Ollscoil Teicneolaíochta an Atlantaigh
Atlantic Technological University

H. Dip. in Science
Software Development

Sorting Algorithms and Benchmarking

Computational Thinking with Algorithms 2023

Jonathan Griffey – G00299043

Contents

Executive Summary

The purpose of this project was to develop a Java application to assess the efficiency of five distinct sorting algorithms, including Bubble Sort, Selection Sort, Insertion Sort, Merge Sort and Counting Sort. The aim of the report is to discuss the chosen algorithms for the application and the performance of it, while also discussing the topics of Algorithms and sorting.

After analysing the performance of the algorithms in Java, it was determined that Bubble Sort had the least favourable time complexity when tested with arrays of different input sizes. On the other hand, Counting Sort consistently exhibited superior performance compared to the other algorithms in the group, regardless of the input size.

Introduction

"Prior to computers, sorting data was a major bottleneck that took huge amounts of time to perform manually" (Wladston Ferreira Filho, 2018, pg86). In computer science, sorting is a fundamental operation used in many applications. "Sorting algorithms are used to rearrange a given array or list according to a comparison operator, which is used to decide the new order of elements in the data" (GeeksforGeeks, 2017).

The purpose of sorting algorithms is to organize data efficiently. Basically, they sort unsorted data into sorted lists. Different sorting algorithms have different efficiencies in terms of speed and memory usage, which means we must be careful to choose the most appropriate algorithm for our specific needs.

You can see a very basic example result of sorting below using my student number (G00299043) as the array to be sorted.

UNSORTED ARRAY

2	9	9	0	4	3

SORTED ASCENDING

0	2	3	4	9	9

SORTED DESCENDING

9	9	4	3	2	0

Efficiency and Performance

Complexity (Time and Space):

When working with sorting, it is vital to be aware of the time and space complexity of algorithms. In simple terms, the time complexity of an algorithm is the amount of time it takes to execute the program and the space complexity is the amount of memory it uses.

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time taken. It is because the total time took also depends on some external factors like the compiler used, processor's speed, etc. (GeeksforGeeks, 2021). It is commonly expressed using big O notation, which gives an upper bound on the growth rate of the number of operations. For example, an algorithm with a time complexity of O(n) means that the number of operations grows linearly with the input size.

Other time complexities include:

- **O(1)** - constant time complexity, where the number of operations does not depend on the input size.
- **O(log n)** - logarithmic time complexity, where the number of operations grows slowly as the input size increases.
- **O(n log n)** - quasi-linear time complexity, where the number of operations grows faster than linear but slower than polynomial.
- **O(n^2)** - quadratic time complexity, where the number of operations grows exponentially with the input size.
- **O(2^n)** - exponential time complexity, where the number of operations grows exponentially with the input size.

Space Complexity: Space Complexity is the total memory space required by the program for its execution. (GeeksforGeeks, 2021). It is also commonly expressed using big O notation, which gives an upper bound on the growth rate of the memory used. For example, an algorithm with a space complexity of O(n) means that the amount of memory used grows linearly with the input size.

Other space complexities include:

- **O(1)** - constant space complexity, where the amount of memory used does not depend on the input size.
- **O(log n)** - logarithmic space complexity, where the amount of memory used grows slowly as the input size increases.
- **O(n log n)** - quasi-linear space complexity, where the amount of memory used grows faster than linear but slower than polynomial.
- **O(n^2)** - quadratic space complexity, where the amount of memory used grows exponentially with the input size.
- **O(2^n)** - exponential space complexity, where the amount of memory used grows exponentially with the input size.

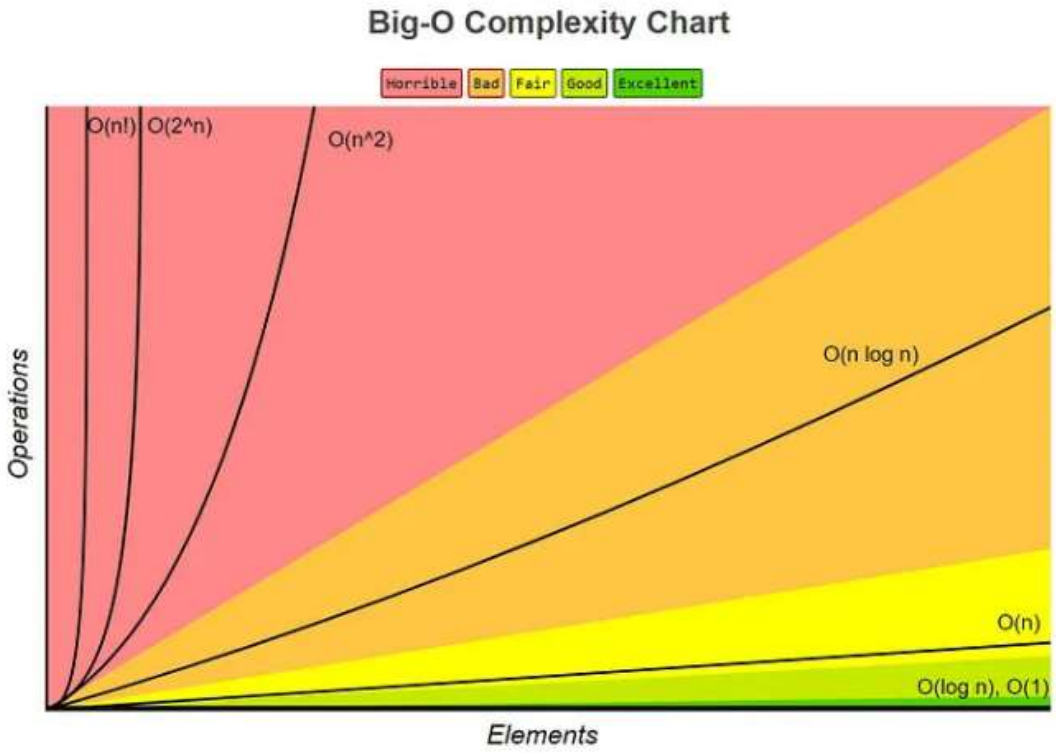


Figure 1. Big-O Complexity Chart (Prado, 2020)

Sorting algorithms have different performance characteristics. Some algorithms are better suited for small datasets, while others are better suited for large datasets. The performance of an algorithm can be affected by the distribution and size of the data, as well as the hardware it is running on. It is important to choose the correct algorithm for the specific problem.

When choosing the right algorithm, there are other factors outside of Time and Space complexity which need to be researched. These include Stability, Recursion and Adaptability.

Stability

The stability of a sorting algorithm is concerned with how the algorithm treats equal elements. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. (Srivastava, 2020)

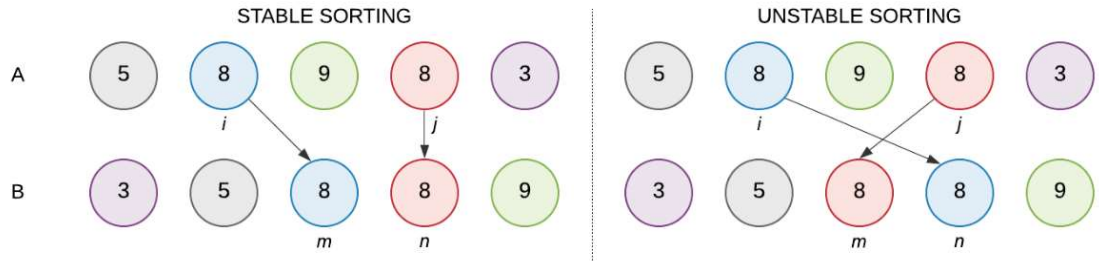


Figure 2. Stable vs unstable sorting

Recursion

A recursive algorithm involves calling itself with reduced input values and then performing basic operations on the returned value to obtain the result for the current input. Typically, a problem that can be solved by applying solutions to smaller instances of the same problem, and where the smaller instances become easier to solve, can be tackled using a recursive algorithm. (Simplilearn.com, n.d.)

Adaptability

A sorting algorithm is adaptive if its run time, for inputs of the same size n, varies smoothly from O (n) to O(n log n) as the disorder of the input varies. It is well accepted that files that are already sorted are often sorted again and that nearly sorted files often occur. (Vladimir Estivill-Castro and Wood, 1996)

In-Place Sorting

An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. (GeeksforGeeks, 2018). An example of this is Quicksort, which was the first In Place sorting algorithm to demonstrate O(n log n) performance (Heineman, Pollice and Selkow, n.d. pg 346).

Comparator Functions

Comparison operators (+, >, <, =, etc) are used in algorithms to determine the relative order of two elements. In sorting algorithms, comparator functions are used to compare elements in order to determine their relative positions in the final sorted output. For example, in the popular Quicksort algorithm, a comparator function is used to partition the array into two subarrays, one with elements greater than or equal to a pivot element, and the other with elements less than the pivot.

Comparison-Based and Non-Comparison-Based Sorts

In comparison based sorting, elements of an array are compared with each other to find the sorted array. The sorting algorithms that sort the data without comparing the elements are called non-comparison sorting algorithms. The examples for non-comparison sorting algorithms are counting sort, bucket sort, radix sort, and others. (OpenGenus IQ: Computing Expertise & Legacy, n.d.)

Sorting Algorithms

This project report discusses 5 sorting algorithms that are used for the benchmarking application. They include:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Comparison based Quick Sort
- Non Comparison based Counting Sort

These algorithms are discussed below with example graphs.

Bubble Sort

Overview

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high (GeeksforGeeks, 2014). Despite its ease of use, Bubble sort is mainly employed as an educational aid due to its inadequate performance in real-world scenarios. It is unsuitable for handling large datasets, with an average and worst-case complexity of O(n2), where n represents the number of items.

How it works

Bubble Sort works by comparing each element with the element on its right, excluding the last element. If the elements in the pair are out of order, it swaps them. This results in the largest element being put at the very end and into its correct place, hence it will not be moved again.

An example of this can be seen in figure 3 below.

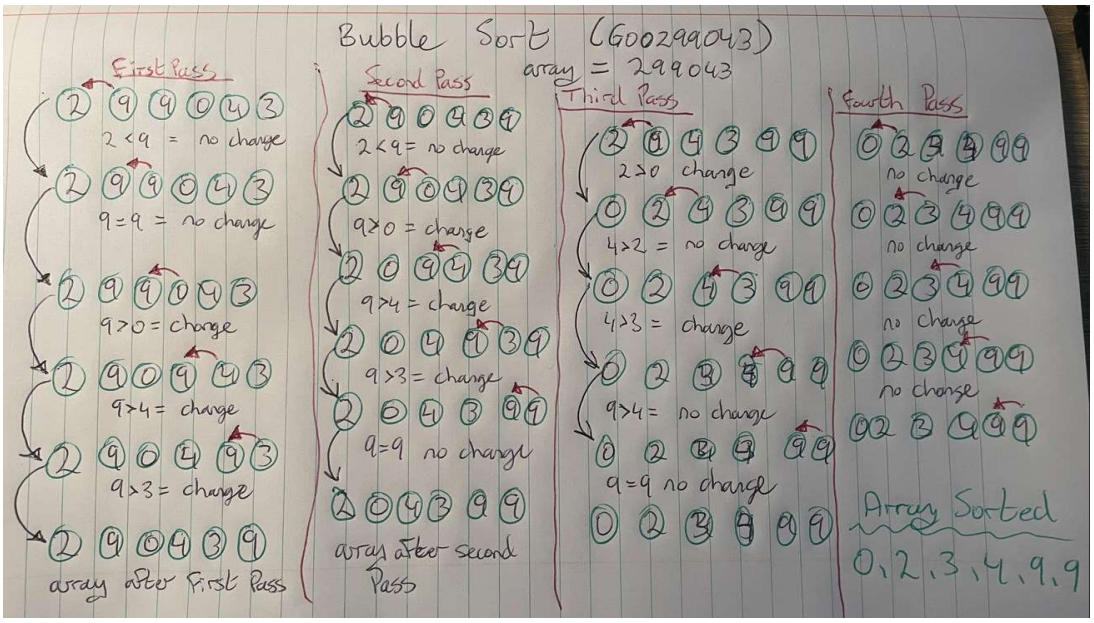


Figure 3. Bubble Sort example with Student code

Code

(www.javatpoint.com, n.d.)

```
public class BubbleSortExample {
   static void bubbleSort(int[] arr) {
      int n = arr.length;
      int temp = 0;
       for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){
                  if(arr[j-1] > arr[j]){
                        //swap elements
                        temp = arr[j-1];
                        arr[j-1] = arr[j];
                        arr[j] = temp;
                  }
            }
       }
   }
```

Pros & Cons

Bubble Sort is a simple enough algorithm to understand as it moves through each index of the array on each pass, which can be suitable for data which is somewhat sorted. However, with an average and worst-case complexity of O(n2), it is not particle for most issues that occur and is very slow.

Selection Sort

Overview

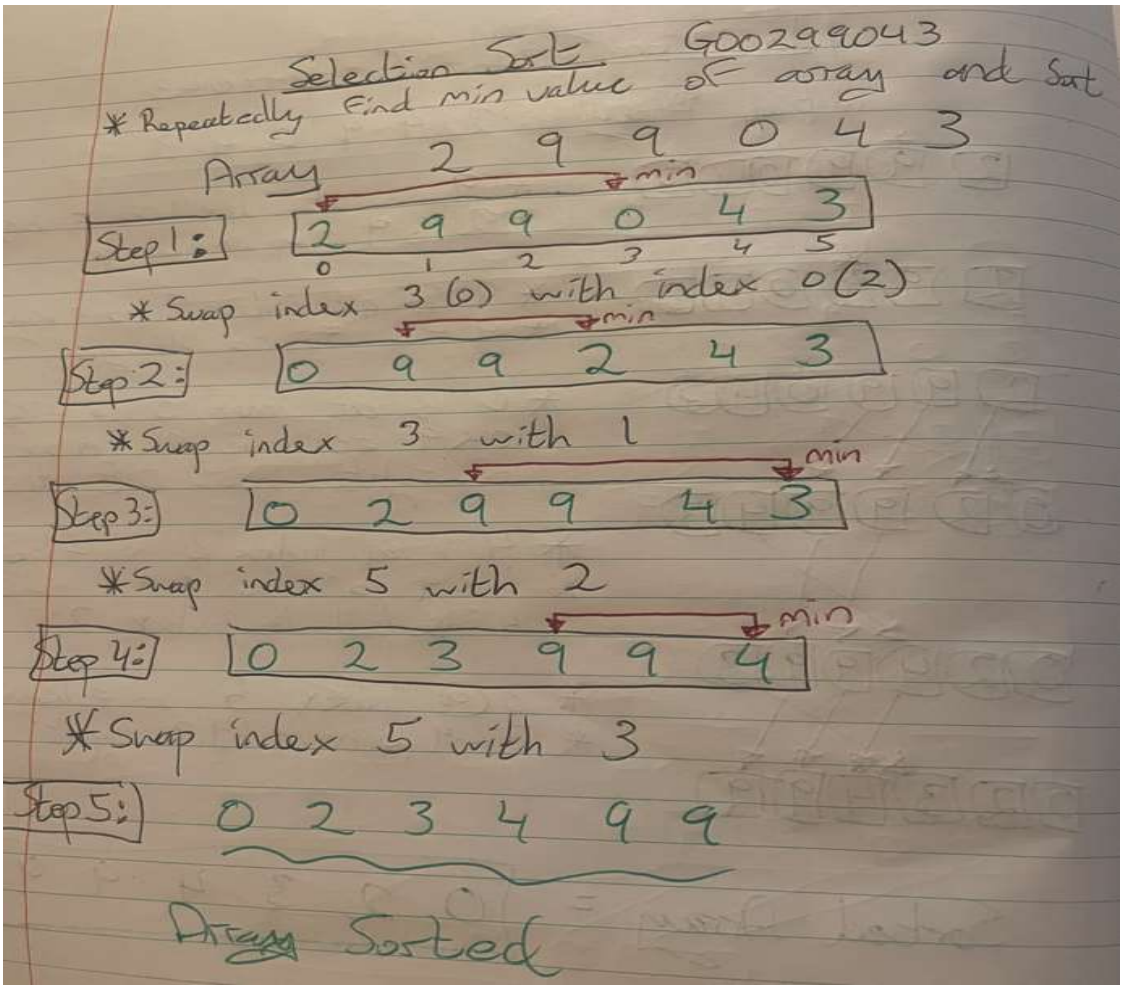
Selection Sort is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end (tutorialspoint.com, 2019). It looks at each index in the array for a minimum value and moves that values to index 0. It repeats this process until the array is sorted.

How it works

Here are the steps for selection sort:

Find the minimum element in the unsorted array. Swap the minimum element with the first element in the unsorted part of array. Repeat the remaining unsorted part of array.

Here's an example of how selection sort works on student number array:



Code

("Selection Sort - Javatpoint")

```
public class SelectionSortExample {

    public static void selectionSort(int[] arr){
        for (int i = 0; i < arr.length - 1; i++)
        {
            int index = i;
            for (int j = i + 1; j < arr.length; j++){
                if (arr[j] < arr[index]){
                    index = j;//searching for lowest index
                }
            }
            int smallerNumber = arr[index];
            arr[index] = arr[i];
```

```
                arr[i] = smallerNumber;
            }
        }

    }
```

Pros and Cons

Pros of selection sort:

- It's simple and easy to understand:
- In-place sorting makes it space-efficient.
- Good for small lists

Cons of selection sort:

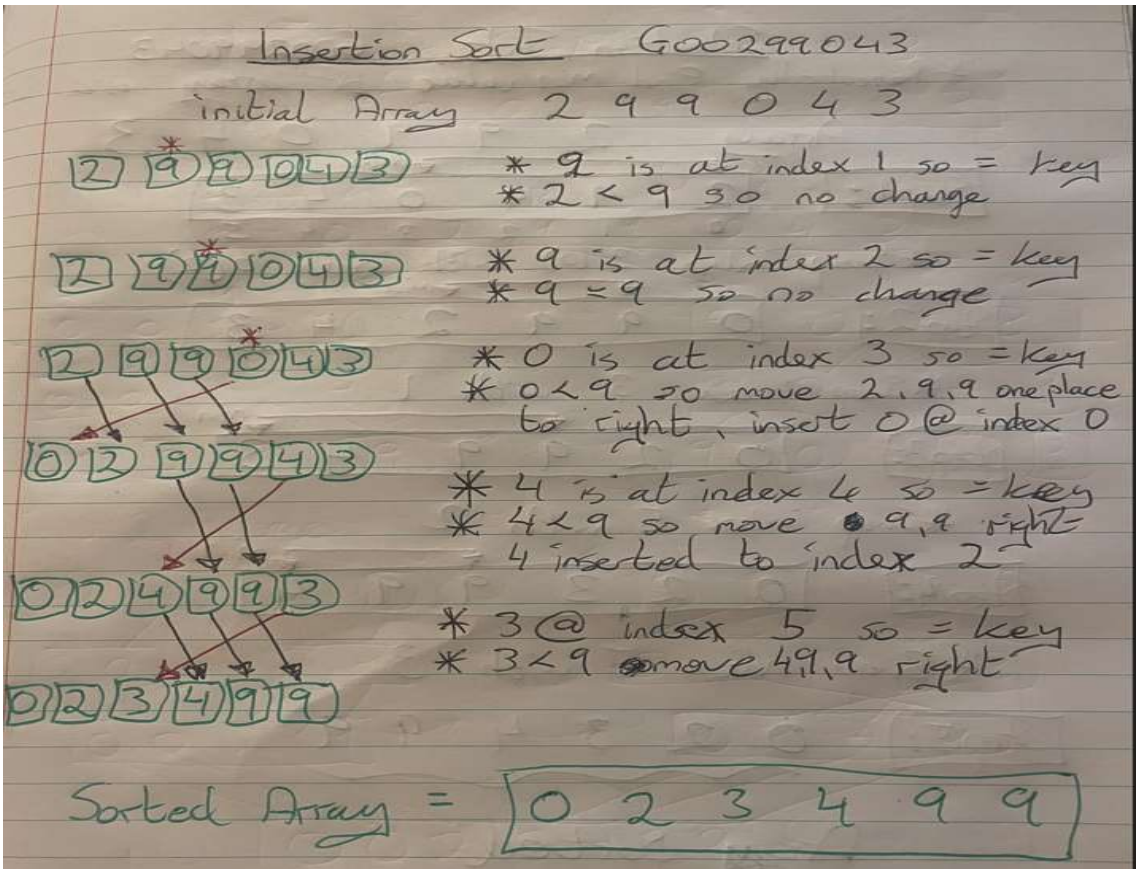
- Selection sort has a time complexity of O(n^2), which makes it inefficient for sorting large lists.
- Selection sort is not a stable sorting algorithm
- Requires many swaps

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part (GeeksforGeeks, 2013). It is an in-place comparison-based sorting algorithm which has simple implementation and is efficient for small data values. It is adaptive which makes it appropriate for data sets which are already partially sorted.

How it works

It takes the first two elements of an array (index 0 and 1) and sorts them in ascending order. It then looks at index 1 and 2 and sorts those the same way by moving index 2 to index 1 if the value is lower. If the value in index 1 is lower then the value at index 0, it will swap these as well. It continues this process until the array is sorted. See example below.



Code

```
public class InsertionSortExample {
   public static void insertionSort(int array[]) {
      int n = array.length;
      for (int j = 1; j < n; j++) {
         int key = array[j];
         int i = j-1;
         while ( (i > -1) && ( array [i] > key ) ) {
            array [i+1] = array [i];
            i--;
         }
         array[i+1] = key;
      }
   }


}
```

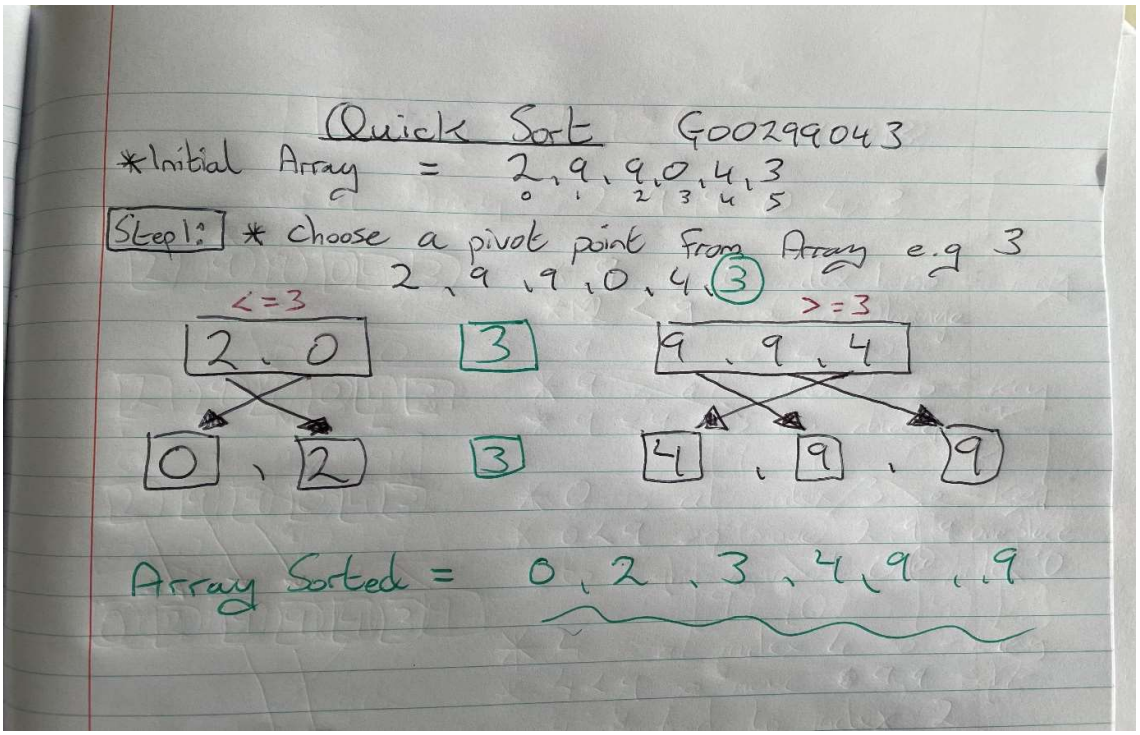
Pros and Cons

Insertion Sort is a stable in place sorting algorithm which is easy to understand and efficient for small lists. However, it has a time complexity of O(n^2), and requires swapping elements, which makes it Inefficient for large lists.

Quick Sort

Overview

Quicksort is widely recognized as one of the most efficient sorting algorithms. Its operation involves dividing an array into smaller partitions and subsequently swapping or exchanging these partitions based on a comparison with a selected 'pivot' element.



Code

```
public class Quick
{  int partition (int a[], int start, int end)
    {
      int pivot = a[end]; // pivot element
       int i = (start - 1);
       for (int j = start; j <= end - 1; j++)
       {
          // If current element is smaller than the pivot
          if (a[j] < pivot)
          {
```

```
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}
/* function to implement quick sort */
void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting
 index, end = Ending index */
{
   if (start < end)
    {
        int p = partition(a, start, end);  //p is partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}
/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        System.out.print(a[i] + " ");
}
```

Pros & Cons

Quicksort is an in place sorting algorithm which has an average and best-case time complexity of O(n log n), making it highly efficient for large datasets. However it is not stable.

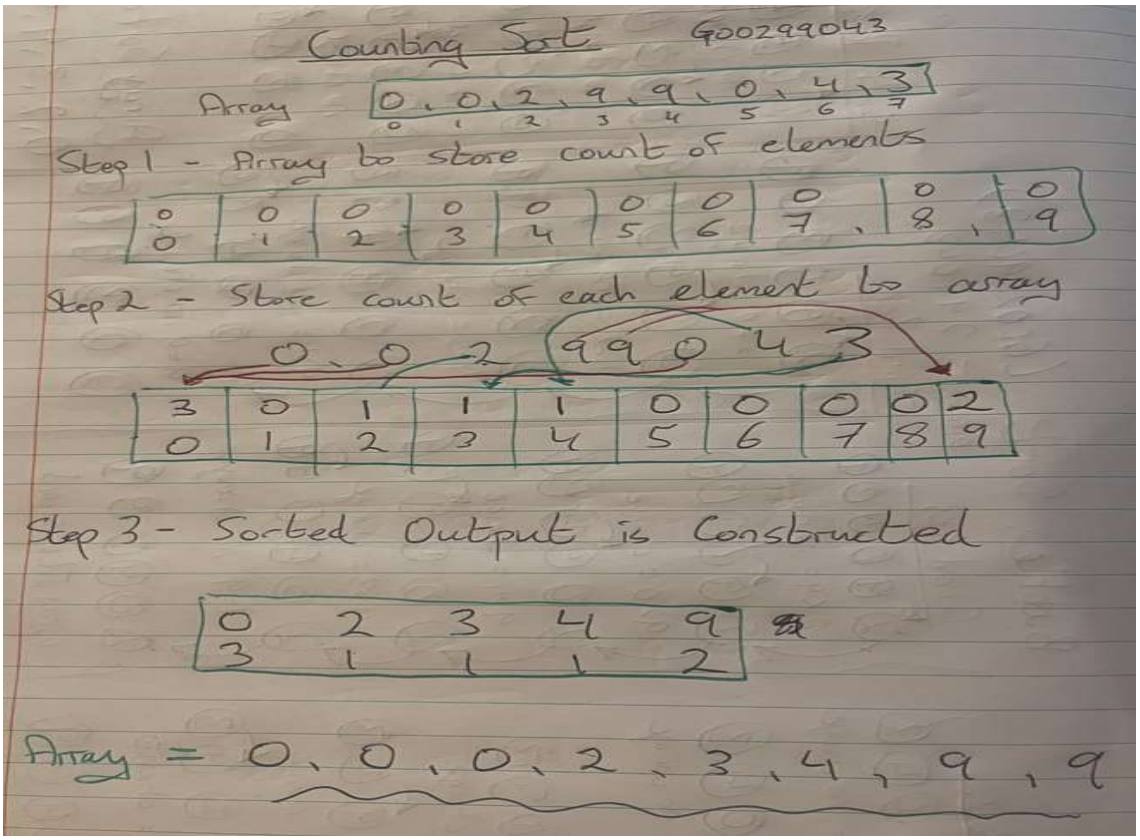
Counting Sort

Overview

Counting Sort is a non-comparison, out-of-place, stable sorting algorithm. Counting sort is effective when range is not greater than number of objects to be sorted. It can be used to sort the negative input values.

How it works

Counting sort creates a second array to store the count of elements in the original array. It then uses arithmetic to go through the original array and add a count of each index to the new array. It then creates a sorted output using the index count. See example of this below.



Code

```
class CountingSort {
        int getMax(int[] a, int n) {
          int max = a[0];
          for(int i = 1; i<n; i++) {
             if(a[i] > max)
```

```
        max = a[i];
  }
  return max; //maximum element from the array
}

void countSort(int[] a, int n) // function to perform counting sort
{
   int[] output = new int [n+1];
   int max = getMax(a, n);
   //int max = 42;
   int[] count = new int [max+1]; //create count array with size [max+1]

  for (int i = 0; i <= max; ++i)
  {
    count[i] = 0; // Initialize count array with all zeros
  }

  for (int i = 0; i < n; i++) // Store the count of each element
  {
    count[a[i]]++;
  }

   for(int i = 1; i<=max; i++)
      count[i] += count[i-1]; //find cumulative frequency

  /* This loop will find the index of each element of the original array in

count array, and
   place the elements in output array*/
  for (int i = n - 1; i >= 0; i--) {
    output[count[a[i]] - 1] = a[i];
    count[a[i]]--; // decrease count for same numbers
}

   for(int i = 0; i<n; i++) {
      a[i] = output[i]; //store the sorted elements into main array
   }
```

```
}

/* Function to print the array elements */
void printArray(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        System.out.print(a[i] + " ");
}
```

Pros & Cons

Counting sort is a non-comparative sorting algorithm, which means it doesn't rely on comparisons between elements in the input array. This makes it faster than comparison-based sorting algorithms like quicksort or merge sort, especially for small inputs or when the range of values in the input is small. It has a time complexity of O(n+k) and is a stable sorting algorithm. However counting sort requires extra space to store the counts of each element in the input array and If the range of values is not known, then counting sort is not applicable. Counting sort may not be efficient when the range of values in the input array is much larger than the number of elements in the array.

Below is a table of the time and space complexities of each algorithm being used for benchmarking.

G00299043	Best	Average	Worst	Space Compexity	Stable
Bubble Sort	O(n)	O(n2)	O(n2)	O(1)	Yes
Selection Sort	O(n2)	O(n2)	O(n2)	O(1)	No
Insertion Sort	O(n)	O(n2)	O(n2)	O(1)	Yes
Quick Sort	O(n*log n)	O(n*log n)	O(n*log n)	O(n)	Yes
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	Yes

Implementation and Benchmarking

Benchmarking, also known as a posteriori analysis, is an empirical method to compare relative performance of algorithm implementations (Dominic Carr 2022). Data can be used to validate theoretical / a priori algorithm analysis but It is prudent to conduct multiple runs of the same experimental setup to ensure you get a representative sample

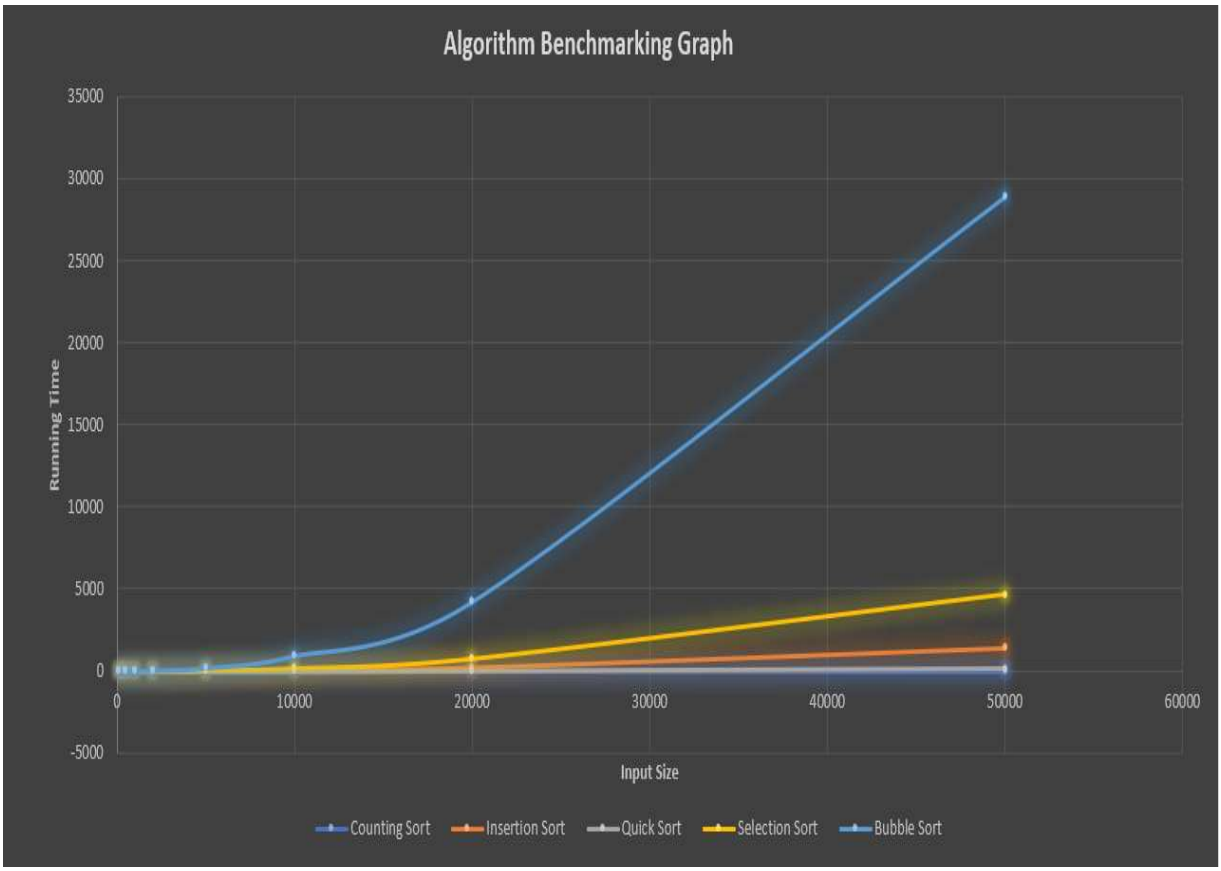
The application includes the five chosen sorting algorithms along with a main method called runner to test each one. It also included a ‘Benchmarking’ class to implement Benchmarking

of Algorithms and print its output to the console. To benchmark the algorithms, arrays of randomly generated integers with varying input sizes 'n' were utilized. The execution time of each algorithm was measured 10 times, and the average of these 10 measurements was then displayed on the console for further analysis

Analysis/Discussion

The results of the benchmarking were as you would expect after reading the theory on each algorithm above. From the graph below it is evident that Bubble Sort is slower in terms of time complexity than other sorting algorithms while the average results table shows that Counting Sort consistently performed better than the other algorithms.

Size	100	500	1000	2000	5000	10000	20000	50000
Counting Sort	0	0	0	0	0	0	1	1
Insertion Sort	0	1	3	2	14	60	226	1394
Quick Sort	0	0	0	0	2	7	23	127
Selection Sort	0	2	2	8	49	187	757	4635
Bubble Sort	1	4	8	29	187	926	4206	28883



References

1. GeeksforGeeks. “Bubble Sort - GeeksforGeeks.” GeeksforGeeks, 2 Feb. 2014, www.geeksforgeeks.org/bubble-sort/.
2. ---. “Insertion Sort - GeeksforGeeks.” GeeksforGeeks, 7 Mar. 2013, www.geeksforgeeks.org/insertion-sort/.
3. ---. “Sorting Algorithms - GeeksforGeeks.” GeeksforGeeks, 2017, www.geeksforgeeks.org/sorting-algorithms/.
4. Heineman, George T, et al. Algorithms in a Nutshell. Sebastopol, Ca, O’reilly, Cop, 2016.
5. “In-Place Algorithm - GeeksforGeeks.” GeeksforGeeks, 26 July 2018, www.geeksforgeeks.org/in-place-algorithm/.
6. “Non Comparison Based Sorting Algorithms.” OpenGenus IQ: Computing Expertise & Legacy, iq.opengenus.org/non-comparison-based-sorting/.
7. Srivastava, Priyank. “Stable Sorting Algorithms | Baeldung on Computer Science.” Www.baeldung.com, 28 Jan. 2020, www.baeldung.com/cs/stable-sorting-algorithms.
8. “Time Complexity and Space Complexity.” GeeksforGeeks, 11 July 2021, www.geeksforgeeks.org/time-complexity-and-space-complexity/.
9. tutorialspoint.com. “Data Structures and Algorithms Selection Sort.” Www.tutorialspoint.com, 2019, www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm.
10. Vladimir Estivill-Castro, and Derick Wood. “An Adaptive Generic Sorting Algorithm That Uses Variable Partitioning∗.” International Journal of Computer Mathematics, vol. 61, no. 3-4, 1 Jan. 1996, pp. 181–194, https://doi.org/10.1080/00207169608804511. Accessed 22 Apr. 2023.
11. “What Is Recursive Algorithm? Types and Methods | Simplilearn.” Simplilearn.com, www.simplilearn.com/tutorials/data-structure-tutorial/recursive-algorithm.
12. Wladston Ferreira Filho. Computer Science Distilled : Learn the Art of Solving Computational Problems. Las Vegas Nevada, Code Energy Llc, 2018.
13. Computational Thinking with Algorithms - Benchmarking in Java lecture notes - Dominic Carr 2022
14. “Selection Sort - Javatpoint.” *Www.javatpoint.com*,www.javatpoint.com/selectionsort.