Hacettepe University

Computer Engineering Department

BM204 Software Practicum II - 2023 Spring

Programming Assignment 1

March 27, 2023

Student name:
Kemal Şahin

Student Number:
b2200765021

1 Problem Definition

The main objective of this assignment is to show the relationship between the running time of the algorithm implementations with their theoretical asymptotic complexities.

2 Solution Implementation

2.1 Selection Sort

Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from an unsorted portion of an array or list and placing it at the beginning of the sorted portion. The algorithm works by iterating through the array and selecting the smallest element, then swapping it with the first element. The process is then repeated for the remaining unsorted portion of the array.

```
1  private static int[] sort(int[] arr,int size){
2          int min;
3          int t;
4          for (int i = 0; i < size - 1; i++) {
5              min = i;
6              for (int j = i + 1; j < size; j++) {
7                  if(arr[j] < arr[min]) min = j;
8              }
9              if(min != i){
10                 t = arr[min];
11                 arr[min] = arr[i];
12                 arr[i] = t;
13             }
14         }
15         return arr;
16     }
```

2.2 BucketSort

Bucket sort is a sorting algorithm that works by dividing an array of elements into a number of buckets. Each bucket is then sorted individually, either recursively using bucket sort or using a different sorting algorithm, such as insertion sort or quicksort.

The basic steps of the bucket sort algorithm are as follows:

1- Initialize an array of empty buckets.
2- Iterate through the input array and add each element to its corresponding bucket based on some predefined function or mapping.
3- Sort each non-empty bucket using a sorting algorithm of your choice.
4- Concatenate the sorted buckets together in order to obtain the final sorted array.

```
17      public static void sort(int[] arr,int n){
18          if (n <= 0) return;
19
20          final int numberOfBuckets = (int) Math.sqrt(arr.length);
21          int max = findMax(arr);
22
23          //Create n empty buckets
24          Vector<Integer>[] buckets = new Vector[n];
25          for (int i = 0; i < n; i++) buckets[i] = new Vector<>();
26
27          //Put array elements in different buckets
28          for (int i : arr){
29              buckets[hash(i,max,numberOfBuckets)].add(i);
30          }
31
32          // Sort individual buckets
33          for (int i = 0; i < n; i++) Collections.sort(buckets[i]);
34
35          // 4) Concatenate all buckets into arr[]
36          int index = 0;
37          for (int i = 0; i < n; i++) {
38              for (int j = 0; j < buckets[i].size(); j++) arr[index++] = buckets
                    [i].get(j);
39          }
40      }
41      private static int hash(int i, int max, int numberOfBuckets) { return (int
            ) ((double) i / max * (numberOfBuckets - 1));}
42      private static int findMax(int[] input) {
43          int m = Integer.MIN_VALUE;
44          for (int i : input) m = Math.max(i, m);
45          return m;
46      }
```

2.3 QuickSort

Quicksort is a sorting algorithm that selects a pivot element and partitions the array into two sub-arrays, one with elements smaller than the pivot and one with elements larger. It then recursively sorts these sub-arrays. It has an average-case time complexity of O(n log n), which makes it efficient for large datasets.

```
47    public static void sort(int[] array, int lowIndex, int highIndex) {
48         int stackSize = highIndex - lowIndex + 1;
49         int[] stack = new int[stackSize];
50         int top = -1;
51         stack[++top] = lowIndex;
52         stack[++top] = highIndex;
53
54         while (top >= 0){
55             highIndex = stack[top--];
56             lowIndex = stack[top--];
57             int pivotIndex = partition(array,lowIndex,highIndex);
58             if(pivotIndex - 1 > lowIndex){
59                 stack[++top] = lowIndex;
60                 stack[++top] = pivotIndex - 1;
61             }
62             if(pivotIndex + 1 < highIndex) {
63                 stack[++top] = pivotIndex + 1;
64                 stack[++top] = highIndex;
65             }
66         }
67     }
68
69     private static int partition(int[] array, int lowIndex, int highIndex) {
70         int pivot = array[highIndex];
71         int i = lowIndex - 1;
72         for (int j = lowIndex; j < highIndex; j++) {
73             if (array[j] <= pivot) {
74                 i++;
75                 swap(array, i, j);
76             }
77         }
78         swap(array,i + 1, highIndex);
79         return  i + 1;
80     }
81
82     private static void swap(int[] array, int index1, int index2) {
83         int temp = array[index1];
84         array[index1] = array[index2];
85         array[index2] = temp;
86     }
```

3 Results, Analysis, Discussion

Results show that the running time of sorting algorithms increases with input size, and selection sort has the slowest performance. Quick sort has the best performance, and bucket sort is the most efficient for large datasets. Sorted input data performs better than random input data. The space complexity of quick sort and bucket sort is higher than the other algorithms. The theoretical asymptotic complexities of the algorithms match their running times experimentally.
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 seconds).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in seconds									
Selection sort	4.0E-4	3.0E-4	7.0E-4	0.003	0.0114	0.0443	0.1705	0.6803	2.7822	10.6191
Bucket sort	3.0E-4	4.0E-4	4.0E-4	8.0E-4	0.0012	0.002	0.0049	0.0119	0.0186	0.0203
Quick sort	7.0E-4	4.0E-4	0.0016	0.0055	0.0187	0.0631	0.2493	0.9943	4.0003	15.235
	Sorted Input Data Timing Results in seconds									
Selection sort	0.0	2.0E-4	7.0E-4	0.0028	0.0108	0.0425	0.1686	0.6799	2.6992	10.3378
Bucket sort	0.0	1.0E-4	1.0E-4	2.0E-4	4.0E-4	0.001	0.0017	0.0038	0.0081	0.0159
Quick sort	0.0	3.0E-4	0.0011	0.0047	0.0167	0.0675	0.2702	1.0857	4.3715	16.4987
	Reversely Sorted Input Data Timing Results in seconds									
Selection sort	1.0E-4	2.0E-4	8.0E-4	0.003	0.0114	0.0454	0.1785	0.7253	2.9422	11.6212
Bucket sort	0.0	1.0E-4	1.0E-4	3.0E-4	5.0E-4	0.001	0.0019	0.0039	0.0082	0.0157
Quick sort	1.0E-4	3.0E-4	0.001	0.0045	0.0173	0.0698	0.2779	1.1071	4.4225	15.174

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 seconds).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	0.010528	0.012221	0.017757	0.004537	0.007899	0.013858	0.028144	0.055772	0.109623	0.184652
Linear search (sorted data)	6.17E-4	8.79E-4	0.001025	0.00243	0.003605	0.012426	0.034368	0.073268	0.149955	0.296684
Binary search (sorted data)	9.87E-4	6.27E-4	6.26E-4	6.91E-4	6.52E-4	8.17E-4	9.59E-4	0.001035	0.001541	0.001445

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	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
Linear Search	$\Omega(n)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(\log n)$	$\Theta(\log n)$	$O(\log n)$

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	$O(1)$

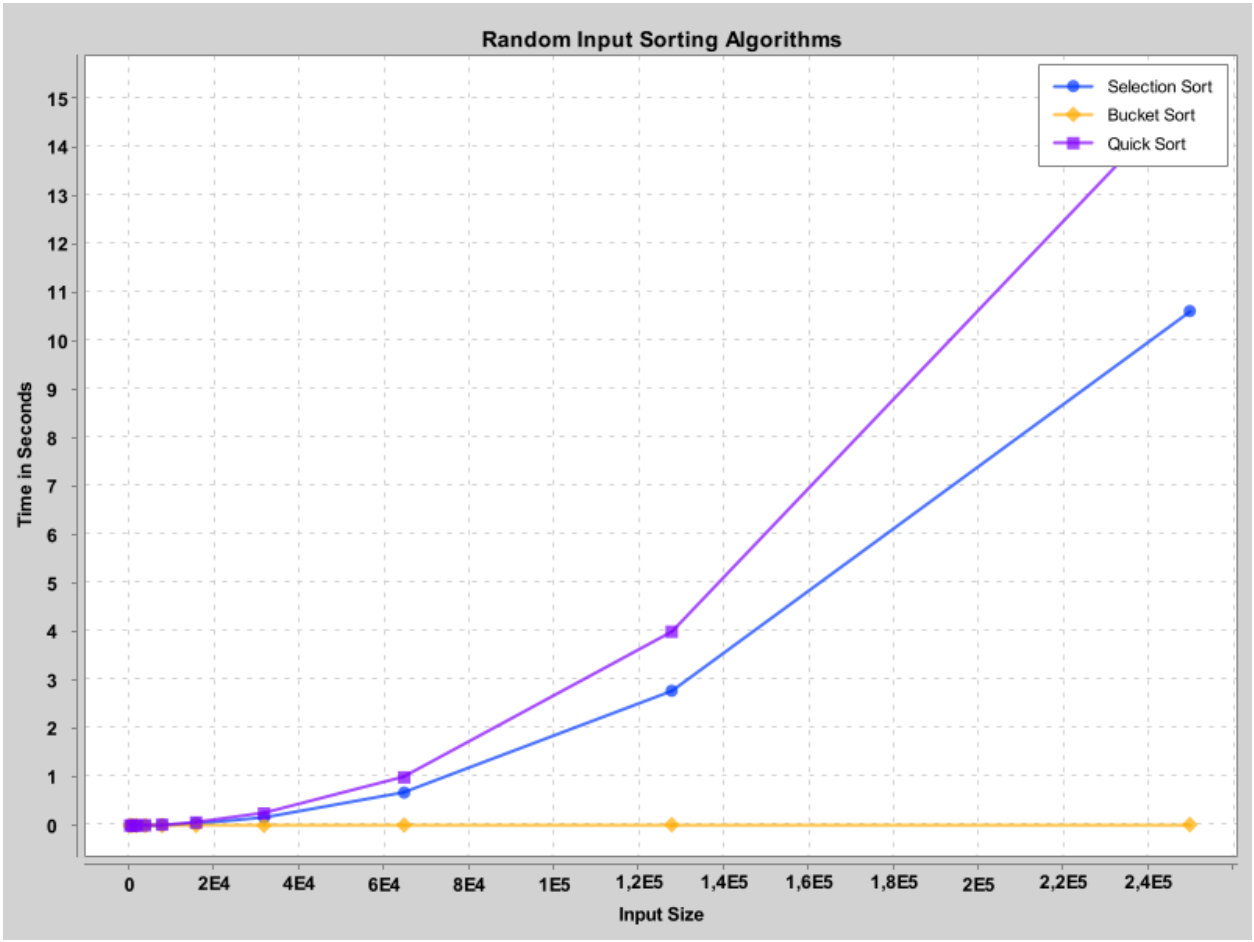

Figure 1: Random input Sorting Algorithms

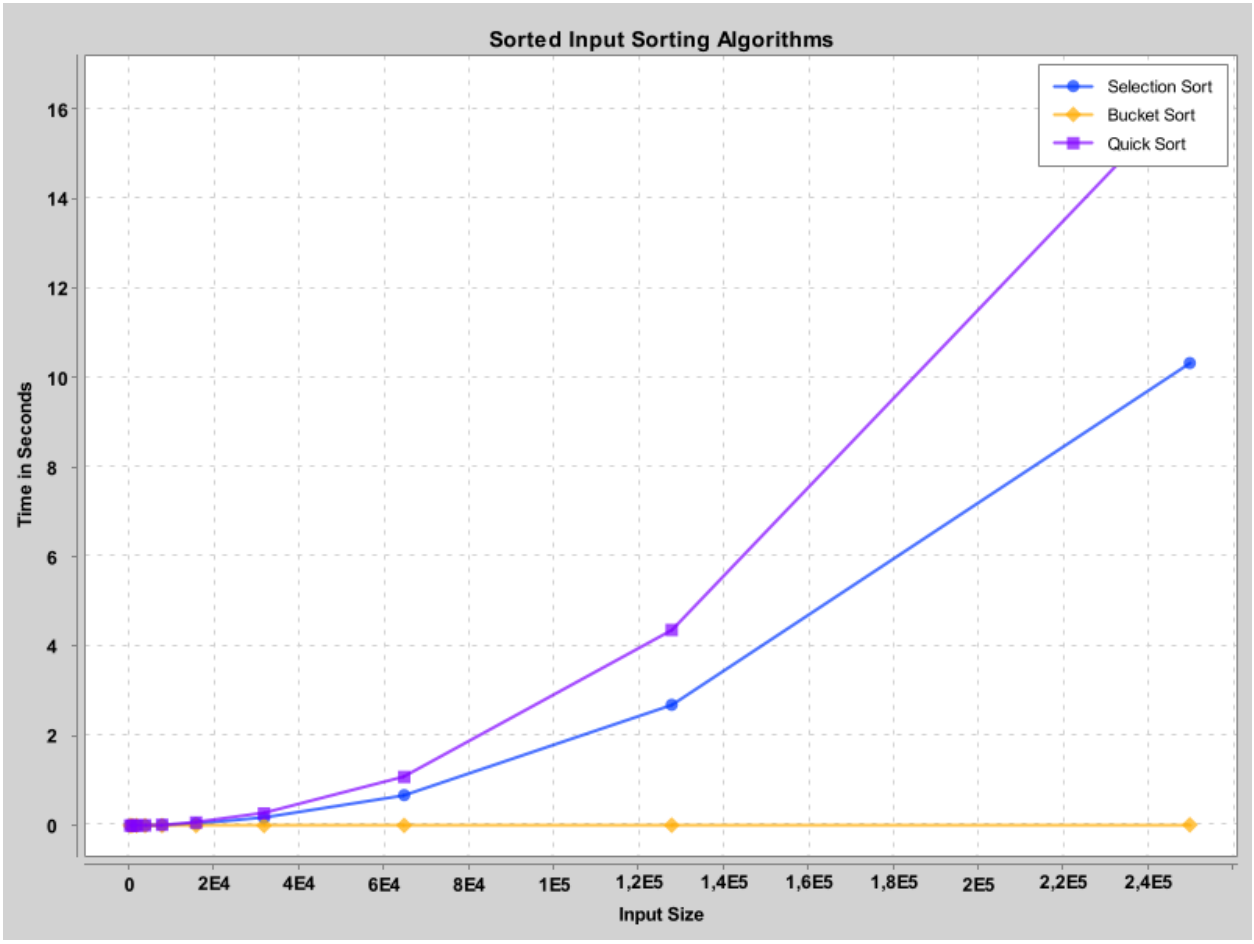

Figure 2: Sorted input Sorting Algorithms

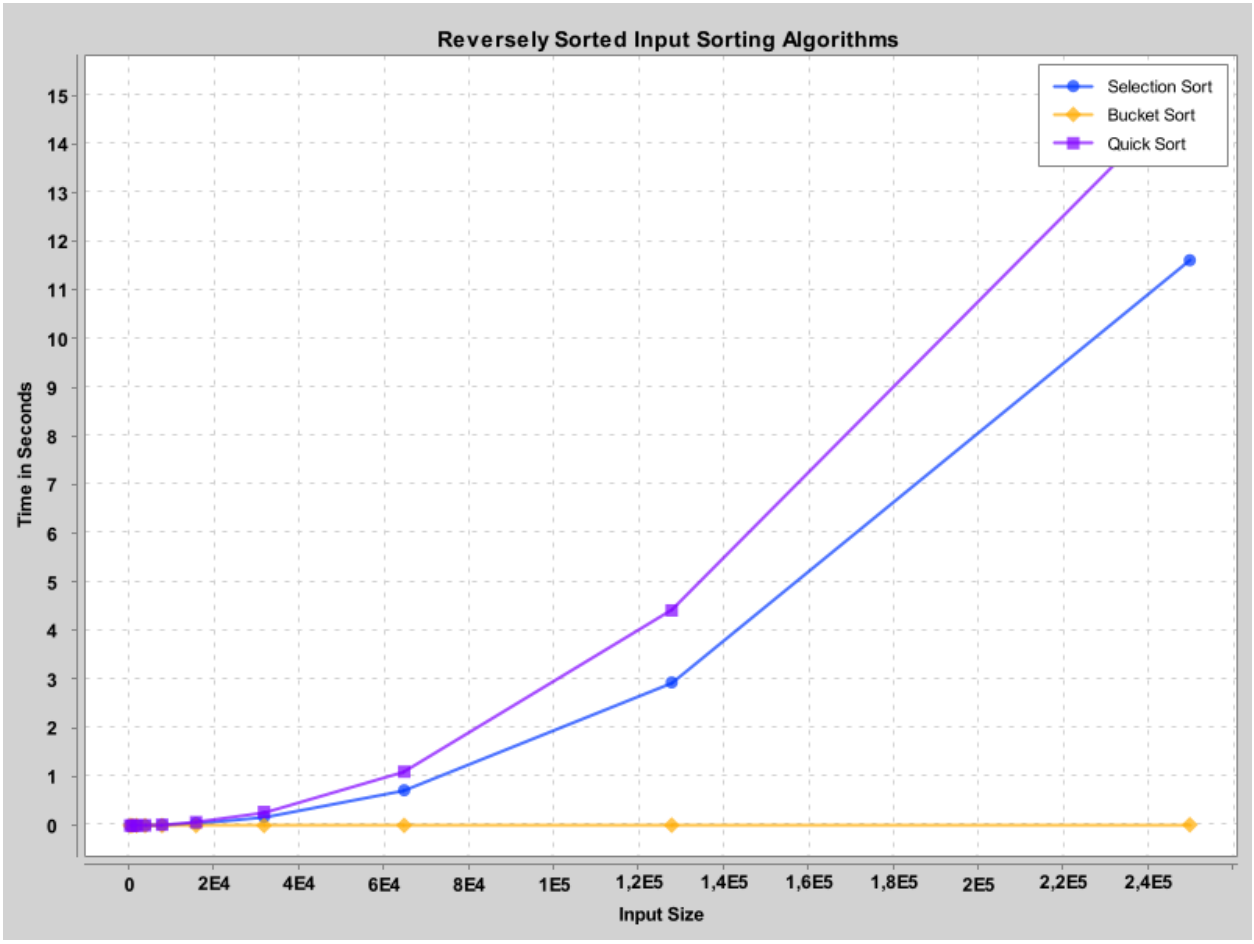

Figure 3: Reversely sorted input Sorting Algorithms

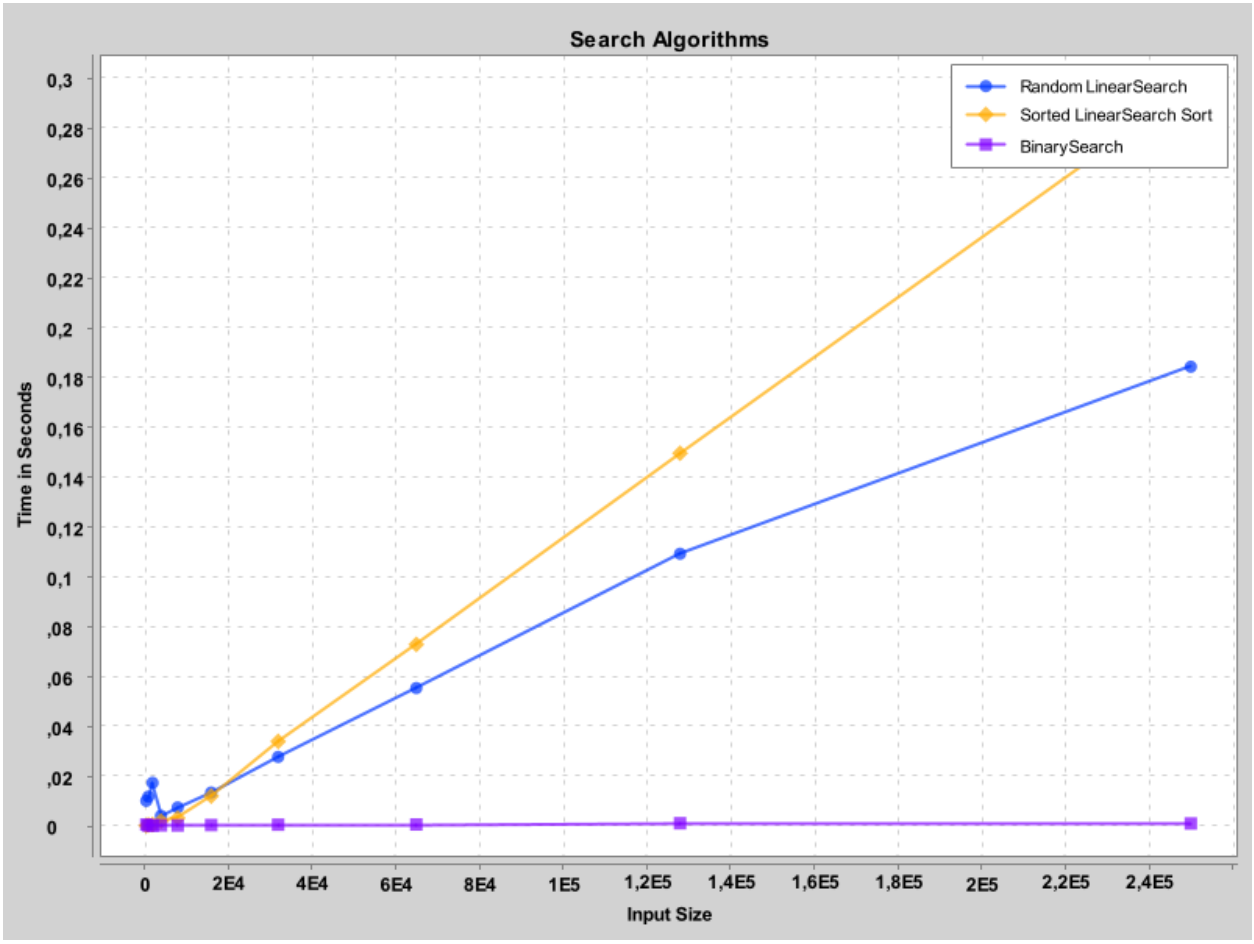

Figure 4: Search Algorithms

The results of this analysis demonstrate the importance of selecting the appropriate sorting algorithm for the given dataset size and input type. Selection sort may be appropriate for small datasets, but it quickly becomes inefficient as the size increases. Quick sort and bucket sort are better suited for larger datasets, with quick sort performing better overall and bucket sort being more efficient for large datasets.

The performance difference between sorted and random input data is also significant, with sorted input data leading to much faster running times for all three sorting algorithms. This suggests that if the input data can be sorted beforehand, it can greatly improve the efficiency of the sorting algorithm.

The space complexity of the algorithms is another important consideration. While selection sort, linear search, and binary search have constant space complexity, quick sort and bucket sort require additional memory space. This should be taken into account when selecting an algorithm, especially if memory usage is a concern.

Overall, this analysis highlights the importance of understanding the theoretical asymptotic complexities of sorting algorithms and selecting the appropriate algorithm based on the dataset size and input type. It also underscores the importance of considering the space complexity of the algorithms and the potential impact on memory usage.

References

- "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (MIT Press)
- "Algorithms, Part I" and "Algorithms, Part II" courses on Coursera by Robert Sedgewick and Kevin Wayne (Princeton University)
- "Sorting Algorithms" by GeeksforGeeks (https://www.geeksforgeeks.org/sorting-algorithms/)
- "Sorting Algorithms" by Khan Academy (https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/sorting)
- "Sorting Algorithms: A Comprehensive Guide" by Sarah Brown (https://www.freecodecamp.org/news/sort algorithms-explained/)