Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 30, 2023

Student name:
Yağız Buğra Boz

Student Number:
b2200356009

1 Problem Definition

I compare 3 different sorting algorithms and 2 different search algorithms running times with the both random and sorted data of different sizes. I also examine the compatibility of the running time of the algorithms with the theoretical time complexity's.

2 Solution Implementation

2.1 Selection Sort Algorithm

```java
    public static double selectionSort(int datalist[]){
        double start = System.currentTimeMillis();

        for (int i = 0; i<datalist.length-1; i++) {
            int minindex = i;
            for(int j = i+1;j<datalist.length;j++) {
                if(datalist[j] < datalist[minindex]) {
                    minindex = j;
                }
            }
            int min = datalist[minindex];
            if(minindex != i) {
                datalist[minindex] = datalist[i];
                datalist[i] = min;
            }
        }
        double now = System.currentTimeMillis();

        return (now-start);

    }
```

2.2 Quick Sort Algorithm

```java
    public static double quickSort(int datalist[], int lowindex, int highindex
        ) {
        double start = System.currentTimeMillis();
        double now;
        if(lowindex >= highindex) {
            now = System.currentTimeMillis();

            return (now-start);
        }

        int pivot = datalist[highindex];
```

```
        int leftpointer = lowindex;
        int rightpointer = highindex;

        while(leftpointer < rightpointer) {
            while (datalist[leftpointer] <= pivot && leftpointer < rightpointer) {
                leftpointer++;
            }
            while (datalist[rightpointer] >= pivot && leftpointer < rightpointer) {
                rightpointer--;
            }
            swap(datalist, leftpointer, rightpointer);

        }
        swap(datalist, leftpointer, highindex);
        quickSort(datalist, lowindex, leftpointer-1);
        quickSort(datalist, leftpointer + 1, highindex);

        now = System.currentTimeMillis();
        return (now-start);
    }
    private static void swap(int datalist[], int index1, int index2) {
        int temp = datalist[index1];
        datalist[index1] = datalist[index2];
        datalist[index2] = temp;
    }
```

2.3 Bucket Sort Algorithm

```
    public static double bucketSort(int datalist[]) {
        double start = System.currentTimeMillis();
        int numberofbuckets = (int) Math.sqrt(datalist.length);
        ArrayList<Integer>[] buckets = new ArrayList[numberofbuckets];
        int max = findmax(datalist);

        for (int i = 0; i < numberofbuckets; i++) {
            buckets[i] = new ArrayList<Integer>();
        }
        for (int i = 0; i < datalist.length; i++) {
            int bucketIndex = hash(i, max, numberofbuckets);
            buckets[bucketIndex].add(datalist[i]);
        }

        for (int i = 0; i < numberofbuckets; i++) {
            Collections.sort(buckets[i]);
        }
```

```

        int k = 0;
        for (int i = 0; i < numberofbuckets; i++) {
            for (int j = 0; j < buckets[i].size(); j++) {
                datalist[k] = buckets[i].get(j);
                k++;
            }
        }
        double now = System.currentTimeMillis();
        return (now-start);
    }

    private static int hash(int i, int max, int numberOfBuckets) {
        int temp = i/max*(numberOfBuckets-1);
        return (int)temp;
    }

    private static int findmax(int[] datalist) {
        int temp = datalist[0];
        for (int i = 1; i < datalist.length;i++) {
            if(temp >= datalist[i]) {
                continue;
            }
            else {  temp = datalist[i];}
        }
        return temp;
    }
```

2.4 Linear Search Algorithm

```
    public static double linearSearch(int datalist[], int x) {
        double start = System.nanoTime();

        for (int i = 0; i<datalist.length;i++) {
            if(datalist[i] == x) {
                double now = System.nanoTime();
                return (now-start);
            }
        }
        double now = System.nanoTime();
        return (now-start);
    }
```

2.5 Binary Search Algorithm

```java
    public static double binarySearch(int datalist[], int x) {
        double start = System.nanoTime();
        int low = 0;
        int high = datalist.length-1;
        while ( high - low > 1) {
            int mid =  (high+low)/2;
            if(datalist[mid] < x) {
                low = mid + 1;
            }
            else {
                high = mid;
            }
        }
        if(datalist[low] == x) {
            double now = System.nanoTime();
            return (now-start);
        }
        else if(datalist[high] == x) {
            double now = System.nanoTime();
            return (now-start);
        }
        else {
            double now = System.nanoTime();
            return (now-start);
        }
    }
```

3 Results, Analysis, Discussion

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

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

Complexity analysis tables to complete (Table 3 and Table 4):

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	0.8	0.1	1.5	5.8	22.5	80.2	284.1	941.8	3678.4	14044.6
Quick sort	0.3	0.0	0.3	0.2	1.1	2.5	5.7	11.0	22.0	48.9
Bucket sort	0.3	0.9	0.5	1.3	1.3	2.3	5.8	9.8	21.1	50.3
	Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.4	1.5	5.7	20.6	101.5	298.5	1375.9	5168.2	19556.0
Quick sort	-	-	-	-	-	-	-	-	-	-
Bucket sort	0.1	0.1	0.1	0.1	0.1	0.5	0.5	1.0	1.6	5.4
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.2	0.4	1.8	7.1	27.9	170.8	426.1	1792	7145.7	29428.3
Quick sort	-	-	-	-	-	-	-	-	-	-
Bucket sort	0.1	0.1	0.1	0.3	0.3	0.6	0.6	1.6	3.2	7.7

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)	2504	204	369	703	1703	2727	5695	14825	29295	114051
Linear search (sorted data)	1986	200	366	707	1443	2728	6263	14682	23952	96590
Binary search (sorted data)	159	100	115	43	45	55	64	90	91	105

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	$\Omega(n + k)$	$\Theta(n + k)$	$O(n^2)$
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	$O(n)$
Linear Search	$O(1)$
Binary Search	$O(1)$

4 Explanations, Results, Plots, Notes

- Figure 1 is the method that will reach the sorting and search algorithms that will enable us to plot graphs and run them in the correct order.
- Figure 3 and Figure 4 do not include the quicksort algorithm. I got the java.lang.StackOverflowError when I ran it with sorted and reversely sorted input. That's why I didn't run the quicksort algorithm. but it works without any problem with random input.
- In order to better compare quicksort and bucketsort with random input, I run Figure 6 without selection sort.

References

- https://www.geeksforgeeks.org/linear-search-vs-binary-search/
- https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/
- https://www.codingninjas.com/codestudio/library/time-space-complexity-of-searching-algorithms

```java
    }
    public static double[][] exec(int datalist[]){
        double[][] timelist = new double[12][10];
        int[] inputsize = {500,1000,2000,4000,8000,16000,32000,64000,128000,250000};
        for (int input = 0; input < 10 ; input++) {
            int[] templist = new int[inputsize[input]];
            for (int i = 0; i<inputsize[input] ;i++) {
                templist[i] = datalist[i];
            }
            //******** RANDOM INPUT SORT
            double timeSR = 0;
            double timeQR = 0;
            double timeBR = 0;
            for(int i = 0; i<10 ;i++) {
                timeSR = timeSR + selectionSort(templist.clone());
                timeQR = timeQR + quickSort(templist.clone(),0,templist.length-1);
                timeBR = timeBR + bucketSort(templist.clone());
            }
            timelist[0][input] = timeSR/10;
            timelist[1][input] = timeQR/10;
            timelist[2][input] = timeBR/10;
            //******* RANDOM INPUT LINEAR SEARCH
            int rand = new Random().nextInt(inputsize[input]);
            double timeLSR = 0;
            for (int i = 0; i<1000 ; i++) {
                timeLSR = timeLSR + linearSearch(templist.clone(),rand);
            }
            timelist[9][input] = timeLSR/1000;
            //******* SORTED INPUT SORT
            bucketSort(templist);
            double timeSS = 0;
            double timeQS = 0;
            double timeBS = 0;
            for(int i = 0; i<10 ;i++) {
                timeSS = timeSS + selectionSort(templist);
                //timeQS = timeQS + quickSort(templist,0,templist.length-1);
                timeBS = timeBS + bucketSort(templist);
            }
            timelist[3][input] = timeSS/10;
            timelist[4][input] = timeQS/10;
            timelist[5][input] = timeBS/10;
            //************* SORTED SEARCH
            double timeLSS = 0;
            double timeBSS = 0;
            for (int i = 0; i<1000 ; i++) {
                timeLSS = timeLSS + linearSearch(templist,rand);
                timeBSS = timeBSS + binarySearch(templist,rand);
            }
            timelist[10][input] = timeLSS/1000;
            timelist[11][input] = timeBSS/1000;
            //************* REVERSELY SORTED SORT
            int[] Rtemp = new int[templist.length];
            for(int i = 0; i< templist.length; i++) {
                Rtemp[templist.length-i-1] = templist[i];
            }
            double timeSRS = 0;
            double timeQRS = 0;
            double timeBRS = 0;
            for(int i = 0; i<10 ; i++) {
                timeSRS = timeSRS + selectionSort(Rtemp.clone());
                //timeQRS = timeQRS + quickSort(Rtemp,0,Rtemp.length-1);
                timeBRS = timeBRS + bucketSort(Rtemp.clone());
            }
            timelist[6][input] = timeSRS/10;
            timelist[7][input] = timeQRS/10;
            timelist[8][input] = timeBRS/10;
            for (int i = 0; i<12;i++) {
                for (int k = 0; k < 10; k++) {
                    System.out.print(timelist[i][k] + "  ");
                }
                System.out.println();
            }
        }


        return timelist;
    }
```

Figure 1: Code part that execute all the algorithms

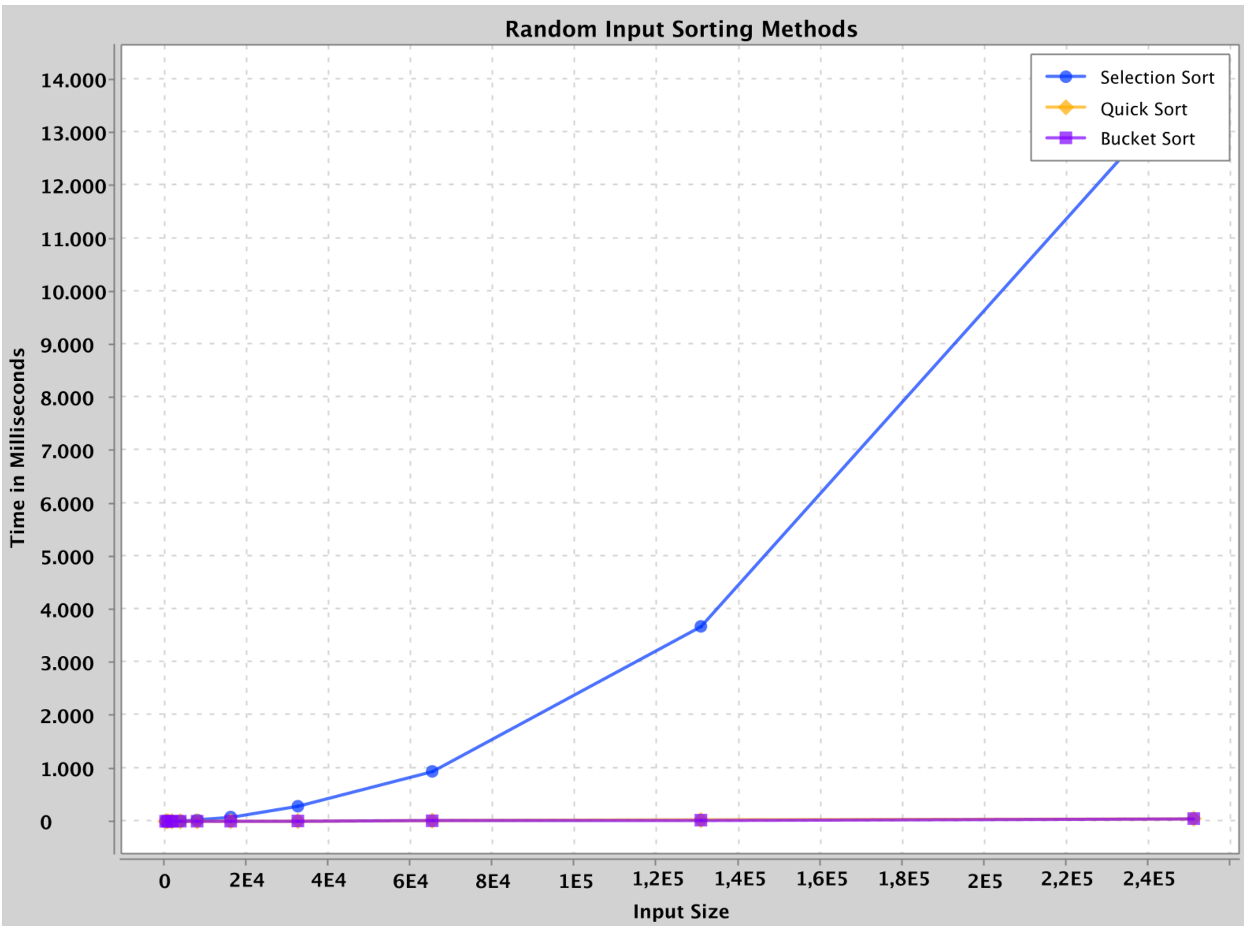

Figure 2: Random input sorting (Selection-Quick-Bucket)

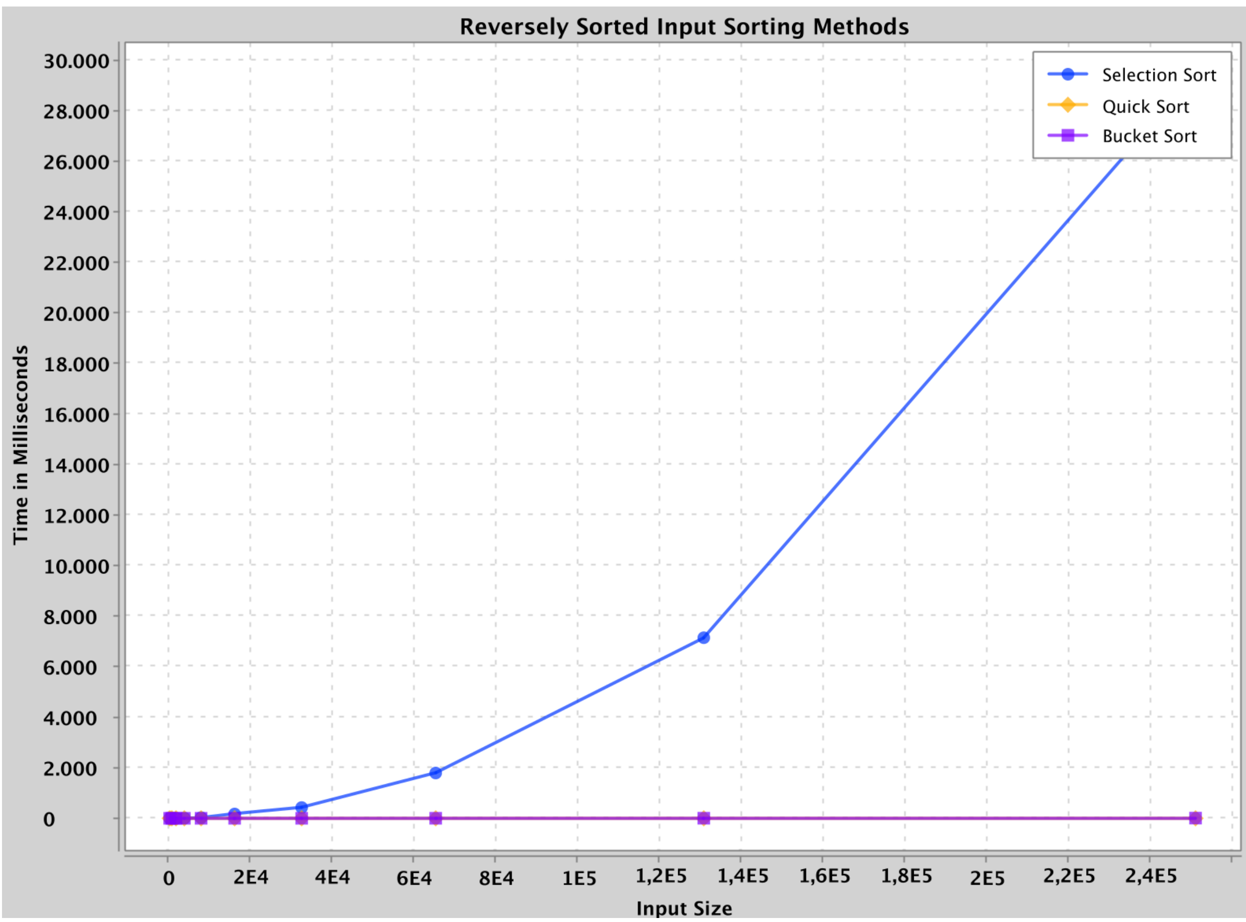

Figure 3: Reversely sorted input sorting (Selection-Bucket)

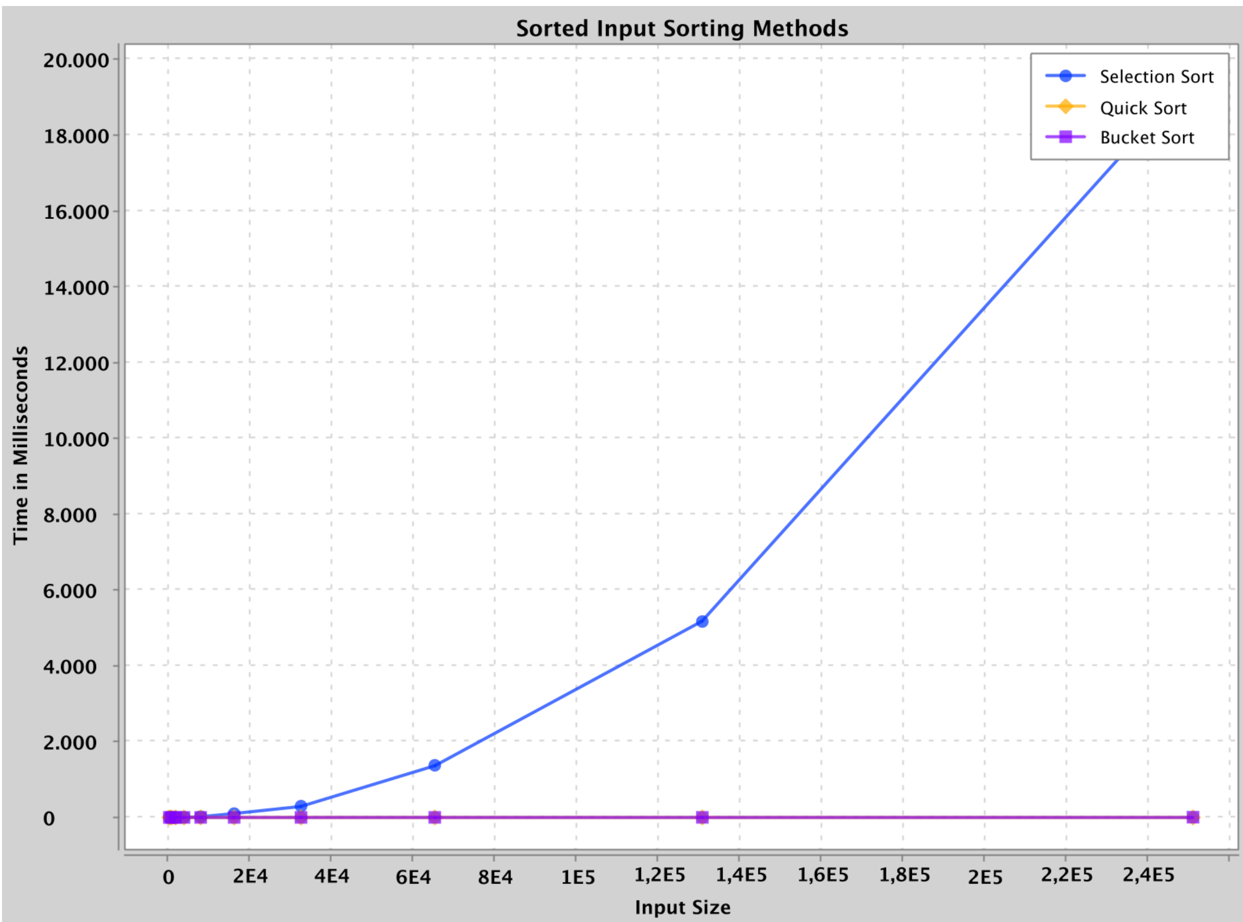

Figure 4: Sorted input sorting (Selection-Bucket)

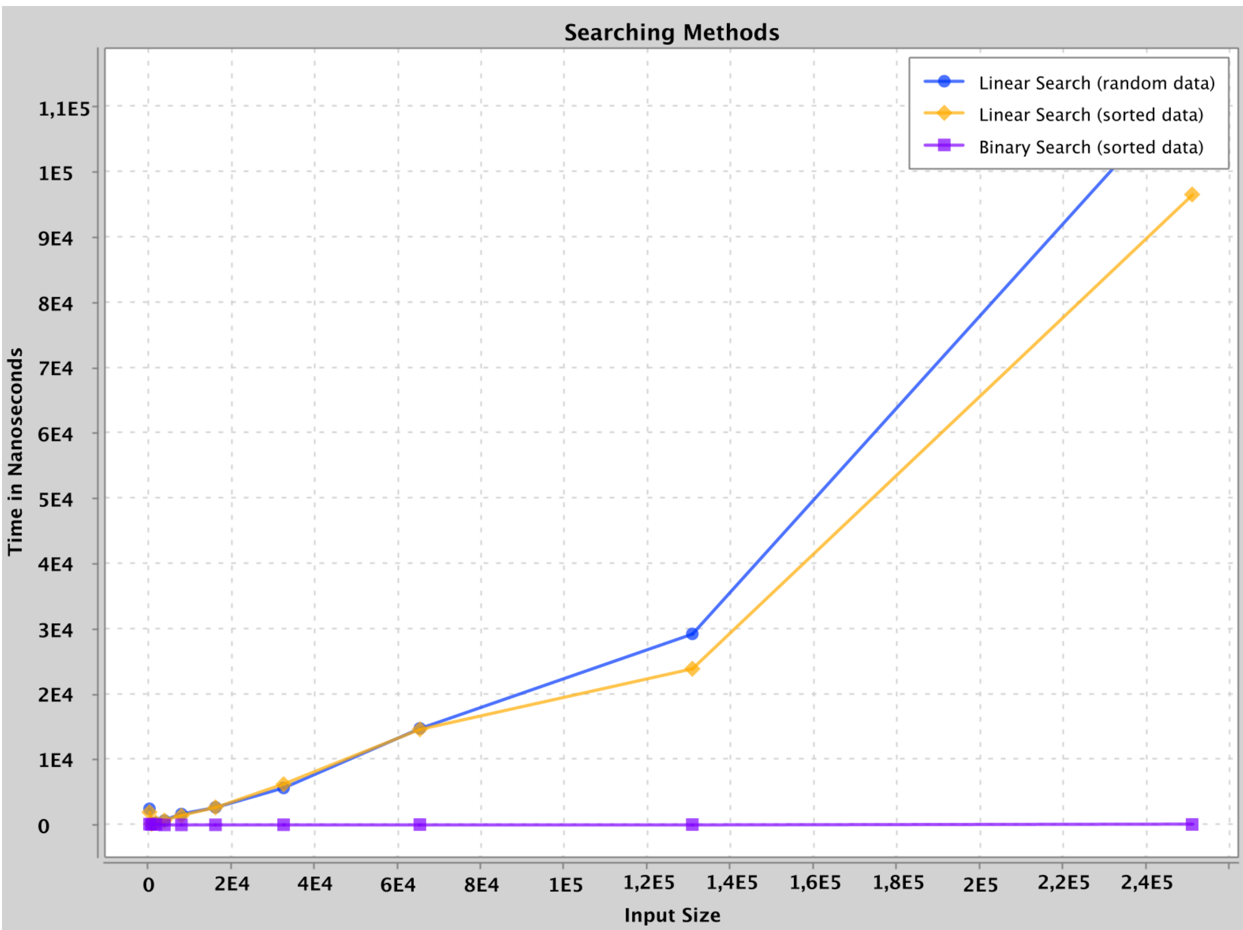

Figure 5: Searching

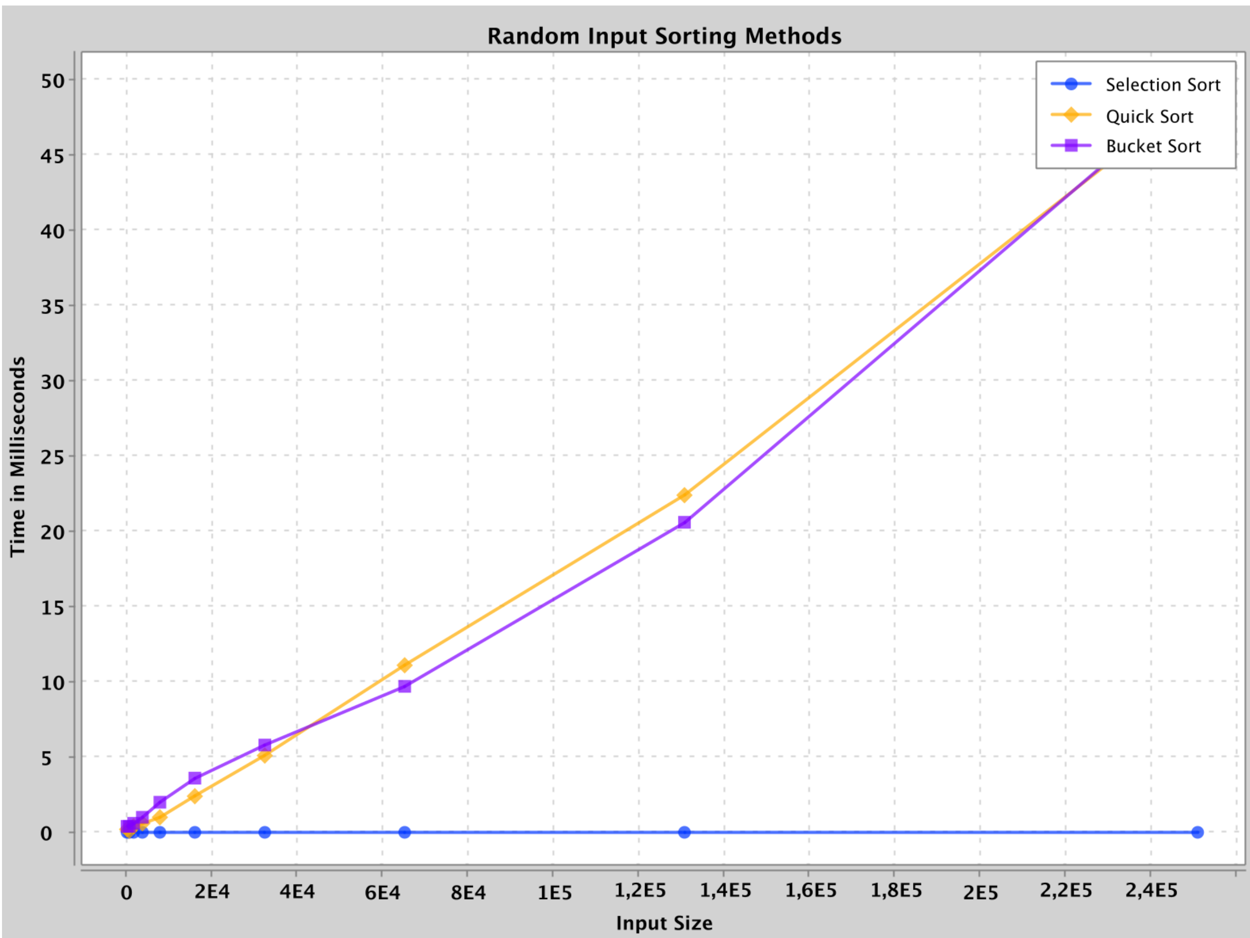

Figure 6: Random input comparison between quick sort and bucket sort