Discussion of Sorting and Searching Algorithms

Author:
İrem Zehra Altun

1 Problem Definition

The purpose of this project is to implement three different sorting algorithms, which are Selection, Quick, and Bucket sort, and to implement two searching algorithm, which are linear and binary search, according to the given pseudo-codes. These sorting algorithms will be tested on different arrays, which are random, sorted and reversed, to compare how long it takes to complete. In searching algorithms, linear and binary search performances will be compared. In addition, linear search will be tested on both random and sorted arrays and their performance will be compared. Thus, we will be able to observe which algorithm will be more efficient in which situation by looking at the performance measurements of these algorithms tested in different arrays.

2 Solution Implementation

As a result of each sorting algorithm, the array is updated as sorted, so no return is used. We also take the average value of 10 iterations to find out how long each sorting algorithm takes. In order to obtain a healthy result, we need to perform each sorting operation on the same array. Therefore, after sorting, I return the array to its original state and make it repeat the same operation.

2.1 Selection Sorting Algorithm

Selection sorting algorithm sorts an array by repeatedly finding the minimum element from unsorted part of the array and putting it at the beginning.

```
1  private static void selectionSort(int[] values){
2          int n = values.length;
3          int min;
4          int temp;
5          for (int i = 0; i<n-1;i++) {
6              min = i;
7
8              for (int j = i+1; j<n;j++){
9                  if (values[j]<values[min]){
10                     min = j;
11                 }
12             }
13             if (min != i){
14                 temp = values[min];
15                 values[min] = values[i];
16                 values[i] = temp;
17
18             }
19         }
20     }
```

2.2 Quick Sorting Algorithm

Quick Sort is a sorting algorithm that uses the divide-and-conquer approach to sort an array. The algorithm works by partitioning the array into two sub-arrays, with one sub-array containing elements smaller than a chosen pivot element, and the other sub-array containing elements larger than the pivot. The pivot element is then placed in its final position in the sorted array, with all elements to the left of the pivot being smaller and all elements to the right being larger.

The iterative version of Quick Sort uses a stack data structure to simulate the recursive function calls used in the recursive version of the algorithm. Instead of calling the quicksort function recursively, the algorithm pushes the start and end indices of the sub-array onto a stack, and then repeatedly pops them off and partitions the sub-array until the stack is empty.

```
21 public static void quickSort(int[] array, int low, int high) {
22         int stackSize = high - low + 1;
23         int[] stack = new int[stackSize];
24         int top = -1;
25 
26         stack[++top] = low;
27         stack[++top] = high;
28 
29         while (top >= 0) {
30             high = stack[top--];
31             low = stack[top--];
32             int pivot = partition(array, low, high);
33 
34             if (pivot - 1 > low) {
35                 stack[++top] = low;
36                 stack[++top] = pivot - 1;
37             }
38 
39             if (pivot + 1 < high) {
40                 stack[++top] = pivot + 1;
41                 stack[++top] = high;
42             }
43         }
44     }
45     private static int  partition(int[] values, int low, int high){
46         int pivot = values[high];
47         int i = (low-1);
48         for (int j = low; j < high; j++){
49             if ( values[j] < pivot){
50                 i++;
51                 swap(values, i, j);
52             }
53         }
54         swap(values, i+1, high);
55         return i+1;
56     }
```

```
    private static void swap(int[] arr, int i, int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

2.3 Bucket Sorting Algorithm

Bucket Sort is a sorting algorithm that works by partitioning an input array into a set of smaller arrays, or "buckets", and then sorting each bucket individually.

```
public static void bucketSort(int[] array) {
        int numberOfBuckets = (int) Math.ceil(Math.sqrt(array.length));
        List<Integer>[] buckets = new List[numberOfBuckets];
        int max = maxVal(array);

        // Creating empty buckets
        for (int i = 0; i < numberOfBuckets; i++) {
            buckets[i] = new ArrayList<Integer>();
        }

        for (int i = 0; i < array.length; i++) {
            int index = hash(array[i], max, numberOfBuckets);
            buckets[index].add(array[i]);
        }
        int index = 0;
        for (int i = 0; i < numberOfBuckets; i++) {
            Collections.sort(buckets[i]);
            for (int j = 0; j < buckets[i].size(); j++) {
                array[index++] = buckets[i].get(j);
            }
        }

}
private static int hash(int value, int max, int numberOfBuckets) {
        return (int) Math.floor(value / max * (numberOfBuckets - 1));
}
private static int maxVal(int[] array) {
        int max = array[0];
        for (int i = 1; i < array.length; i++) {
            if (array[i] > max) {
                max = array[i];
            }
        }
        return max;
}
```

2.4 Linear Searching Algorithm

Linear Search works by iterating through an array or a list of elements in a sequential manner, comparing each element with the target value until a match is found or the entire list is searched.

```
97  public static int linearSearch(int[] values, int x) {
98          int len = values.length;
99          for (int i = 0; i < len; i++) {
100             if (values[i] == x) {return i;}
101         }
102         return -1;
103 }
```

2.5 Binary Searching Algorithm

Binary Search works on a sorted array of elements. The basic idea behind Binary Search is to divide the search range in half with each iteration, by comparing the middle element of the array with the target value being searched. If the middle element matches the target value, the search is successful. If the middle element is greater than the target value, the search continues in the lower half of the array. Similarly, if the middle element is less than the target value, the search continues in the upper half of the array. The process is repeated until the target value is found, or until the search range is exhausted and the target value is not found.

```
104 public static int binarySearch(int[] arr, int x) {
105         int low = 0;
106         int high = arr.length - 1;
107 
108         while (high - low > 1) {
109             int mid = (high + low) / 2;
110 
111             if (arr[mid] < x) {
112                 low = mid + 1;
113             } else {
114                 high = mid;
115             }
116         }
117         if (arr[low] == x) {return low;}
118         else if (arr[high] == x) {return high;}
119 
120         return -1;
121 }
```

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	0.05587	0.217804	0.798529	3.043208	11.767504	45.261404	179.699695	695.825233	2772.247775	10757.486208
Quick sort	0.00977	0.030083	0.038779	0.119312	0.295291	0.656654	1.876524	5.764841	15.521912	27.842587
Bucket sort	0.063416	0.134949	0.272491	0.535629	1.158903	2.304066	4.333787	6.524408	13.982187	30.09975
	Sorted Input Data Timing Results in ms									
Selection sort	0.050641	0.199699	0.776812	2.877062	11.447054	44.643179	179.875867	713.516233	2844.217795	10852.811929
Quick sort	0.036591	0.088591	0.42227	0.788991	4.196916	13.70607	16.153541	21.641362	1894.678924	13037.128891
Bucket sort	0.008916	0.010079	0.0245	0.031083	0.043545	0.090525	0.173491	0.33367	2.686891	1.571283
	Reversely Sorted Input Data Timing Results in ms									
Selection sort	0.068566	0.272483	0.977108	3.652296	14.043771	56.429216	227.292095	942.081691	4095.603183	19419.968283
Quick sort	0.054345	0.221154	0.806296	3.086787	12.590178	50.77472	203.590329	817.413862	3272.175775	12533.404308
Bucket sort	0.01525	0.027458	0.04465	0.05915	0.070249	0.119733	0.283375	0.700258	1.553612	5.833229

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)	182	273	350	677	1343	2695	5484	10996	21674	42386
Linear search (sorted data)	95	181	343	672	1342	2721	5446	10084	21476	42374
Binary search (sorted data)	75	102	104	94	43	100	19	18	23	87

Running time test results for search algorithms are given in Table 2.
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	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$
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	$O(1)$

– The Selection Sort algorithm has a time complexity of $O(n^2)$ in all cases: best case, worst case, and average case. This is because the algorithm always requires two nested loops, and the number of iterations for both loops is proportional to the square of the input size.

– The best case scenario for Quick Sort happens when the pivot element divides the array into two equal parts, resulting in each partitioning step splitting the array in half. This leads to an optimal time complexity of $\Omega(n \log n)$. The worst case for Quick Sort happens when the pivot element is either the smallest or largest element in the array. This leads to partitions with only one element and the rest of the array, resulting in a time complexity of $O(n^2)$.

– The best case computational complexity of Bucket Sort is $\Omega(n + k)$ where n is the number of elements to be sorted and k is the number of buckets. This happens when all the elements are uniformly distributed in the buckets. Average case computational complexity is $\Theta(n + k)$, where n is the number of elements in the input array and k is the number of buckets. This means that the algorithm runs in linear time proportional to the size of the input array and the number of buckets. It involves dividing the input into smaller sub-arrays or buckets, sorting each bucket individually, and then merging them back together in sorted order.The worst case computational complexity of Bucket Sort is $O(n^2)$ when all the elements fall into the same bucket, and we need to sort this bucket using sorting algorithm. The auxiliary space 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 is because we are creating k buckets and storing n elements in them.

– Linear search has its best-case computational complexity of O(1) when the element to be searched is at the first index of the array. However, in the average and worst-case scenarios, where the element is not at the first index, the algorithm would have to examine each element of the array until it finds the target element, thus leading to a computational complexity of O(n), where n is the number of elements in the array. The algorithm uses no extra space and operates directly on the input array, so its auxiliary space complexity is O(1).

– When the element being searched for is located in the middle of the array, the binary search algorithm can find it quickly with just one comparison, resulting in the best-case computational complexity of O(1). However, in the worst-case scenario, the element is not present in the array or is located at either end, requiring the algorithm to divide the array repeatedly until it can determine that the element is not present, resulting in a computational complexity of O(log n). The average case computational complexity of O(log n) is achieved when the elements in the array are uniformly distributed. As it is a iterative implementation of binary search, its auxiliary space complexity is O(1).

Example how to include and reference a figure: Fig. 1.

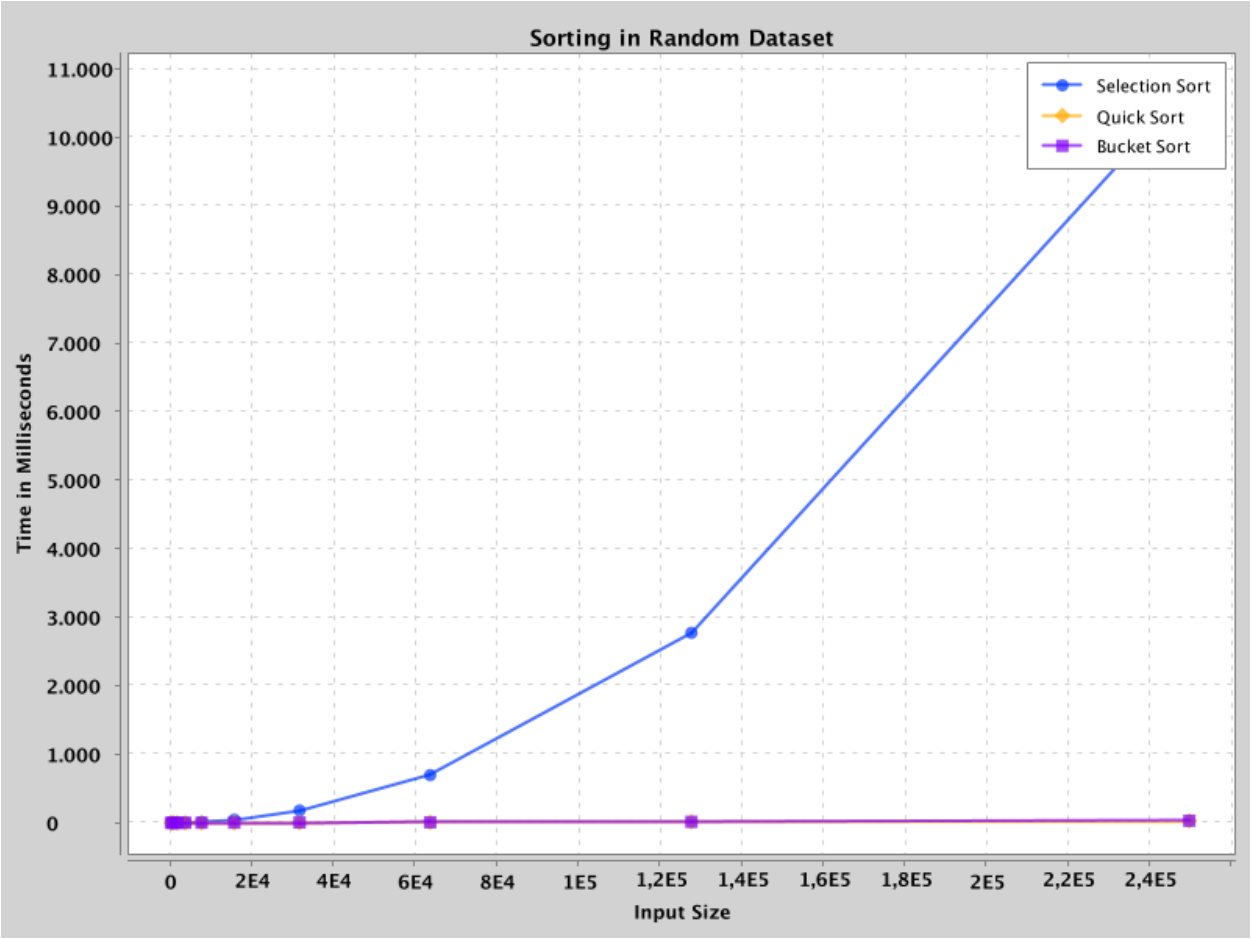

Figure 1: Sorting on random dataset in different sizes.

In Figure 1 on random data, selection sort increases quadratically as the element length of the array increases, while bucket and quick sort progress linearly. When we make a performance comparison, selection sort performs badly, while quick and bucket sort performs much better. (Bucket sort and quick sort lines overlap.)

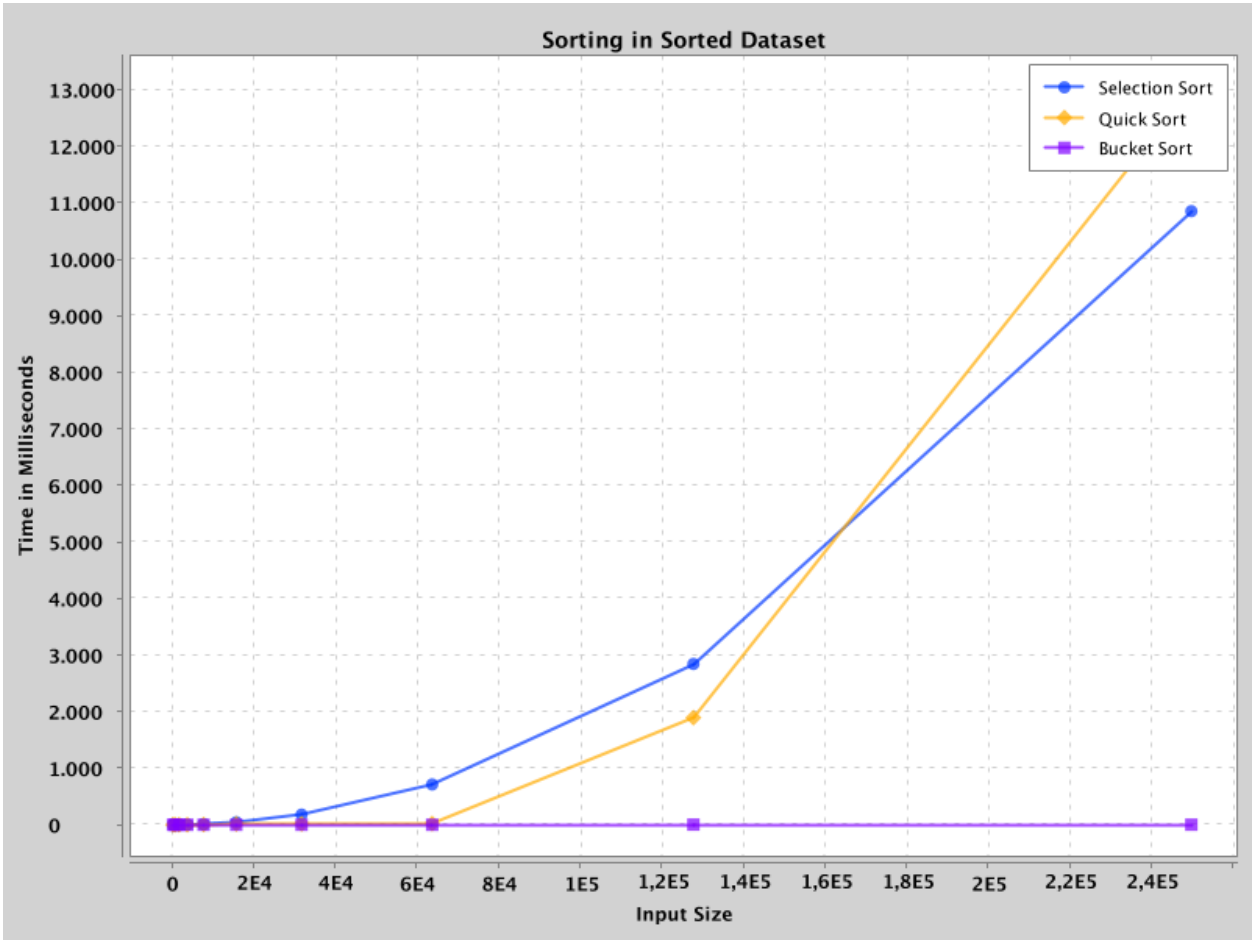

Figure 2: Sorting on sorted dataset in different sizes.

In Figure 2, we observe that on the sorted data, selection sort increases quadratic as the element length of the array increases, while the quick sort array length starts to increase quadratic when it reaches a certain length. In short, if we need to perform sorting on sorted data, we should use bucket sort that performs best.

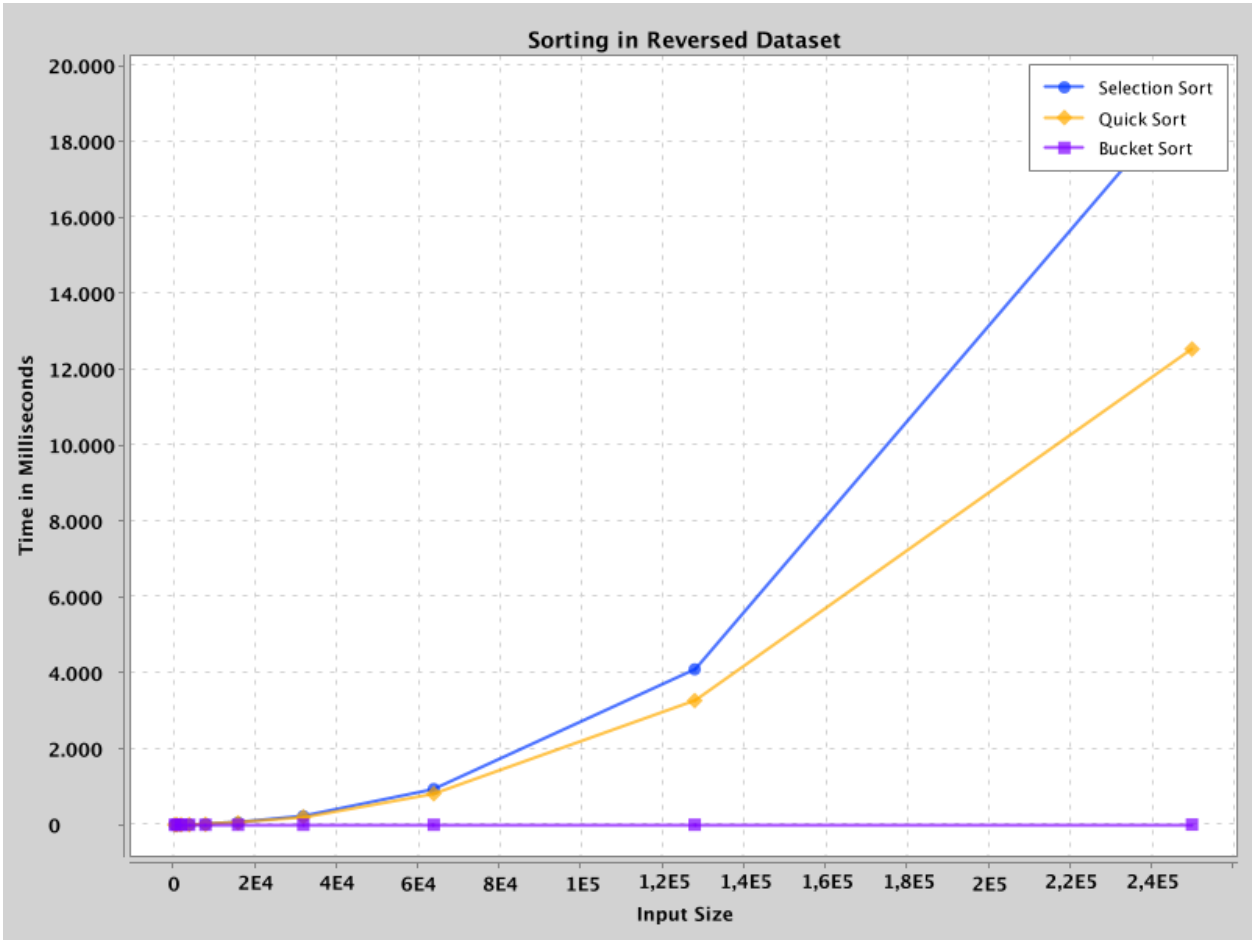

Figure 3: Sorting on reversed dataset in different sizes.

In Figure 3, while we observe that on the reversed data, selection and quick sort increases quadratically bucket sort again shows a linear progression and becomes the sort algorithm with the best performance.

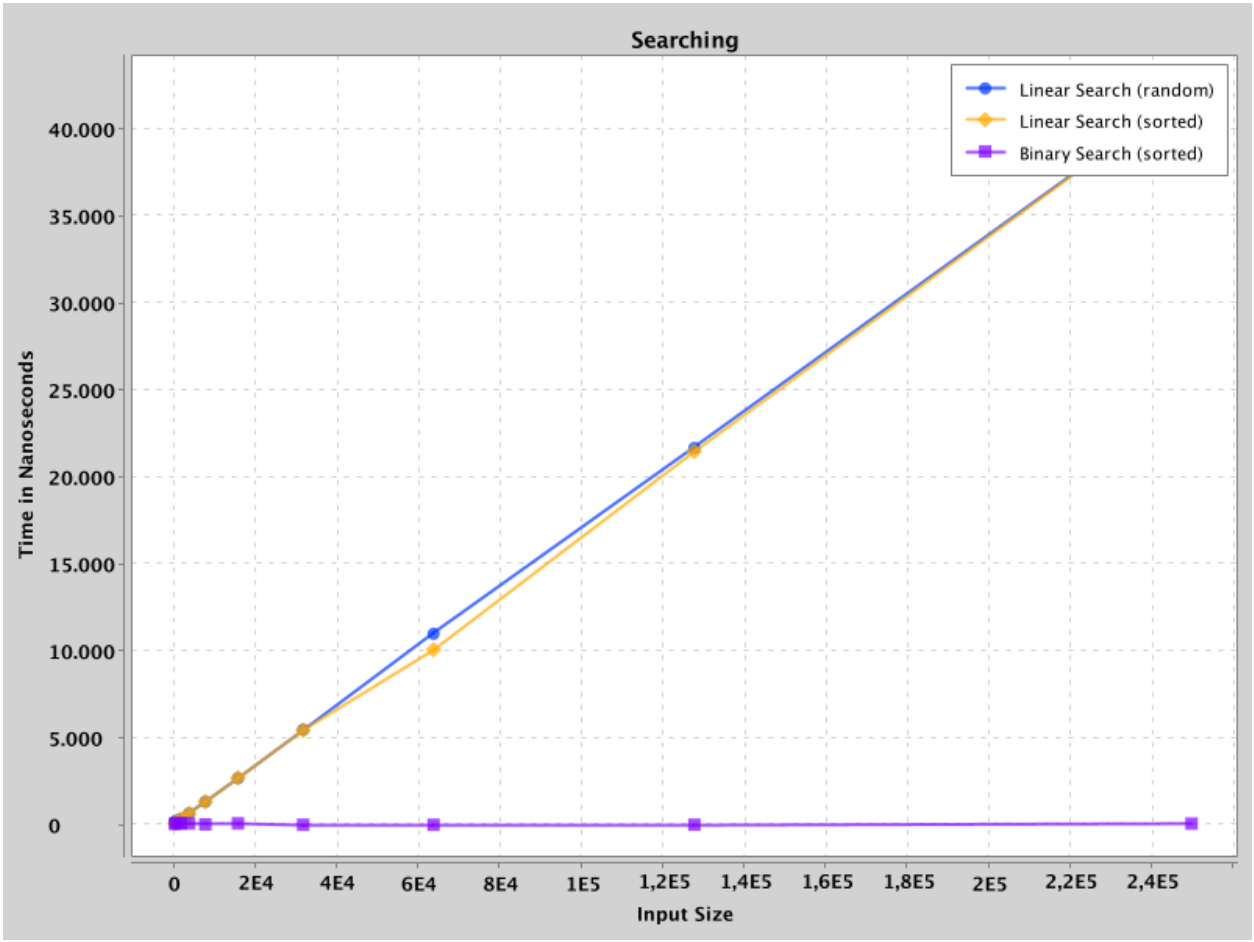

Figure 4: Searching on dataset in different sizes.

In Figure 4, we observe that linear search performs close to each other in both sorted data and random data. When we compare binary search and linear search on sorted data, we observe that binary search performs better.

4 Results Analysis and Discussion

• What are the best, average, and worst cases for the given algorithms in terms of the given input data to be sorted/searched?

As can be seen from table 3, figure 1, figure 2 and figure 3,

Selection sort: Best, average and worst case cases are (n^2) since it shows a quadratic increase in all cases.

Quick sort: Although it has the best case status on random data, it performs poorly on sorted and reversed data. Also, if the input data is already sorted, selection sort may perform better than quick sort. That's why quick sort's best case and average cases are $n \log n$, while the worst case case is n^2.

Bucket sort: showed the best performance in all 3 cases compared to other sorting algorithms. Bucket sorting's best and average cases are n+k, and the worst case is n^2. However, the reason why the bucket sort does not show the worst case on the given inputs is because not all items in the array have the same value or because the maximum value of the array is not small compared to the bucket count. So the average case is n+k instead of n^2.

As can be seen from table 3 and figure 4,

Linear search: In linear searching, its best case is O(1) and its worst case is n. However, since the probability of encountering the best case situation is low, the average case status is also n. It showed O(n) status on both sorted and random input.

Binary search: binary search occurs on sorted input. and its best case is O(1), its worst case is (log n). Therefore, binary search on sorted input performs better than linear search.

• Do the obtained results (running times of your algorithm implementations) match their theoretical asymptotic complexities?

Yes, my experiments and theoretical asymptotic complexities matched.

5 Notes

I understand from this assignment result that, If I need to work on large inputs and space complexity is important, I should use selection sort. If space complexity doesn't matter, I should use bucket sort. In the search process, if my inputs are sorted, I should use binary search to search in the fastest way.

References

- https://www.simplilearn.com/tutorials/data-structure-tutorial/bucket-sort-algorithm
- https://www.scaler.com/topics/data-structures/bucket-sort/
- https://iq.opengenus.org/time-complexity-of-binary-search/
- https://www.upgrad.com/blog/linear-search-vs-binary-search/
- https://iq.opengenus.org/time-complexity-of-selection-sort/
- https://mycareerwise.com/programming/category/sorting/quicksort-iterative