Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 30, 2023

Student name:
Zeynep Yeşilkaya

Student Number:
b2210356048

1 Problem Definition

Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be sorted. Furthermore, modern computing and the internet have made accessible a vast amount of information. The ability to efficiently search through this information is fundamental to computation. The efficiency of a sorting algorithm can be observed by applying it to sort datasets of varying sizes and other characteristics of the dataset instances that are to be sorted. In this assignment, you will be classifying the given sorting and searching algorithms based on two criteria:

- Computational (Time) Complexity: Determining the best, worst and average case behavior in terms of the size of the dataset.
- Auxiliary Memory (Space) Complexity: Some sorting algorithms are performed "inplace" using swapping. An in-place sort needs only O(1) auxiliary memory apart from the memory used for the items being sorted. On the other hand, some algorithms may need O(log n) or O(n) auxiliary memory for sorting operations. Searching algorithms usually do not require additional memory space.

The main objective of this assignment is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities. It is expected to implement the algorithms given as pseudocodes, and perform a set of experiments on the given datasets to show that the empirical data follows the corresponding asymptotic growth functions. To reduce the noise in your running time measurements, consideration will have to be given to how it can be done, and the results will need to be plotted to demonstrate and analyze the asymptotic complexities

2 Solution Implementation

2.1 Quick Sort Algorithm

```
int[] sort(int[] arr){
    int low=0;
    int high=arr.length-1;
    int[] stack = new int[-low+high+1];
    int top=-1;
    stack[++top] = low;
    stack[++top] = high;
    while (top>=0){
        high = stack[top--]; //pop
        low = stack[top--];
        int pivot = partition(arr,low,high);

        if(pivot-1>low){ //that checks if there exist a left side of the array
            stack[++top] = low; //push
            stack[++top] = pivot-1;
        }
        if(pivot+1<high){ //that checks if there exist a right side of the
           array
            stack[++top] = pivot+1;
            stack[++top] = high; //push
        }
    }
    return arr;
}
int partition(int[] arr, int low, int high){
    int pivot = arr[high];
    int i = low - 1;

    for(int j=low;j<high;j++){
        if(arr[j] <= pivot){
            i++;
            int temp = arr[i];//swap
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i+1];//swap
    arr[i+1] = arr[high];
    arr[high] = temp;

    return i+1;
}
```

2.2 Bucket Sort Algorithm

```java

int[] sort(int[] arr){

    int n = (int)Math.sqrt(arr.length);

    int m = arr[0];
    for (int num:arr){
        m = Math.max(num,m);
    }

    final int max = m;

    Map<Integer, List<Integer>> buckets = new HashMap<>();

    IntStream.range(0,n).forEach(i -> buckets.put(i, new ArrayList<>()));//
        create buckets

    for(int num:arr){
        buckets.get(hash(num,max,n)).add(num);//add numbers to the buckets
    }
    buckets.values().forEach(Collections::sort);//sort the buckets

    return Arrays.stream(buckets.values().stream().flatMap(Collection::stream)
        .toArray())
    .mapToInt(o -> (int)o).toArray();//merge elements and return
}

int hash(int i, int max, int numOfBuckets){
    return i/max *(numOfBuckets-1);
}
```

2.3 Selection Sort Algorithm

```
int[] sort(int[] arr){

    for(int i=0;i<arr.length;i++){
        int minIndex = i;
        for(int j=i+1; j<arr.length; j++){
            if(arr[j]<arr[minIndex]) minIndex = j;
        }

        if(minIndex != i){
            int temp = arr[minIndex];
            arr[minIndex] = arr[i] ;
            arr[i] = temp;
        }
    }
    return arr;
}
```

2.4 Linear Search Algorithm

```
int search(int[] arr, int x){

    int n = arr.length;
    for(int i=0;i<n;i++){
        if(arr[i] == x) return i;
    }
    return -1;
}
```

2.5 Binary Search Algorithm

```
    int search(int[] arr, int x){

        int l=0;
        int r=arr.length-1;

        while (l<=r){
            int m = (r+l)/2;
            if(arr[m] == x) return m;
            else if(arr[m]<x) l=m+1;
            else if(arr[m]>x) r=m-1;
        }
        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									
Selection sort	0.4	0.5	0.9	4.0,	15.1	79.8	241.4	901.7	3634.1	13816.8
Quick sort	0.4	0.5	0.2	0.2	0.5	1.2	7.4	22.0	42.9	71.1
Bucket sort	1.2	1.0	1.2	1.6	3.0	4.3	8.4	13.6	32.8	85.1
	Sorted Input Data Timing Results in ms									
Selection sort	0.0	0.1	0.8	3.7	13.9	54.5	219.6	884.8	3719.4	13819.3
Quick sort	0.3	1.1	2.2	8.2	26.5	99.7	400.8	1615.4	6441.8	24571.2
Bucket sort	0.0	0.0	0.1	0.2	0.1	0.6	, 0.6	1.4	2.7	14.9
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.2	1.0	4.1	15.3	64.7	269.6	1102.3	4754.3	21671.2
Quick sort	0.3	0.4	0.9	2.5	6.4	12.4	38.4	165.0	635.2	1337.9
Bucket sort	0.0	0.0	0.0	0.0	0.1	0.4	0.7	1.5	2.7	6.7

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

Table 1 shows the results of running time tests for three sorting algorithms: Selection sort, Quick sort and Bucket sort. The tests were performed for different input sizes (ranging from 500 to 250,000). The results are presented in three categories based on the input data: random input data, sorted input data and reversely sorted input data. The values in the table are the time taken in milliseconds (ms) for each algorithm to execute for a given input size and data type.

Overall, the results show that Bucket sort is the fastest algorithm for all three input types. Bucket sort also performed well, especially for random and sorted input data. The execution time of all three algorithms increases as the input size increases. For the same input size, the execution time is highest for reversely sorted input data, followed by random input data and then sorted input data.

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)	254.0	555.0	1197.9	2080.6	4107.1	8347.7	13990.8	25460.5	46875.2	89826.1
Linear search (sorted data)	1685.6	844.8	1243.3	1934.5	2926.5	4108.6	8099.0	13394.2	27710.3	56297.8
Binary search (sorted data)	358.2	369.0	444.0	723.6	1456.0	1931.9	2628.6	2662.9	3356.2	3427.9

The table displays the running time (in nanoseconds) of three different search algorithms (linear search with random data, linear search with sorted data, and binary search with sorted data) for varying input sizes (500 to 250,000). The table indicates that for random data, linear search has the slowest running time, while binary search has the fastest running time.

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**
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Bucket Sort	$\Omega(n + k)$	$\Theta(n + n^2/k + k)$	$O(n^2)$
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**
Selection Sort	$O(1)$
Quick Sort	$O(n)$
Bucket Sort	$O(n + k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

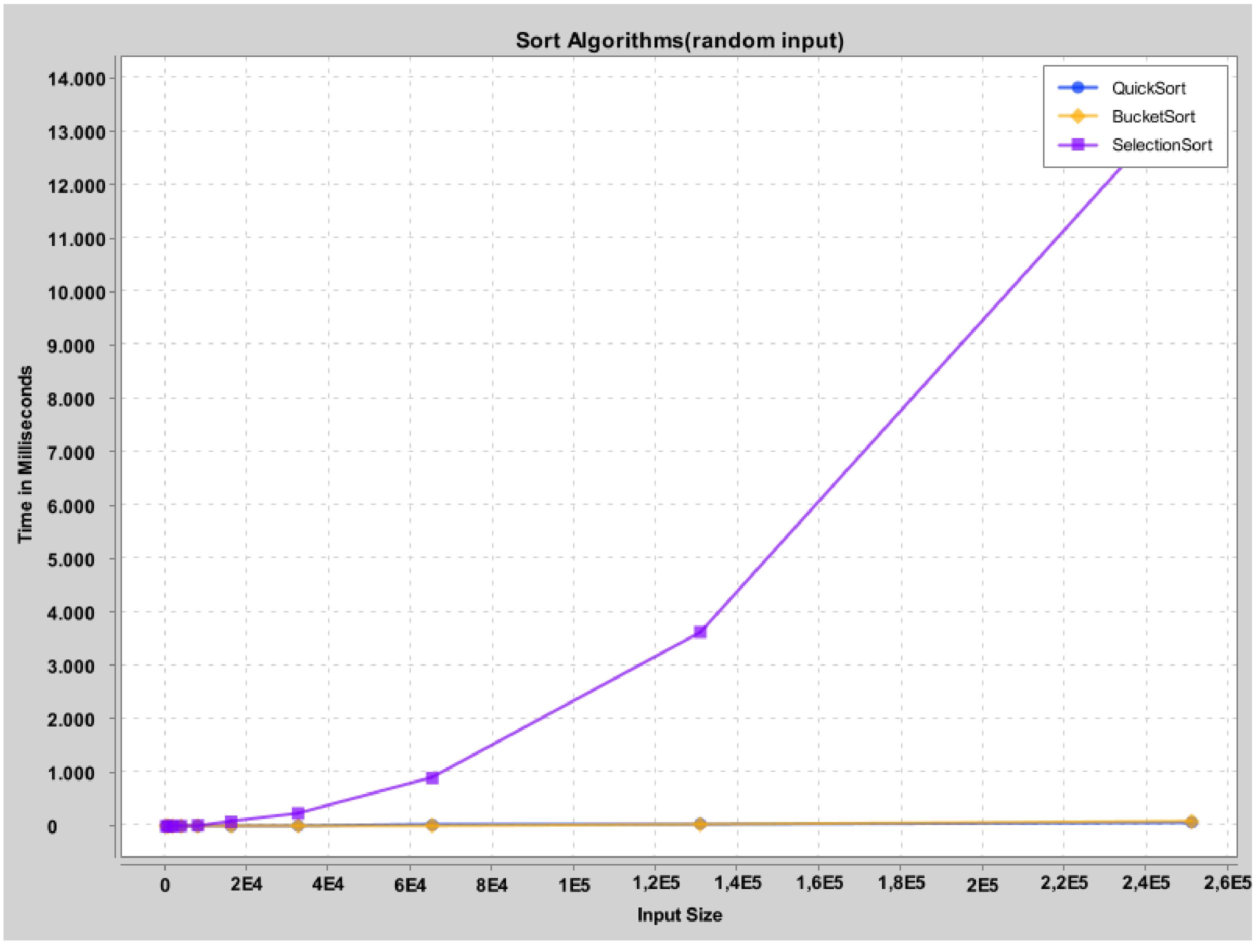

Figure 1: A smaller plot of the functions.

Figure 1 displays the performance of Quick Sort, Bucket Sort, and Selection Sort algorithms on a randomly generated array. Selection Sort exhibits the slowest performance, while Bucket Sort and Quick Sort appear to have similar performances that overlap with each other.

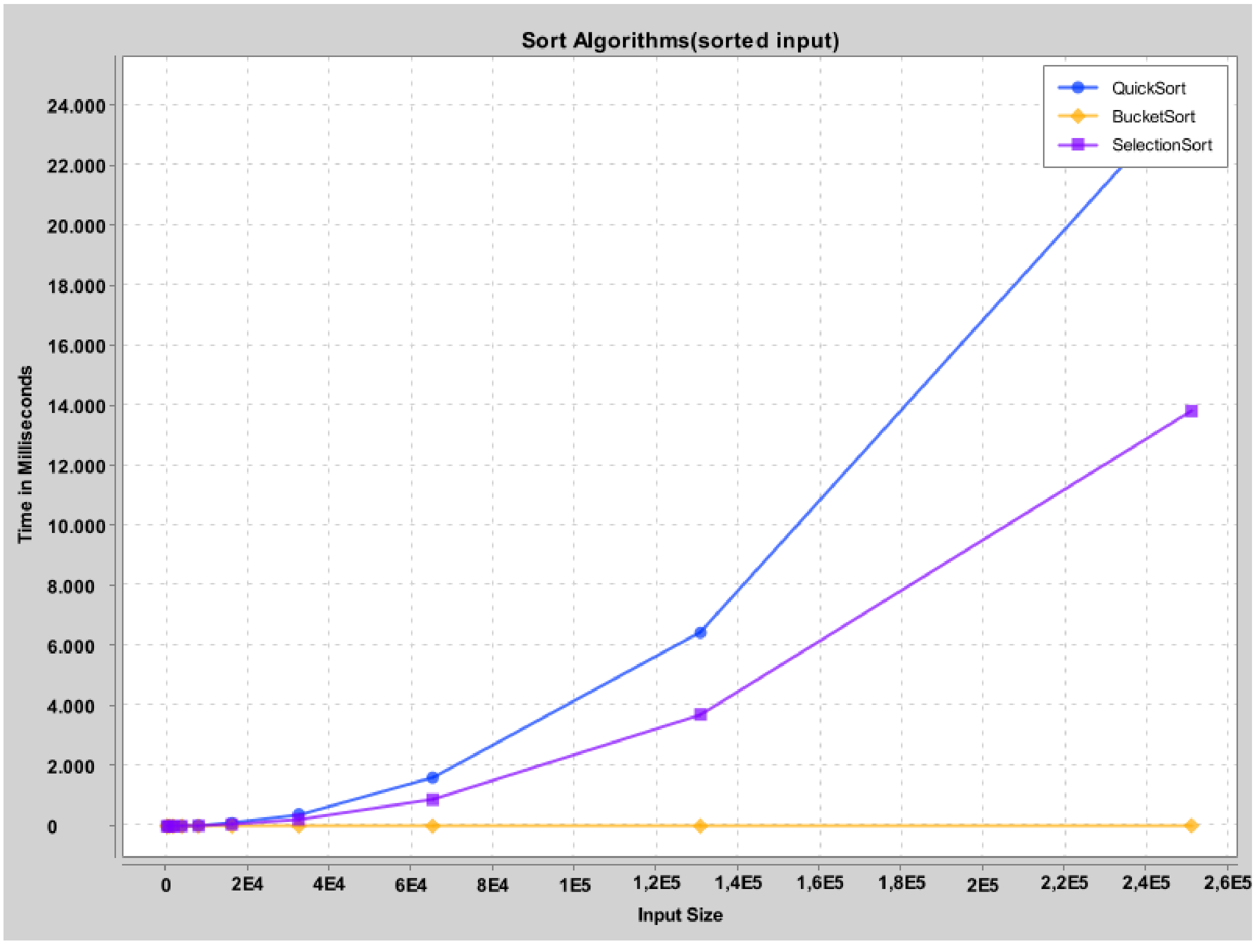

Figure 2: Plot of the functions.

Displayed in Figure4 is a comparison of the performance of Quick Sort, Bucket Sort, and Selection Sort algorithms on a sorted array. It is worth noting that Quick Sort displays the slowest performance due to its worst-case scenario. On the other hand, Selection sort and Bucket Sort show similar performance to that of the random array.

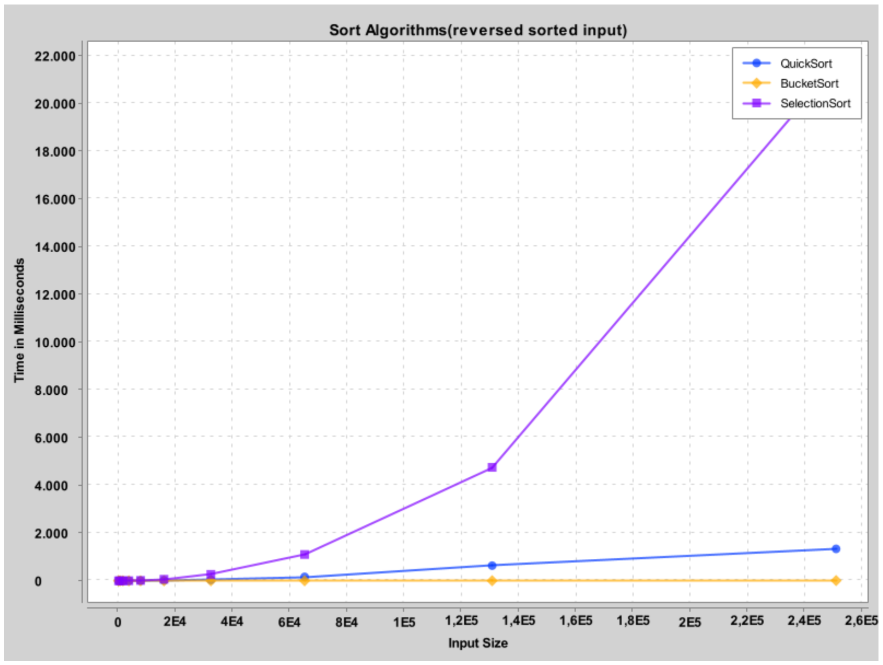

Figure 3: Plot of the functions.

Figure4 illustrates how the algorithms Quick Sort, Bucket Sort, and Selection Sort perform on an input array that is sorted in reverse order. Selection Sort exhibits the slowest performance as it is in its worst-case scenario. Both Quick Sort and Selection Sort have a worst-case time complexity of O(n^2) when sorting a reverse-sorted array. This is because in these cases, each element in the array needs to be compared with all other elements, resulting in a large number of comparisons and swaps. However, Quick Sort is faster than Selection Sort even on this worst-case input because of its more efficient partitioning scheme, which reduces the number of swaps required to sort the array. As shown in the table, Quick Sort consistently outperforms Selection Sort on reverse-sorted input data, despite both algorithms having the same time complexity.

On the other hand, Bucket sort behaves similarly to the random and sorted arrays. The performance of Bucket sort is not affected by the input array because it partitions the input into "buckets" and sorts each bucket separately. The algorithm assigns elements to buckets based on their values, and since the values are uniformly distributed in the input array, the elements are evenly distributed across the buckets. This results in a uniform distribution of work across the buckets.

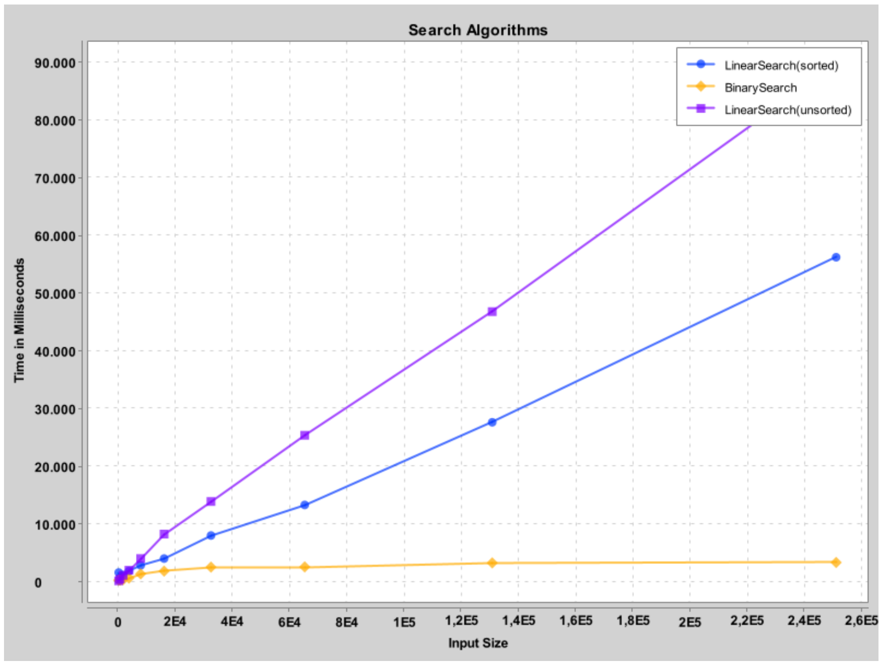

Figure 4: Plot of the functions.

I compared my linear search graphs with those of my friends to gain a better understanding of the results. It is not uncommon to observe varying timings when comparing the performance of linear search on sorted and unsorted data among different people, even when using the same algorithm and input size. This may be attributed to several factors, such as differences in hardware, operating systems, or programming environments. Moreover, the specific input values generated by each person's random number generator can also affect the algorithm's performance. Furthermore, the order of input values may influence the number of iterations required by the algorithm depending on the search value used. Thus, it is expected to see discrepancies in the performance results of linear search on sorted versus unsorted data when comparing results among different individuals.

References

- stackoverflow.com/questions/12169038/read-csv-file-column-by-column
- stackoverflow.com/questions/3382954/measure-execution-time-for-a-java-method
- stackoverflow.com/questions/46898/how-do-i-efficiently-iterate-over-each-entry-in-a-java-map
- wikipedia.org/wiki/Quicksort
- wikipedia.org/wiki/Selection$_{s}ort$
- wikipedia.org/wiki/Bucket$_{s}ort$