COMPUTATIONAL THINKING WITH ALGORITHMS

HIGHER DIPLOMA IN COMPUTING - SOFTWARE DEVELOPMENT

Edivagner Ribeiro
Email: g00411275@gmit.ie

CONTENTS

INTRODUCTION

With the advent of computing, many everyday tasks were transported to a digital interface. Today we have a phenomenal amount of data produced and stored all the time. And finally, to process this data, the systematic organization of information becomes fundamental to minimize the computational cost. If you think of a simple task like finding a phone number in a list, this task can be tedious in an unsorted list, and dramatically increase the time to search for information.

Technically we can say that in a sorted series the previous element must be smaller than its successor, so if we must sort elements of type X:

$$X_i < X_j \quad \text{where } i < j$$

The sorting of the series of elements *X* is obtained by permuting the elements so that the sorted series of *X* is composed of all the elements of the initial series. If there are identical elements (same value) in the series, they must be placed sequentially in the series (Heineman, et al., 2016). In this way, sorting algorithms are employed to place the data collection elements in a certain order.

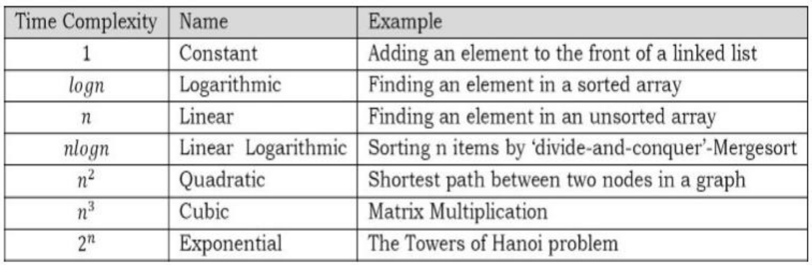
It is possible to find several methods to solve the same task. Basic everyday things like preparing lunch or the trajectory and transport (car, bicycle, etc) from home to work. For each type of task, we can have a sequence of very well-defined steps without ambiguity to reach the final objective. This sequence of steps is the algorithms. Therefore, to sort a collection of data we have several possible algorithms, so we create a new problem, how to differentiate and classify them. Here an important concept emerges, called Space and Time Complexity.

According to Karumanchi (Karumanchi, 2017) comparing algorithms or solutions in terms of execution time is not the best way, since this can change depending on the computer's specifications. The number of statements is also not the best way, as it depends on the language the algorithm is being programmed, however, if the execution time for a given algorithm is dependent on the size of the input data, this function would be characteristic for each algorithm and independent of the programming language and the computer we are executing.

Therefore, time complexity or computational complexity can be expressed as a function of task execution time where the execution time of each task element has a constant time and the time is solely dependent on the size of the input data (wikipedia_Time_complexity, 2022) (Sipser, 2013).

The variation in the processing time of an algorithm that grows with the increase in the amount of data, but also with the number of processes and loops is called the Rate of Growth. Depending on the process involved in the algorithms, we have different time complexity which were summarized in the table.

Table 1: List of growth rates (Karumanchi, 2017, p. 21)



Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

To define the complexity of the algorithm with multiple processes, we adopt the highest degree of complexity. If in an algorithm we need to multiply a matrix (Time Complexity cubic = n^3), but also find an element in the middle of the matrix, which is disordered (Time Complexity linear = n), the complexity of the algorithm will be the value of the highest complexity of the system, so we have a cubic complexity for this algorithm.

One way to describe and characterize the algorithm is called **Big O**, this notation relates the size of the input data and the number of processes involved in the algorithm to perform the task. The algorithms that have constant time are called **O(1)** the linear ones of **O(n)**, while the quadratics are the **O(n^2)** (Downey, 2017).

Sorting algorithms can be divided into two main types, comparative, and non-comparative. Comparative algorithms take an element and compare it with the others, looking for who is greater or equal, thus finding its new position in the series. Non-comparative methods are based on the distribution of information. For example, to sort playing cards where A < 2 < 3…< 10 < J < Q < K, and clubs (♣) < diamonds (♦) <hearts (♥) < spades (♠). Here the direct comparison can increase the time to perform the task, as we have an alphanumeric and symbolic sequence, that is, two different indices to be able to sort.

With a non-comparative algorithm, we can distribute the cards in alphanumeric piles (A, 2, 3, …, 10, J, Q, K). Then we collect the cards in the correct order (from A to K) and distribute them again now in 4 piles according to the symbol. As in the previous collection, the cards were already ordered by the first alphanumeric condition, when we collect now in the order of the symbols, we have the playing cards ordered by the two criteria.

SORTING ALGORITHMS

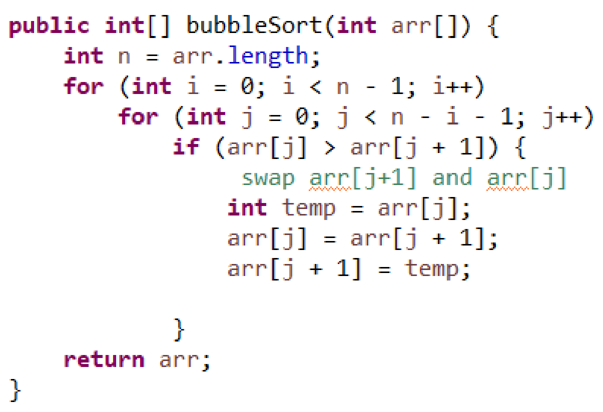
As mentioned before, sorting information may be necessary to speed up some other tasks. There are dozens of algorithms to solve this problem, so the choice of tool always depends on its application, in this case, some information about the data to be sorted (NIST, 2022).

Bubble Sort

This is a simple comparison algorithm, where each element is compared with its adjacent element and swaps places if its element is bigger.

17	33	30	38	13	Compare the first two values (17 and 33), as they are already in order, let's go to the next ones.
17	33	30	38	13	Compare 33 and 30, then put them in order by switching places
17	30	33	38	13	The next ones are already in order
17	30	33	38	13	Then we compare the last two and switch places to the correct order
17	30	33	13	38	the next iterations are necessary to move element 13 to the beginning of the series following the same logic of comparing each pair of elements.
17	30	13	33	38	
17	13	30	33	38	
13	17	30	33	38	

Among the algorithms proposed for this work, this is the least efficient of all, but an excellent example of a sorting algorithm. However, it works perfectly for nearly ordered data and small sets. Figure 1 shows the implementation of the algorithm in java.



```
public int[] bubbleSort(int arr[]) {
    int n = arr.length;                        // n = size of array
    for (int i = 0; i < n - 1; i++)            // loop all array ------> T1 = n * T2
        for (int j = 0; j < n - i - 1; j++)   // loop inside array ---> T2 = n * T3
            if (arr[j] > arr[j + 1]) {         // swap element --------> T3 = a
                 swap arr[j+1] and arr[j]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

            }
    return arr;
}
```

Figure 1: Implementation of the bubble sorting algorithm.
The code was adapted from geeksforgeeks (Mishra, 2022)

The bubble sorting algorithm takes O(n2) time due to the two loops to solve the problem as we can demonstrate.

$$T(n) = T_1 = n * \ T_1 = n * n = an^2$$

$$O\big(T(n)\big) = \ O(n^2)$$

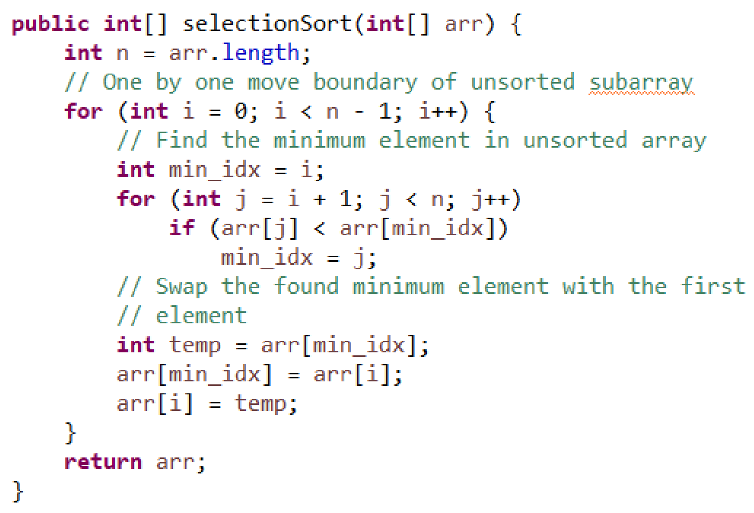
Selection Sort

This is one of the simplest sorting algorithms, its strategy is to search for the smallest element and reposition it in the data set. In a way, the selection sort algorithm divides the data set into two parts, one ordered and one unordered. Looking for the smallest elements in the unordered group and moving this element to the end of the ordered portion, as we can see in the figure below.

17	33	30	38	13	Initial Array

17	33	30	38	13	First finds the lowest value (13) and swaps position with the first in the list
13	33	30	38	17	Search for the lowest value in the rest of the series (17)
13	17	33	30	38	moves the 17 to the second position and shifts the rest of the series.
13	17	30	33	38	repeats the process until the end has the ordered series.
13	17	30	38	13	

This algorithm has the same characteristics as bubble sorting, however as it does not query the sorted data again, the execution time is shorter, however, it also has **O(n^2)** complexity time due to the two loops of the algorithm (Figure 2).



```
public int[] selectionSort(int[] arr) {
    int n = arr.length;
    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
    return arr;
}
```

Figure 2: Implementation of the Selection sorting algorithm. The code was adapted from stackabuse.com (Zobenica, 2022)

The complexity analysis here can be determined as follows: for a vector of 5 elements as we demonstrated earlier, we need to do 4 comparisons to find the minimum. In the next iteration, the vector is already partially sorted and now we do 3 comparisons, then 2 comparisons, and finally the last comparison so that the vector is fully coordinated. That is, the total number of comparisons is given by:

$$n * (n - 1) * \frac{1}{2} \;=\; \frac{n^2}{2} - \frac{n}{2}$$

As the time complexity we take only the highest degree and disregard the constants, we again have a quadratic time **O(n^2)** (Woltmann, 2020).

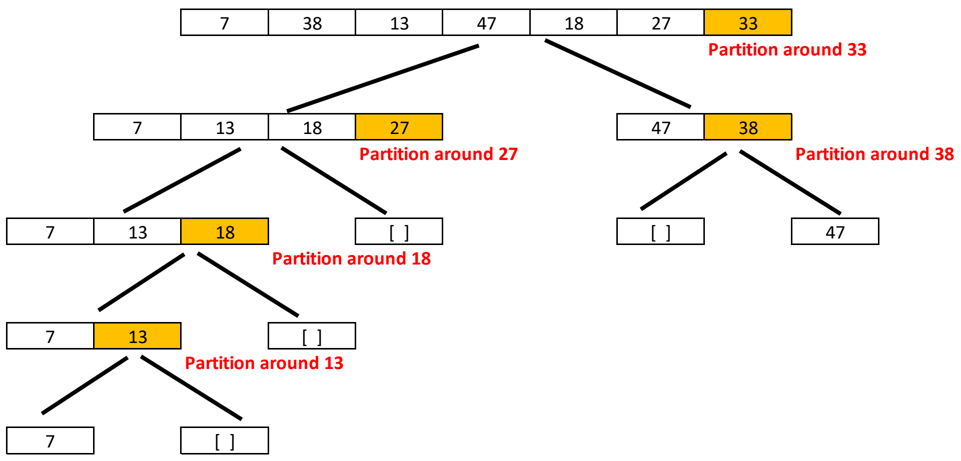
Quicksort

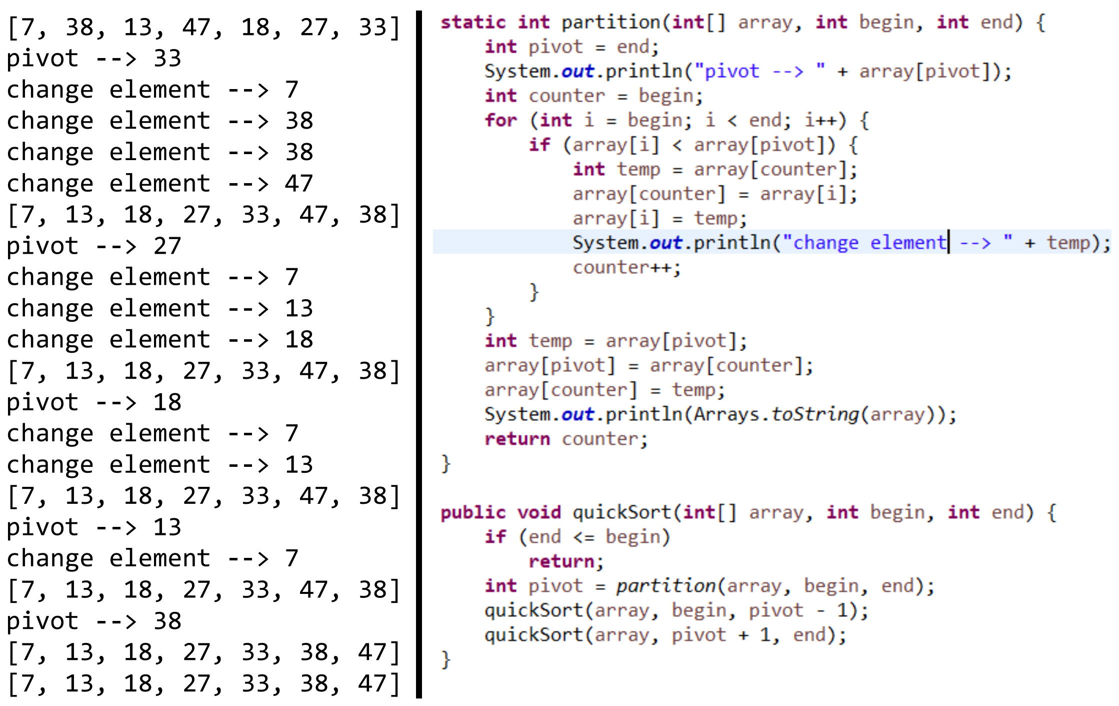
Divide and conquer, this is the great secret of success since the Roman Empire, passing through Napoleon and reaching the computer age. Solving a big problem can be much more complicated than a small problem, this principle applies here.

The Quicksort algorithm remains a comparison algorithm, whose strategy is to divide and conquer, transforming, in this case, a bid data set of numbers into two smaller sets and then dividing again in two and so on. The algorithm method can be described as follows:

First, pick a special element called a pivot. Compare this element with the others and divide the series by eating the smallest elements to the left and the largest ones to the right. we repeat the process in the new series formed (choose a new pivot and divide the series) until we have only a pair of elements as shown in Figure 3.

In this example, when we take an element to pivot, we always choose the element with the highest index, however, it could be a random choice, and finding the best pivot can be decisive for the speed of the algorithm.





```
[7, 38, 13, 47, 18, 27, 33]
pivot --> 33
change element --> 7
change element --> 38
change element --> 38
change element --> 47
[7, 13, 18, 27, 33, 47, 38]
pivot --> 27
change element --> 7
change element --> 13
change element --> 18
[7, 13, 18, 27, 33, 47, 38]
pivot --> 18
change element --> 7
change element --> 13
[7, 13, 18, 27, 33, 47, 38]
pivot --> 13
change element --> 7
[7, 13, 18, 27, 33, 47, 38]
pivot --> 38
[7, 13, 18, 27, 33, 38, 47]
[7, 13, 18, 27, 33, 38, 47]
```

```
static int partition(int[] array, int begin, int end) {
    int pivot = end;
    System.out.println("pivot --> " + array[pivot]);
    int counter = begin;
    for (int i = begin; i < end; i++) {
        if (array[i] < array[pivot]) {
            int temp = array[counter];
            array[counter] = array[i];
            array[i] = temp;
            System.out.println("change element --> " + temp);
            counter++;
        }
    }
    int temp = array[pivot];
    array[pivot] = array[counter];
    array[counter] = temp;
    System.out.println(Arrays.toString(array));
    return counter;
}

public void quickSort(int[] array, int begin, int end) {
    if (end <= begin)
        return;
    int pivot = partition(array, begin, end);
    quickSort(array, begin, pivot - 1);
    quickSort(array, pivot + 1, end);
}
```

Figure 3: Implementation of the QuickSort algorithm and the partition tree. The code was adapted from stackabuse.com (Zobenica, 2022)

This is an efficient algorithm with good performance; however, the time complexity here depends on the size of the data, and on the value of the pivot with respect to the dataset. For this algorithm we have the following equation for the time complexity (Zobenica, 2022):

$$T(n) = T(k) + T\,(n - k - 1) + O(n)$$

as the pivot can be any element of the dataset, the problem varies from [0] to [n-1] and k is the size of each new dataset until the end of the n iterations.

In the best scenario, we have a balanced system, where the pivot divides the dataset into two groups of approximately the same size. Therefore, in general, we can have a time complexity.

$$T(n) = O(n \log n)$$

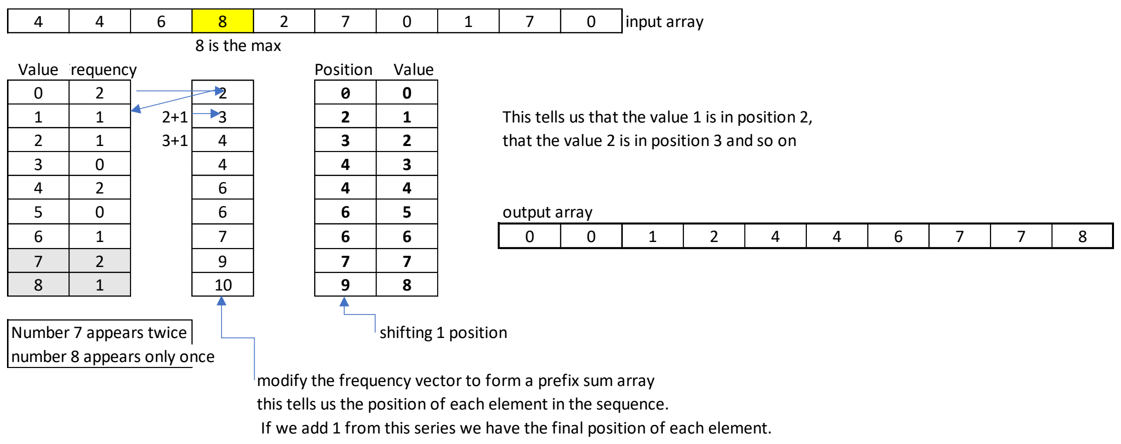
The worst scenario can exist when the pivot is approximately zero or the maximum value of the dataset, which would imply working with practically the entire dataset in subsequent partitions. This scenario can be expressed by a quadratic temporal complexity.

$$T(n) = O(n^2)$$

Counting Sort

This algorithm belongs to the non-comparative sorting group, its principle is in the search for the frequency of unique elements in the data set.

Its process is quite interesting, first, we identify the biggest element in the dataset. We then created an array of zeros assuming we would have one element of each value randomly distributed in the original dataset. Then we visit all the elements, and we go count how many times each unique element appears in the original dataset, and finally rewrite the array with the values according to the count. We see an example illustrated in Figure 4.



input array
8 is the max
2+1
3+1
Number 7 appears twice
number 8 appears only once
modify the frequency vector to form a prefix sum array
this tells us the position of each element in the sequence.
If we add 1 from this series we have the final position of each element.
shifting 1 position
This tells us that the value 1 is in position 2,
that the value 2 is in position 3 and so on
output array

4	4	6	8	2	7	0	1	7	0

Value	requency	
0	2	2
1	1	3
2	1	4
3	0	4
4	2	6
5	0	6
6	1	7
7	2	9
8	1	10

Position	Value
0	**0**
2	**1**
3	**2**
4	**3**
4	**4**
6	**5**
6	**6**
7	**7**
9	**8**

0	0	1	2	4	4	6	7	7	8

```
[4, 4, 6, 8, 2, 7, 0, 1, 7, 0]
largest element of the array: 4
largest element of the array: 6
largest element of the array: 8
largest element of the array: 8
largest element of the array: 8
largest element of the array: 8
largest element of the array: 8
largest element of the array: 8
largest element of the array: 8
count array:
[0, 0, 0, 0, 0, 0, 0, 0, 0]
Store the count of each element
count array:
[2, 1, 1, 0, 2, 0, 1, 2, 1]
Store the cumulative count of each
array
count array:
[2, 3, 4, 4, 6, 6, 7, 9, 10]
Find the index of each element of
the
original array in count array
count array:
[0, 2, 3, 4, 4, 6, 6, 7, 9]

[0, 0, 1, 2, 4, 4, 6, 7, 7, 8]
```

```java
public int[] countSort(int array[], int size) {

    int[] output = new int[size + 1];

    // Find the largest element of the array
    int max = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max)
            max = array[i];
    }
    int[] count = new int[max + 1];

    // Initialize count array with all zeros.
    for (int i = 0; i < max; ++i) {
        count[i] = 0;
    }
    // Store the count of each element
    for (int i = 0; i < size; i++) {
        count[array[i]]++;
    }
    // Store the cummulative count of each array
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }
    // Find the index of each element of the
    // original array in count array, and
    // place the elements in output array
    for (int i = size - 1; i >= 0; i--) {
        output[count[array[i]] - 1] = array[i];
        count[array[i]]--;
    }

    // Copy the sorted elements into original array
    for (int i = 0; i < size; i++) {
        array[i] = output[i];
    }
    return array;
}
```

Figure 4: Implementation of the Counting algorithm. Code was adapted from (Programiz, 2022)

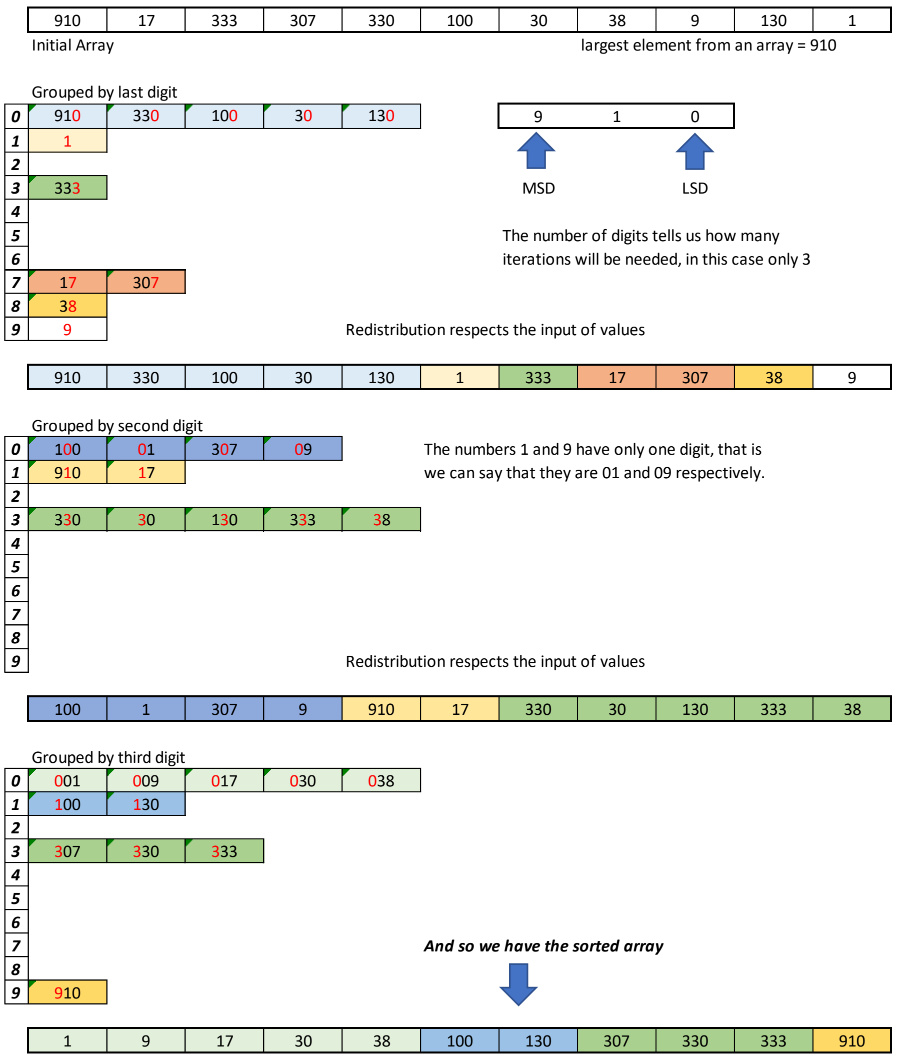
Counting Sort is a linear time complexity algorithm that depends on the number of elements ***n*** and the range ***k***. However, the loops for doing the prefix sum on the count array take ***O(k)*** time. And the other loops for the total number of elements take ***O(n)*** times (Mangal, 2017). In this way we can write the following equation:

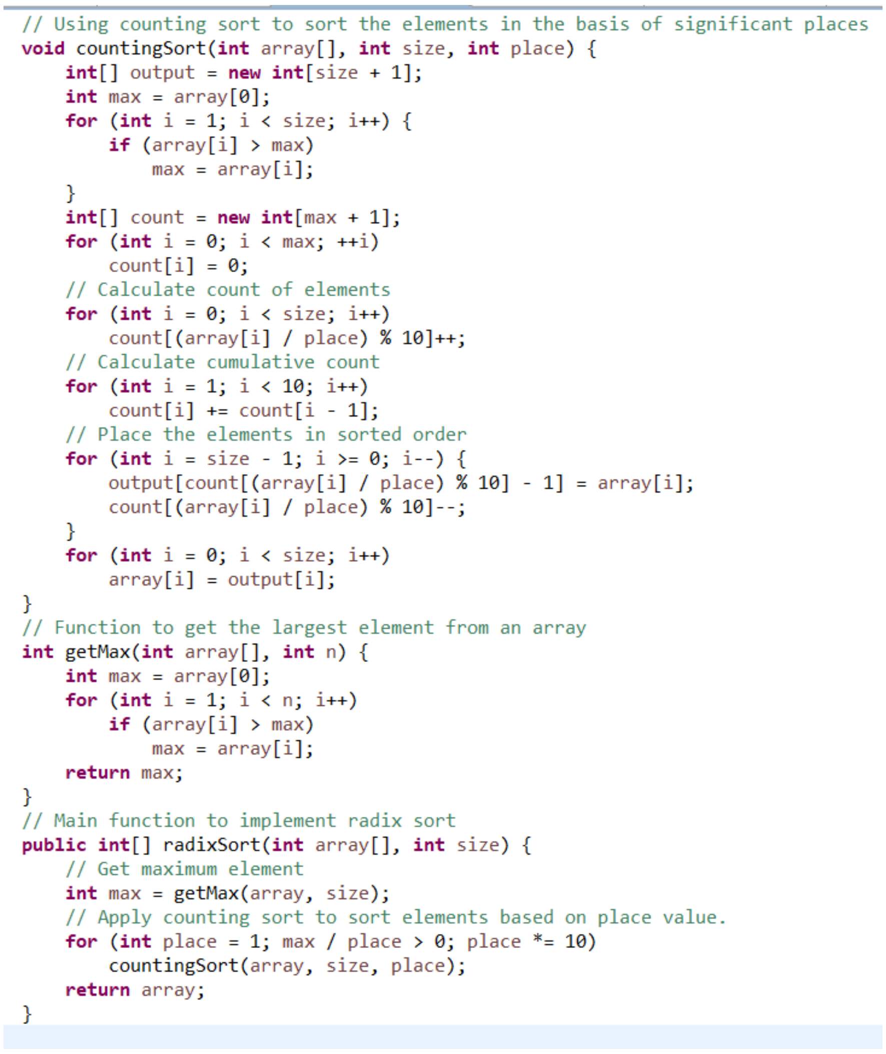
$$O(k) + O(n) + O(k) + O(n) = O(n+k)$$

Radix Sort

Radix is a non-comparative algorithm, which sorts elements digit by digit (NIST, 2022). The idea is quite simple, for example, if we have a list of names alphabetically unsorted. We can separate the names into 26 groups, using the 26 characters of the alphabet as a key to form an order relative to the first characters of the name.

Here we already have the list sorted by the first character that composes the name. Now within each group we look at the second character and separate it again into 26 groups, this would give names with the initials AA, AB, AC, AD... Then we can do the same process over and over until all the subgroups and their respective subgroups are sorted. In practice, this algorithm works from the least significant digit (LSD) to the most significant digit (MSD). In Figure 5 we show an example with a numeric array.





```
// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place) {
    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;
    // Calculate count of elements
    for (int i = 0; i < size; i++)
        count[(array[i] / place) % 10]++;
    // Calculate cumulative count
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
    // Place the elements in sorted order
    for (int i = size - 1; i >= 0; i--) {
        output[count[(array[i] / place) % 10] - 1] = array[i];
        count[(array[i] / place) % 10]--;
    }
    for (int i = 0; i < size; i++)
        array[i] = output[i];
}
// Function to get the largest element from an array
int getMax(int array[], int n) {
    int max = array[0];
    for (int i = 1; i < n; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}
// Main function to implement radix sort
public int[] radixSort(int array[], int size) {
    // Get maximum element
    int max = getMax(array, size);
    // Apply counting sort to sort elements based on place value.
    for (int place = 1; max / place > 0; place *= 10)
        countingSort(array, size, place);
    return array;
}
```

Figure 5: Implementation of the Radix Sort algorithm. Code was adapted from programiz (Programiz, 2022)

The Radix algorithm seems to me to be a very efficient strategy, however it has Counting Sort as its background, which is responsible for counting each element, identifying the index, and distributing them in the correct order, as we showed in detail in the previous section.

The difference here is that in the case of numbers, we need to distribute the information only with the values from 0 to 9, as we use one digit at a time. The same would happen with words, we would have 26 groups and the number of loops depends on the size of the largest existing word in the series.

Since this algorithm is based on counting sort, the temporal complexity is also linear, dependent on the number of elements **n** and the size of the largest element **k** in the array. But in radix, we must call counting sort in a for loop repeatedly equal to the **d** number of digits of the largest element in the dataset.

$$\text{Radix sort} = O(n+k) * d$$

$$O\big(d(n+k)\big)$$

IMPLEMENTATION & BENCHMARKING

As mentioned before, to perform the same task, we can have different tools, however the choice of tool depends on the task. In this case, each type of algorithm has its limitations regarding the size of the dataset. As we will see, some have an acceptable performance for small arrays, others are independent of the size. However, a complete study for algorithm analysis should ideally be tested in different scenarios causing unexpected real data situations. (Heineman, et al., 2016). In this report, we focus on the study of the running time of the sorting algorithms with different array sizes.

Figure 6 shows the flowchart that we used to obtain the running time for the different sorting algorithms. To study the running time of algorithms in a systematic way we first generate an array of random numbers ranging from 0 to 500000. So, in each test set, we took a part of this primary array and tested it on all the algorithms (Figure 7), with this strategy, we have a systematic evaluation of each algorithm.

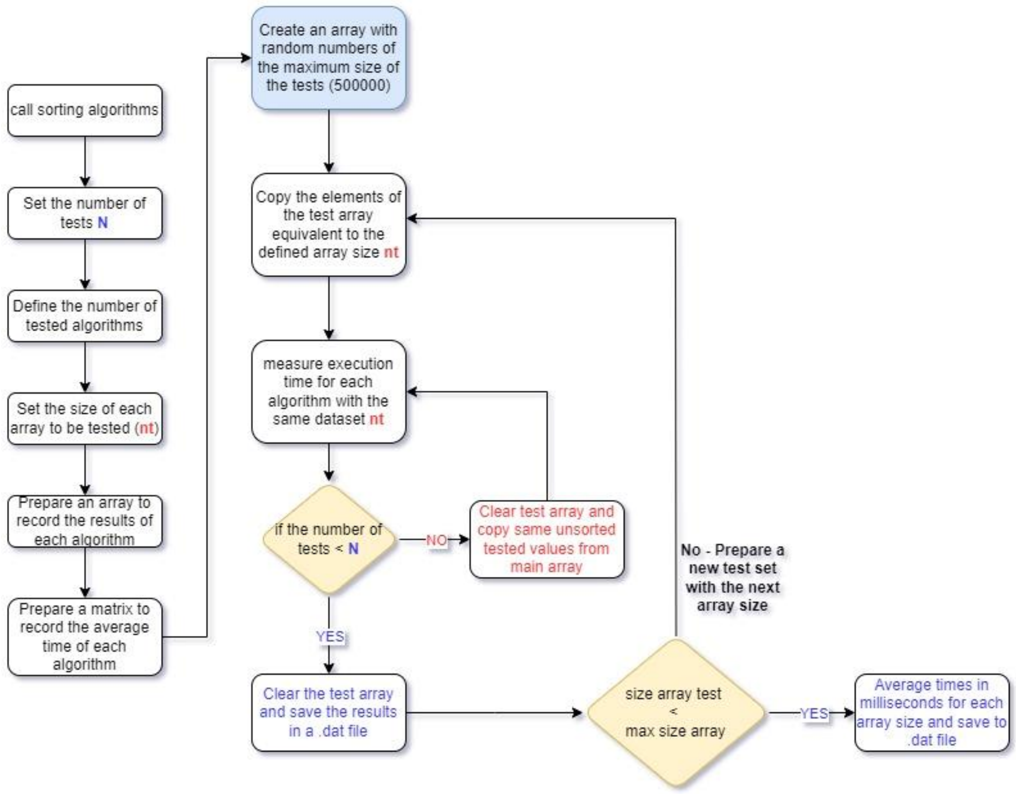

call sorting algorithms
Set the number of tests N
Define the number of tested algorithms
Set the size of each array to be tested (nt)
Prepare an array to record the results of each algorithm
Prepare a matrix to record the average time of each algorithm
Create an array with random numbers of the maximum size of the tests (500000)
Copy the elements of the test array equivalent to the defined array size nt
measure execution time for each algorithm with the same dataset nt
if the number of tests < N
NO
Clear test array and copy same unsorted tested values from main array
YES
Clear the test array and save the results in a .dat file
No - Prepare a new test set with the next array size
size array test < max size array
YES
Average times in milliseconds for each array size and save to .dat file

Figure 6: Flowchart of the study of running times of sorting algorithms.

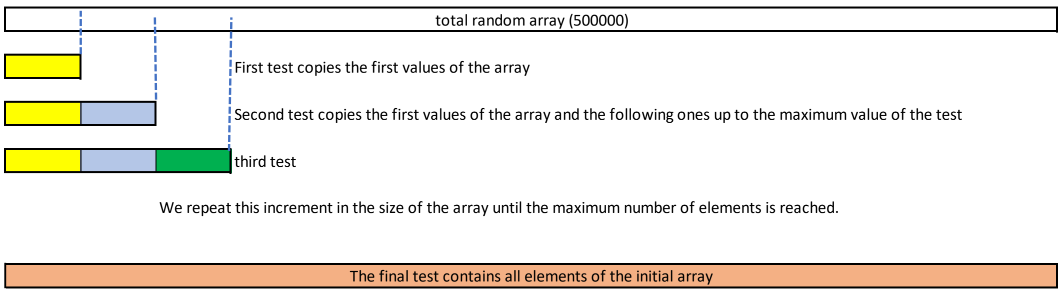

total random array (500000)
First test copies the first values of the array
Second test copies the first values of the array and the following ones up to the maximum value of the test
third test
We repeat this increment in the size of the array until the maximum number of elements is reached.
The final test contains all elements of the initial array

Figure 7: Technique to generate random test arrays for multiple tests.

The Table 2 shows the values obtained for one of the tests performed with the bubble algorithm and its respective graph (Figure 8) with the values of each of the tests. The tests can be easily fitted by a quadratic polynomial function, which is consistent with the time complexity of ***O(n^2)***. For an array of 10000 elements, we have a time of the order of 0.1 seconds, which is perfectly acceptable, however, as this algorithm has a quadratic bias, the time increases dramatically with the increase in the number of elements.

Table 2: Running times in milliseconds with different array sizes with bubble sort algorithm

ARRAY SIZE	TEST 01	TEST 02	TEST 03	TEST 04	TEST 05	TEST 06	TEST 07	TEST 08	TEST 09	TEST 10
100	0.216	0.178	0.198	0.180	0.164	0.164	0.190	0.195	0.178	0.169
250	0.781	0.235	0.197	0.169	0.152	0.182	0.208	0.234	0.207	0.196
500	0.779	0.779	0.777	0.787	0.781	0.745	0.997	0.752	0.738	1.026
750	1.724	1.566	1.862	1.974	2.471	1.893	1.624	1.659	1.692	1.758
1000	3.539	1.245	1.149	1.163	1.121	1.055	1.096	1.200	0.993	0.905
1250	1.347	1.497	1.548	1.597	1.584	1.522	1.550	1.551	1.497	1.316
2500	5.708	5.520	5.534	6.615	5.157	4.773	5.587	5.155	4.997	4.970
3750	11.400	11.915	13.146	10.878	10.834	10.759	11.975	11.468	10.831	11.354
5000	23.258	22.220	22.064	24.204	21.118	22.023	22.040	21.058	22.715	20.574
6250	34.510	34.936	36.041	33.708	34.459	37.196	41.223	34.057	36.180	35.145
7500	58.692	57.798	71.178	60.855	53.291	56.599	52.450	54.171	53.252	54.447
8750	77.984	75.528	73.105	76.152	80.919	81.309	77.577	74.620	71.683	75.337
10000	100.086	100.812	101.377	110.441	100.866	100.112	98.934	98.445	110.761	104.523

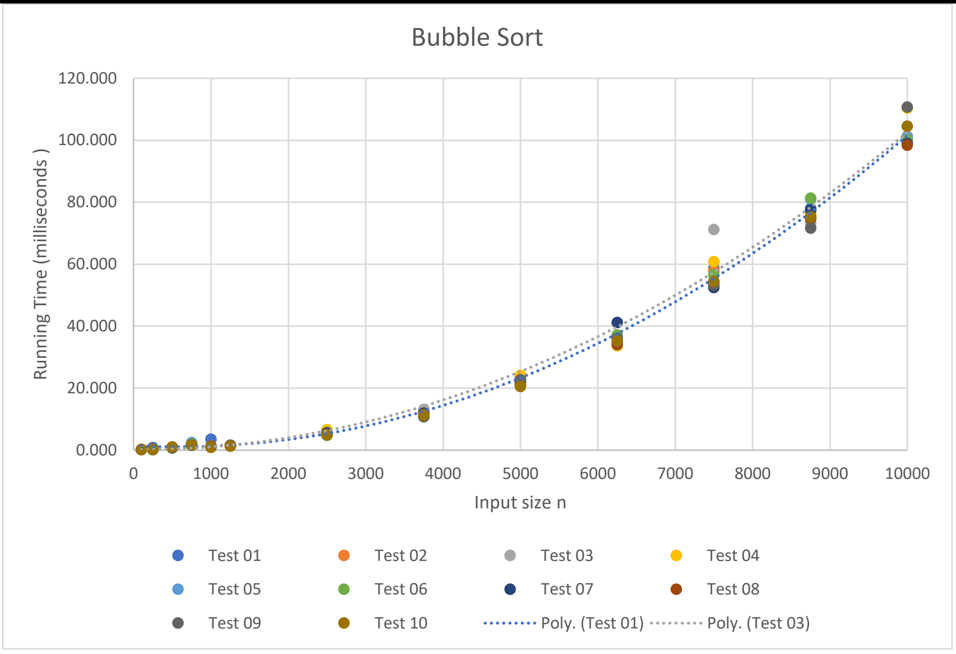


Figure 8: Running times for the bubble algorithm

Figures 9 and 10 show the comparison of the average times of the five algorithms that can be read in detail in the respective tables. It is interesting to note the linear curve of the Quicksort, Counting sort, and Radix sort algorithms. This linear tendency guarantees a better execution time even with the increase of the dataset size, wherewith 500000 elements we have an approximate time of 45 milliseconds.

Table 3: Comparative average times of the algorithms

Size	*100*	*250*	*500*	*750*	*1000*	*1250*	*2500*	*3750*	*5000*	*6250*	*7500*	*8750*	*10000*
Bubble	0.183	0.256	0.816	1.822	1.346	1.501	5.402	11.456	22.127	35.745	57.273	76.421	102.636
Selection	0.087	0.188	0.214	0.215	0.382	0.587	2.15	3.898	7.49	10.887	16.388	21.082	26.958
Quick	0.028	0.021	0.016	0.027	0.049	0.062	0.168	0.196	0.282	0.444	0.458	0.49	0.629
Counting	0.182	0.078	0.081	0.063	0.057	0.026	0.033	0.043	0.044	0.042	0.031	0.038	0.06
Radix	0.249	0.183	0.28	0.111	0.138	0.167	0.292	0.295	0.306	0.383	0.357	0.429	0.682

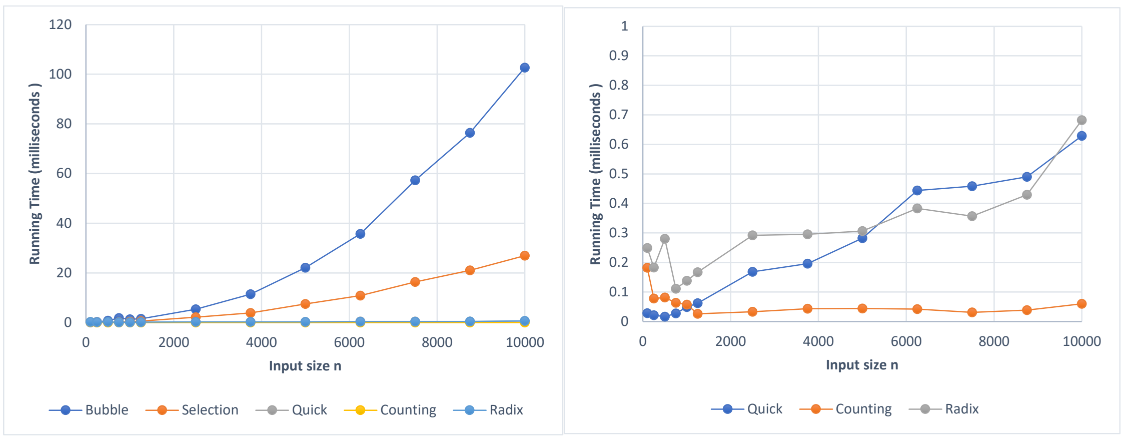


Figure 9: Comparative average times of the algorithms (input size 10000)

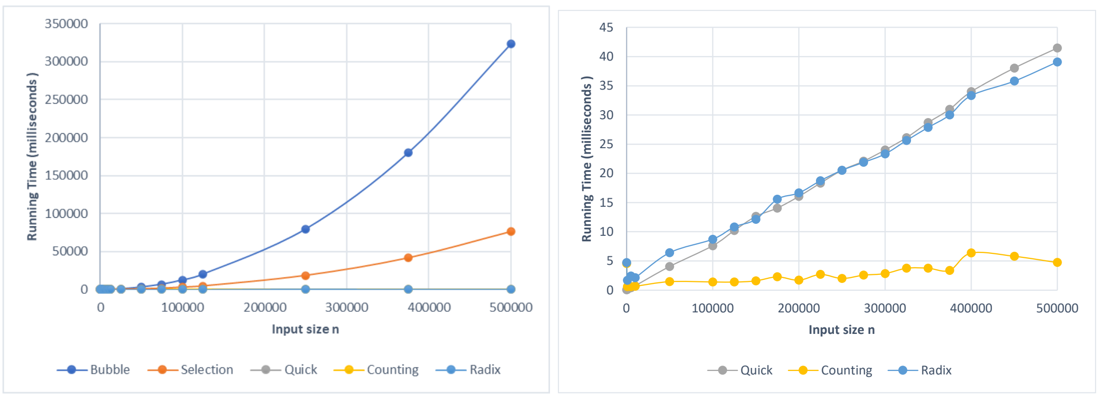


Figure 10: Comparative average times of the algorithms (input size 500000)

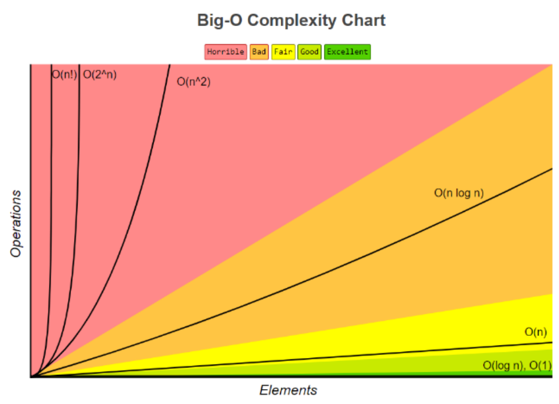
Size	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**1000**	**1200**	**1500**	**2000**	**3000**
Bubble	0.163	0.283	0.224	0.166	0.259	0.356	0.579	0.746	0.931	1.136	1.995	3.278	7.988
Selection	0.09	0.196	0.141	0.11	0.105	0.168	0.237	0.226	0.347	0.444	0.805	1.208	2.446
Quick	0.037	0.019	0.013	0.014	0.021	0.025	0.023	0.035	0.046	0.055	0.089	0.131	0.157
Counting	4.574	0.572	0.513	0.485	0.84	0.531	0.614	0.754	0.51	0.572	0.49	0.723	0.687
Radix	5.898	1.291	1.679	1.538	3.398	1.79	1.77	1.589	3.121	1.546	1.087	1.699	1.748

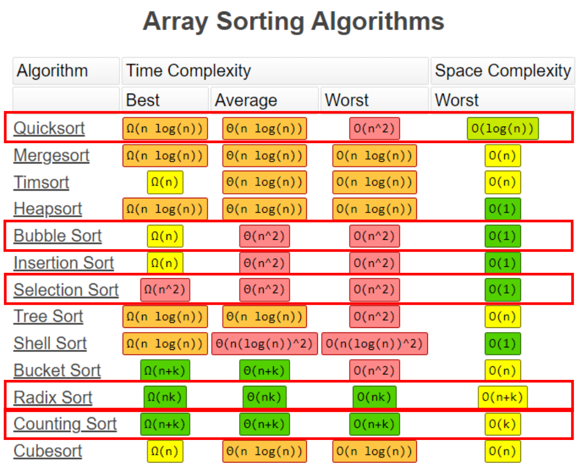
Size	**5000**	**7500**	**10000**	**12500**	**25000**	**50000**	**75000**	**100000**	**125000**	**250000**	**375000**	**500000**
Bubble	21.732	57.451	104.974	171.659	736.437	3149.196	6788.939	12356.16	19862.52	79337.27	179918.4	323272.1
Selection	6.713	15.001	28.145	41.975	191.351	656.726	1442.774	2915.844	4486.678	18458.01	41567.48	76322.02
Quick	0.282	0.43	0.612	0.806	1.926	3.45	6.239	7.359	8.949	48.524	30.25	41.272
Counting	0.564	0.572	0.504	0.573	0.787	1.028	1.312	1.865	1.776	7.582	3.916	5.678
Radix	2.215	1.925	2.424	2.32	4.138	5.33	7.395	14.994	12.714	57.954	32.85	44.79

Size	**100**	**1000**	**5000**	**10000**	**50000**	**100000**	**125000**	**150000**	**175000**	**200000**	**225000**	**250000**	**275000**	**300000**	**325000**	**350000**	**375000**	**400000**	**450000**	**500000**
Quick	0.034	0.112	0.399	0.734	4.033	7.59	10.196	12.615	14.049	16.039	18.318	20.526	22.077	23.993	26.117	28.687	30.972	33.985	38.034	41.471
Counting	4.539	0.589	0.605	0.649	1.449	1.397	1.376	1.584	2.269	1.696	2.687	1.992	2.561	2.831	3.752	3.754	3.382	6.36	5.803	4.728
Radix	4.711	1.666	2.412	2.136	6.407	8.713	10.813	12.106	15.569	16.642	18.742	20.53	21.878	23.316	25.64	27.864	29.994	33.331	35.803	39.086

CONCLUSIONS

The tests performed agree with the models described in the report for the Big O values. Considering the table in the figure below, the data when analysed by QuickSort, when demonstrating a linear trend, we can conclude that the input data could not reproduce the worst scenario.





Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure 11: Big O diagram and table comparing different algorithms (Rowell, 2022)

When dealing with a small amount of data, the speed of the algorithms in classifying the data is very balanced, however, the Radix algorithm can present great difficulty for few data depending on the range of values. As the Radix algorithm depends on the number of digits, in a dataset, even if small, but with high values, that is, with many digits, the running time tends to be high, as it would have to run a loop for each digit, even with a linear trend, we would have a loss of efficiency.

BIBLIOGRAPHY

Downey, A. B., 2017. *Think Data Structures.* First Edition ed. s.l.:O'Reilly .

Heineman, G. T., Pollice, G. & Selkow, S., 2016. *Algorithms in a Nutshell - Second Edition.* s.l.:O'Reilly.

Karumanchi, N., 2017. *Data Structures And Algorithms Made Easy In JAVA.* s.l.:CareerMonk Publications..

Mangal, P., 2017. *Codingeek.* [Online]
Available at: https://www.codingeek.com/algorithms/counting-sort-explanation-pseudocode-and-implementation/
[Accessed 12 05 2022].

Mishra, R., 2022. *geeksforgeeks.org.* [Online]
Available at: https://www.geeksforgeeks.org/bubble-sort/
[Accessed 11 05 2022].

NIST, 2022. *NIST - radix sort.* [Online]
Available at: https://xlinux.nist.gov/dads/HTML/radixsort.html
[Accessed 12 05 2022].

NIST, 2022. *NIST: National Institute of Standards and Technology.* [Online]
Available at: https://xlinux.nist.gov/dads/HTML/sort.html
[Accessed 11 05 2022].

Programiz, 2022. *Programiz.* [Online]
Available at: https://www.programiz.com/dsa/counting-sort
[Accessed 10 05 2022].

Programiz, 2022. *Radix Sort Algorithm.* [Online]
Available at: https://www.programiz.com/dsa/radix-sort
[Accessed 13 05 2022].

Rowell, E., 2022. *Big O Cheat Sheet.* [Online]
Available at: https://www.bigocheatsheet.com/
[Accessed 13 05 2022].

Sipser, M., 2013. *Introduction to the Theory of Computation.* 3 ed. ed. Boston: Cengage Learning.

wikipedia_Time_complexity, 2022. *https://en.wikipedia.org/wiki/Time_complexity.* [Online]
Available at: https://en.wikipedia.org/wiki/Time_complexity
[Accessed 05 10 2022].

Woltmann, S., 2020. *HappyCoders - Selection Sort – Algorithm, Source Code, Time Complexity.* [Online]
Available at: https://www.happycoders.eu/algorithms/selection-sort/
[Accessed 12 05 2022].

Zobenica, D., 2022. *Stack Abuse.* [Online]
Available at: https://stackabuse.com/sorting-algorithms-in-java/
[Accessed 11 05 2022].