Al Maaref University

Faculty of Sciences

Department of Computer Science

Comparative Study of Sorting Algorithms

Done by: Fatima Wisam Khadra

Course: CSC 420 Algorithms

Instructor: Dr Imad Jawhar

Spring 2021-2022

Table of Contents

Table of Tables

Table of Figures

Introduction

Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output (Cormen et al.).

One of the problems that can be solved using algorithms is the sorting problems. There are many ways to solve it thus sort objects. This study will focus on the following set of sorting algorithms which are: Insertion Sort, Merge Sort, Heap Sort, Bubble Sort, Quick Sort, Counting Sort and Selection Sort. All these are sorting algorithms each sorts in a different manner and procedure.

The objective of this paper is to compare these different sorting algorithms with respect to their average execution time to answer the question which algorithm is the fastest?

Hypothesis

The hypothesis of this study will be set based on previous information and knowledge about the selected algorithms.

Analyzing an algorithm and studying its performance has come to mean predicting the resources -memory, communication bandwidth, computer hardware- that the algorithm requires (Cormen, Leiserson, Rivest, & Stein). Analyzing different algorithms for one problem permits identifying the most efficient one. In this study, algorithms are analyzed and classified based on their measured computational time which is mainly affected by the size of the input. In general, the time taken by an algorithm grows with the size of the input. Thus, the running time of a program is measured with respect to the size of its input. Input size depends on the studied problem. It is the number of items in the input. For sorting algorithms, it's the array size n. The running time of an algorithm on a specific input is the number of steps or operations executed.

Because the running time interests us and characterizes an algorithm's efficiency, the order of growth is studied. The order of growth is the relation that studies the variation of the input size with respect to the running time of an algorithm using asymptotic notations. It allows

us to compare the performance of alternative algorithms and compare how the running time increases with the input size (Cormen, Leiserson, Rivest, & Stein).

Each algorithm, has a best-case scenario, average case and worst case. Each case has its running time. Table 1 shows the asymptotic running time of the selected sorting algorithms in all their cases. We use the O notation which indicates an asymptotic upper bound.

Table 1. Order of Growth of different sorting algorithms

Sorting Algorithm	Best Case	Average Case	Worst Case
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Because asymptotic notations describe running time in terms of functions, the order of growth in Table 1 will be displayed in a graph. Figure 1 shows the results.

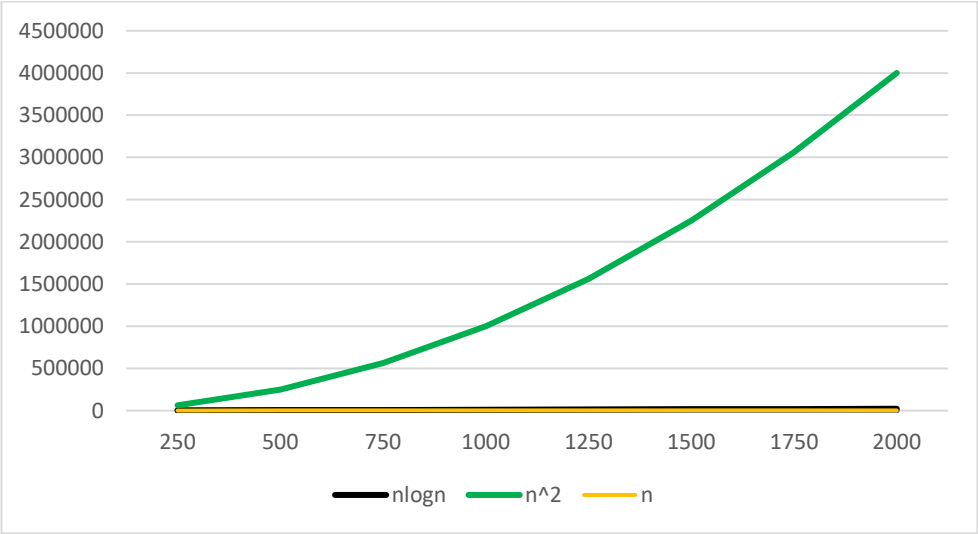


Figure 1. The order of growth functions graphs

Based on the results of the above graph, n^2 is the upper bound of nlogn the upper bound of n. In general, an algorithm whose running time has a higher order of growth might take more running time for large input. For example, $O(n^2)$ algorithm takes more time to run an input of size 1000 than a $O(n)$ algorithm. Thus, we hypothesize that Counting Sort is faster in execution than Insertion Sort and Bubble Sort. Selection sort is the slowest for large n. Merge, Heap and Quick Sort are so close in execution time. This paper will conduct a small study to test this hypothesis.

Methodology

The aim of this study is to evaluate the behavior of some sorting algorithms with respect to n number of elements. First, the sorting algorithms -insertion sort, merge sort, heap sort, quick sort, selection sort, bubble sort, counting sort- are programmed using Java. Using OOP-Java permits having each algorithm in a separate class which will organize more the interaction between these algorithms in different classes in the preceding steps. Then, the evaluator is also programmed using Java in a separate class called Evaluation. The aim of this class is to evaluate the computation time of each of the above sorting algorithms at different sizes. This class follows a set of steps:

1. *n* variable is declared representing the number of elements. For clearer observations and efficient results, *n* ranges between 50 and 2000 incrementing 10 at each step.
2. At each *n*, an array of size *n* made up of random elements is initialized. Random arrays are generated rather than user-defined arrays to assure the proper work of each algorithm.
    2.1. Each random array generated will invoke each sorting algorithm respectively. At the same time, the time -in nano seconds- taken once invoking the algorithm and once it is done executing is calculated. Each algorithm runs 500 times to ensure the minimum accuracy of the Java compiler when executing in case it is affected by any other running application.
    2.2. The average computation time of each algorithm is saved in a Java Map. Maps are chosen rather than Arrays because Maps are more feasible in adding

elements to them. Also, in this case, the average time which is the value belonging to a sorting algorithm which is the key. So, the Map allows saving such mapped data in an easier efficient way.

2.3. Then, the average computation time of all algorithms for each *n* is saved in a Java Map as well.

3. The average computation time for each *n* is displayed and saved in a JSON file for farther usages. JSON -JavaScript Object Notation- is a file format used to represent JS objects, arrays and data. JSON is chosen in this study because it is relatively easy to read and write for humans and software.

Note that, all Java classes for this study are not designed to execute on any input type array, i.e. the classes are not Generic. For the sake of simplicity, *int* primitive type is only considered.

After collecting the data of the average computation time for every algorithm for each *n*, the results must be displayed. Python Language is used to plot the results because Java has no ready packages for this purpose unlike Python.

Code

This section will explain explicitly the implementation of the algorithms, the evaluator and the plotter.

Insertion Sort

One of the ways to solve a sorting problem is using insertion method. Insertion sort is an efficient algorithm for sorting a small number of elements in place (Cormen et al.). The algorithm takes a sequence of numbers set in an array *A* which also contains the sorted output sequence when the procedure is finished. Figure 2 shows the implementation of the Insertion Sort algorithm. It is implemented in a separate class of its name. The *InsertionSort* class has two declared static variables *key* and *I* in addition to two methods: *insertionSort* which applies the sorting algorithm and the *print* which handles printing the output in a specific format. The aim of *key* is to save the value at a specific position. The *insertionSort* method is set *static* in order to access it without a calling object. Its return type is void because the algorithm applies its sorting

in place on the input array. Thus, no need to return any new object. The method also takes one input parameter which is of type *int* called *terms* in this case. *Terms* is the array that holds the elements to be sorted. The algorithm sorts them by looping over *terms*. It starts iterating from 1 not 0 considering that the element at index 0 is a subarray trivially sorted. This subarray is initially the base start of the comparison. Every element at every index is initialized to the declared variable *key* and *i* is initialized to the value of the index *j*-1. Then, as far as *i* is positive and the element at it is greater than *key*, push this element to the position directly after and decrement *i.* Finally, set the value held by *key* in the *i*+1 position.

In order to print the output result, the print method iterates over its input parameter array, an appends the elements at every index respectively to the object *str* of the class type *StringBuilder*. Then it prints *str.*

```
public class InsertionSort {
    static int key,i;

    public static void insertionSort(int[] terms){
        for(int j=1;j<terms.length;j++){
            key=terms[j];
            i=j-1;
            while(i>=0 && terms[i]>key){
                terms[i+1]=terms[i];
                i=i-1;
            }
            terms[i+1]=key;
        }
    }
    public static void print(int[] terms){
        StringBuilder str=new StringBuilder();
        for(int i=0;i<terms.length;i++){
            str.append(terms[i]+", ");
        }
        System.out.println(str);
    }
```

Figure 2. InsertionSort Class

To test this algorithm if working properly, a *main* method is added. An array of type *int* is declared and initialized having 6 elements. This array is then passed as an argument to the methods *insertionSort* to be sorted and the *print* to print the sorted array. Figure 3 shows the main method for this class. Figure 4 shows the output which validates the work of the algorithm. The elements are set in order after calling the insertion sort algorithm.

```java
public static void main(String[] args) {
    int[] terms={5,2,4,6,1,3};
    System.out.println("The first array before sorting: ");
    print(terms);
    System.out.println("The first array after sorting: ");
    insertionSort(terms);
    print(terms);
```

Figure 3. The main method of the InsertionSort Class

```
The first array before sorting:
5, 2, 4, 6, 1, 3,
The first array after sorting:
1, 2, 3, 4, 5, 6,
```

Figure 4. The output of the insertion sort algorithm

Merge Sort

Merge sort algorithm uses the divide and conquer approach in sorting elements. it recursively divides the problem into smaller ones then merges their solutions forming a single sorted array. In order to apply this approach, this algorithm is divided into two methods: *Merge* and *Merge-Sort* which respectively apply divide and conquer. Figure 5 shows the class *MergeSort* implementing the *merge* method. This method takes four input parameters: an array of type *int* representing the array to be sorted, and three integers that respectively represent the start of the array, mid and end. The method's body declares two integers *n1* and *n2* and initializes two arrays of their sizes. These arrays represent the right and left subarrays, where each will have a copy of the elements from the main input array. This copying process which assembles divide step in the approach is done using the for loops in lines 7 through 12. Declaring a MAX_VALUE -infinity- at the end of each subarray permits having no smaller card in case one array is over. Then a for loop iterates over the input array. If element in the right subarray is greater than that in the left, the left element is set to the input array at its corresponding *k* position. Else, the right element is set.

```java
public class MergeSort {
    private static void merge(int[] arr,int p, int q, int r){
        int n1=q-p+1;
        int n2= r-q;
        int[] l = new int[n1+1];
        int[] R = new int[n2+1];
        for(int i=0;i<n1;i++){
            l[i]=arr[p+i];
        }
        for(int j=0;j<n2;j++){
            R[j]=arr[q+1+j];
        }
        Integer maxNb=Integer.MAX_VALUE;
        l[n1]= maxNb;
        R[n2]= maxNb;
        int i=0;int j=0;
        for(int k=p;k<= r;k++){
            if(l[i]<= R[j]){
                arr[k]=l[i];
                i=i+1;}
            else
            {
                arr[k]= R[j];
                j=j+1;
```

Figure 5. MergeSort Class-Merge Method

But in order to keep dividing the problem to save time, another method is defined which is *mergesort* in Figure 6. It takes as parameters an array of type int and two integers representing the start and end of the array. In order to execute, it makes sure that the start didn't reach the end yet. It initializes the variable *q* to the mid value of the array. Recursively, *merge* is invoked until a subarray of 1 element is reached, i.e. the start and end will then be equal, so, the method won't execute. When the base case is reached -having one-element-subarray-, *merge* is invoked to order and sort the elements.

```java
public static void mergesort(int[] arr,int p, int r){
    if(p<r) {
        int q = (int) Math.floor((p + r) / 2);
        mergesort(arr, p, q);
        mergesort(arr, p: q + 1, r);
        merge(arr, p, q, r);
    }
}

public static void print(int [] terms){
    StringBuilder str=new StringBuilder();
    for(int i=0;i<terms.length;i++){
        str.append(terms[i]+", ");
    }
    System.out.println(str);
}
```

Figure 6. The MergeSort Class -mergeSort method

Figure 7 shows a sample *int* array passed to the *mergeSort* to test the algorithm. Figure 8 shows the output where it is clear that elements of the array are sorted. Thus the algorithm is working properly.

```java
public static void main(String[] args){
    int arr[] = { 6, 5, 12, 10, 9, 1 };

    mergesort(arr, p: 0, r: arr.length-1);
    System.out.println("Sorted array: ");
    print(arr);
}
```

Figure 7. The main method of the MergeSort Class

```
Sorted array:
1, 5, 6, 9, 10, 12,
```

Figure 8. The output of the merge sort algorithm

Heap Sort

Heap is an array viewed as a binary tree where every node is an element in the array (Cormen et al). The heap sort algorithm sorts a heap after ensuring that it's a heap. So, the heap sort algorithm invokes the build max heap algorithm to build a max heap with the help of the max heapify algorithm that maintains the max-heap property. Figure 9 shows the *HeapSort* class that has its corresponding implementation. The three methods *parent, left* and *right* return the number of the respective nodes. *maxheapify* method takes an array of *int* which represents the array to be sorted and two integers which represent the starting node and the size of the array respectively. The results of the *left* and *right* methods are initialized to the declared variables *l* and *r.* By comparing the element at each node -left and right nodes- to that at the parent *i,* the largest node is identified. Based on the largest node, swapping is done in order to maintain the largest at the top. This method is called recursively to go over the whole heap and maintain its max property. Figure 10 shows the *buildMaxHeap* method which converts an array into a max heap in a bottom up manner by running *maxHeapify* on each node (Cormen, Leiserson, Rivest, & Stein). Also, the figure shows the *heapsort* method. This method sorts its input array parameter. First, it builds a max heap from the array elements. Then, it iterates over the heap from bottom to up and swaps the element at top which is the largest with the smallest element set at the bottom. After this swap, it is essential to call *maxHeapify* to maintain again the max heap property and select the largest root. In order to test whether *these* algorithms are sorting properly, a *main* method is added as shown in Figure 11. It prints the output using the *print* method which does the same as explained before. Figure 12 validates the algorithms. Numbers are sorted properly by *buildMaxHeapify* and *heapSort*. Notice that the result of the *maxHeapify* validates the max heap property. Each node(i) is greater than its left(2i) and right(2i+1).

```java
public class HeapSort {

    private static double parent(int i) { return Math.floor(i/2); }
    private static int left(int i) { return (2*i)+1; }
    private static int right(int i) { return (2*i)+2; }

    private static void maxHeapify(int[] arr, int i, int size){
        int l=left(i);
        int r=right(i);
        int largest;
        if(l< size && arr[l]>arr[i])
            largest=l;
        else
            largest=i;
        if(r< size && arr[r]>(arr[largest]))
            largest=r;
        if(largest!=i)
        {
            int temp=  arr[largest];
            arr[largest]=arr[i];
            arr[i]=  temp;
            maxHeapify(arr,largest,size);
        }
    }
```

Figure 9. The HeapSort Class-maxHeapify method

```java
private static void buildMaxHeap(int[] arr){
    for(double i=Math.floor(arr.length/2)-1; i>-1;i--){
        maxHeapify(arr,(int)Math.round(i), arr.length);
    }
}

public static void heapSort(int[] arr){
    buildMaxHeap(arr);
    int length= arr.length;
    for(int i= length-1;i>0;i--){
        int temp=arr[0];
        arr[0]=arr[i];
        arr[i]=temp;
        length=length-1;
        maxHeapify(arr, i: 0,i);

    }
}
```

Figure 10. The HeapSort Class- build-max-heap and heapSort methods

```java
public static void print(int[] terms){
    StringBuilder str=new StringBuilder();
    for(int i=0;i<terms.length;i++){
        str.append(terms[i]+", ");
    }
    System.out.println(str);
}
public static void main(String[] args){
    System.out.println("Max Heapify:");
    int[] arr ={27,17,3,16,13,10,1,5,7,12,4,8,9,0};
    maxHeapify(arr, i: 2, arr.length);
    print(arr);

    System.out.println("\n################Build Max Heapify");
    int[] arr2={4,1,3,2,16,9,10,14,8,7};
    buildMaxHeap(arr2);
    print(arr2);

    System.out.println("\n#############Heap Sort");
    int[] arr3={16,14,10,8,7,9,3,2,4,1};
    heapSort(arr3);
    print(arr3);
}
```

Figure 11. The main and print methods of the HeapSort Class

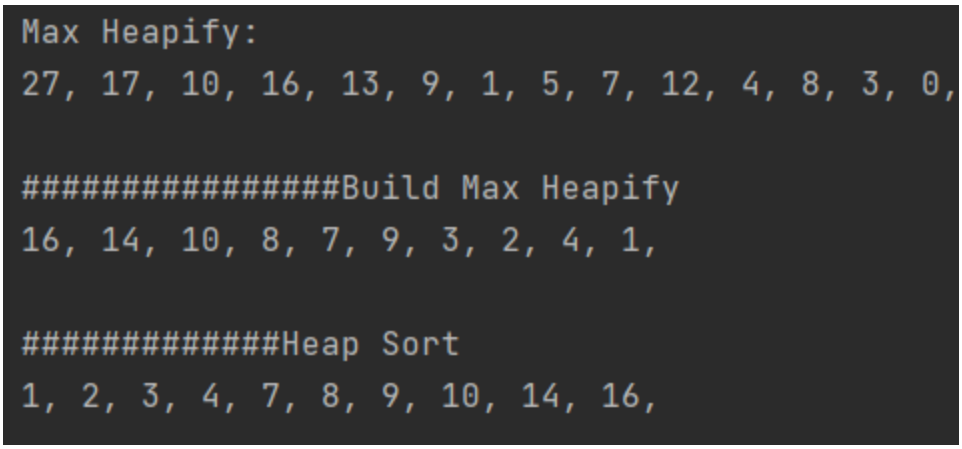


Figure 12. The output of the HeapSort Class

Quick Sort

Quick sort applies divide and conquer as merge sort. Figure 13 shows its implementation. It divides the array using *partition* method and then conquers the subarrays by recursively calling *quicksort* (Cormen, Leiserson, Rivest, & Stein). *Partition* takes an array with its start and end points as parameters. It returns an integer which is initialized the *int q*. *q* identifies the borders of the subarray that are to be divided recursively in *quicksort. Partition* starts by selecting a pivot element *x*. Then, the for loop iterates over the array. Once the comparison is satisfied, elements are exchanged to maintain order. The pivot is then swapped with the leftmost element greater than *x,* thereby moving the pivot into its correct place in the partitional array and returning its new index (Cormen, Leiserson, Rivest, & Stein). This swapping divides the array into two subarrays where the pivot is greater than its leftmost elements and less than its right most element. Figure 14 shows the *main* method of the class as well as the output which is properly sorted.

```java
public class QuickSort {
    public static  void quickSort(int[] arr, int p, int r){
        if(p<r){
            int q=partition(arr,p,r);
            quickSort(arr,p, r: q-1);
            quickSort(arr, p: q+1,r);
        }
    }
    private static  int partition(int[] arr, int p, int r){
        int x=arr[r];
        int i=p-1;
        for(int j=p;j<r;j++){
            if(arr[j]<=x){
                i++;
                int temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
        int temp=arr[i+1];
        arr[i+1]=arr[r];
        arr[r]=temp;
        return i+1;
    }
```

Figure 13. The QuickSort Class

```
    public static void print(int[] terms){
        StringBuilder str=new StringBuilder();
        for(int i=0;i<terms.length;i++){
            str.append(terms[i]+", ");
        }
        System.out.println(str);
    }

    public static void main(String[] args) {
        System.out.println("Quick Sort");
        int[] arr ={2,8,7,1,3,5,6,4};
        quickSort(arr, p: 0,  r: arr.length-1);
        print(arr);
    }
}
```

```
QuickSort
C:\Users\user\.jdks\openjdk-15.0.2\bin\java.exe
Quick Sort
1, 2, 3, 4, 5, 6, 7, 8,
```

Figure 14. The main and print methods of the QuickSort class and its output

Counting Sort

The below figure shows the class implementing the counting sort algorithm.

```
public class CountingSort {
public static void countSort(int[] array, int size) {
    int[] output = new int[size + 1];
    int max = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max)
            max = array[i];
    }
    int[] count = new int[max + 1];
    for (int i = 0; i < max; ++i) {
        count[i] = 0;
    }
    for (int i = 0; i < size; i++) {
        count[array[i]]++;
    }
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }
    for (int i = size - 1; i >= 0; i--) {
        output[count[array[i]] - 1] = array[i];
        count[array[i]]--;
    }
    for (int i = 0; i < size; i++) {
        array[i] = output[i];
    }
```

Figure 15. The CountingSort Class

Figure 16 shows the main method of this class. The output validates the work of the algorithm where it shows that the numbers in the input array are properly sorted.

```
    public static void print(int[] terms){
        StringBuilder str=new StringBuilder();
        for(int i=0;i<terms.length;i++){
            str.append(terms[i]+", ");
        }
        System.out.println(str);
    }

    public static void main(String[] args) {
        System.out.println("Counting Sort");
        int[] arr ={ 4, 2, 2, 8, 3, 3, 1 };
        countSort(arr,arr.length);
        print(arr);
    }
}
```

```
CountingSort
C:\Users\user\.jdks\openjdk-15.0.2\bin\java.exe
Counting Sort
1, 2, 2, 3, 3, 4, 8,
```

Figure 16. The main and print method of the Counting Sort with the output

Bubble Sort

The figure below also shows the implementation of the bubble sort algorithm in a class of its name.

```java
public class BubbleSort{
    public static  void bubbleSort(int[] arr){
        int temp;
        for(int i=0;i< arr.length;i++){
            for(int j= arr.length-1;j>i;j--){
                if(arr[j]<arr[j-1])
                { temp=arr[j];
                    arr[j]=arr[j-1];
                    arr[j-1]=temp;}
            }
        }
    }
    public static void print(int[] terms){
        StringBuilder str=new StringBuilder();
        for(int i=0;i<terms.length;i++){
            str.append(terms[i]+", ");
        }
        System.out.println(str);
    }
    public static void main(String[] args) {
        System.out.println("Bubble Sort");
        int[] data = { -2, 45, 0, 11, -9 };
        bubbleSort(data);
        print(data);
    }
```

Figure 17. The BubbleSort class

Notice in the main method the input array include a set of numbers that are put in random. We predict that invoking Bubble Sort will set these numbers in increasing order to be: -9, -2, 0, 11, 45.

Figure 18 validates the prediction.

```
Bubble Sort
-9, -2, 0, 11, 45,
```

Figure 18. The output of the Bubble Sort

Selection Sort

The below figure shows the implementation of the corresponding algorithm.

```
public class SelectionSort {
    public static  void selectionSort(int[] arr){
        for(int i=0;i< arr.length;i++){
            int smallest=i;
            for(int j=i+1;j< arr.length;j++)
            {
                if(arr[j]<arr[smallest]){
                    smallest=j;
                }
            }
            int temp=arr[i];
            arr[i]=arr[smallest];
            arr[smallest]=temp;
        }
    }
    public static void print(int[] terms){
        StringBuilder str=new StringBuilder();
        for(int i=0;i<terms.length;i++){
            str.append(terms[i]+", ");
        }
        System.out.println(str);
    }
```

Figure 19. The SelectionSort Class

Figure 20 shows the input and the output. The order of numbers varied between the input and the output after the invocation of the selection sort. This indicates that, this algorithm is properly sorting the elements.

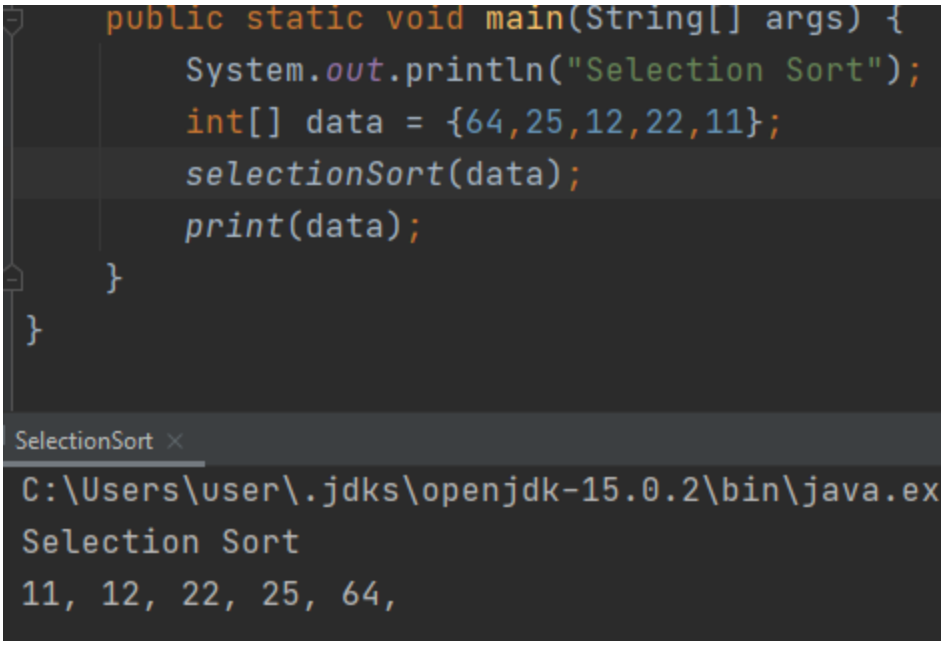


Figure 20. The output of the selection sort algorithm

Evaluator

The evaluation of the computation time of the sorting algorithms is implemented in the *Evaluation* class. Figure 21 shows the import of different required classes from Java packages which will be used later. Figure 22 shows the *main* method of the *Evaluation* class. First, a *Map* object named *finalTable* is declared. *finalTable Map* stores objects of class types *Integer* and *Map.* In other words, *finalTable* has its keys of type *Integer* and its values of type *Map.* The later *Map* stores *String* keys and *Double* values. *finalTable Map* is designed to *Integers* and another *Map* object because *finalTable* will store the number of elements *n* and its average computation time for every algorithm. The *Map* object parameter stores the sorting algorithm name as a key and its average computation time as a value. Then a for loop loops from 50 to 2000 incrementing 10 after each iteration for more accurate results. A *Map* object called *average_Execution_at_Each_n* is initialized. *average_Execution_at_Each_n* stores *Strings* and *Doubles* which are its keys and values respectively. The key represents the name of the sorting algorithm and the value is the algorithm's average computation time. This *Map* is declared inside the for loop. Thus, every *n* -number of elements- has a *Map* object saving its average execution time for every algorithm. The sorting algorithms are invoked respectively using a for loop. Each iteration number refers to an algorithm. Every algorithm runs 500 times to ensure a minimum

error in time computation by the compiler. Java compiles fast and this may affect the reading of the time taken to execute a method in addition to the hardware specifications that may play a role as well. That's why it's more accurate to run the method more than one time. Before running any algorithm, a random array of *int* and size *n* is generated. Random arrays are generated rather than user-defined arrays to assure the proper work of each algorithm. The timer starts counting and the algorithm is invoked via *choice* method. The timer class used is from the *System* package and is measured in nanosecond. Figure 23 shows the *choice* method. It takes two parameters: an *int* that identifies which algorithm to be called and an array of type *int* which is to be sorted. The for loop of key *k* will loop from 1 to 8. Each key will be passed as an argument for the method *choice*. So, all algorithms will be called respectively. The *choice* method invokes the respective algorithm that sorts the array. This method return a *String* which holds the name of the algorithm invoked to be passed later to the *average_Execution_at_Each_n Map* when storing the average time. Then the timer is stopped. *Execution* variable is calculated. It represents the time taken for execution. So, it is the difference between the end and the start time. Then after executing the algorithm 500 times, the average of these executed times is calculated and saved in the declared variable *averageExecution*. Thus, this variable holds the average execution time of every algorithm at a specific n. This average with the name of the sorting algorithm belonging to is added to the *average_Execution_at_Each_n Map* using the *put()* method. After invoking all algorithms at a specific *n* -number of elements in an array- and computing their average execution times, *n* and *average_Execution_at_Each_n Map* are passed to the *finalTable Map* to be saved. The *finalTable Map* invokes the *convertor* method. Figure 24 shows the implementation of this method. Its aim is to save the parameter's data in a JSON file. In this case, the *convertor* method reads a parameter of type *Map* whose key is an *int* and value is a *String-Double Map*. This parameter is read by the method and saved in a JSON file. The type of the *finalTable* is compatible with the method's parameter. The method creates a *JSONObject* named *json*. In this case, *finalTable* is passed to *json*'s constructor. Then, a file is created of name results.json. The *json*'s data is then written to the created file using *write* method and flushed to its destination. This block is put in a try catch to handle IOException just in case. As a result, the average execution time for every algorithm at every n is saved in a JSON file for feasible usages

later. Figure 25 shows a sample of the results in the JSON file named results. The data is saved in its proper JSON format. For example, the figure shows n=1540. So, the random array generated is of size 1540. The average execution time of Bubble Sort for n=1540 is equal to `2254479.0 ns` and so on.

```
import org.json.simple.JSONObject;

import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.Random;
import java.util.stream.IntStream;
```

Figure 21. The Evaluation Class imported classes

```
public static void main(String[] args) {
    Map<Integer, Map<String, Double>> finalTable = new HashMap<Integer, Map<String, Double>>();
    int n;
    for( n=50;n<=2000;n+=10){
        //generate array of random numbers
        int N=n;
        Map<String, Double> average_Execution_at_Each_n = new HashMap<~>();
        String sortAlgorithm = null;
        for(int k=1;k<8;k++){
            double execution=0;
            for(int i=0;i<500;i++){
                int[] randomIntsArray = IntStream.generate(() -> new Random().nextInt( bound: (int)N/2)).limit(n).toArray();
                double start = System.nanoTime();
                sortAlgorithm=choice(k,randomIntsArray.clone());
                double end = System.nanoTime();
                execution+=(end - start);
            }
            double averageExecution=execution/500;
            average_Execution_at_Each_n.put(sortAlgorithm,averageExecution);
        }
        finalTable.put(n,average_Execution_at_Each_n);
    }
    convertor(finalTable);
}
```

Figure 22. The main method of the Evaluation Class

```java
private static String choice(int x, int[] arr){
    if(x==1){ Merge.mergeSort(arr, l: 0, r: arr.length-1); return "Merge Sort"; }
    if(x==2){ InsertionSort.insertionSort(arr); return "Insertion Sort"; }
    if(x==3){ HeapSort.heapSort(arr); return "Heap Sort"; }
    if(x==4){
        QuickSort.quickSort(arr, p: 0,  r: arr.length-1);
        return "Quick Sort";
    }
    if(x==5){
        BubbleSort.bubbleSort(arr);
        return "Bubble Sort";
    }
    if (x==6){
        SelectionSort.selectionSort(arr);
        return "Selection Sort";
    }
    if (x==7){
        CountingSort.countSort(arr, arr.length);
        return "Count Sort";
    }
    return "No algorithm selected";
}
```

Figure 23. The choice method of the Evaluation Class

```java
private static void convertor(Map<Integer, Map<String, Double>> table){
    JSONObject json =  new JSONObject(table);

    //save the results in a JSON file
    try (FileWriter file = new FileWriter( fileName: "results.json")) {
        file.write(json.toJSONString());
        file.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

Figure 24. The convertor method of the Evaluation Class

```
{
  "1540": {
    "Bubble Sort": 2254479.0,
    "Merge Sort": 188522.6,
    "Insertion Sort": 473669.4,
    "Count Sort": 6931.2,
    "Heap Sort": 207735.8,
    "Quick Sort": 108724.8,
    "Selection Sort": 1401048.2
  },
  "1030": {
    "Bubble Sort": 1217689.0,
    "Merge Sort": 115873.8,
    "Insertion Sort": 242401.0,
    "Count Sort": 5890.2,
    "Heap Sort": 144579.4,
    "Quick Sort": 116194.6,
    "Selection Sort": 604390.2
  },
  "520": {
    "Bubble Sort": 284028.8,
    "Merge Sort": 42515.0,
    "Insertion Sort": 51848.6,
    "Count Sort": 2169.6,
    "Heap Sort": 36487.0,
    "Quick Sort": 25269.2,
    "Selection Sort": 123241.4
  },
```


Figure 25. Sample of the resulted JSON file

Plotter

In order to display the results and comprehend them, Python is used to plot the results. Figure 26 shows the first step of plotting. The json library is imported to access its functions. The JSON file *results* is opened using open function. And loaded to a declared list *listData.* The List object of Python is chosen to store the data in because it saves data as keys and values, thus more feasible to handle in upcoming tasks. The values of the list are saved in *Values* and same for the list's keys.

```python
import json
```

```python
# Opening JSON file
f = open(r'F:\Al Maaref Uni\Second Year 2021-2022\Spring 21-22\CSC 420 Algo\Project\SortingAlgorithms\results.json')
```

```python
listData=[]
listData = json.load(f)
Values=list(listData.values())
Keys=list(listData.keys())
```

Figure 26. Reading JSON file in Python

Figure 27 shows how the data in the list are classified accordingly. A set of arrays are created referencing each sorting algorithm in addition to sizes array. When displaying the results.json file, we notice that *n* is not saved in order starting from 50. That's why we set a for loop to iterate over the same values of n. This value which represents the number of elements is appended to the array sizes. Then, for each i -n number of elements-, the value of the key equal to each sorting algorithm -as named in the json file- is accessed from the list and appended to its corresponding array. For example, for i=50, the loop will retrieve the value of the key="Insertion Sort" from the list and append it to the array of insertion sort and so on for all algorithms at 50. After that, using the numpy library in Python, numpy arrays are created for each sorting algorithm

where the initial arrays are passed as parameters. Creating numpy arrays permits accessing various functions thus making tasks easier.

```
import numpy as np
insertion = []
merge = []
quick = []
heap = []
bubble = []
selection = []
counting = []
sizes = []

for i in range(50, 2000, 10):
    sizes.append(i)
    insertion.append(listData[str(i)]['Insertion Sort'])
    merge.append(listData[str(i)]['Merge Sort'])
    heap.append(listData[str(i)]['Heap Sort'])
    quick.append(listData[str(i)]['Quick Sort'])
    bubble.append(listData[str(i)]['Bubble Sort'])
    selection.append(listData[str(i)]['Selection Sort'])
    counting.append(listData[str(i)]['Count Sort'])

# create numpy arrays from the original arrays
npArrayInsertion = np.array(insertion)
npArrayMerge = np.array(merge)
npArrayHeap = np.array(heap)
npArrayQuick = np.array(quick)
npArrayBubble = np.array(bubble)
npArraySelection = np.array(selection)
npArrayCounting = np.array(counting)
```

Figure 27. Classifying the results using Python

After saving the results properly, plotting them is now accessible. Figure 28 shows the Python code for plotting the results. In order to plot, the matplotlib library must be imported. Figure 1 is initialized and the plots to be assembled on this figure are added. The plot function took as parameters: the data corresponding to the x-axis which is the array of sizes, the data corresponding to the y-axis which are the numpy arrays holding the average execution times and

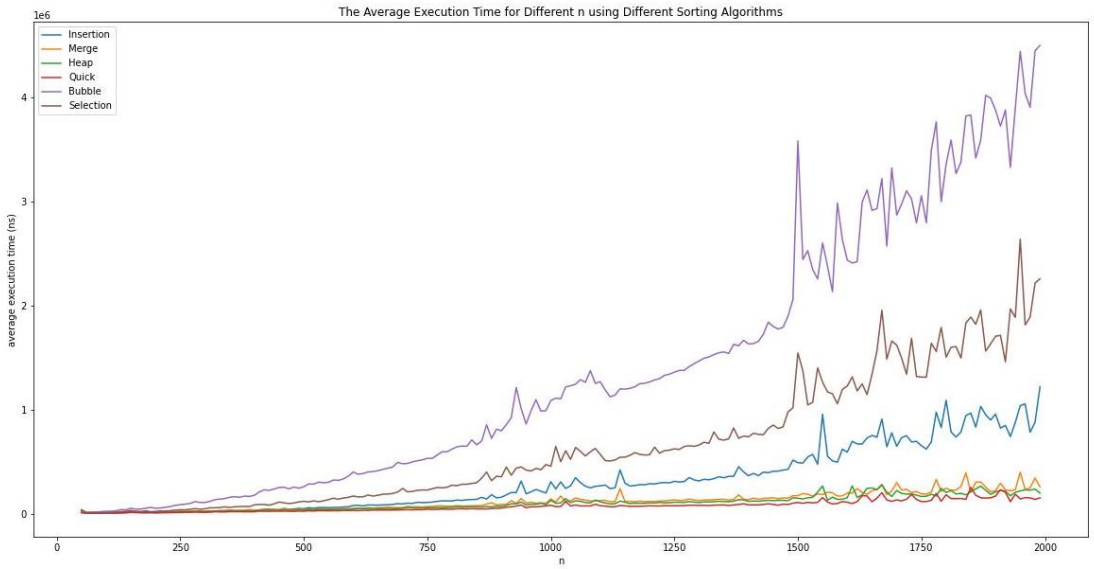
the label to point to. Another figure, Figure 2 is initialized to plot the counting sort average execution time separately.

```
import matplotlib.pylab as plt
fig=plt.figure(1)
plt.plot(sizes, npArrayInsertion, label='Insertion')
plt.plot(sizes, npArrayMerge, label='Merge')
plt.plot(sizes, npArrayHeap, label='Heap')
plt.plot(sizes, npArrayQuick, label='Quick')
plt.plot(sizes, npArrayBubble, label='Bubble')
plt.plot(sizes, npArraySelection, label='Selection')
plt.xlabel('n')
plt.ylabel("average execution time (ns)")
plt.title("The Average Execution Time for Different n using Different Sorting Algorithms")
plt.legend()
plt.savefig('sorting algorithms.jpg')
fig2 = plt.figure(2)
plt.plot(sizes, npArrayCounting, label="Counting")
plt.xlabel('n')
plt.ylabel("average execution time (ns)")
plt.title("The Average Execution Time for Different n using Counting Sort")
plt.legend()
plt.savefig('counting sort.jpg')
plt.show()
```

Figure 28. Plotting the results using Python

Figure 29 shows the resulted graph of plot 1. Figure 30 shows the resulted graph of plot 2.

Figure 29. The average execution time for different n using different sorting algorithms



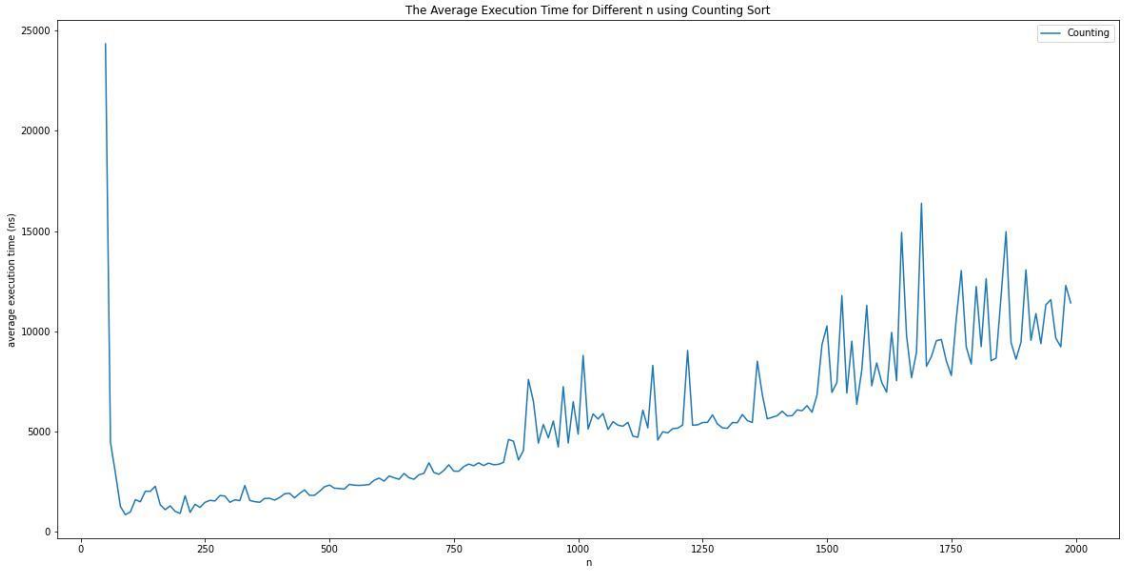


Figure 30. The average execution time for different n using counting sort

Some of the results are also displayed in the below table.

Table 2. Average computation time -ns- for different algorithms at specific n

Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Bubble Sort	Selection Sort	Counting Sort
500	48276.8	40883.6	33903.6	23741.6	259060.0	119320.2	2341.8
750	107964.6	68379.6	56884.6	43101.6	531556.0	226656.8	3037.4
1000	307139.0	141761.2	128138.6	80982.6	1086049.6	455184.0	4883.0
1250	311098.2	132113.0	110440.6	76460.4	1359236.6	622952.4	5460.6
1500	490005.2	172959.0	148386.8	107527.4	3579925.4	1543348.4	10275.8
1750	653219.6	190512.2	164866.6	116186.4	3053542.8	1312096.6	7803.4
2000	1005381.0	245491.2	306053.4	199282.6	3557665.2	2113979.4	9210.6

Results

The above graph shows that the Quick Sort algorithm is the fastest sorting algorithm whereas n increases, the execution time fluctuates on the same value near zero. It's increasing similarly to nlogn. On the other hand, Bubble Sort is considered a slow algorithm. As n increases, it takes more time to execute that reaches 4 times more than that taken by the quick sort for the same n. This increase in execution time is not a good significance because it indicates that the algorithm requires a lot of time as the number of elements increases. In terms of speed and efficiency it's not fast nor efficient. The merge and heap sort algorithms almost tie. They both increase in an nlogn manner. But yet Heap sort is slightly faster for large n. For example, at n=1750, the heap sort took 164866.6 ns to execute whereas the merge sort took 190512.2 ns. So, as n increases, heap sort is taking less time to execute unlike merge sort. Insertion Sort is increasing gradually. In other words, when n=1000 the average execution time reached approximately 307139 ns while for n=2000 the average execution time reached approximately 1005381 ns. This means that the execution time is more than doubling with n. So, for a very large n, this algorithm takes more time to execute. Yet it is faster than the selection sort at each n.

The graph of the counting sort shows that it is increasing in a linear manner. The table of results shows that for all n its execution time is less than that of other algorithms. This indicates that, it is faster than all other algorithms.

Conclusion

This paper aims to study and compare a set of sorting algorithms based on their execution time. The sorting algorithms selected are: insertion, merge, heap, quick, selection, bubble and counting. First, they are implemented using Java. Then, also using Java, random arrays are generated and invoked the algorithms respectively while measuring the time of execution. The average computation time at each number of elements for each sorting algorithm is stored in a JSON file. Finally, this file is read using Python. Its data is classified and plotted using Python matplotlib library. The resulted graphs and values are analyzed and the results match the theoretical ones. Therefore, the sorting algorithms are ordered decreasingly in terms of speed as

the following: Counting Sort, Quick Sort, Heap Sort, Merge Sort, Insertion Sort, Selection Sort and Bubble Sort for large n.

References

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (n.d.). *Introduction to Algorithms.* MIT Press.