**Efficiency of Sorting and Searching Algorithms**

Rafael Garcia, [rgarc821@my.epcc.edu](mailto:rgarc821@my.epcc.edu) – 915-227-0003

**Abstract**

This article focuses on the efficiency of using sorting algorithms by comparing six different of them, as well as the efficiency of two of the major search algorithms. The sorting algorithms used are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Shell Sort, and the search algorithms are Linear Search and Binary Search. These algorithms are described on their own section alongside their history and theoretical efficiency. Further into this article are the test cases, experiments, and results performed in order to test these algorithms, and based on the results of the experiments, it can be concluded that the analysis of these algorithms is important for modern day life.

Keywords: Searching, Sorting, Algorithm, Search, Sort, Binary, Quick, Merge, Linear, Shell, Bubble, Selection, Insertion, Java, Big-Oh, Big-Ω, Performance, Efficiency.

1. **Introduction**

Sorting and Searching Algorithms are very useful tools to manage data structures in everyday life, especially in the age of information. Today, it is required to have the data organized and stored so it can be accessed efficiently. Since the world is growing each day, and alongside it its information, humans require help to take on the overwhelming tasks that come with managing data. For this, there are computers, which can store and process large quantities of information for humans to use, but computers also have limitations. When it comes to having data organized, as well as accessing it, computers consume a lot of processing, and therefore time, especially in very large quantities of data. For this, there are various algorithms that look onto tackling a better performance from the machine.

Some algorithms to sort and search can consume great quantities of computer performance. Because of this, it is necessary to know what algorithms give the most efficient performance, both in time and computations. For this, this article will take on looking at the performance of six major sorting algorithms, and 2 major searching algorithms.

1. **Problem and solving approach**

The main problem, or task in this case, is the fact that the practical performance of each algorithm is not known, and for this, it is required to explore some of the major algorithms out there that help with the tasks of searching and sorting. In other words, the problem requires the computational time for some sorting and searching algorithms in order to determine which one is faster for each task.

In this case, the algorithms that will be used in order to solve the problem are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Shell Sort for sorting, and Linear Search and Binary Search for searching. These algorithms will be placed in the class “SortingAlgorithms” in order to improve the readability of the program.

The main method of the program will be under the “SortingTester” class, and it will be responsible for obtaining the data to sort, and for obtaining the time to sort with each algorithm. To determine the time, the method “System.nanoTime()” will be used to determine the starting time and end time in nanoseconds, and the double “totalTime” will register the difference of both, which will be the time elapsed per algorithm. Also, each case will use the 641,392 words from the “dictionary.txt” document, which are processed by the scanner, and copied into the String array “sort1”. Further on, this array of Strings will be copied into other 5 arrays named “sort2” - “sort5”. Afterwards, the main will call each sorting method from the class “SortingAlgorithms”, assigning as a parameter one of the string arrays for each. On each method call, the start, and and total time will be recorded and printed into the terminal.

After all the sorting algorithms are done, the main will use the search algorithms, testing on each the time, and assigning them with one of the sorted String arrays and a target String as parameters. When this is completed, the results will be printed out to the terminal for each target string, as well as with whether it has been found or not.

This seems to be the best approach, since some of the operations might take fractions of a second to perform, and with this there is more precision on the performance difference between algorithms. Also, with the amount of data provided, it gives a more concise result on the algorithms that might just take a couple of seconds to perform, and having it stored in copied arrays avoids the necessity of reading the “dictionary.txt” document more than one time, avoiding other type of performance issues that won’t be discussed in this document.

1. **Algorithm selection**

As stated before, there are six algorithms that will be used. Here is some information about each of them:

1. **Bubble Sort**

Bubble sort consists of pushing or “bubbling” the largest element to the end of the list by comparing the current element with the next element on the list and determining if both of them are in order. If they are, the program moves to the next comparison. If they’re not, the program will swap the elements so they can be in order. This process is done for each of the elements of the list [1]. This type of algorithm takes a collection of items as a parameter and outputs, at least in memory, the collection of items sorted in order. In this case, the input is an array of Strings, and the output is said array but in ASCII ascending order.

This algorithm hasn’t always been known as bubble sort. The term was coined by Iverson in 1962, but the algorithm has existed since the late 1950’s, when it was known as “sorting by exchange” or “exchange sorting” due to the fact that it constantly exchanges values in order to fully sort a data list [2].

In terms of theoretical runtime analysis, bubble sort has a complexity of O(n2), Ω(n), and Θ(n2) [3].

1. **Selection sort**

Selection sort uses a collection of items as an input as well, and outputs it in the necessary order. For this, the program assigns the current element as the smallest value. Then it iterates through the following elements finding the index of the smallest value. Next, the swap between the current element and the smallest value occurs and it move into the next element. This process is done iteratively until the last element is reached.

Unfortunately, not much history is available about this method, but Pramod and Sachin Kadam show in their article that the earliest use of selection sort was in 1962, but no authorship or circumstances are provided [4].

When it comes to its complexity, the selection sort algorithm presents O(n2), Ω(n2), and Θ(n2) [5].

1. **Insertion Sort**

Insertion sort also uses a collection of data and outputs it as its sorted version. This program needs to start at the second element. For each element, the program takes a look at the previous element, and if the current element is lower than the previous element, then it performs the swap. After the swap, the current element is checked again with the next previous element and performs the same operations. In order to move to the following element, one of two conditions must be met. The first, the current element must have reached the beginning of the list, or second, where the current element is higher than the previous element.

The first implementation of the Insertion Sort algorithm can be traced back to K Zuse in 1945, when he worked on the world’s first commercial computer, z4 [4].

The Insertion Sort algorithm has a performance complexity of O(n2), Ω(n), and Θ(n2) [6].

1. **Merge Sort**

Merge Sort as well uses a collection as data as input, the starting index and ending index, and as output it produces the ordered collection. Merge sort consists of the “mergeSort” method, which obtains the midpoint and creates left and right partitions by calling itself recursively and updating the starting and ending values with the midpoint respectively. The “mergeSort” method calls then the “merge” method, which is responsible for inserting the values of the partition into a merged list with the stack.

The idea of merge sort was born in 1938 with Jame W. Bryce, where he used a machine called collator to merge cards from two different stations [4]. Later on, this method was formally invented by John Von Neumann in 1945, but the more detailed version of it can be traced back to 1948 [7].

The complexity of this method is O(nlogn), Ω(nlogn), and Θ(nlogn) [8].

1. **Quick Sort**

Quick Sort, similar to Merge Sort, uses a collection of data and the starting and ending indexes, and for the output it creates the sorted collection of data. Quick sort is a recursive method which base case is reached when the starting index is higher than the ending index, which means that the array is sorted. Else, the “quickSort” method will obtain a pivot value by assigning it with the value returned by the “partition” method, and recursively call quicksort, first for the left partition with the original starting index and the pivot as the new ending index, then for the right partition will take as parameters the pivot plus 1 as the start index and the original ending index. The “partition” method is responsible for returning the pivot value, which is usually the midpoint of the partition. It also checks if the numbers before it and after it are lower than the pivot. If the lower and higher indexes meet, or the lower index is greater than the higher, the pivot will be assigned with the value of the higher index. Else, the values of the low and high indexes will be swapped, and the lower index will increase by one and the higher index will decrease by one. This comparison and swapping will be repeated until the lower index is equal to or greater than the higher index.

This method was created by Tony Hoare in 1959 but published until 1961 due to him thinking it was too simple. For this, he use an Algol60 compiler, which was the first to support recursion [9].

The runtime complexity that this algorithm has is O(n2), Ω(nlogn), and Θ(nlogn) [10].

1. **Shell Sort**

Shell sort will take a collection of data and will output said collection sorted. This is similar to insertion sort, except it uses a divide and conquer method, in which the program iterates through segments called gaps. These gaps indicate the middle index of the array, and it keeps iterating the division of the index in 2 after it reaches 0. While iterating each gap, an inner loop iterates from the current value up to the last index. This loop assigns a key value with the value of the array at the current index of the inner loop. Then a control index for a second inner loop is assigned with the current index of the first inner loop. This second nested loop will only be exited if the current index of the second inner loop is less than the gap, or if the value of the array at the position of the current index of the second inner loop minus the gap index value is lower than the key value. If the loop executes, then the value of the array at the current position of the second inner loop will be assigned with the value of the array at the position of the current index of the second inner loop minus the gap index, and the current index of the second inner loop is decreased by the gap index. When this second inner loop finishes executing or it if it doesn’t execute at all, in the first inner loop the value of the current position of the second inner loop will be assigned with the key value.

Donald Shell created this algorithm and published it in 1959, with which he received his Ph.D. in Mathematics from the University of Cincinnati [11].

The runtime complexity is O(n2), Ω(nlogn), and Θ(nlogn) [12].

Alongside our sorting algorithms, there are also the search algorithms which are the following:

1. **Linear search**

Linear search requires only two inputs, a collection of data and a target value, in this case an array of Strings and a String to search. This outputs, whether the string was or wasn’t found in the array. How this method works is by iterating through the entire array and returning true when the string was found. If the iteration is complete, then the string was not found, and therefore this method should return false.

Unfortunately there is no historical information about this algorithm, but this is perfectly understandable since it is a common human task to perform this kind of search in everyday life.

In terms of complexity analysis, the notation for this on is O(n), Ω(1), and Θ(n/2) [13]

1. **Binary Search**

Binary search takes the collection of data as well and a target value. Also for this case it will be the array of Strings and the target String. This as well, provides the Boolean value on whether the target is in the array or not. For this method you need the mid, low, and high indexes of the array, with the low and mid initialized to 0, and the high to the length of the array minus 1. Then, a while loop will execute until high is lesser than low. In the loop, the mid will be assigned with the value of the sum of the high and the low between 2. Then the program checks if the value of the array at the mid index is lower than the target. If yes, low index will be assigned with the value of mid plus 1. Else, the program will check if the element at the mid position is greater than the target. In this case, the high index will be assigned with the mid index plus 1. In the case that the value at the mid index of the array equals the target, then the method returns true, since it has been found at the midpoint. In case the program gets out of the loop, then the method will return false, since the element wasn’t found.

The first idea of this algorithm came from John Mauchly in a discussion of non-numerical programming methods in 1946 [14].

The complexity that this algorithm possesses is O(logn), Ω(1), and Θ(logn) [15].

1. **Performance of our Approach**

The performance for Bubble Sort in the worst case scenario is O(n2) in number of comparisons and O(n2) swaps. In the best case scenario we have Ω(n) comparisons and Ω(1) swaps. The average performance is Θ(n2) comparisons and Θ(n2) swaps [16].

For Selection sort, we have O(n2) comparisons and O(n) swaps for the worst case scenario. In the best case scenario it does Ω(n2) comparisons and Ω(1) swaps. Its average performance tends to be Θ(n2) comparisons and Θ(n) swaps [5].

For Insertion sort the worst case scenario is O(n2) for both swaps and comparisons. Best case scenario, it will perform Ω(n) comparisons and Ω(1) swaps. When it comes to the average performance, it is Θ(n2) comparisons and swaps [6].

On Merge Sort, the worst case performance in O(nlogn). Its best case performance is Ω(nlogn) the typical, Ω(n) a natural variant. Its average performance can be defined by Θ(nlogn) [8].

On Quick Sort, the worst case performance is O(n2), while its best case performance is Ω(nlogn), but it has an average performance of Θ(nlogn) [10].

The performance on the worst case scenario for Shell Sort is O(n2) or O(nlog2n). In the best case scenario with most gap sequences the performance is Ω(nlogn) and in the worst case gap sequence Ω(nlog2n). The average performance will depend on the gap sequence.

On linear search, the worst case performance is O(n), while the best case performance is Ω(1). Its average performance is Θ(n/2) [13].

For binary search, its worst case complexity is O(log n), its best case complexity is Ω(1)), and its average performance is Θ(log n) [15].

1. **Testing Strategy**

The experiments conducted to obtain this will include one case per sorting algorithm and four cases per searching algorithm. The cases for the sorting algorithms will involve:

1. And if statement that takes the user input to execute the methods one by one in case of required interruption of the running.
2. The double startTime = System.nanoTime();
3. The method call with a copy of the array of Strings scanned from “dictionary.txt” (In the case of Merge and Quicksort, the indexes of the start and end).
4. The double endTime = System.nanoTime();
5. The double totaltTime = endTime-statTime;
6. A print statement that shows the total time in nanoseconds.

The cases for the searching algorithms are very similar, with the exception of having the method call within the print statement, since they’re boolean methods. In the parameters, one of the sorted lists was used as the array of strings, and one of the following four words as a test case:

* yellow-earth
* AMARyYO
* amarillo
* yellow

1. **Experiments and Results**

Each of the cases was run with little to no interaction with the machine to dedicate maximum performance to the algorithms. The results are shown in Table 1 for sorting.

Table 1. Sorting Results

|  |  |
| --- | --- |
| Algorithm | Time of execution (nanoseconds) |
| Bubble Sort | 2.2084264033e13 |
| Selection Sort | 3.9200645721e12 |
| Insertion Sort | 3.7417020801e12 |
| Merge Sort | 3.520508e8 |
| Quicksort | 2.943407e8 |
| Shell Sort | 1.0273453e9 |

Results for the search algorithms are displayed in table 2.

Table 2. Search results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Linear Search | | Binary Search | |
| Word | Time | Result | Time | Result |
| yellow-earth | 2.10114e7 | True | 4.243e5 | True |
| AMARyYO | 1.4469e7 | False | 2.514e5 | False |
| amarillo | 1.5346e7 | True | 2.063e5 | True |
| yellow | 1.01942e7 | True | 2.109e5 | true |

1. **Conclusion**

Based on the results of the experiments, and the research conducted, it is easy to determine that quick sort is one of the most efficient algorithms to sort data, followed by merge sort, shell sort, insertion sort, selection sort, and with the poorest performance, bubble sort. In terms of searching algorithms, it can be said that it is more efficient to use binary search instead of linear search, since it uses less computational time to search.

With these experiments also one can tell how the data performs in a more real environment, since most of the modern software requires a lot of data in order to function. And people need that software to be fast enough to keep up with the rapid changes in the world of the 21st century, since it has become commonplace to access information in a quick and efficient way.

1. **Implementation Annexes**

public class SortingAlgorithms{

//bubble sort

public static void bubbleSort(String [] a) {

String temp = " ";

for(int i=0;i<a.length;i++){

for(int j=0;j<a.length-1;j++) {

if(a[j].compareTo(a[j+1])>0){

temp=a[j];

a[j]=a[j+1];

a[j+1]=temp;

}

}

System.out.print(i+" ");

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Title: Selection Sort

\* Author: Servin, C.

\* Date: 10/11/2021

\* Code version: Uknown

\* Availability: https://learn.zybooks.com/zybook/EPCCCOSC1437Fall2021/chapter/7/section/2

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static void selectionSort(String [] a) {

int indexSmallest;

String temp;

for (int i = 0; i < a.length - 1; ++i) {

indexSmallest = i;

for (int j = i + 1; j < a.length; ++j) {

if (a[j].compareTo(a[indexSmallest])<0) {

indexSmallest = j;

}

}

temp = a[i];

a[i] = a[indexSmallest];

a[indexSmallest] = temp;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Title: Insertion Sort

\* Author: Servin, C.

\* Date: 10/11/2021

\* Code version: Uknown

\* Availability: https://learn.zybooks.com/zybook/EPCCCOSC1437Fall2021/chapter/7/section/3

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static void insertionSort(String [] a) {

String temp;

for (int i = 1; i < a.length; ++i) {

int j = i;

while (j > 0 && a[j].compareTo(a[j-1])<0) {

temp = a[j];

a[j] = a[j - 1];

a[j - 1] = temp;

--j;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Title: Merge Sort

\* Author: Servin, C.

\* Date: 10/11/2021

\* Code version: Uknown

\* Availability: https://learn.zybooks.com/zybook/EPCCCOSC1437Fall2021/chapter/7/section/5

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static void merge(String [] a, int i, int j, int k) {

int mergedSize = k - i + 1;

String [] mergedNumbers = new String[mergedSize];

int mergePos = 0;

int leftPos = i;

int rightPos = j + 1;

while (leftPos <= j && rightPos <= k) {

if (a[leftPos].compareTo(a[rightPos])<0) {

mergedNumbers[mergePos] = a[leftPos];

++leftPos;

}

else {

mergedNumbers[mergePos] = a[rightPos];

++rightPos;

}

++mergePos;

}

while (leftPos <= j) {

mergedNumbers[mergePos] = a[leftPos];

++leftPos;

++mergePos;

}

while (rightPos <= k) {

mergedNumbers[mergePos] = a[rightPos];

++rightPos;

++mergePos;

}

for (mergePos = 0; mergePos < mergedSize; ++mergePos) {

a[i + mergePos] = mergedNumbers[mergePos];

}

}

public static void mergeSort(String [] a, int i, int k) {

int j;

if (i < k) {

j = (i + k) / 2;

mergeSort(a, i, j);

mergeSort(a, j + 1, k);

merge(a, i, j, k);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Title: Quick Sort

\* Author: Servin, C.

\* Date: 10/11/2021

\* Code version: Uknown

\* Availability: https://learn.zybooks.com/zybook/EPCCCOSC1437Fall2021/chapter/7/section/4

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static int partition(String [] a, int i, int k) {

int l=i;

int h=k;

int midpoint=i + (k - i) / 2;

String pivot=a[midpoint];

String temp;

boolean done=false;

while (!done) {

while (a[l].compareTo(pivot) < 0) {

++l;

}

while (pivot.compareTo(a[h]) < 0) {

--h;

}

if (l >= h) {

done = true;

}

else {

temp = a[l];

a[l] = a[h];

a[h] = temp;

++l;

--h;

}

}

return h;

}

public static void quicksort(String [] a, int i, int k) {

int j;

if (i >= k) {

return;

}

j = partition(a, i, k);

quicksort(a, i, j);

quicksort(a, j + 1, k);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Title: Shell Sort

\* Author: Baeldung

\* Date: 10/11/2021

\* Code version: Uknown

\* Availability: https://www.baeldung.com/java-shell-sort

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static void shellSort(String [] a) {

int n = a.length;

for (int gap = n / 2; gap > 0; gap /= 2) {

for (int i = gap; i < n; i++) {

String key = a[i];

int j = i;

while (j >= gap && a[j - gap].compareTo(key) > 0) {

a[j] = a[j - gap];

j -= gap;

}

a[j] = key;

}

}

}

public static boolean linearSearch(String [] a, String target) {

for(int i=0;i<a.length;++i) {

if(a[i].equals(target)){

return true;

}

}

return false;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Title: Binary Search

\* Author: Servin, C.

\* Date: 10/11/2021

\* Code version: Uknown

\* Availability: https://learn.zybooks.com/zybook/EPCCCOSC1437Fall2021/chapter/8/section/2

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public static boolean binarySearch(String [] a, String target) {

int mid = 0;

int low =0;

int high = a.length-1;

while (high>=low){

mid = (high+low)/2;

if(a[mid].compareTo(target) < 0)

low = mid +1;

else if(a[mid].compareTo(target) > 0)

high = mid -1;

else

return true;

}

return false;

}

}

import java.util.\*;

import java.io.\*;

public class SortingTester{

public static void main(String[]args) throws IOException{

//variable initialization

Scanner scnr = new Scanner(new File("dictionary.txt"));

double startTime = 0;

double endTime = 0;

double totalTime = 0;

int totElements = 0;

while(scnr.hasNextLine()) {

scnr.nextLine();

++totElements;

}

// copy from file to array and multiple array copies

String [] sort1 = new String [totElements];

Scanner scnr2 = new Scanner(new File("dictionary.txt"));

for(int i=0;i<sort1.length;++i) {

sort1[i]=scnr2.nextLine();

}

String [] sort2 = new String [totElements];

for(int i=0;i<sort2.length;++i) {

sort2[i]=sort1[i];

}

String [] sort3 = new String [totElements];

for(int i=0;i<sort3.length;++i) {

sort3[i]=sort1[i];

}

String [] sort4 = new String [totElements];

for(int i=0;i<sort4.length;++i) {

sort4[i]=sort1[i];

}

String [] sort5 = new String [totElements];

for(int i=0;i<sort5.length;++i) {

sort5[i]=sort1[i];

}

String [] sort6 = new String [totElements];

for(int i=0;i<sort6.length;++i) {

sort6[i]=sort1[i];

}

// method implementation

Scanner input = new Scanner(System.in);

System.out.print("Do you want to execute bubble sort? (y/n) ");

char userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

SortingAlgorithms.bubbleSort(sort1);

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Bubble Sort = " + totalTime + "nanoseconds ");

System.out.println((totalTime/1000000000)+" seconds");

System.out.println();

}

System.out.print("Do you want to execute selection sort? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

SortingAlgorithms.selectionSort(sort2);

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Selection Sort = " + totalTime + "nanoseconds ");

System.out.println((totalTime/1000000000)+" seconds");

System.out.println();

}

System.out.print("Do you want to execute insertion sort? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

SortingAlgorithms.insertionSort(sort3);

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Insertion Sort = " + totalTime + " nanoseconds");

System.out.println((totalTime/1000000000)+" seconds");

System.out.println();

}

System.out.print("Do you want to execute merge sort? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

SortingAlgorithms.mergeSort(sort4, 0, sort4.length-1);

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Merge Sort = " + totalTime + " nanoseconds");

System.out.println((totalTime/1000000000)+" seconds");

System.out.println();

}

System.out.print("Do you want to execute quicksort? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

SortingAlgorithms.quicksort(sort5, 0, sort5.length-1);

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("QuickSort = " + totalTime + " nanoseconds");

System.out.println((totalTime/1000000000)+" seconds");

System.out.println();

}

System.out.print("Do you want to execute shell sort? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

SortingAlgorithms.shellSort(sort6);

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Shell Sort = " + totalTime + " nanoseconds");

System.out.println((totalTime/1000000000)+" seconds");

System.out.println();

}

System.out.print("Do you want to execute Linear search? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

System.out.println("Result for yellow-earth: " + SortingAlgorithms.linearSearch(sort4, "yellow-earth"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Linear search = " + totalTime + " nanoseconds");

startTime = System.nanoTime();

System.out.println("Result for AMARyYO: " + SortingAlgorithms.linearSearch(sort4, "AMARyYO"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Linear search = " + totalTime + " nanoseconds");

startTime = System.nanoTime();

System.out.println("Result for amarillo: "+SortingAlgorithms.linearSearch(sort4, "amarillo"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Linear search = " + totalTime + " nanoseconds");

startTime = System.nanoTime();

System.out.println("Result for yellow: "+SortingAlgorithms.linearSearch(sort4, "yellow"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Linear search = " + totalTime + " nanoseconds");

System.out.println();

}

System.out.print("Do you want to execute Binary search? (y/n) ");

userInput = input.next().charAt(0);

if(userInput == 'y') {

startTime = System.nanoTime();

System.out.println("Result for yellow-earth: " + SortingAlgorithms.binarySearch(sort4, " yellow-earth"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Binary search = " + totalTime + " nanoseconds");

startTime = System.nanoTime();

System.out.println("Result for AMARyYO: " + SortingAlgorithms.binarySearch(sort4, "AMARyYO"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Binary search = " + totalTime + " nanoseconds");

startTime = System.nanoTime();

System.out.println("Result for amarillo: "+SortingAlgorithms.binarySearch(sort4, "amarillo"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Binary search = " + totalTime + " nanoseconds");

startTime = System.nanoTime();

System.out.println("Result for yellow: "+SortingAlgorithms.binarySearch(sort4, "yellow"));

endTime = System.nanoTime();

totalTime = endTime-startTime;

System.out.println("Binary search = " + totalTime + " nanoseconds");

System.out.println();

}

}

}

# References

|  |  |
| --- | --- |
| [1] | B. Miller and D. Ranum, "6.7 Bubble Sort," Runestonne Academy, 01 October 2013. [Online]. Available: https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html. [Accessed 14 October 2021]. |
| [2] | O. Astrachan, "Bubble Sort: An Archeological Algorithmic Analysis," Duke University, [Online]. Available: https://users.cs.duke.edu/~ola/bubble/bubble.html. [Accessed 14 October 2021]. |
| [3] | GeeksforGeeks, "Bubble Sort," GeeksforGeeks, 24 September 2021. [Online]. Available: https://www.geeksforgeeks.org/bubble-sort/. [Accessed 14 October 2021]. |
| [4] | P. Kadam and S. Kadam, "Root to fruit (2): An Evolutionary Approach for Sorting algorithms," Oriental Journal Of Computer Science An Technology, 26 December 2014. [Online]. Available: http://www.computerscijournal.org/vol7no3/root-to-fruit-2-an-evolutionary-approach-for-sorting-algorithms/. [Accessed 14 October 2021]. |
| [5] | StarryGrandma, "Selection Sort," Wikipedia, 14 July 2021. [Online]. Available: https://en.wikipedia.org/wiki/Selection\_sort. [Accessed 14 October 2021]. |
| [6] | Ancarda, "Insertion Sort," Wikipedia, 3 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Insertion\_sort. [Accessed 14 October 2021]. |
| [7] | M. Tusilram, "Sorting Algorithm," Indiana State Univeristy, 3 December 2014. [Online]. Available: http://cs.indstate.edu/~mtulsiram/index.html. [Accessed 14 Octorber 2021]. |
| [8] | Bubba73, " Merge Sort," Wikipedia, 3 September 2021. [Online]. Available: https://en.wikipedia.org/wiki/Merge\_sort. [Accessed 14 October 2021]. |
| [9] | Nic, "Quicksort: the history and implementations," dev.to, 25 October 2019. [Online]. Available: https://dev.to/snj/quicksort-the-history-and-implementations-28o2. [Accessed 14 October 2021]. |
| [10] | Glibg10b, "Quicksort," Wikipedia, 1 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Quicksort. [Accessed 14 October 2021]. |
| [11] | L. J.A.N., "Donald Lewis Shell," IEEE Computer Society, 2021. [Online]. Available: https://history.computer.org/pioneers/shell.html. [Accessed 14 October 2021]. |
| [12] | Programiz, "Shell Sort Algorithm," Programiz, 2021. [Online]. Available: https://www.programiz.com/dsa/shell-sort. [Accessed 14 October 2021]. |
| [13] | Felida97, "Linear Search," Wikipedia, 7 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Linear\_search. [Accessed 14 October 2021]. |
| [14] | M. Lucas, "History of Science and Mathematics," Stack Exchange, 16 April 2015. [Online]. Available: https://hsm.stackexchange.com/questions/2200/what-is-the-first-historical-reference-to-the-binary-search-algorithm. [Accessed 14 October 2021]. |
| [15] | SunDawn, "Binary Search Algorithm," Wikipedia, 11 October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Binary\_search\_algorithm. [Accessed 14 October 2021]. |