<Integer> list){
         int size = list.size();
         int max = 0;
         for(int i = 0; i < size; i++){
             if(list.get(i) > max) max = list.get(i);
         }
         return max;
    }
```

2.4 Linear Search

```java
public static int linearSearch(List<Integer> A, int x){
         for(int i = 0; i < A.size(); i++){
             if(A.get(i) == x) return i;
         }
         return -1;
    }
```

2.5 Binary Search

```java
public static int binarySearch(List<Integer> A, int x){
         int high = A.size() - 1;
         int low = 0;
         while(high - low > 1){
             int mid = (high + low)/2;
             if(A.get(mid) < x){
                 low = mid;
             }
             else{
                 high = mid;
             }
         }
         if(A.get(low) == x) return low;
         else if(A.get(high) == x) return high;
         else return -1;
    }
```

3 Results, Analysis, Discussion

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	2	2	5	28	113	535	1396	4471	20950	118616
Merge sort	0	0	1	2	6	21	75	281	1075	4075
Counting sort	234	268	263	266	202	152	133	136	140	146
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	1
Merge sort	0	0	1	2	7	23	79	274	1055	3962
Counting sort	131	131	132	132	132	133	133	135	139	140
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	2	10	49	235	806	2662	9427	37136	146888
Merge sort	0	0	0	1	6	20	72	274	1061	3987
Counting sort	133	132	134	132	133	133	137	139	138	141

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)	6182	1986	911	1429	1819	3150	6241	12434	24805	51958
Linear search (sorted data)	186	313	578	1139	2123	4315	8223	17093	32814	106502
Binary search (sorted data)	629	216	200	208	232	271	252	314	433	276

Complexity analysis tables to complete (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	$\Omega(1)$	$\Theta(logn)$	$O(logn)$

Table 4: Auxiliary space complexity of the given algorithms.

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

Here our Merge sort takes n space in the form of multiple sub lists together forming a list parallel to the main list: 22. Also our Counting sort makes the new array count 51 that has a size equal to the max element of the list.

In Figure 1 1, we can see how algorithms perform sorting a randomly sorted list, and as we can see, counting sort does the best while insertion sort is far worse having nearly $O(n^2)$ complexity

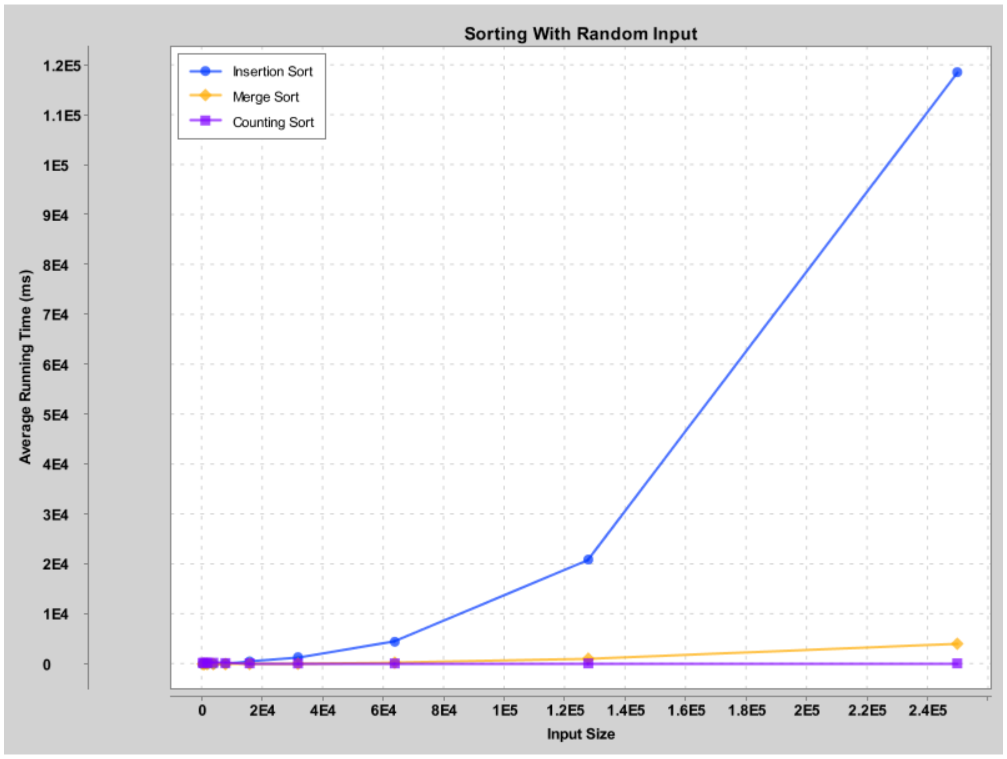

Figure 1: A sample plot showing running time results for varying input sizes for three sorting algorithms on randomly sorted list.

When the input is sorted, we can see (Figure 2 2) how insertion sort becomes even faster than counting sort since it has less computations and has same time complexity, while merge sort is the worse with $O(nlogn)$ complexity

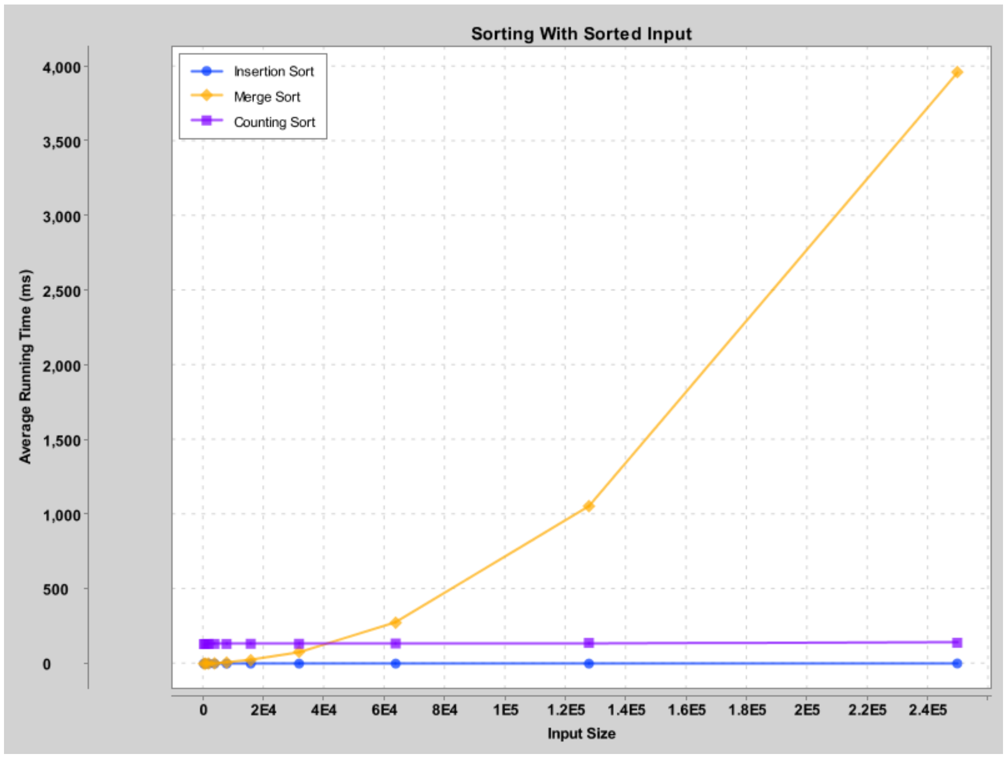

Figure 2: A sample plot showing running time results for varying input sizes for three sorting algorithms on naturally sorted list.

With reversely sorted input (Figure 3 3), we can see insertion sort becoming much worse than the other algorithms although we have a plot similar to Figure 1 where input is random (Figure 1 1).

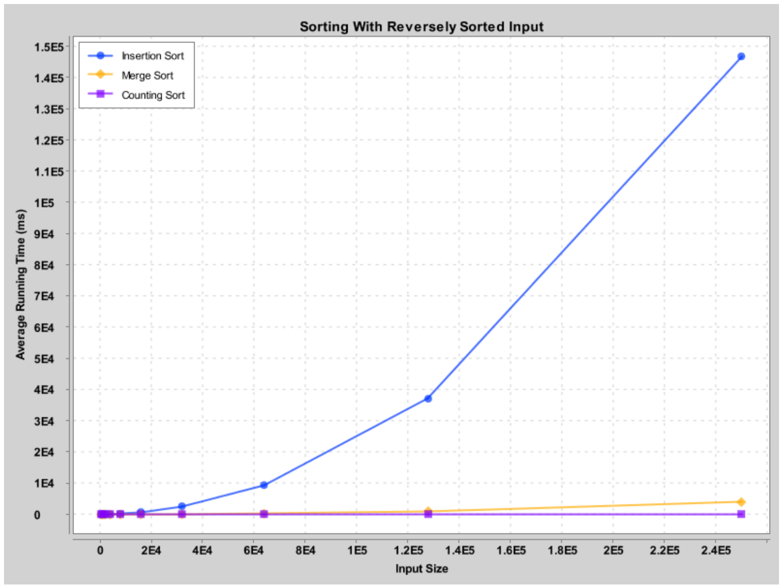

Figure 3: A sample plot showing running time results for varying input sizes for three sorting algorithms on reversely sorted list.

For searching algorithms (Figure 4 4) we can see how binary search out performs linear search while both sorted and random.

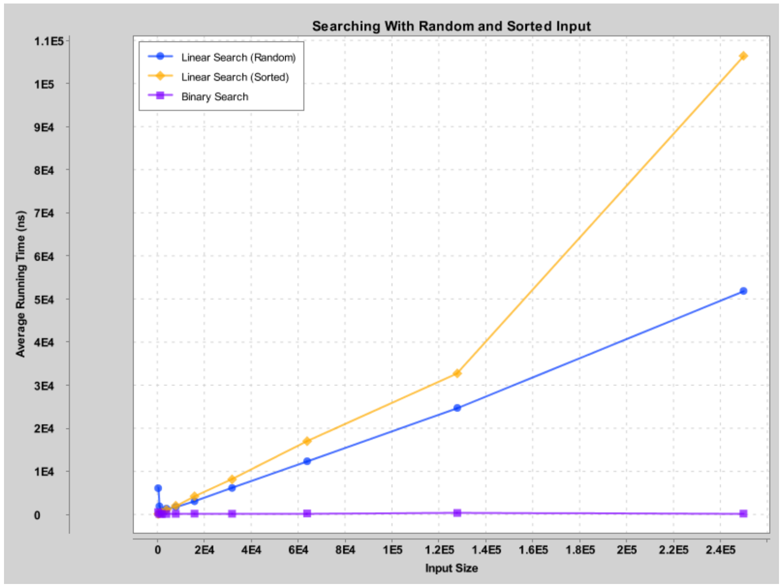

Figure 4: A sample plot showing running time results for varying input sizes for two searching algorithms on randomly/naturally sorted list.

Results analysis, explanations...

As we saw from our plots, most of the algorithms follow the time complexities in Table 3 except for some like Counting sort, which at first seems like it is O(1) but that's probably because of our dataset not being really a range dataset so our algorithm was running at its worst all the time due to getting a high number at the start, while we can see, even if by marginal differences between input sizes, more of a linear complexity when the input is sorted, but performance across all sorting ways is overall close. Merge sort is always $O(nlogn)$.
Insertion sort has its worst performance with reversely sorted and a bit better performance when input is randomly sorted with both having $O(n^2)$ complexity. However, if the list is already sorted or semi-sorted, insertion sort becomes the best with $O(n)$ complexity and lower overhead computation than Counting sort.

For searching algorithms, as expected, binary search performs greatly better than linear search having a complexity of $O(logn)$ (we dont have the best case of $O(1)$ since it wasn't implemented to stop when number is found in our function) compared to $O(n)$ (Figure 4). While linear search when sorted looks like its polynomial, We expect that with more experiments the plot will converge to a linear line.

References

- https://www.javatpoint.com/