Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 27, 2023

Student name:
Nurullah Başer

Student Number:
b2200356077

1 Problem Definition

In this assignment, 4 sorting algorithms including quick sort, bucket sort, selection sort, and java builtin sort and 2 search algorithms including linear search, binary search were compared. The comparisons were tested in different data sizes and evaluated in 3 different types: random, sorted, and reversely sorted.

2 Sort Solution Implementation

2.1 Abstract Sort Class

```
public abstract class Sorts {

    public double[] randomInputTimes = new double[10];
    public double[] sortedInputTimes = new double[10];
    public double[] reverselySortedInputTimes = new double[10];
    int randomIndex = 0;
    int sortedIndex = 0;
    int reverselyIndex = 0;

    protected void swap(int[] arr, int i , int j) {
        arr[j] ^= arr[i];
        arr[i] ^= arr[j];
        arr[j] ^= arr[i];
    }

    public abstract void sort(int[] arr);

    public void manager(int[] arr,InputType inputType,String name) {
        long totalTime = 0;

        for(int i = 0; i < 10 ; i++) {
            int[] copyArr = Arrays.copyOf(arr,arr.length);

            long startTime = System.nanoTime();
            sort(copyArr);
            long finishTime = System.nanoTime();

            totalTime += finishTime - startTime;
        }
        totalTime = TimeUnit.NANOSECONDS.toMillis(totalTime);

        switch (inputType) {
            case random:
                System.out.println(name + " " + arr.length + " " + "random" +
                    " " + ((double)totalTime/10));
```

```
                randomInputTimes[randomIndex++] = ((double)totalTime/10);
                break;
            case sorted:
                System.out.println(name + " " + arr.length + " " + "sorted" +
                    " " + ((double)totalTime/10));
                sortedInputTimes[sortedIndex++] = ((double)totalTime/10);
                break;
            case reverselySorted:
                System.out.println(name + " " + arr.length + " " + "reversely"
                     + " " + ((double)totalTime/10));
                reverselySortedInputTimes[reverselyIndex++] = ((double)
                   totalTime/10);
                break;
        }
    }
}
```

2.2 Selection Sort

```
public class SelectionSort extends Sorts {
    @Override
    public void sort(int[] arr) {
        int n = arr.length;

        for(int i = 0; i < n-1; i++) {
            int index = i;
            for(int j = i+1; j < n; j++) {
                if (arr[j] < arr[index]) index = j;
            }
            swap(arr,i,index);
        }
    }
}
```

2.3 Quick Sort

```
62  public class QuickSort extends Sorts{
63      @Override
64      public void sort(int[] arr) {
65          int low = 0;
66          int high = arr.length - 1;
67          int top = -1;
68
69          int[] stack = new int[high-low+1];
70
71          stack[++top] = low;
72          stack[++top] = high;
73
74          while (top >= 0) {
75              high = stack[top--];
76              low = stack[top--];
77
78              int p = partition(arr,low,high);
79
80              if (p - 1 > low) {
81                  stack[++top] = low;
82                  stack[++top] = p-1;
83              }
84
85              if (p+1 < high) {
86                  stack[++top] = p + 1;
87                  stack[++top] = high;
88              }
89          }
90      }
91
92
93      public int partition(int[] arr, int low, int high) {
94          int pivot = arr[high];
95
96          int i = low-1;
97          for(int j = low; j < high; j++) {
98              if (arr[j] <= pivot) {
99                  i++;
100                 swap(arr,i,j);
101             }
102         }
103         swap(arr,i+1,high);
104         return i + 1;
105     }
106 }
```

2.4 Bucket Sort

```
107 public class BucketSort extends Sorts{
108     @Override
109     public void sort(int[] arr) {
110         int n = (int) (Math.sqrt(arr.length));
111 
112         ArrayList<Integer>[] bucket = new ArrayList[n];
113 
114         for(int i = 0; i < n; i++) {
115             bucket[i] = new ArrayList<>();
116         }
117 
118         int max = arr[0];
119         for(int element : arr) {
120                max = Math.max(max,element);
121         }
122 
123         for (int element : arr) {
124             int bucketIndex = hash(element,max,n);
125             bucket[bucketIndex].add(element);
126         }
127 
128         for(int i = 0; i < n; i++) {
129             Collections.sort(bucket[i]);
130         }
131 
132         int index = 0;
133         for(int i = 0; i < n; i++) {
134             for (int j = 0, size = bucket[i].size(); j < size; j++) {
135                 arr[index++] = bucket[i].get(j);
136             }
137         }
138     }
139 
140     private int hash(int i, int max, int n) {
141         return (int) ((double) i / max * (n-1));
142     }
143 
144 }
```

2.5 Java Built-in Sort

```
145 public class JavaSort extends Sorts{
146     @Override
147     public void sort(int[] arr) {
148         Arrays.sort(arr);
149     }
150 }
```

3 Search Solution Implementation

3.1 Abstract Search Class

```
151 public abstract class Searches {
152     public double[] randomInputTimes = new double[10];
153     public double[] sortedInputTimes = new double[10];
154     int randomIndex = 0;
155     int sortedIndex = 0;
156 
157     public abstract int search(int[] arr, int target);
158 
159     public void manager(int[] arr, InputType inputType, String name) {
160         long totalTime = 0;
161 
162         for(int i = 0; i < 1000; i++) {
163             int index = (int)(Math.random() * ((arr.length)-1));
164             long startTime = System.nanoTime();
165             search(arr,arr[index]);
166             long finishTime = System.nanoTime();
167             totalTime += finishTime - startTime;
168         }
169 
170         switch (inputType) {
171             case random:
172                 System.out.println(name + " " + arr.length + " " + "random" +
                        " " + ((double)totalTime/1000));
173                 randomInputTimes[randomIndex++] = ((double)totalTime/1000);
174                 break;
175             case sorted:
176                 System.out.println(name + " " + arr.length + " " + "sorted" +
                        " " + ((double)totalTime/1000));
177                 sortedInputTimes[sortedIndex++] = ((double)totalTime/1000);
178                 break;
179         }
180     }
181 }
```

3.2 Linear Search

```
182 public class LinearSearch extends Searches{
183     @Override
184     public int search(int[] arr, int target) {
185         for(int i = 0; i <  arr.length-1; i++)
186             if(arr[i] == target)
187                 return i;
188         return -1;
189     }
190 }
```

3.3 Binary Search

```
191 public class BinarySearch extends Searches{
192     @Override
193     public int search(int[] arr, int target) {
194         int low = 0;
195         int high = arr.length - 1;
196 
197         while(low <= high) {
198             int mid = low + (high-low)/2;
199 
200             if(arr[mid] == target) return mid;
201             if(arr[mid] > target) high = mid-1;
202             else if(arr[mid] < target) low = mid+1;
203         }
204         return -1;
205     }
206 }
```

4 Results, Analysis, Discussion

4.1 Tables Of Algorithms Running Times

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	0.3	1.2	4.4	18.0	59.3	238.9	1021.2	3725.9	14573.2
Quick sort	0.0	0.1	0.1	0.2	0.7	2.9	5.7	22.3	62.8	154.8
Bucket sort	0.4	0.3	0.4	0.6	1.4	2.4	4.5	9.1	16.6	36.8
Java sort	0.1	0.0	0.1	0.4	0.5	1.3	1.3	3.4	6.2	15.7
	Sorted Input Data Timing Results in ms									
Selection sort	0.0	0.2	1.2	4.3	17.7	59.2	236.8	1061.8	3871.1	14662.2
Quick sort	0.1	0.6	3.4	10.3	37.9	150.0	607.1	2408.8	9654.3	36709.2
Bucket sort	0.0	0.1	0.2	0.0	0.1	0.4	0.3	2.0	2.3	4.8
Java sort	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.4	0.7	1.3
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.1	0.5	1.9	7.2	25.2	109.4	435.0	1746.5	6994.4	29335.8
Quick sort	0.1	0.6	2.5	10.7	37.5	149.7	614.8	2379.0	9365.8	35095.9
Bucket sort	0.0	0.0	0.1	0.2	0.2	0.3	0.6	1.4	2.9	5.8
Java sort	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2

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)	1377.3	134.9	207.6	459.4	629.2	1508.6	2369.1	5216.5	8887.8	14710.6
Linear search (sorted data)	1211.9	219.2	226.6	391.3	670.2	1412.6	2662.1	6723.1	11191.9	23207.3
Binary search (sorted data)	342.8	204.4	217.0	120.8	112.8	116.1	111.9	120.0	133.6	141.0

4.2 Tables Of Algorithms Time and Space Complexity

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	$O(1)$

Selection sort is a simple sorting algorithm that sorts elements by repeatedly finding the smallest element from an unsorted list and placing it at the beginning of the sorted list. The algorithm then moves on to the remaining unsorted portion of the list and repeats the same process until the entire list is sorted. The time complexity of the selection sort is $O(n^2)$. This makes it inefficient for large lists, as the number of comparisons and swaps needed to sort the list grows quadratically with the number of elements. The space complexity of the selection sort is O(1), as it only requires a constant amount of additional memory to perform the sorting.

Quick sort is a sorting algorithm that sorts elements by dividing the unsorted list into two sublists, one containing elements smaller than a chosen pivot element, and the other containing elements larger than the pivot. The pivot is then placed in its final position in the sorted list, and the same process is repeated on the two sublists until the entire list is sorted. The average time complexity of quick sort is $O(nlogn)$. However, in the worst-case scenario where the pivot is consistently chosen poorly, the time complexity can degrade to $O(n^2)$. The space complexity of quick sort is $O(logn)$ on average, as it requires a stack to keep track of the recursive calls. However, in the worst-case scenario where the recursive calls are deeply nested, the space complexity can be as high as $O(n)$.

Bucket sort is a non-comparative sorting algorithm that works by distributing elements into a number of buckets based on their value, and then sorting the elements within each bucket. The time complexity of bucket sort is $O(n+k)$, where n is the number of elements to be sorted and k

is the number of buckets. This makes bucket sort particularly efficient when the input is uniformly distributed over a range of values, as the number of elements in each bucket will be relatively small. The space complexity of bucket sort is $O(n + k)$, as it requires an array of buckets and may also require additional memory for the sorting algorithm used within each bucket.

Linear search is a simple algorithm for finding a target value in a list by checking each element one by one until the target is found or the end of the list is reached. The time complexity of the linear search is $O(n)$, where n is the length of the list. This means that the worst-case scenario is when the target value is not in the list, in which case the algorithm must check every element. On average, linear search will find the target value after checking half of the elements in the list. The space complexity of linear search is $O(1)$, as it only requires a constant amount of additional memory to perform the search, regardless of the size of the input list. Linear search is an efficient algorithm for small lists, but for larger lists, more sophisticated algorithms such as binary search are generally more appropriate.

Binary search is a search algorithm that works by repeatedly dividing a sorted list in half and discarding one half based on the comparison of the target value with the midpoint of the list, until the target value is found or determined to not be in the list. The time complexity of the binary search is $O(logn)$. This is because the algorithm divides the search space in half with each comparison, resulting in a search time that grows logarithmically with the size of the input. The space complexity of the binary search is $O(1)$, as it only requires a constant amount of additional memory to perform the search, regardless of the size of the input list. Binary search is an efficient algorithm for searching large sorted lists, as it can quickly eliminate large portions of the list and converge to the target value with only a few comparisons.

4.3 Analysis Of Sort Algorithms

4.3.1 Sort Algorithms Random Input

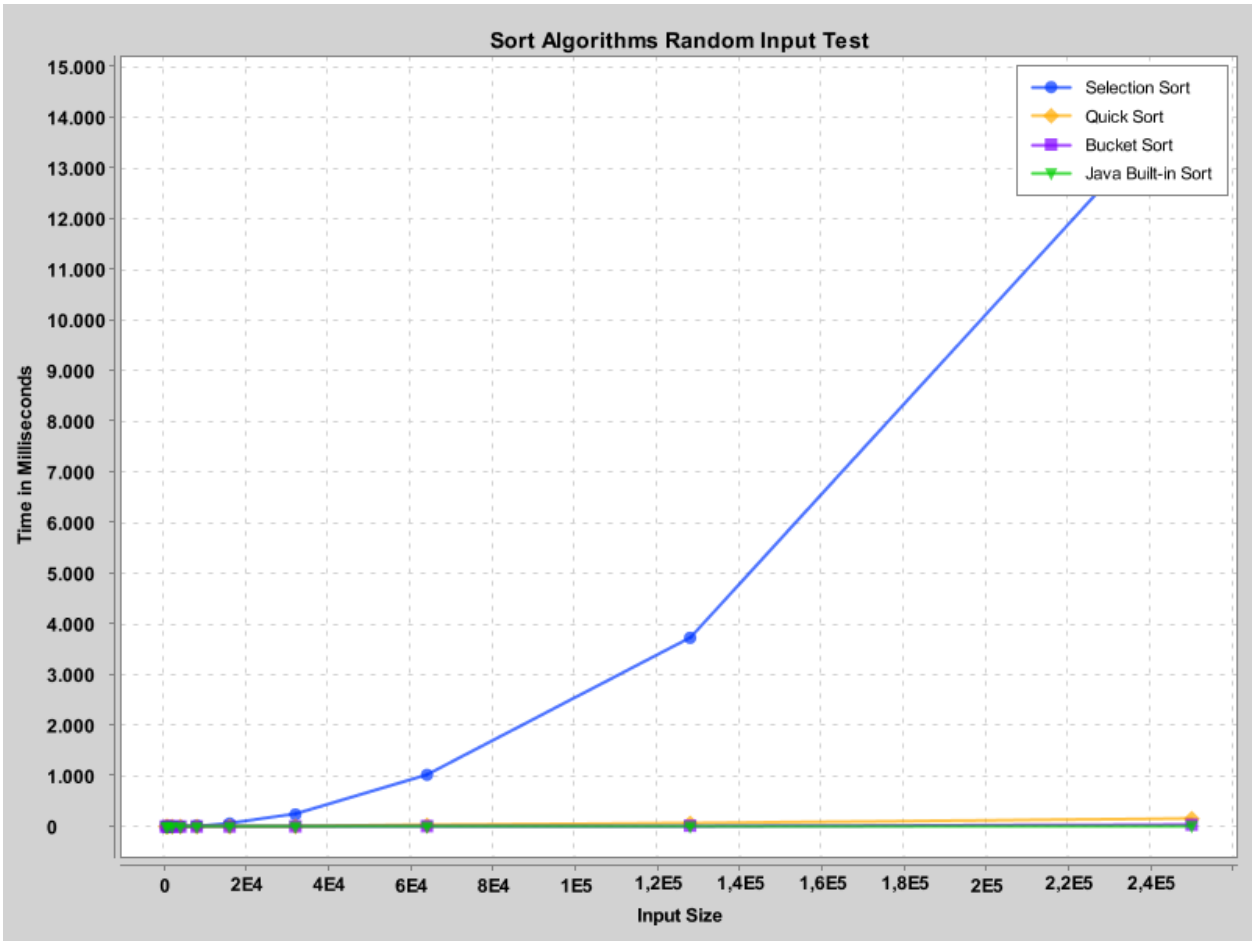

Figure 1: Sort Algorithms Random Input

Quick sort, bucket sort and java sort run in about the same time on random input. However, this may change as the input size increases. Because the complexity of bucket sort and quick sort is different from each other. Also, selection sort works slower than other algorithms because it works n^2.

4.3.2 Analysis Of Sort Algorithms Sorted Input

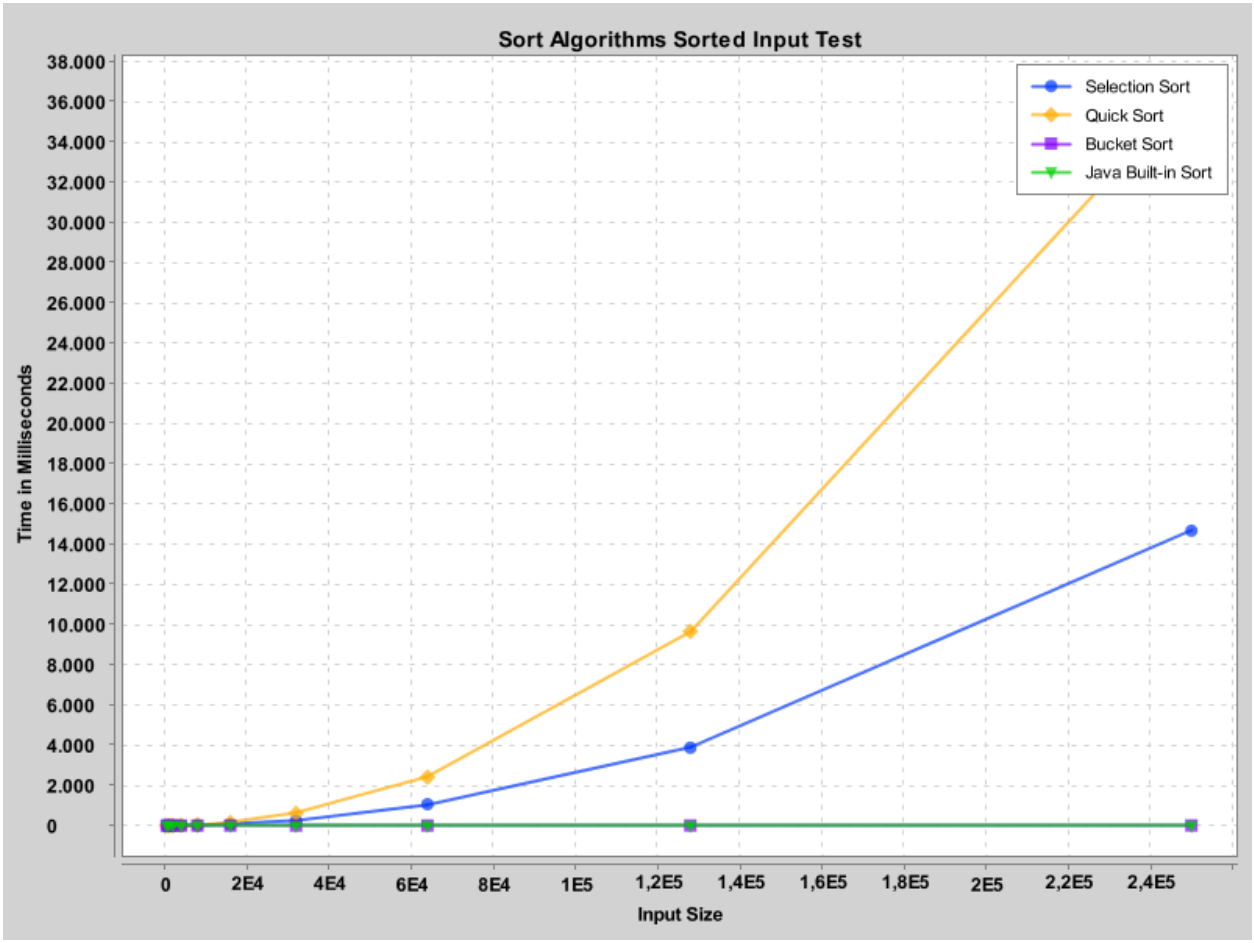

Figure 2: Sort Algorithms Sorted Input

Selection search is running slow as it is still n^2. Bucket sort and java sort work much faster here than other sorts. Because the sorted input cannot affect them badly, on the other hand, quick sort becomes the worst case and works slower than random input because while it is the worst case, its complexity is n^2.

4.3.3 Analysis Of Sort Algorithms Reversely Sorted Input

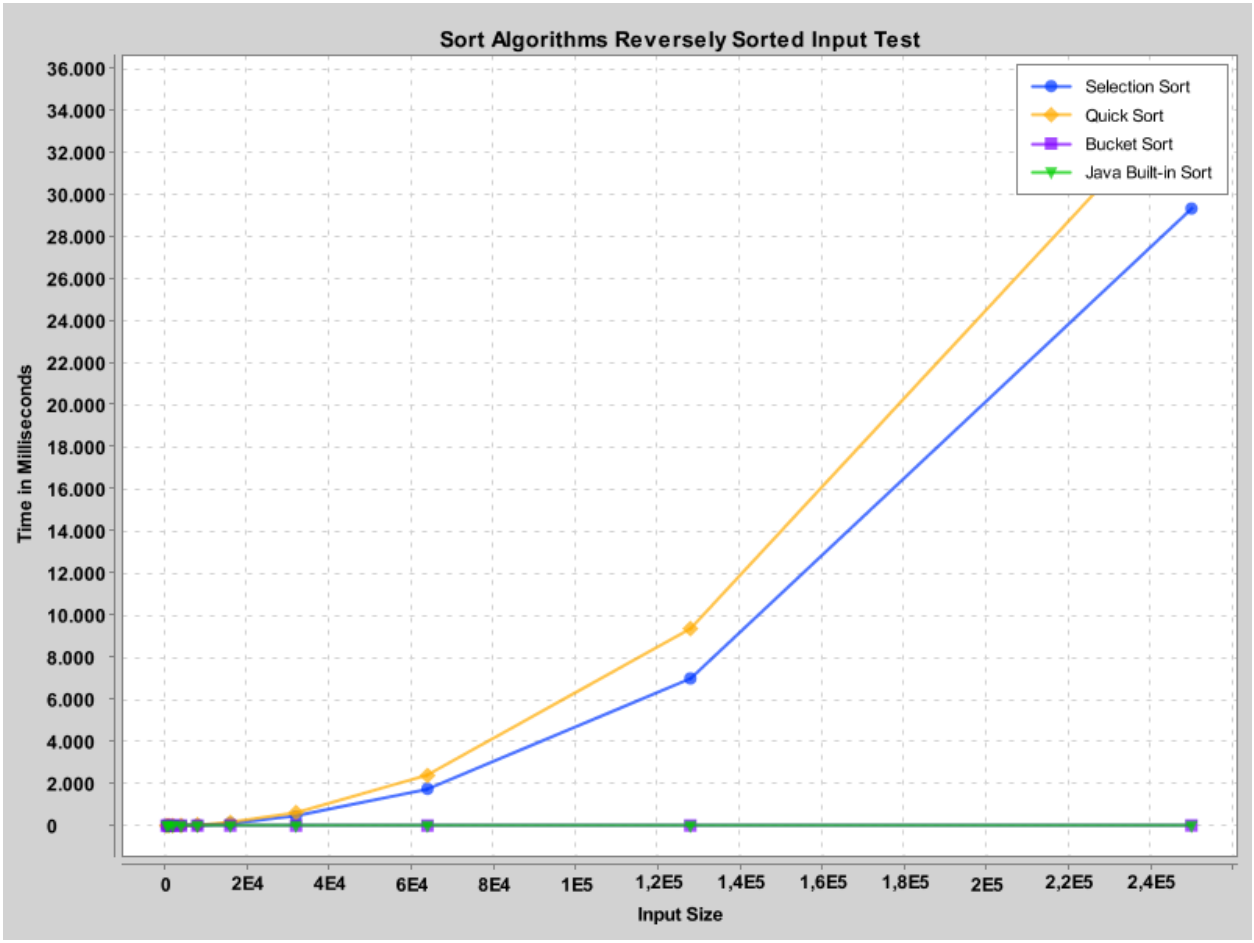

Figure 3: Sort Algorithms Reversely Sorted Input

Selection sort works n^2 here too. Quick sort here also works n^2 because there is a worst case. Bucket and java sort work fast here too.

4.3.4 Analysis Of Selection Sort Algorithm

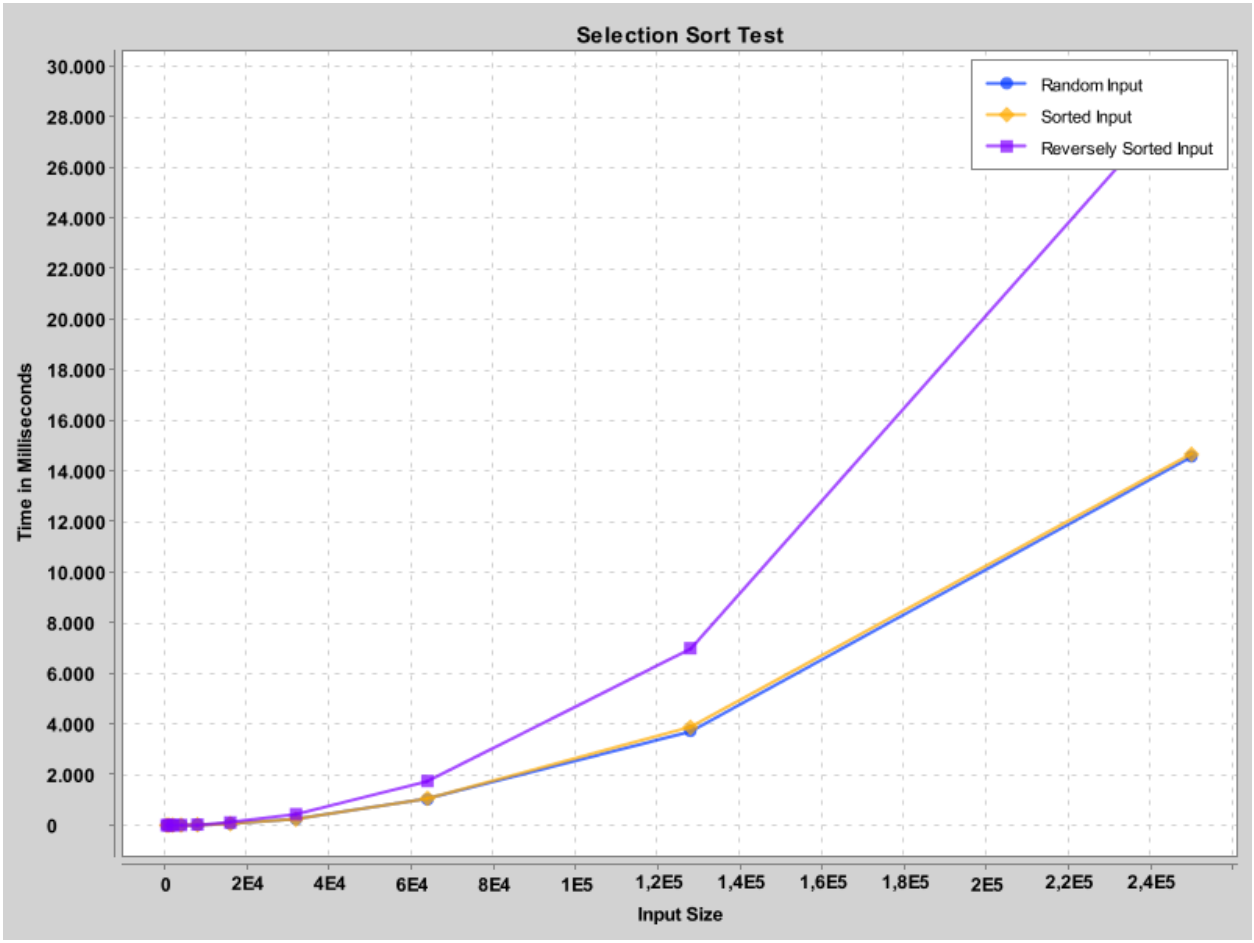

Figure 4: Selection Sort Algorithm

Selection sort works regardless of whether the input is in random, sorted or reversely sorted. Because the selection sort's best, worst and average are the same. $O(n^2)$

4.3.5 Analysis Of QuickSort Algorithm

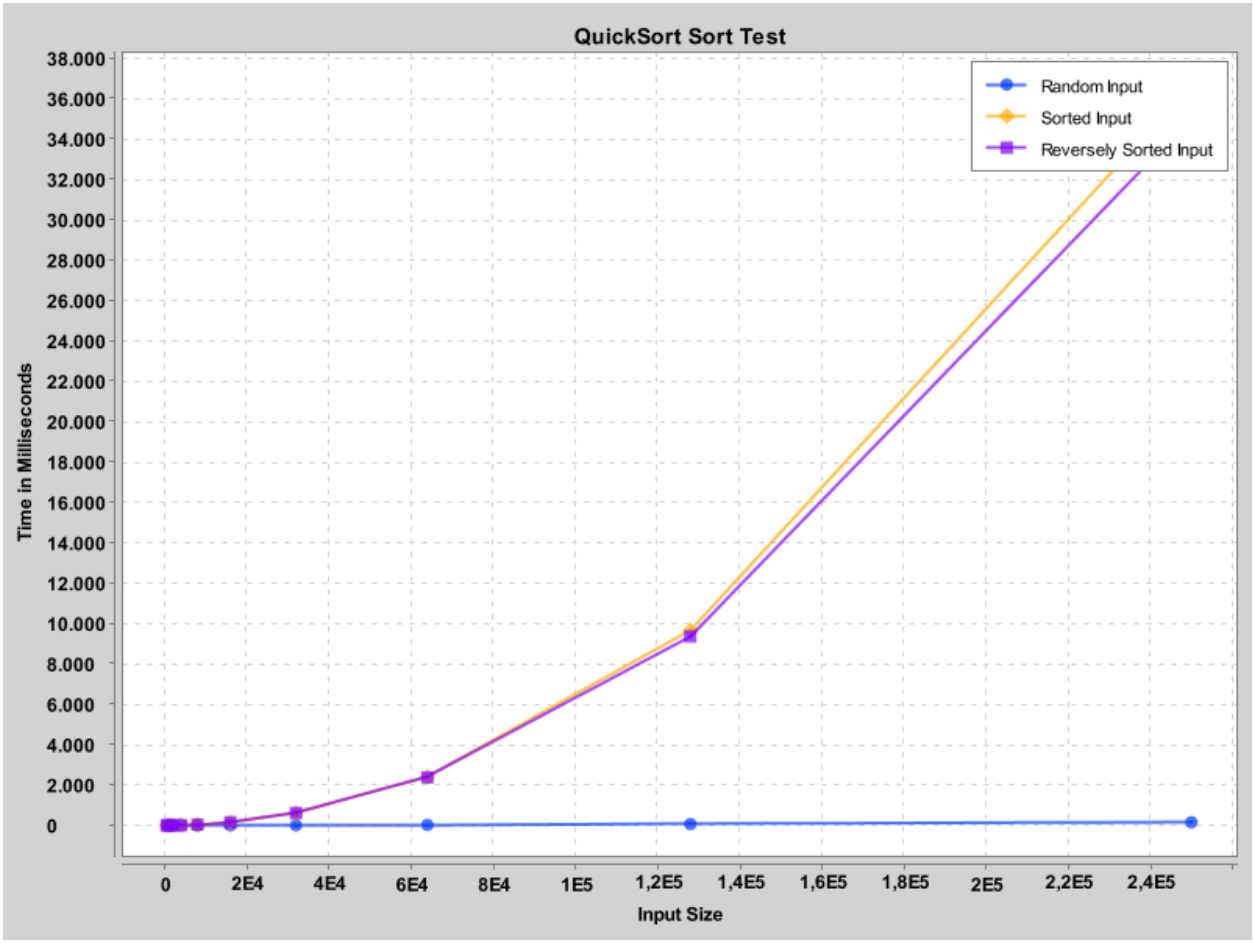

Figure 5: QuickSort Algorithm

The worst case happens when the Quicksort algorithm passes the input sorted or reversely sorted, and its complexity is $O(n^2)$. Therefore, it works slower than random input when sorted or reversely sorted.

4.3.6 Analysis Of Bucket Sort Algorithm

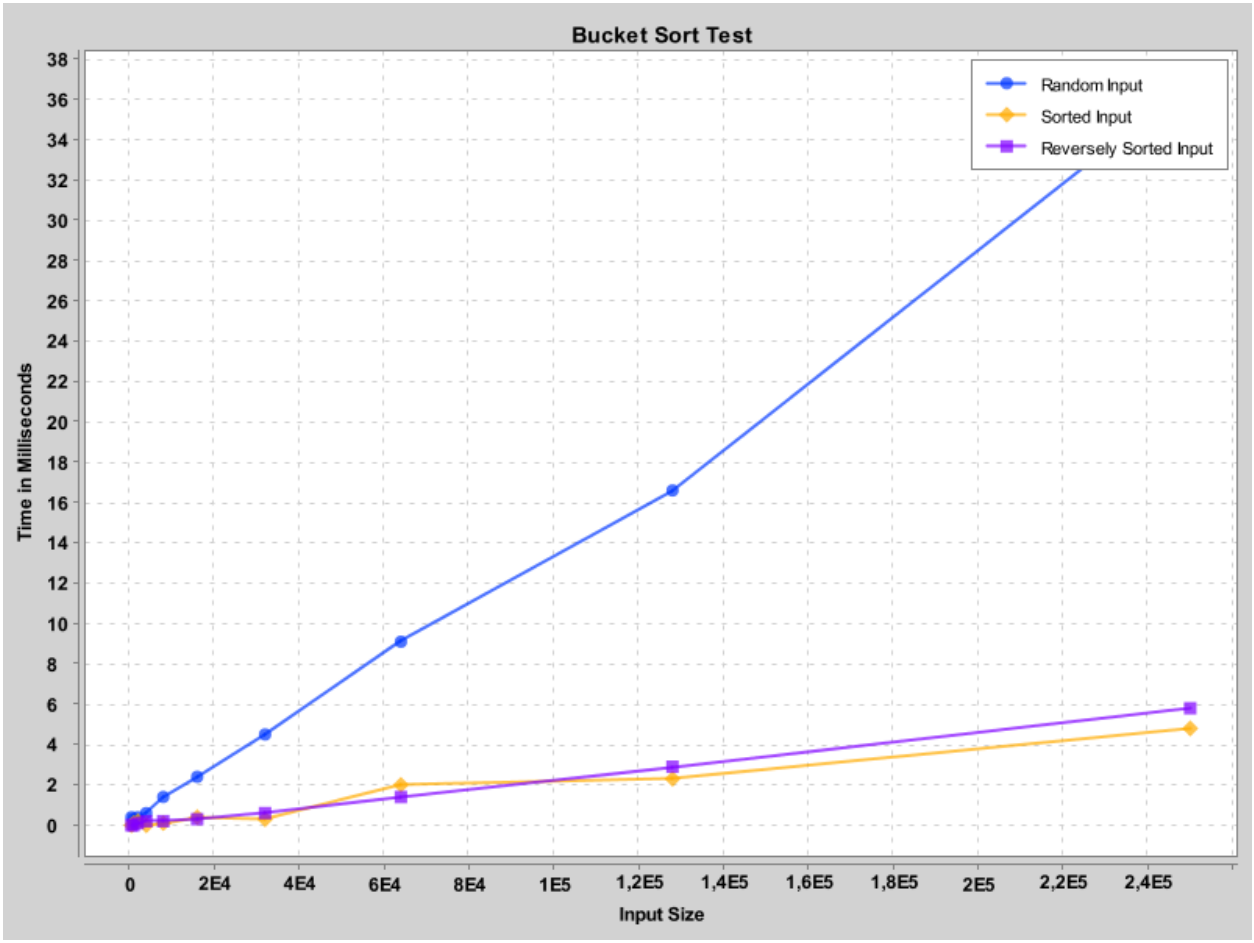

Figure 6: Bucket Sort Algorithm

In order for the bucket sort algorithm to be a worst case, each input element must fall into the same bucket from the hash result. Since this does not happen in the written algorithm, the bucket sort algorithm works the same for random, sorted and reversely sorted.

4.4 Analysis Of Search Algorithms

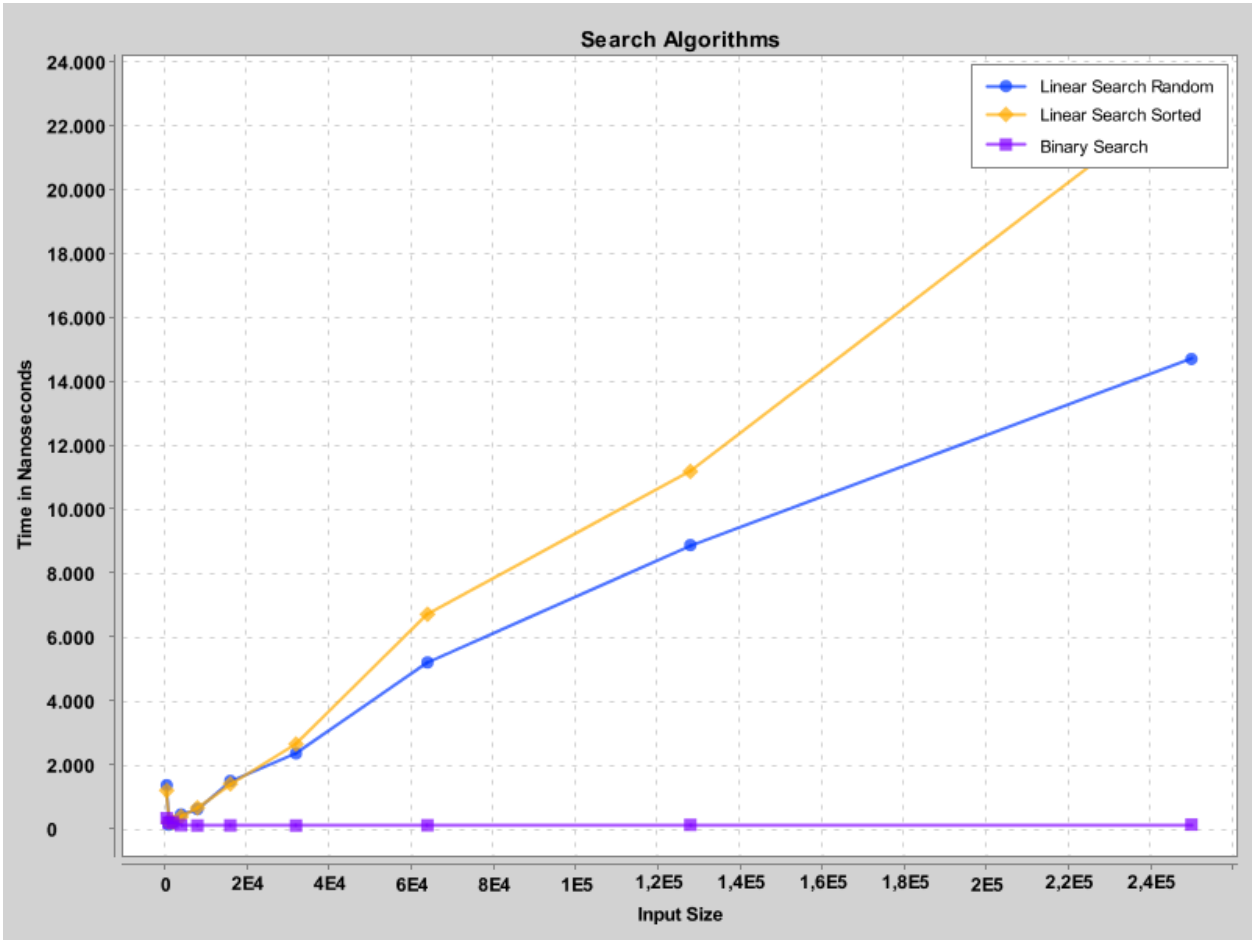

Figure 7: Seacrh Algorithms

In these two search algorithms, binary search only works on sorted inputs. For linear search, there is no need for such a condition. Linear search gives almost the same result in sorted and random inputs. Only the order of the searched value in the input changes this result. If it is at the beginning of the input, it will find it faster. If it is at the end of the input or not in the input, it will take much longer. Binary search, on the other hand, works independently of where the searched value is because it uses the middle method and is much faster than linear search. But its disadvantage is that it only works on sorted inputs.

4.5 Discussion

Using a built-in sort algorithm is generally better. Because the built-in functions are shaped according to the input with the conditions in themselves and try to sort them accordingly.

If our input is sorted, it will be much more advantageous to use binary search. But if the input is not sorted, we will either sort first or use linear search.

5 Notes

In my opinion, built-in functions are the best :D