Report

Benchmarking Sorting Algorithms

Computational Thinking with Algorithms

G00376368 – Karolina Szafran-Belzowska

Table of contents

[1. Introduction 3](#_Toc40180903)

[1.1. The concept of sorting 3](#_Toc40180904)

[1.2. Time and Space Complexity 4](#_Toc40180905)

[1.2.1. Bubble Time complexity 5](#_Toc40180906)

[1.2.2. Selection Sort Complexity 5](#_Toc40180907)

[1.2.3. Insertion Sort Complexity 5](#_Toc40180908)

[1.2.4. Merge Sort Complexity 5](#_Toc40180909)

[1.2.5. Counting Sort Complexity 6](#_Toc40180910)

[1.2.6. Big O Complexity 6](#_Toc40180911)

[1.3. In-place algorithm 8](#_Toc40180912)

[1.4. Stable sorting 9](#_Toc40180913)

[1.5. Comparison-based and non-comparison-based sorting algorithms 10](#_Toc40180914)

[2. Sorting Algorithms 11](#_Toc40180915)

[2.1. Bubble Sort 11](#_Toc40180916)

[2.2. Selection Sort 13](#_Toc40180917)

[2.3. Insertion Sort 14](#_Toc40180918)

[2.4. Merge Sort 16](#_Toc40180919)

[2.5. Counting Sort 18](#_Toc40180920)

[3. Implementation and Benchmarking 19](#_Toc40180921)

[3.1. Python code 20](#_Toc40180922)

[3.1.1. Random arrays 20](#_Toc40180923)

[3.1.2. Timing the algorithm and benchmark 22](#_Toc40180924)

[3.1.3. Average time 22](#_Toc40180925)

[3.1.4. DataFrame 22](#_Toc40180926)

[3.1.5. Plot Graph 23](#_Toc40180927)

[3.1.6. Final results 23](#_Toc40180928)

[4. References 25](#_Toc40180929)

# Introduction

## The concept of sorting

A sorting algorithm is a methods for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest or shortest-to-longest distance. Sorting algorithms take a list of items as input data, perform specific operations on those lists and deliver ordered arrays as output.[1]

A sorting algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.[2][3]

Choosing the best sorting algorithm is as about knowing what you are sorting as it is about the relative performance of the algorithms.

The output of any sorting algorithm must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element),
2. And the output is a [permutation](https://en.wikipedia.org/wiki/Permutation) (a reordering of the original elements) of the input.

The input data is often stored in an [array](https://en.wikipedia.org/wiki/Array_data_type), which allows [random access](https://en.wikipedia.org/wiki/Random_access), rather than a list, which only allows [sequential access](https://en.wikipedia.org/wiki/Sequential_access), though many algorithms can be applied to either type of data after suitable modification.

Sorting algorithms are often classified by:

* [**Computational complexity**](https://en.wikipedia.org/wiki/Computational_complexity_theory) ([worst, average and best](https://en.wikipedia.org/wiki/Best,_worst_and_average_case) behavior) in terms of the size of the list (*n*). For typical serial sorting algorithms good behavior is O(*n* log *n*), with parallel sort in O(log2 *n*), and bad behavior is O(*n*2). Ideal behavior for a serial sort is O(*n*), but this is not possible in the average case. Optimal parallel sorting is O(log *n*). [Comparison-based sorting algorithms](https://en.wikipedia.org/wiki/Comparison_sort) need at least Ω(*n* log *n*) comparisons for most inputs.
* [**Computational complexity**](https://en.wikipedia.org/wiki/Computational_complexity_theory)**of swaps** (for "in-place" algorithms).
* [**Memory**](https://en.wikipedia.org/wiki/Memory_(computing))**usage** (and use of other computer resources). Some sorting algorithms are "[in-place](https://en.wikipedia.org/wiki/In-place_algorithm)". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(*n*)) additional memory is considered "in-place".
* **Recursion.** Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
* **Stability:** [stable sorting algorithms](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability) maintain the relative order of records with equal keys (i.e., values).
* Whether or not they are **a**[**comparison sort**](https://en.wikipedia.org/wiki/Comparison_sort)**.** A comparison sort examines the data only by comparing two elements with a comparison operator.
* **General method:** insertion, exchange, selection, merging, *etc.* Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.
* Whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
* **Adaptability:** Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be [adaptive](https://en.wikipedia.org/wiki/Adaptive_sort).

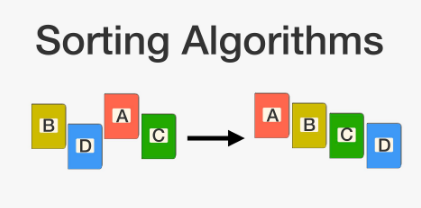


Image 1: Sorting algorithms: <https://brilliant.org/wiki/sorting-algorithms/> on 08/05/2020

## Time and Space Complexity

The complexity of an algorithm is the function f(n) which gives the running time and storage space requirement of the algorithm in terms of the size n of the input data. Complexity refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size n of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. **Best Case** : The minimum possible value of f(n) is called the best case.
2. **Average Case** : The expected value of f(n).
3. **Worst Case** : The maximum value of f(n) for any key possible input.

Best, worst, and average cases of a given [algorithm](https://en.wikipedia.org/wiki/Algorithm) express what the [resource](https://en.wikipedia.org/wiki/Resource_(computer_science)) usage is *at least*, *at most* and *on average*. Average performance and worst-case performance are the most used in algorithm analysis.[4]

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input. To compare standard sorting algorithms on the basis of space, the Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort and Heap Sort use O(1) auxiliary space. Space complexity of all these sorting algorithms is O(n) though.[5]

### Bubble Time complexity

**Worst and Average Case Time Complexity:**O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

Bubble sort takes minimum time (Order of n) when elements are already sorted.

### Selection Sort Complexity

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)

Selection sort never makes more than O(n) swaps and can be useful when memory write is a costly operation.

### Insertion Sort Complexity

**Time Complexity:** O(n\*2)

**Auxiliary Space:**O(1)

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

### Merge Sort Complexity

**Time Complexity:**  Sorting arrays on different machines.

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.   
T(n) = 2T(n/2) + 

Time complexity of Merge Sort is  in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

**Auxiliary Space:** O(n)

### Counting Sort Complexity

**Time Complexity:** O(n+k) where n is the number of elements in input array and k is the range of input.  
**Auxiliary Space:** O(n+k)

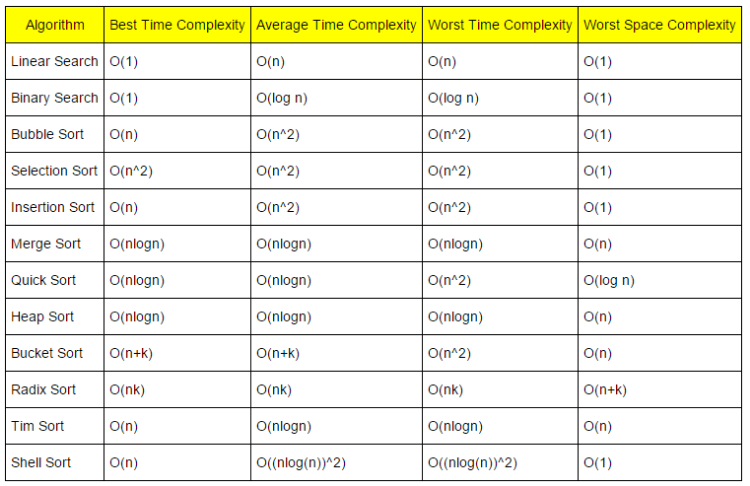


Image 2: Time complexity. Taken from: <https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/> on 08/05/2020

### Big O Complexity

Big O notation is a mathematical notation that describes the [limiting behavior](https://en.wikipedia.org/wiki/Asymptotic_analysis) of a [function](https://en.wikipedia.org/wiki/Function_(mathematics)) when the [argument](https://en.wikipedia.org/wiki/Argument_of_a_function) tends towards a particular value or infinity.[6] Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used by an algorithm.

Below are examples of Big O notation[7][8]:

1. O(1), constant time algorithm - describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.



1. O(N), linear time algorithm - describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.



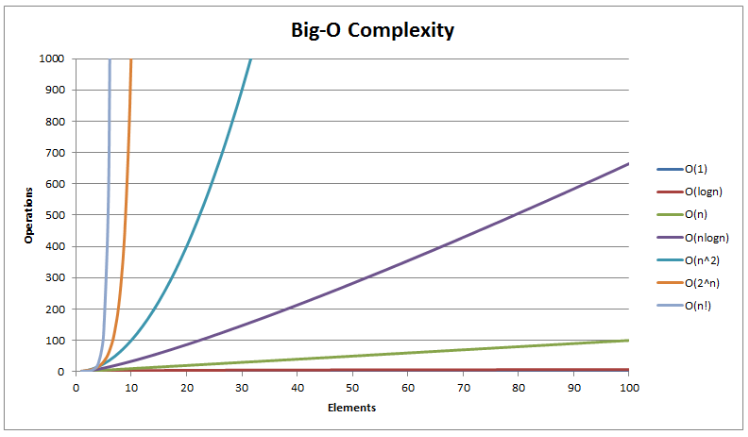
1. O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

If the list has two letters in it the program will take four operations to run, but if the list has four trillion letters, it may never finish running.



1. O(2N) denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2N) function is the recursive calculation of Fibonacci numbers.
2. Logarithms, logarithmic time algorithm - [**Binary search**](https://en.wikipedia.org/wiki/Binary_search) is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it.  It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set. This type of algorithm is described as **O(log N)**. The bigger the input, the smaller proportion of the actual input your program has to go through.[9]

**Orders of growth**

****

**Image 3: Big O Complexity. Taken from:** <https://www.hackerearth.com/practice/notes/sorting-and-searching-algorithms-time-complexities-cheat-sheet/> on 08/05/2020

## In-place algorithm

**In-place algorithm** is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) which transforms input using no auxiliary [data structure](https://en.wikipedia.org/wiki/Data_structure). However a small amount of extra storage space is allowed for auxiliary variables. The input is usually overwritten by the output as the algorithm executes. In-place algorithm updates input sequence only through replacement or swapping of elements. An algorithm which is not in-place is sometimes called not-in-place or out-of-place.

An in-place algorithm is an algorithm that does not need extra space and produces an output in the same memory that contains the data by transforming the input “in-place”.

An example of in-place algorithm.

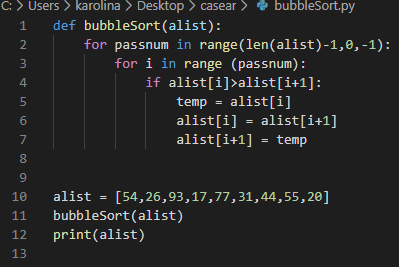
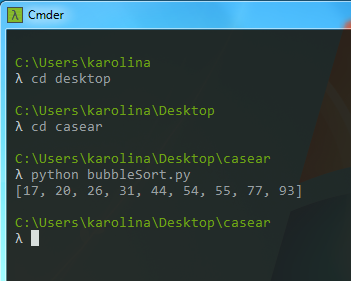


Image 4: Taken from the module Computational Thinking with Algorithms (week 10), GMIT

And the result of the code above is:



An example of in-place algorithms: [Bubble sort](http://www.geeksforgeeks.org/bubble-sort/), [Selection Sort](http://www.geeksforgeeks.org/selection-sort/), [Insertion Sort](http://www.geeksforgeeks.org/insertion-sort/), [Heapsort](https://www.geeksforgeeks.org/heap-sort/), QuickSort.

Not In-Place : [Merge Sort](http://www.geeksforgeeks.org/merge-sort/).

## Stable sorting

A stable sorting algorithm is said to be sorted if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

The stability of a sorting algorithm is concerned with **how the algorithm treats equal (or repeated) elements**. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don’t.

All sorting algorithms use a key to determine the ordering of the elements in the collection, called the sort key. [10]

Stable Sorting Algorithms: Insertion Sort, Merge Sort, Bubble Sort, Tim Sort, Counting Sort.

Unstable Sorting Algorithms: Heap Sort, Selection Sort, Shell Sort, Quick Sort.

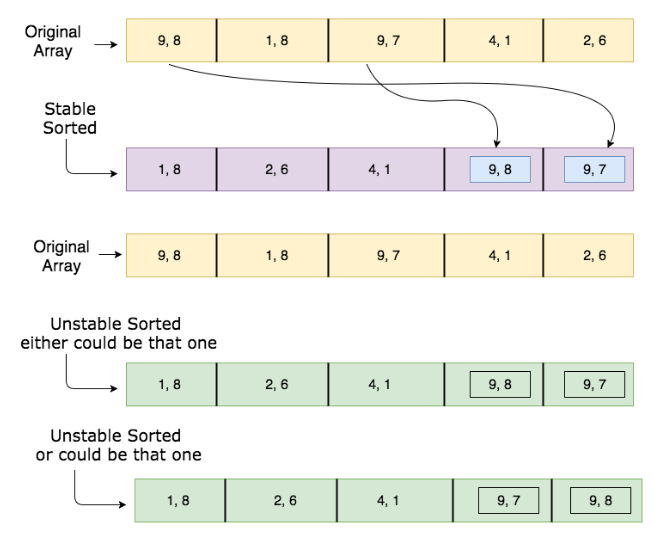


Image 5: Stable and unstable sorting. Taken from:

<https://stackoverflow.com/questions/1517793/what-is-stability-in-sorting-algorithms-and-why-is-it-important> on 11/05/2020

## Comparison-based and non-comparison-based sorting algorithms

A comparison sort is a type of [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that only reads the list elements through a single abstract comparison operation that determines which of two elements should occur first in the final sorted list.

A comparison sort algorithm sorts items by comparing values between each other. It can be applied to any sorting cases. The best complexity is O(n\*log(n)).

A non-comparison sort algorithm uses the internal character of the values to be sorted. It can only be applied to some particular cases, and requires particular values. The best complexity is probably better depending on cases, such as O(n).[11]

All sorting problem that can be sorted with non-comparison sort algorithm can be sorted with comparison sort algorithm, but not the other way around.

One of the most critical differences between these two sorting algorithms is speed. Non-comparison sorting is usually faster than comparison sorting because of not doing the comparison. The limit of speed for comparison-based sorting algorithm is O(NlogN) while for non-comparison based algorithms its O(n) i.e. linear time.[12]

Examples of comparison-based sorting: Quick Sort, Merge Sort, Heap Sort, Selection Sort, Bubble Sort, Insertion Sort, Block Sort, Shell Sort, Heap Sort,

Examples of non-comparison-based sorting is: Radix Sort, Counting Sort, Bucket Sort, Postman Sort, Flash Sort, Burst Sort.

1. Sorting Algorithms

## **Bubble Sort**

Bubble Sort, one of the simplest algorithms for sorting an array, consists of repeatedly exchanging pairs of adjacent array elements that are out of order until no such pair remains. The serial software implementation of bubble sort has a time complexity that is:

**Worst and Average Case Time Complexity:** O(n\*2). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

**Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

**Sorting In Place:**Yes

**Stable:** Yes

This type of sort is a slow-and-predictable sorting algorithm. Is often used to introduce the concept of a sorting algorithm.[13]

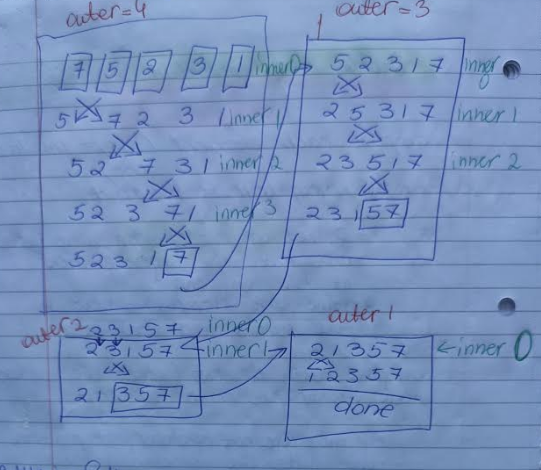


Image 6: Bubble Sort. Taken from my notes, module Computational Thinking with Algorithms (week 10), GMIT.

An implementation of Bubble sort is shown below:

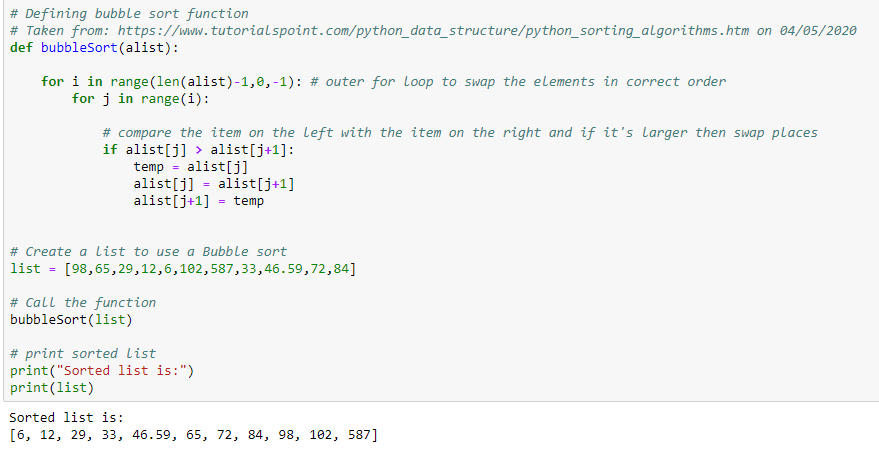


Image 7: An Implementation of Bubble Sort. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

## Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.[13]

**Time Complexity:** O(n2) as there are two nested loops.

**Auxiliary Space:** O(1)

It never makes more than O(n) swaps and can be useful when memory write is a costly operation.

[**Stability:**](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/)The default implementation is **not stable**.

**In Place :**Yes, it does not require extra space.

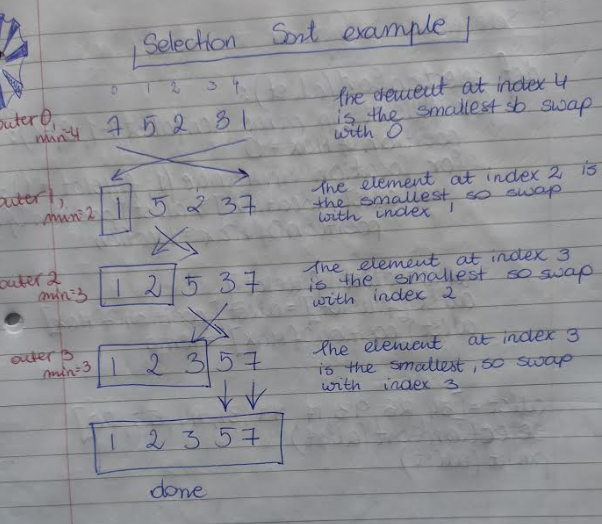


Image 8: Selection Sort. Taken from my notes, module Computational Thinking with Algorithms (week 10), GMIT.

An implementation of Selection sort is shown below:

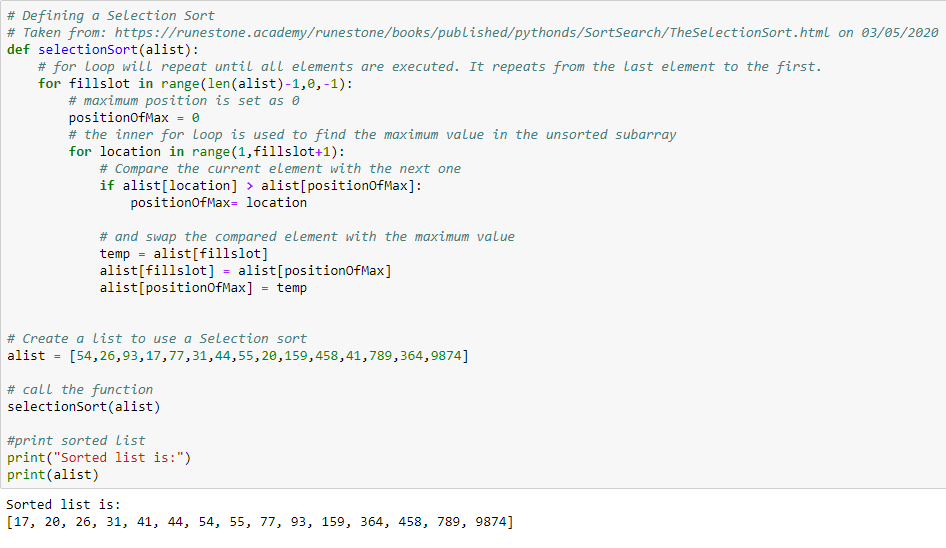


Image 9: An Implementation of Selection Sort. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

## Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands. Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time when elements are already sorted. Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.[13]

It involves finding the right place for a given element in a list. At the beginning the function compares the first two elements and sorts them by comparing them. Then the third element needs to find its proper position among the previous two sorted elements. This way more and more elements are added to the already sorted list by putting them in their proper position.[16]

**Time Complexity:** O(n\*2)

**Auxiliary Space:**O(1)

**Algorithmic Paradigm:** Incremental Approach

**Sorting In Place:** Yes

**Stable:** Yes

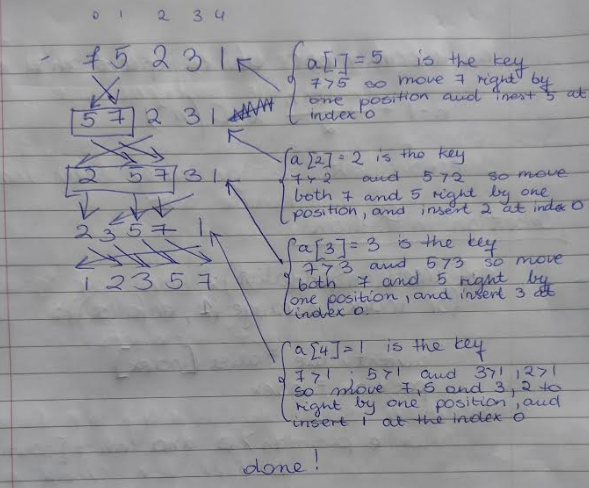


Image 10: Insertion Sort. Taken from my notes, module Computational Thinking with Algorithms (week 10), GMIT.

An implementation of Insertion sort is shown below:

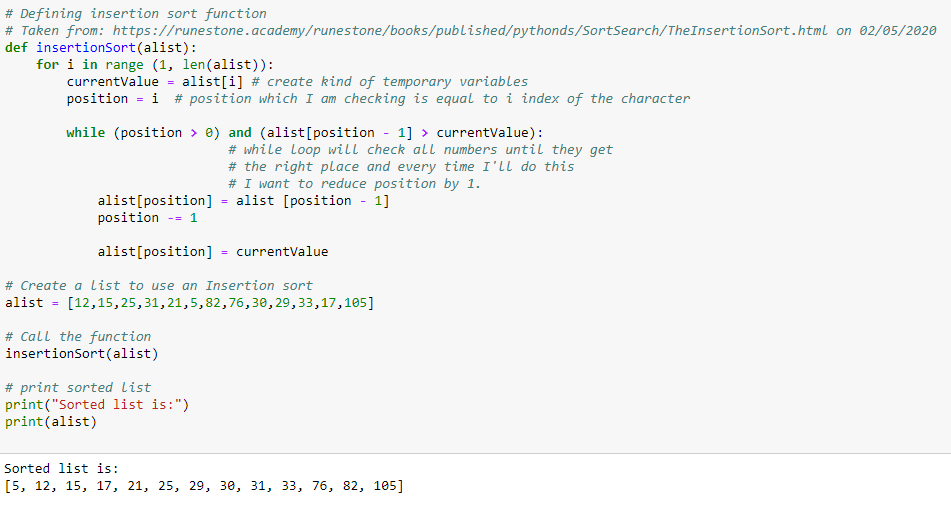


Image 11: An Implementation of Insertion Sort. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

## Merge Sort

Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The **merge() function** is used for merging two halves.[13]

It repeatedly breaks the array down into smaller chunks for an input array. After complete deconstruction arrays are recombined but not in their original order. The key to the merge sort algorithm is that when we begin the reconstruction we do not just put the elements back in their original order, instead we insert the elements in their appropriate sorted order. Merge is faster when it has to merge two sorted halves of an array than to sort the full array in place.

In the original input array in implementation will use recursion.

**Time Complexity:** Sorting arrays on different machines. Time complexity can be expressed as following recurrence relation. T(n) = 2T(n/2) + 

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is 

Time complexity of Merge Sort is  in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

Every element in the input approximately log in times giving n times log n.

**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer (recursive)

**Sorting In Place:** No in a typical implementation

**Stable:** Yes

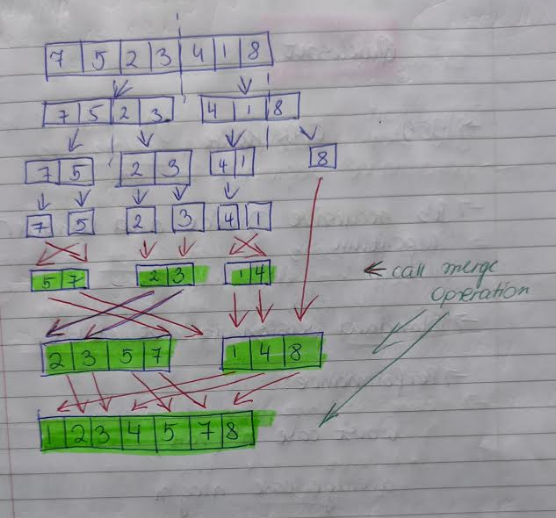


Image 12: Merge Sort. Taken from my notes, module Computational Thinking with Algorithms (week 11), GMIT.

An implementation of Merge sort is shown below:

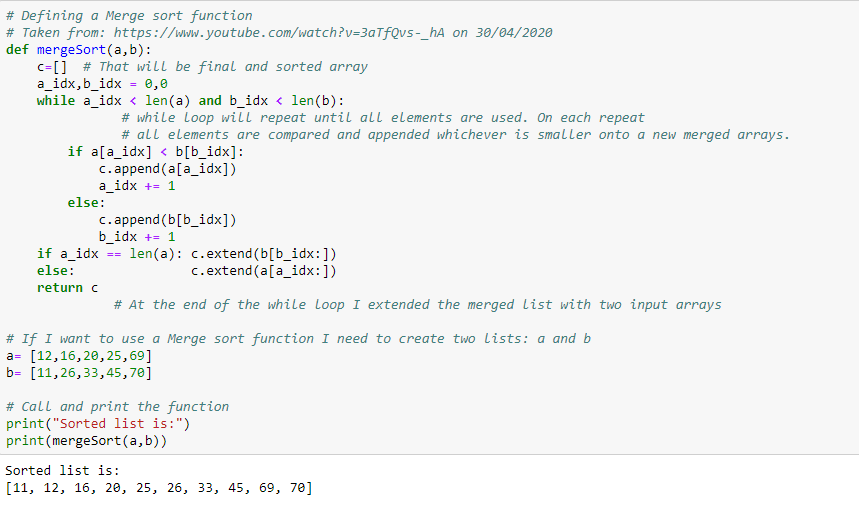


Image 13: An Implementation of Merge Sort. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

## Counting Sort

Counting Sort runs in O(n) time, amking it asymptotically faster than comparison-based sorting algorithms. It only works when the range of potential items in the input is known ahead of time.[14]

Counting sort works by iterating through the input, counting the number of times each item occurs, and using those counts to compute an item's index in the final, sorted array.

Counting sort allows to do something which seems impossible – sort a collection of items in linear time.

**Worst case time**: n + k

**Best case time**: n + k)

**Average case time**: n + k

**Space**: n + k

**Stable**: Yes



Image 14: Counting Sort. Taken from my notes, module Computational Thinking with Algorithms (week 11), GMIT.

An implementation of Counting sort is shown below:

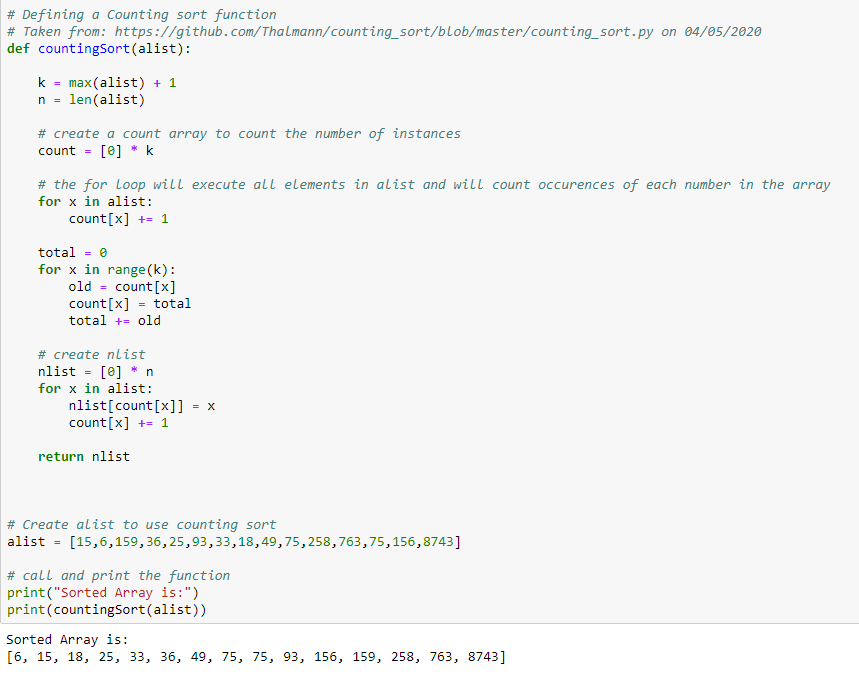


Image 15: An Implementation of Counting Sort. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

# Implementation and Benchmarking

This project contains a Python codes which will be used to benchmark five different sorting algorithms. I have implemented and benchmarked five sorting algorithms with random array. I have chosen the fallowing algorithms:

1. A simple comparison-based sort:

* Bubble Sort
* Selection Sort (my choice)
* Insertion Sort(my choice)

1. An efficient comparison-based sort:

* Merge Sort

1. A non-comparison sort:

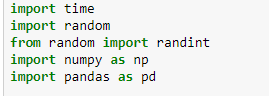
* Counting Sort

## Python code

The implementation of each sorting algorithms are shown above (in Sorting Algorithms section). The main idea of benchmarking is to figure out how fast the code executes and where the bottlenecks are. These actions lead to optimization. There are situations where you need your code to run faster because your business needs have changed, and you need to figure out what parts of your code are slowing it down.[15] Benchmarking involves running sorting algorithms for random numbers with different input sizes and measuring the time needed to run.

The running time was measured ten times and I have used numpy.mean to return the average of an array of ten run times for each algorithm and input size. I have used a variety of different input sizes, e.g. size = 100,250,500,750,1000,1250,2500,3750,5000,6250,7500,8750 and 10000. I used different sizes to test the effect of the input size on the running time of each algorithm.

At the beginning of Benchmarking sorting algorithms (coding) I imported all libraries important for the project.



* time is used for counting each algorithm
* random is used to generate random numbers(arrays)
* numpy to generate lists of random arrays
* pandas to create DataFrame for final results.

### Random arrays

I used *random\_array* to return an array of size n. The random module is imported with the randint function to generate random integers in the range, here between 0 and 99.

Those integers were appended to an array. Those arrays were used to generate integers with different input sizes “size” (in my project). I also used them to run time for each algorithm in milliseconds.

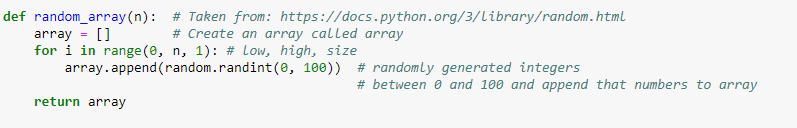


Image 16: Random array.. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

I also created *create\_array* to generate randomized arrays to implement and benchmark each algorithm in seconds. It is more to save time when it is executed and see how it all work.

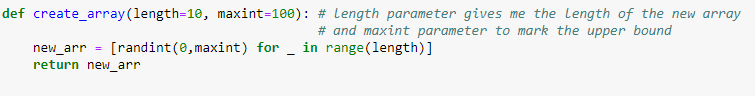
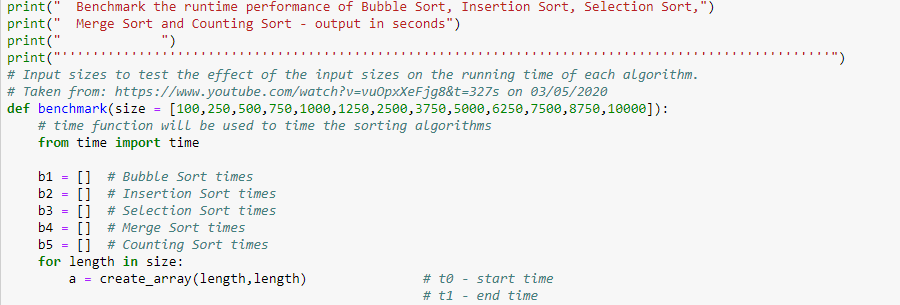


Image 17: An Create array. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

This helped me more to understand benchmarking. Results are below.



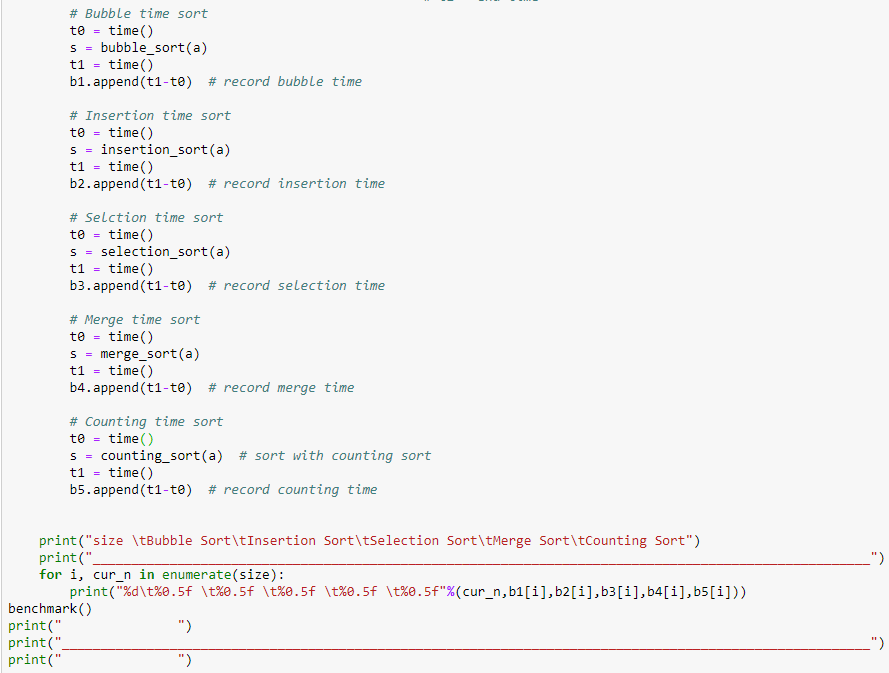


Image 18: The benchmark performance in seconds. Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

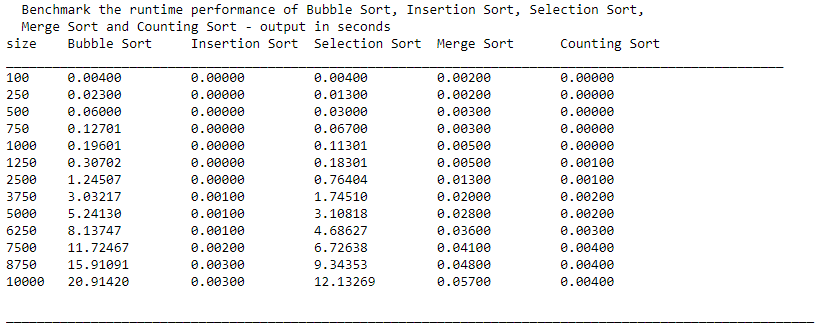


Image 19 : The results of benchmarking in seconds, Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

### Timing the algorithm and benchmark

To measure the running time of all sorting algorithms I used time module to record the start time (t0) and the end time (t1). This gives the results in seconds.

*Benchmark\_a* - used all sorting algorithms and for each input size run them 10 times and measured the time taken for it to sort randomly generated arrays.

It was used in for loop to execute an array 10 times for each algorithm.

I called time.time ()twice, before sort algorithms were started to measure the start time and when they ended to measure the end time.. Arrays generated from *random\_array* were appended into lists which were created to store results.

### Average time

*avg\_time\_bubble\_sort* – (here just for bubble sort but it is done for all algorithms) – returns average running time of the lists which store results from timing. It tastes input sizes and sorting algorithms as inputs. I used numpy.mean to return the average figures.

To get results in milliseconds I multiplied by 1000.

Those results were appended again into different lists which I created before for each sorting algorithm.

### DataFrame

The results are recorded and stored in a data table. The data set is organized into columns and rows. Rows correspond to sorting algorithms and columns correspond to the size of the array. The data set type is Pandas DataFrame and is allocated to a variable named df\_average.

The output is formatted into 3 decimal places and transposed. I used .round(3).transpose to get it.

Names given to the table:

1. Size = size of the array
2. Bubble Sort
3. Insertion Sort
4. Selection Sort
5. Merge Sort
6. And Counting Sort

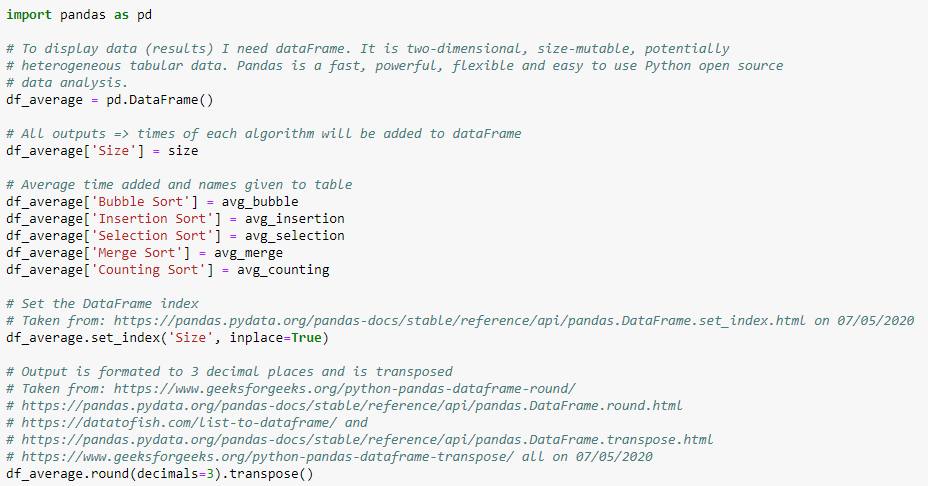


Image 19: DataFrame Implemantation,

Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

### Plot Graph

At the very end I generated a plot to show all results in graph. To do this I imported matplotlib.pyplot to create a line plot of the results. I added title, xlabel with fontsize = 13, xticks, ylabel with fontsize =13 and yticks. I called grid which exposes the graph as very readable and also called gca().legend() to show which line belongs to which sort.

### Final results

As I mentioned before I decided to choose 5 sorting algorithms, which are:

* Bubble Sort,
* Insertion Sort,
* Selection Sort,
* Merge Sort,
* And Counting Sort.

Below are shown results of my research. This contains average times of ten runs of each sorting algorithm. It is measured for each input size (size= 100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750 and 1000) and results are in milliseconds.

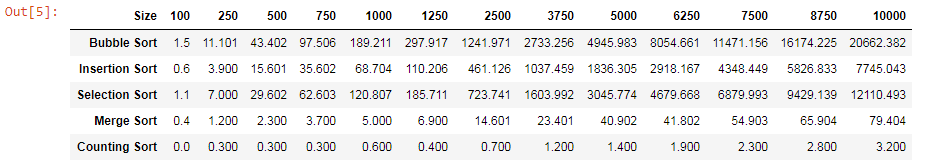


Image 20 : The results of benchmarking in milliseconds, Taken from my Jupyter notebook – CTA – Project,

<https://github.com/karolinaszafranbelzowska/CTA-project>

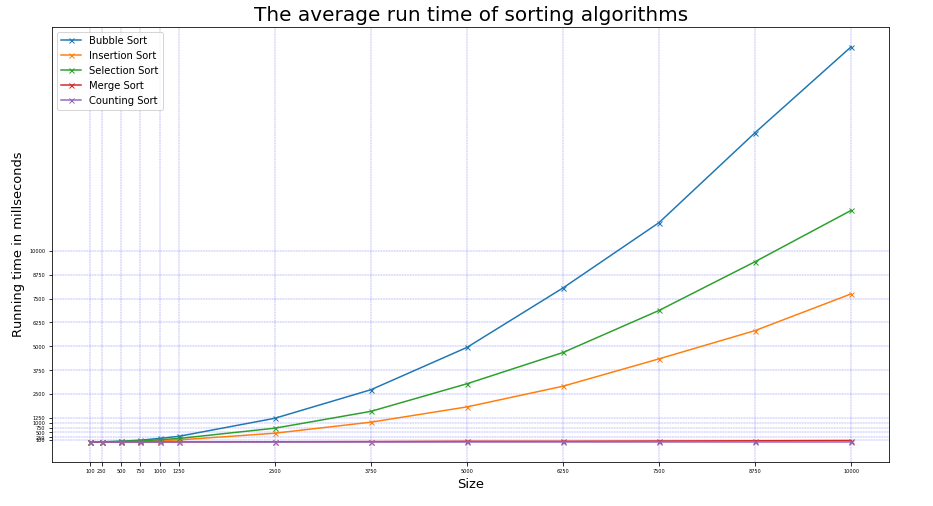


Image 21 : Plot graph – the average running time for each of five sorting algorithms for input sizes,

Taken from my Jupyter notebook – CTA – Project, <https://github.com/karolinaszafranbelzowska/CTA-project>

Looking at the results I can say that Bubble Sort had the longest running time of all algorithms.

For the very large input sizes it needs almost twice more time than Selection Sort. Bubble Sort is suitable for small input sizes and in this case it needed much more time than others, and I was expecting this. It is extremely slow and probably because of this algorithm it took around 40 minutes to execute outputs.

Selection Sort had the second longest time in my research. It usually gives better performance than Bubble Sort and so it did in here. Like Bubble it is suitable for small data, input sizes.

Insertion Sort as the algorithm is a simple comparison-based algorithm and it is suitable for small lists which are close to be sorted. Very inefficient for large random arrays and like Bubble and Selection Sorts needs more time to run input instances.

Those three sort algorithms I would not recommend to use in benchmarking as it needs a lot of time to execute large numbers. Bubble, Selection and Insertion Sorts suffer from exponential time complexity in the worst cases, these algorithms begin to slow down substantially as the input array size increases.

Merge had the second best time performance. The best, worst and average cases are very similar (n logn). Merge sort implementation shows that is significantly faster for larger input array sizes than Bubble, Selection and Insertion Sort. For the smaller sizes I see a much closer competition. Average time in Merge Sort is very close to average time in Insertion Sort at an input array size of 100. The larger the size is the greater the difference between each algorithm is. It is more efficient for large input instances and I could see that in my study.

Counting Sort had the best average time of all algorithms and I was expecting this. Counting is suitable for large input instances and it allows to execute something which seems impossible.

# References

[1] <https://whatis.techtarget.com/definition/sorting-algorithm>

[2] <https://www.studytonight.com/data-structures/introduction-to-sorting>

[3] <https://en.wikipedia.org/wiki/Sorting_algorithm>

[4] <https://en.wikipedia.org/wiki/Best,_worst_and_average_case>

[5] <https://www.geeksforgeeks.org/g-fact-86/>

[6] <https://en.wikipedia.org/wiki/Big_O_notation>

[7] <https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/>

[8] <https://medium.com/better-programming/a-gentle-explanation-of-logarithmic-time-complexity-79842728a702>

[9] <http://mca.kiet.edu/webguru/unit-1ds.html>

[10] <https://www.baeldung.com/cs/stable-sorting-algorithms>

[11] <https://stackoverflow.com/questions/25788781/definition-of-non-comparison-sort>

[12] <https://en.wikipedia.org/wiki/Comparison_sort>

[13] <https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/>

[14] [https:/www.interviewcake.com/concept/java/counting-sort](https://www.interviewcake.com/concept/java/counting-sort)

[15] <https://www.blog.pythonlibrary.org/2016/05/24/python-101-an-intro-to-benchmarking-your-code/>

[16] <https://www.tutorialspoint.com/python_data_structure/python_sorting_algorithms.htm>

**Other Web sides:**

[17] <https://numpy.org>

[18] <https://www.w3schools.com>

[19] <https://pandas.pydata.org>

[20] <https://www.geeksforgeeks.org>

[21] <https://datatofish.com>

[22] <https://matplotlib.org>

[23] <https://docs.python.org>

[24] <https://stackoverflow.com>

[25] <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html>

<https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheSelectionSort.html>

<https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html>

<https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html>

**Videos**

[1] <https://www.youtube.com/watch?v=AthG28-_RuM&t=445s>

[2] <https://www.youtube.com/watch?v=AgtzMtrzhzs>

[3] <https://www.youtube.com/watch?v=JxTghISBmI8>

[4] <https://www.youtube.com/watch?v=3aTfQvs-_hA>

**Github Repositories**

<https://github.com/jennifer-ryan/benchmarking-sorting-algorithms>

<https://github.com/andkoc001/CTA_sorting_project>

<https://github.com/johndunne2019/CTA-Project-Benchmarking-Sorting-Algorithms>

<https://gist.github.com/haandol/a5df913cfd278820e43e>

<https://github.com/Thalmann/counting_sort/blob/master/counting_sort.py>