# Hacettepe University Computer Science Department

## BBM 204 – Programming Assignment 1

*Name:* Mert

*Surname:* ÇÖKELEK

*ID:* 21727082

*Email:* b21727082@cs.hacettepe.edu.tr

*Due Date:* 25.03.2019

*Subject:* Analysis of sorting and searching algorithms.

*Advisor:* Burçak ASAL, Alaettin UÇAN.

## Problem Definition:

In this assignment, we are expected to write java codes for binary search, some sorting algorithms and ‘calculate their execution times’ with respect to the sizes of the arrays and finally ‘visualize the results’ we obtained for three different cases: Best (Sorted), Average (Random), Worst(Reverse-Sorted) arrays.

The results will give us idea about the time complexities of the algorithms and their dependencies on the three cases.

With these operations, we will improve our ‘algorithm analysis skills’ and experience the corresponding time complexities.

## Findings:

For this purpose, I created 3 different test cases and 10 different array sizes for every single algorithm.

The three cases for the sorting algorithms are:

1- Worst Case: Reverse Sorted Array

2- Average Case: Shuffled Array

3- Best Case: Ordered Array

The three cases for binary search:

1- Worst Case: The searching key does not exist in the array

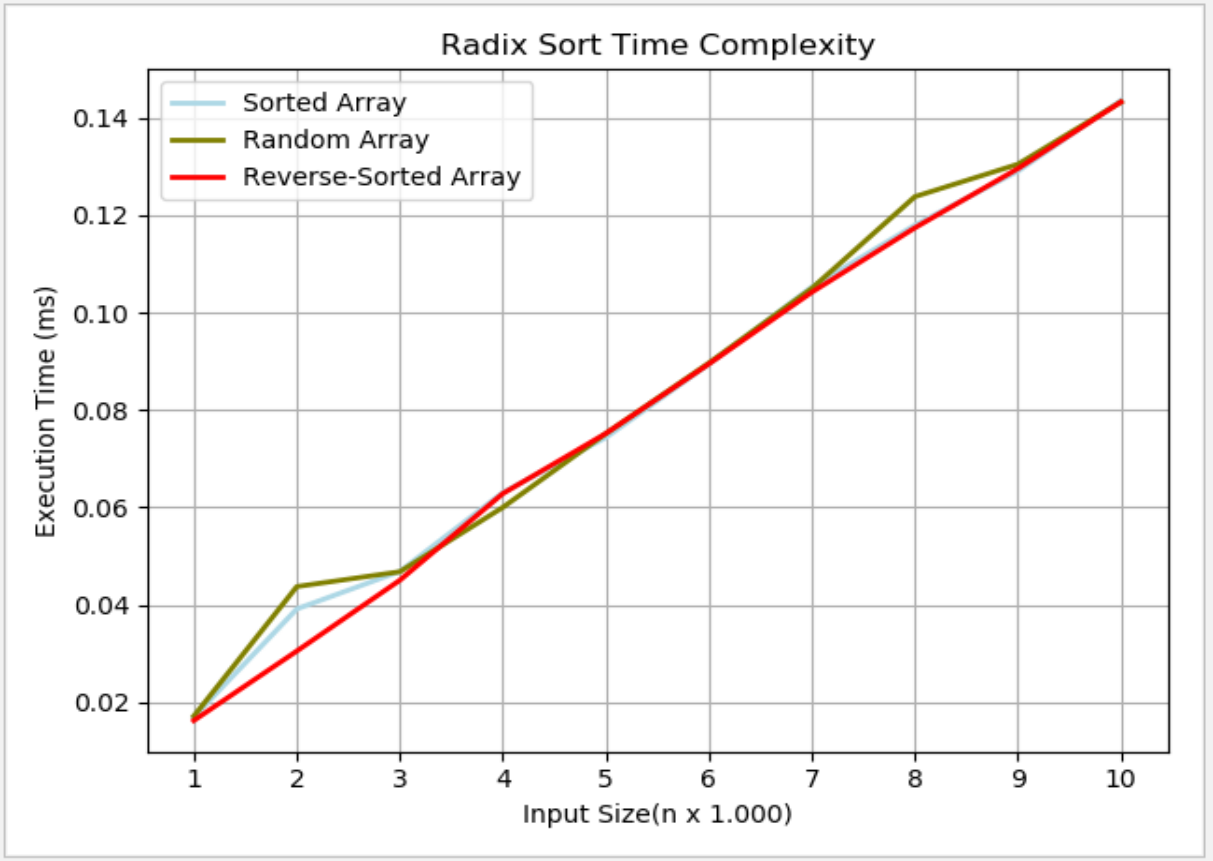
2- Average Case: Searching for any element in the array

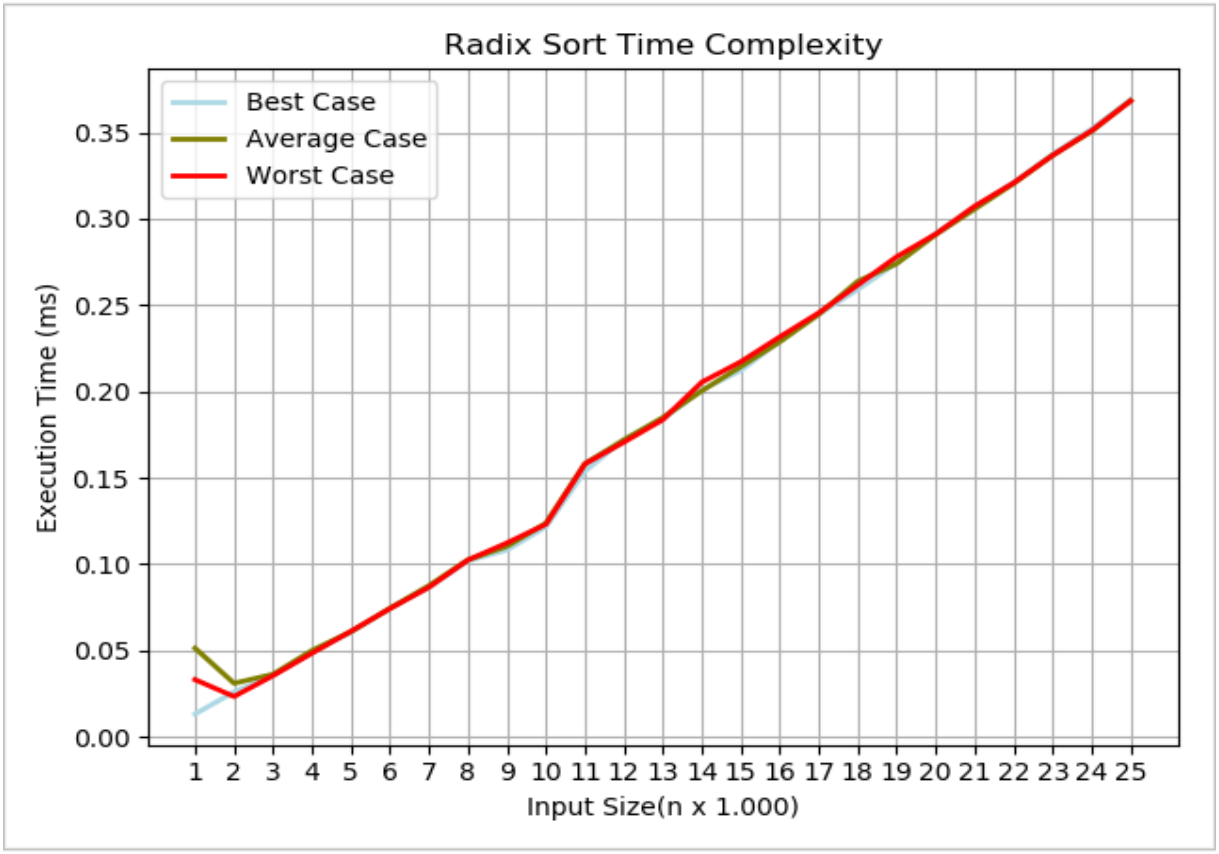
3- Best Case: Searching the middle element in the array

These three cases are used on 10 different sizes ranging from 1000 to 10000 for sorting, and 100.000 to 1.000.000 for searching algorithms.

For every size and case combinations, the execution time calculations are done 10 different times, and the average value for them is taken and plotted.

Here are the corresponding plots:


Radix Sort Time Complexity
Sorted Array
Random Array
Reverse-Sorted Array
Execution Time (ms)
0.02
0.04
0.06
0.08
0.10
0.12
0.14
1
2
3
4
5
6
7
8
9
10
Input Size(n x 1.000)


Radix Sort Time Complexity
Best Case
Average Case
Worst Case
Execution Time (ms)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Input Size(n x 1.000)

**For Radix Sort [O(n*k)]:**

***The General Logic of the Algorithm:***

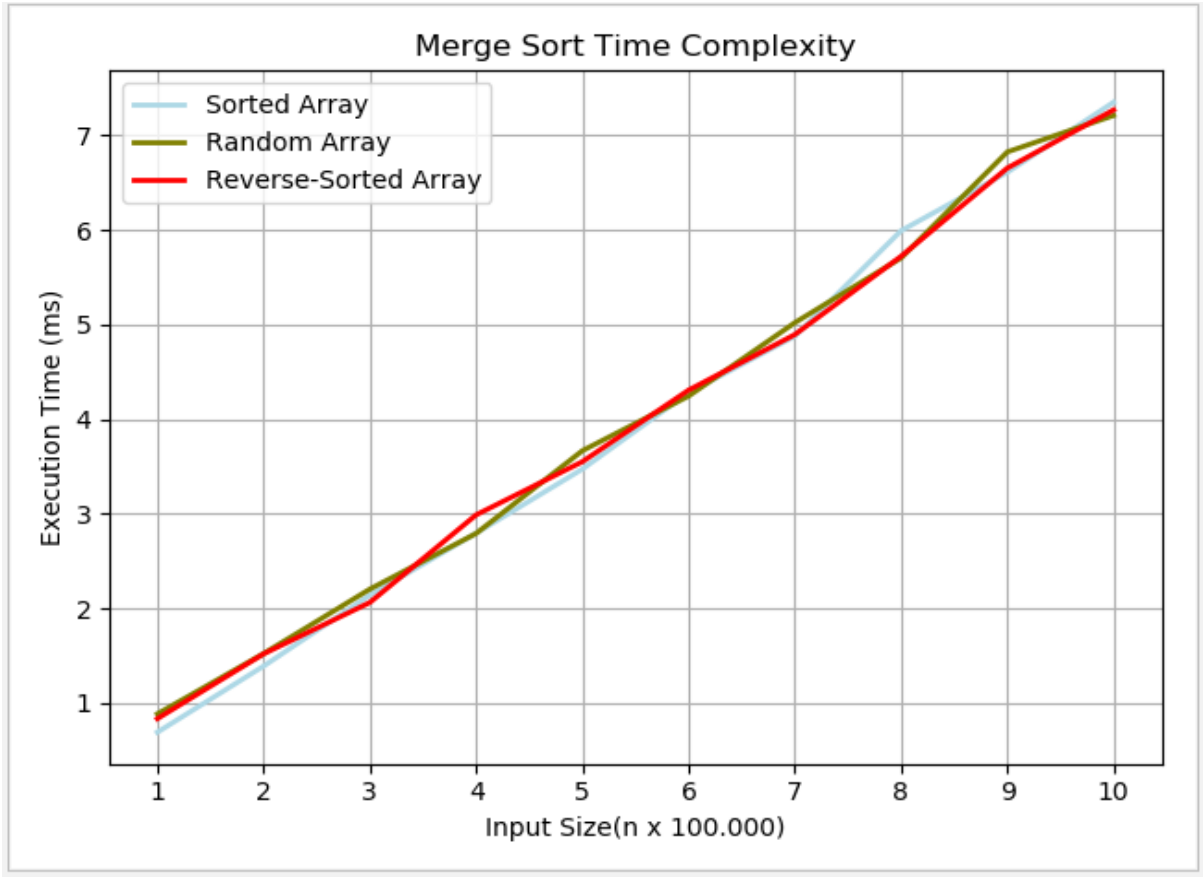
Radix Sort is based on sorting the elements by their digits, from the least significent digit to the most. To do so, it uses “Counting Sort” which is a linear time, not comparison based sorting algorithm. It works by counting the number of elements which have same values. And after some arithmetic, calculates the positions of each object for the returning array.

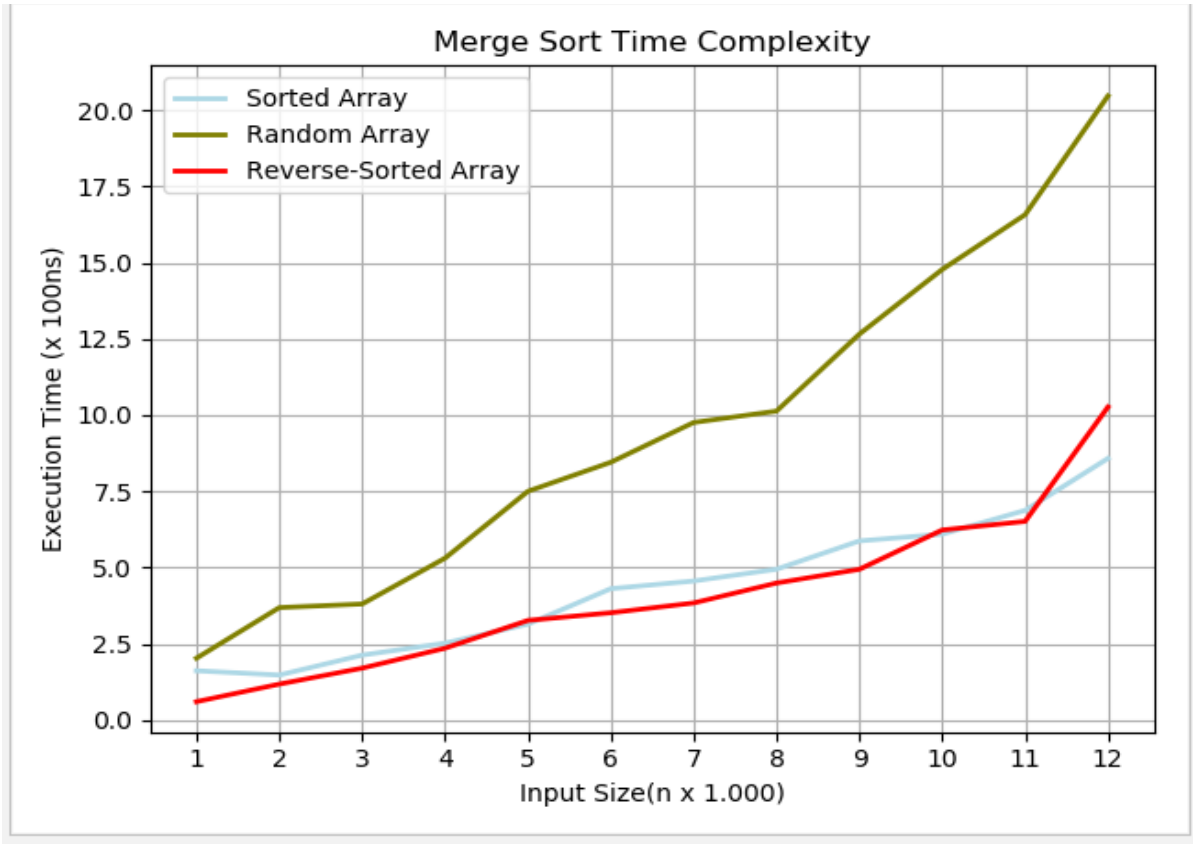
***The Analysis of the Graph:***

We can see that the execution times for three different array combinations do not differ, this shows that Radix Sort Algorithm is input-independent. We can see this from the code, radix sort uses “counting sort” for every digits of the elements which has a complexity O(n+k) when the elements are in range(1, k).

Say the largest number in the array has ‘d’ digits. Then, Radix sort will use Counting Sort d times (Complexity: O(d * (n+k)). K is in range 10 for decimal numbers. When the array size goes infinity, the overall complexity is limited by O(nlogbn) == O(n*k) since d is slightly smaller than n. (Linear Time Complexity)

We can see this from the graph, the points are in a line form.


Merge Sort Time Complexity
Sorted Array
Random Array
Reverse-Sorted Array
Execution Time (ms)
Input Size(n x 100.000)


Merge Sort Time Complexity
Sorted Array
Random Array
Reverse-Sorted Array
Execution Time (x 100ns)
Input Size(n x 1.000)

**For Merge Sort [O(nlogn)]:**

The General Logic of the Algorithm:

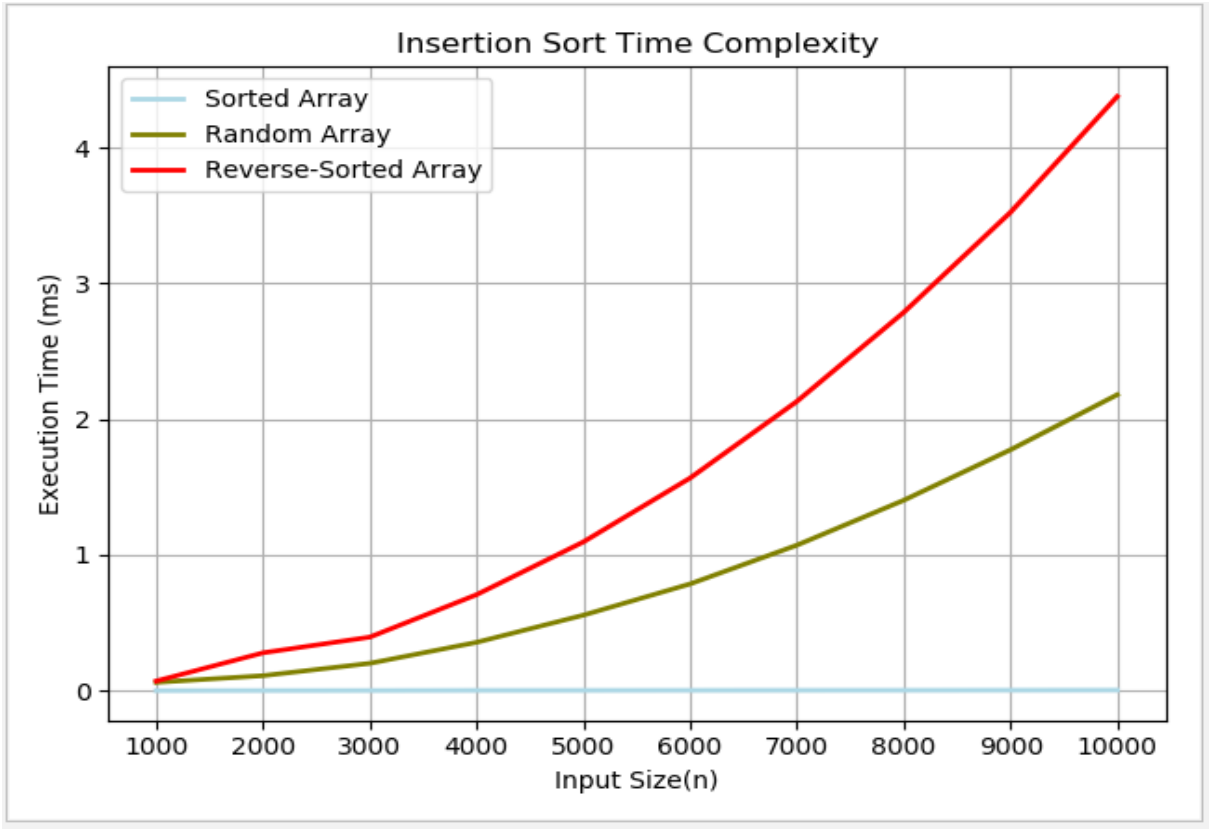
Merge Sort is a recursive algorithm that works by dividing the current array into two halves until reaching arrays of 1 element. So we can imagine this as a recursion tree. From the bottom to the top of the tree, the elements are combined such that the upper level array is the ordered collection of its children. This operation is processed until reaching the root of the tree (The complete array).

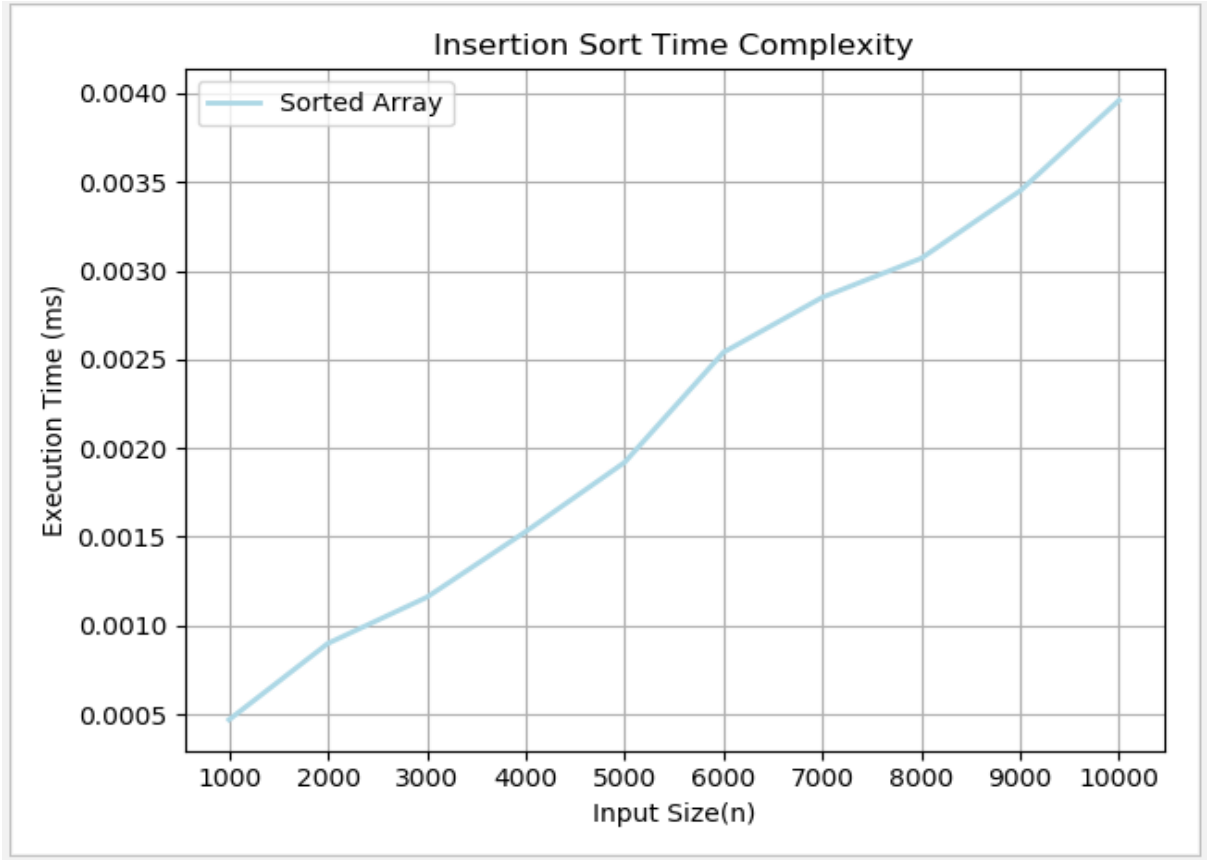
We can also see that the execution times for three different cases are so close. This shows that the Merge Sort Algorithm is also input-independent. Because no matter how the array elements are ordered, it always divides the current arrays into two parts and finally merges them in an ordered way. The time is nearly-linearly (because the logarithmic part does not effect the visual so much.) increasing by the input size.

From the graph, we see that the case with shuffled array is a bit slower than the reversed and ordered ones. This is because the program enters the if-else blocks, makes more array accesses at shuffled case. This happens when array size is around (1000- 10000). When the array sizes are around (100000- 1000000), their complexities are closing to each other as we can see from the two graphs above.

We can also see the complexity of this algorithm by the code:

At each step, it takes n times to merge the child arrays and the total number of merges (height of the recursion tree) is logn(base 2), because at each step we divide current arrays into 2 halves. For each step, there are n comparisons for an initial array of size “n” and there are logn steps. So, the overall complexity of the algorithm is **O(nlogn),** which is linearithmic.


Insertion Sort Time Complexity
Sorted Array
Random Array
Reverse-Sorted Array
Execution Time (ms)
0
1
2
3
4
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input Size(n)


Insertion Sort Time Complexity
Sorted Array
Execution Time (ms)
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
0.0035
0.0040
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input Size(n)

**For Insertion Sort [O(n) - O(n^2)]:**

We can see that the times are different for three cases:

Best Case is linear [O(N)],

Average Case is quadratic,

Worst Case is exactly double of Average Case.

Which shows us that Insertion sort is input-dependent and for worst case, it is a quadratic(O(N^2)) algorithm.

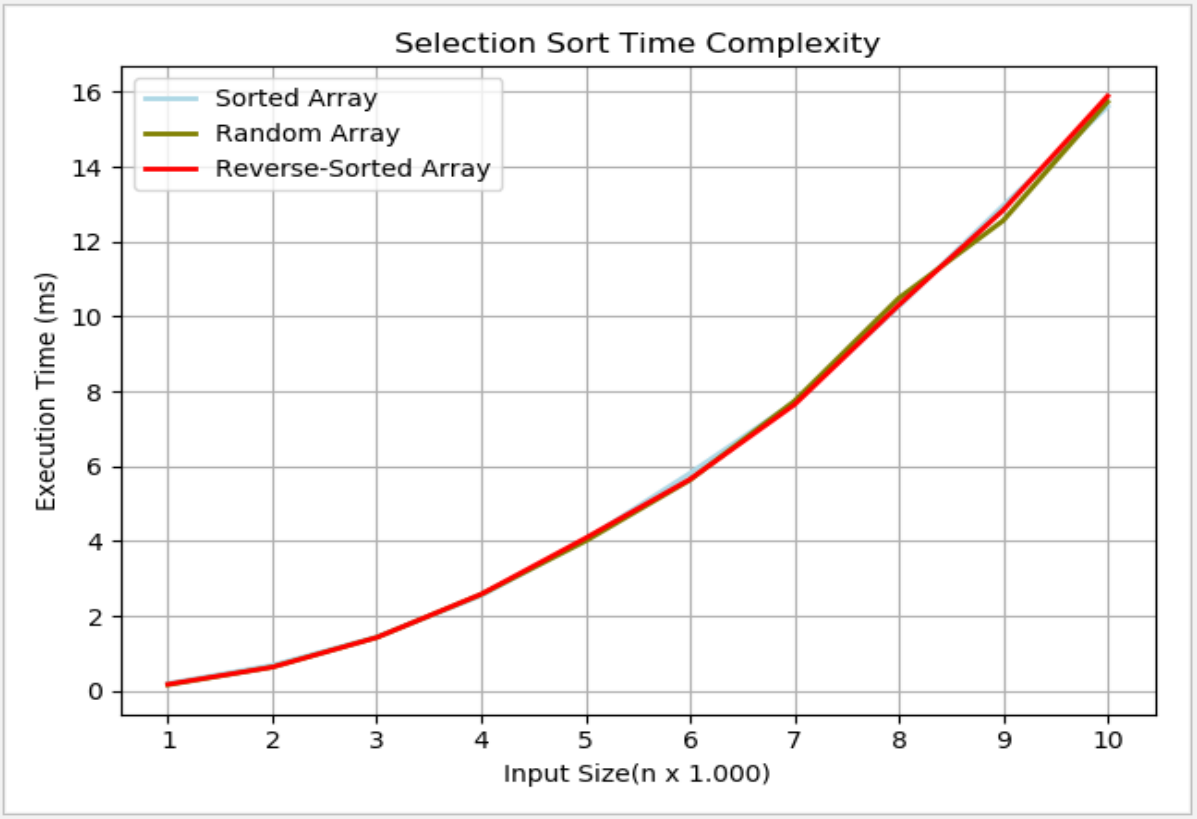
```
private void sort(int[] array){
    int n = array.length;
    int key, j;
    for(int i = 0; i < n; i++){
        key = array[i];
        j = i - 1;
        while(j >= 0 && array[j] > key){
            array[j+1] = array[j];
            j-=1;
        }
        array[j+1] = key;
    }
}
```

We can see this from the code, there are 2 for loops inside, and first one (i) goes from 0 to n-1, the second (j) goes from (i+1) to n. And at each step, all the largest elements at the right of the I'th element are swapped with the I'th element.

**For worst case,** it is a reversed ordered array and needs to swap (n-i-1) elements each step. There are n steps and it totally goes to **(n-1) + (n-2) + (n-3) + …. + (2) + (1) = (n-1) * (n) / 2 == ½ n^2.**

**For average case:** it is a shuffled array and for any step, there are approximately (n-i+1)/2 larger elements at the right side of the I'th element, so the total number of swap operations are: **(n-1)/2 + (n-2)/2 + … + (2)/2 + (1)/2 == 1/4(n^2)**

**For best case:** it is an ordered array, so it will iterate over the array until finding a larger element at the right of the I'th element, and it won't find any. So the required time is only to traverse the array of n items. **Complexity: n**




**For selection Sort [O(n^2)]:**

We can see that the execution time does not depend on the three cases. All the execution times are growing [O(N^2)].

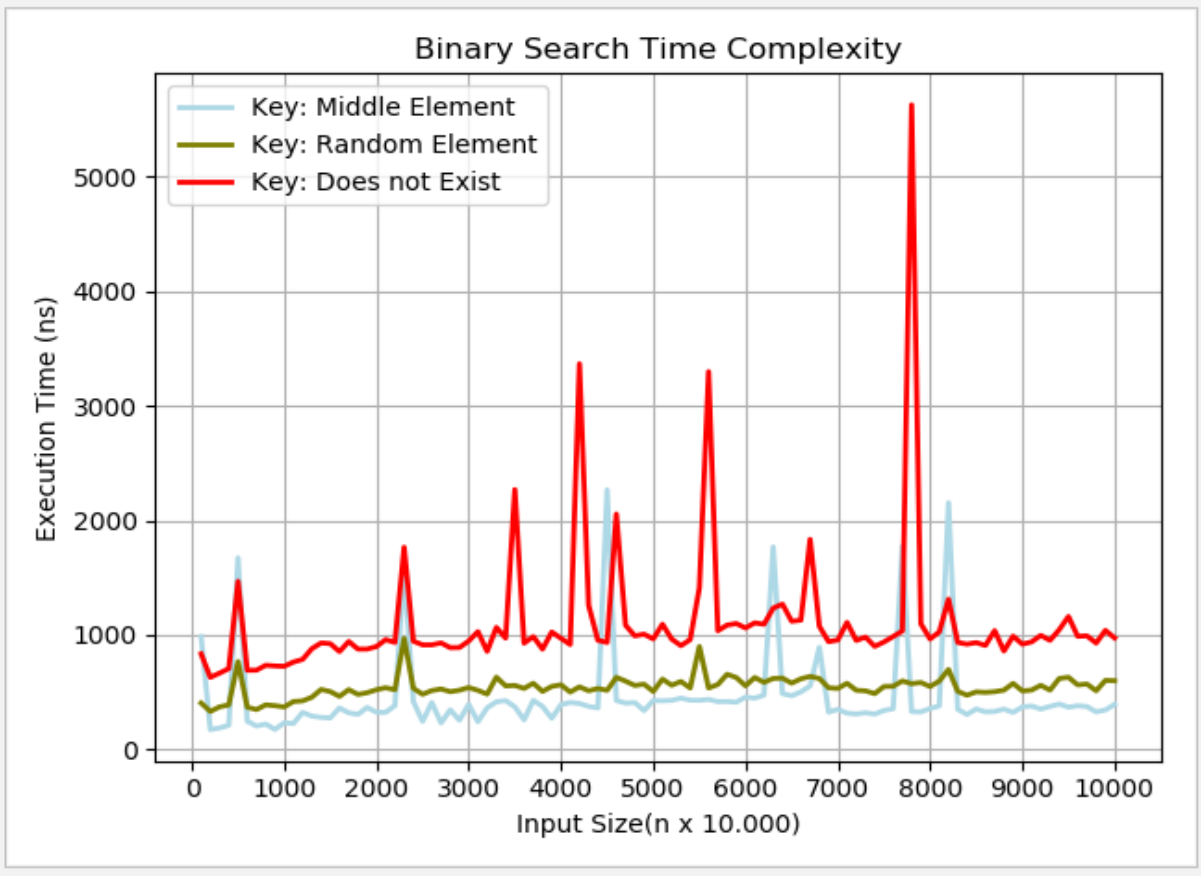
```
private void sort(int[] array){
    int n = array.length;
    int min;
    for(int i = 0; i < n-1; i++) {
        min = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[min])
                min = j;
        }
        swap(array, min, i);
    }
}
```

We can see this from the code: No matter how the array is ordered, it always uses 2 for loops, first one (i) is ranging from 0 to n and the second one (j) is from i+1 to n.

At each step, it searches for the minimum largest element at the right side of the i'th element.

**For any case,** it will use 2 for loops (1 for iterating for the current element and 2nd for iterating at the right side of the current element) and at every time, to find the minimum larger element at the right side of the current element, no matter what the case is, the

program will iterate over (n-i-1) elements for i in range n, so the overall complexity is: (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1 = n * (n-1) / 2 == (½ n^2). = [O(N^2)]




**For Binary Search [O(1) - O(logn)]:**

Binary search is a recursive algorithm that compares the key with the middle element of the array until finding the correct match. If the key and the mid element are not the same, the ordered array is decremented to its half, so at each time the complexity decremented by ½. For initial n elements:

**Best Case:** Finding the key at the first time (middle element): The complexity is O(1), no other operations needed.

**Worst Case:** The key does not exist: The program will look for the key at logn(base2) compares, finally it will return -1 since there are no more elements to compare with.

**Average Case:** The key is randomly placed in the array, so it will take more than O(1) and less than O(logn) because the possible maximum number of compares is logn(base2).

[½ logn]

**The irregularity of binary search graphs** is because the binary search is a very fast algorithm (logn), its execution time can be calculated by nanoseconds. And since nanoseconds are very sensitive, the result is effected from the background processes, CPU clock cycles, current network situation.. In best case of binary search, it's O(1) but the reasons above can effect the correct result.

## Discussion:

The most important part – about how to analyse the algorithms and calculate their complexities – of this assignment is this part. The sample code for calculation is shown as below:

```
void results(double[] worst, double[] avg, double[] best){
    long et = 0, st = 0;
    /* st: starting time */
    /* et: ending time  */
    for(int Time = 0; Time < 10; Time++){
        for(int i = 0; i < 10; i++) {
            /* create an array of (i * 1000) size */
            int[] arr = create_random_array( size: (i+1) * 1000);

            /* Average Case, Array is shuffled. */
            st = System.nanoTime();
            sort(arr);
            et = System.nanoTime();
            avg[i] += (et - st) / 10e6;

            /* Now the array is sorted, Best Case */
            st = System.nanoTime();
            sort(arr);
            et = System.nanoTime();
            best[i] += (et - st) / 10e6;


            arr = reverse_array(arr);

            /*Now the array is reversed, Worst Case*/
            st = System.nanoTime();
            sort(arr);
            et = System.nanoTime();
            worst[i] += (et - st) / 10e6;
        }
    }
}
```

- This is the corresponding method that calculates the execution times of the sorting algorithms. It takes 3 parameters for the arrays, which are holding the exe times for worst, average and best cases.
- ***In the first loop***, "Time" variable is updated 10 times and for each time, in the second for loop, an array is created with different sizes. (i = loop variable and every array is at size of thousand times i).
- ***In the second for loop***, the sort methods are called three times (Worst, Average, Best cases) and each time, the starting and finishing times are stored in the corresponding arrays.

- ***The idea behind this method*** is to create different arrays to see the difference on different sizes, by this way we can have ideas about the overall complexity of the corresponding algorithm over input size.
- These operations are being done 10 times for every algorithm, to increment the accuracy of the execution times. This will make the results more precise.

The exexution times are taken in nanoseconds and divided by 10e6 to get a more clear number in terms of milliseconds.

My Codes in Detailed Explanation:

In my main class (Assignment1.java) I defined 5 methods for printing the results of the corresponding sorting algorithms.

**`Private static void print_selection(),`**

**`Private static void print_insertion(),`**

**`Private static void print_merge(),`**

**`Private static void print_radix(),`**

**`Private static void print_search().`**

The parameters of these methods are:

Output file and 3 arrays holding the execution times for worst, average and best cases.

Their common purpose is to print all the cases of all the algorithms to “output.txt”.

Other than the main class, there is an abstract class “Sort”, which holds the general methods required in almost every sorting algorithm.

These are:

**`Private static int binary_search(int[] arr, int key):`** Returns the index of the key in the array. If it does not exist, returns –1. Used for analysis and “create_random_array” method.

**`Static int[] reverse_array(int[] arr):`** Returns the reversed version of the parameter.

**`Static int[] create_random_array(int n):`** Returns a shuffled array of size n.

**`Static int[] create_ordered_array(int n):`** Returns an ordered array of size n.

**`Static int[] shuffle_array(int[]arr):`** Shuffles and returns the parameter.

**`Static void swap(int[] array, int j, int min):`** Swaps the j’th and min’th elements in array.

The Sort classes that extend this class are:

Merge.java, Insertion.java, Selection.java, Radix.java.

These classes also contain the unique sorting codes, and a spesific method named “Results”.

This Result Method has been explained at “Discussion” Part.

And the last class is Binary_Search.

It contains methods for binary search and “Results”.