Course: Computational Thinking and Algorithms

Student Name: Niamh O’Leary

Student ID: G00376339

Project: Benchmarking Sorting Algorithms

1. **Introduction**

“Numerous computations and tasks become simple by properly sorting information in advance. The search for efficient sorting algorithms dominated the early days of computing. Indeed, much of the early research in algorithms focused on sorting collections of data that were too large for the computers of the day to store in memory. Because today’s computers are so much more powerful than the ones of 50 years ago, the size of the data sets being processed is now on the order of terabytes of information. Although you may not be called on to sort such huge data sets, you will likely need to sort large numbers of items. In this chapter, we cover the most important sorting algorithms and present results from our benchmarks to help you select the best sorting algorithm to use in each situation.”[[1]](#endnote-1)

* 1. **Sorting**

Essentially sorting arranges data in a sequence which makes searching easier, for example in ascending or descending order. As humans realised the importance of searching quickly, the need for efficient sorting became more sought after.[[2]](#endnote-2) Therefore sorting was the most fundamental algorithmic problem that was faced in the early days of computing. The development of sorting algorithim helped specify the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

* Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names , so that the names can be searched easily.
* Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.[[3]](#endnote-3)

Since the explosion of modern technologies, computer algorithms have expanded and can be found in nearly every aspect of life, hence he need to find the most efficient method of sorting.

All sorting algorithms can be measured as to their complexity and performance.

* 1. **Complexity**

The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. There are two main complexity measures of the efficiency of an algorithm:

**Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. In layman’s terms, We can say time complexity is sum of number of times each statements gets executed.

**Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. When we say “this algorithm takes constant extra space,” because the amount of extra memory needed doesn’t vary with the number of items processed.[[4]](#endnote-4)

When identifying the complexity of an algorithm, it is important to identify the most expensice computation within an algorithm to determine its classification. The overall performance of the algorithm must be classified as quadratic. Figure 1., shows the typically, algorithmis complexity of a number of polynominal and exponential algrothims. The most popular metric for calculating time complexity is Big O notation. Time complexity is measured in terms of the number of operations an algorithm uses.



Figure 1.

It is important to consider the characteristics of the data type in the input as well as the size of the input. An algorithm which takes an array as an input and returns the first item in the array will run in constant time (denoted by O(1)), regardless of the size of the input, it will always take the same amount of time to run. An algorithm which takes an input array and loops through each item to find the sum of the items in the array will have a run time that varies in direct proportion to the size of the input data. This is denoted by O(n). Finally, an algorithm that uses a nested loop to determine if a dataset contains duplicates can have complexity that varies in proportion to the square on the size of the input. This is denoted by O(n2).

* 1. Performance

While complexity can be considered the theoretical measure of algorithim efficiency, performance can be seen as the practical measure. Performance of an algorithm is measured by implementing the algorithm. The amount of time, disk space and memory consumed when a program is run. Performance can be affected by the computer specification, the compiler used to run the code and the efficiency of the code itself.

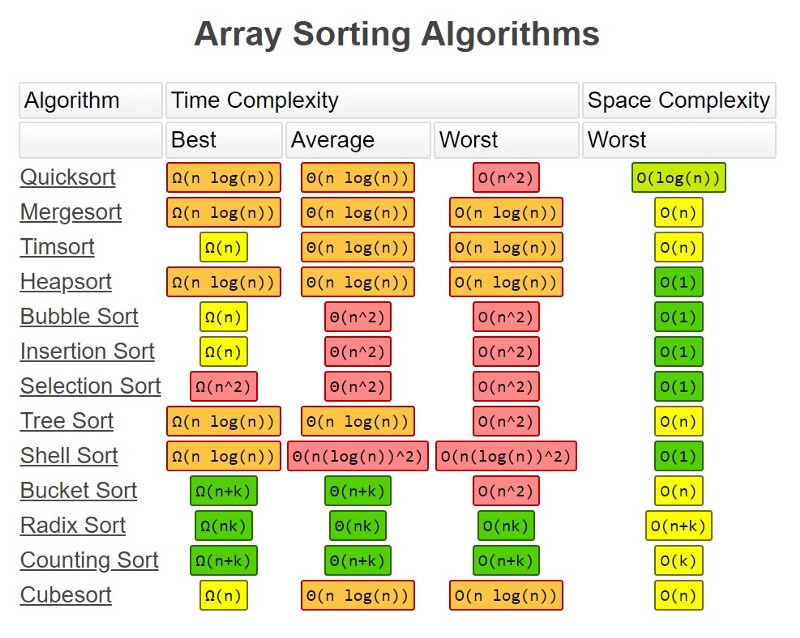
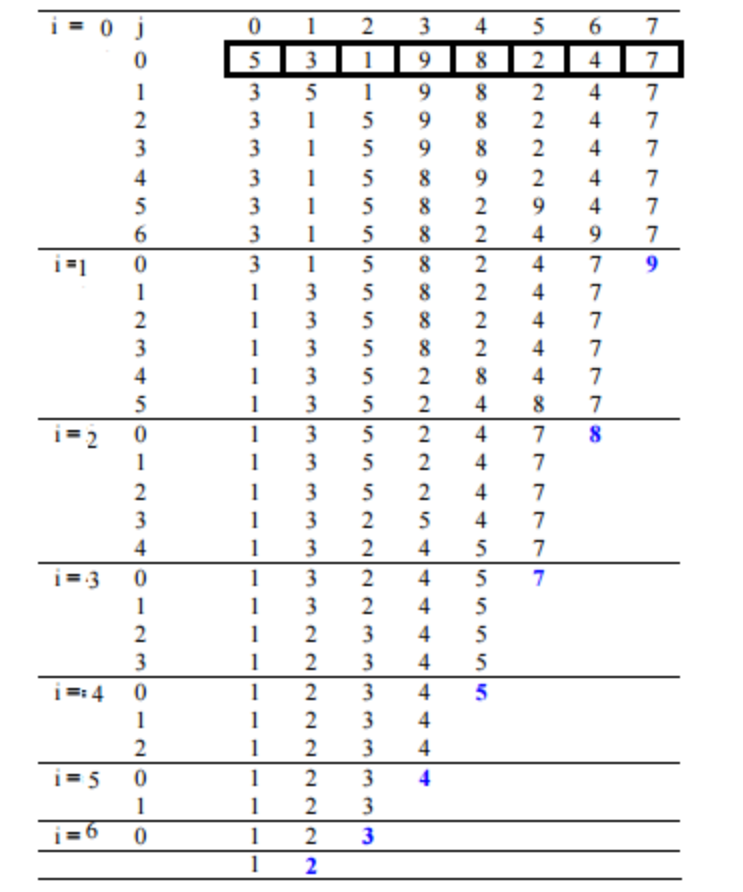


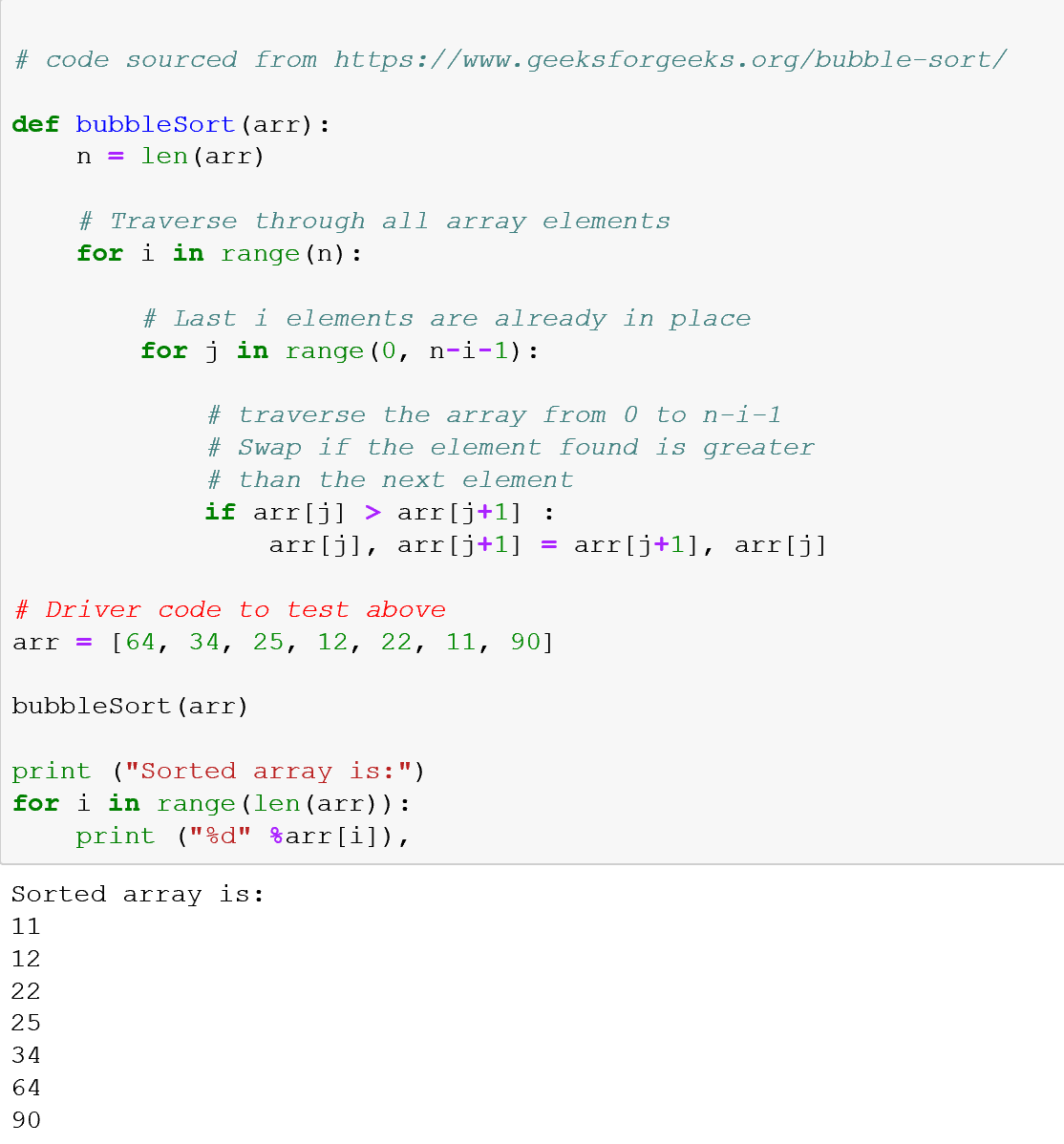
Figure 2.

Figure 2., shows a summary if the performance of different algorithms. Five of these alogrithms will be investigated further in this project and a detailed analysis of the code needed to run, the complexity and performance will be discussed on detail.

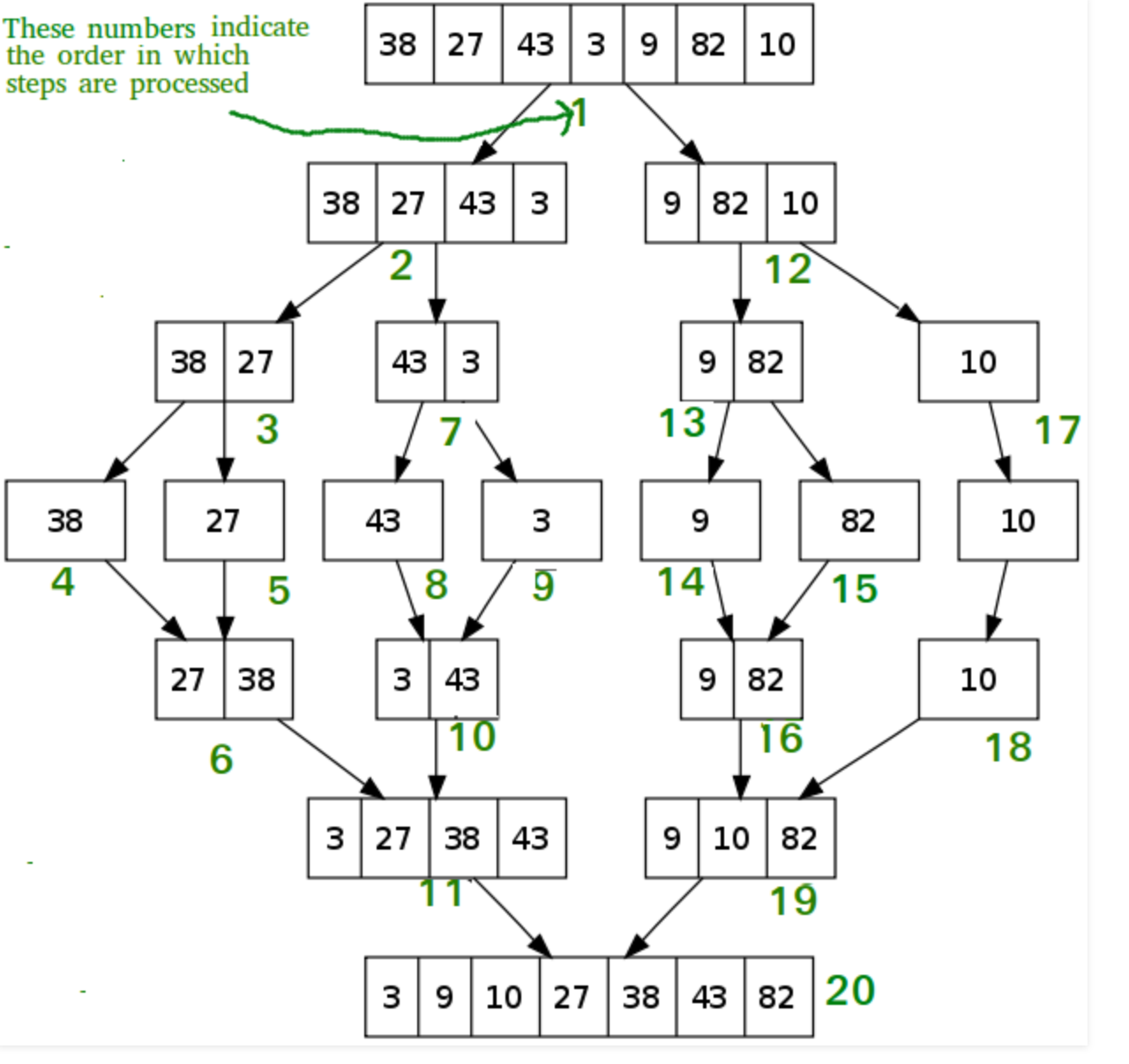
* 1. In-place sorting
  2. Stable sorting
  3. Comparable elements and comparator functions
  4. Comparison based or non-comparision based sortinggithub
  5. github

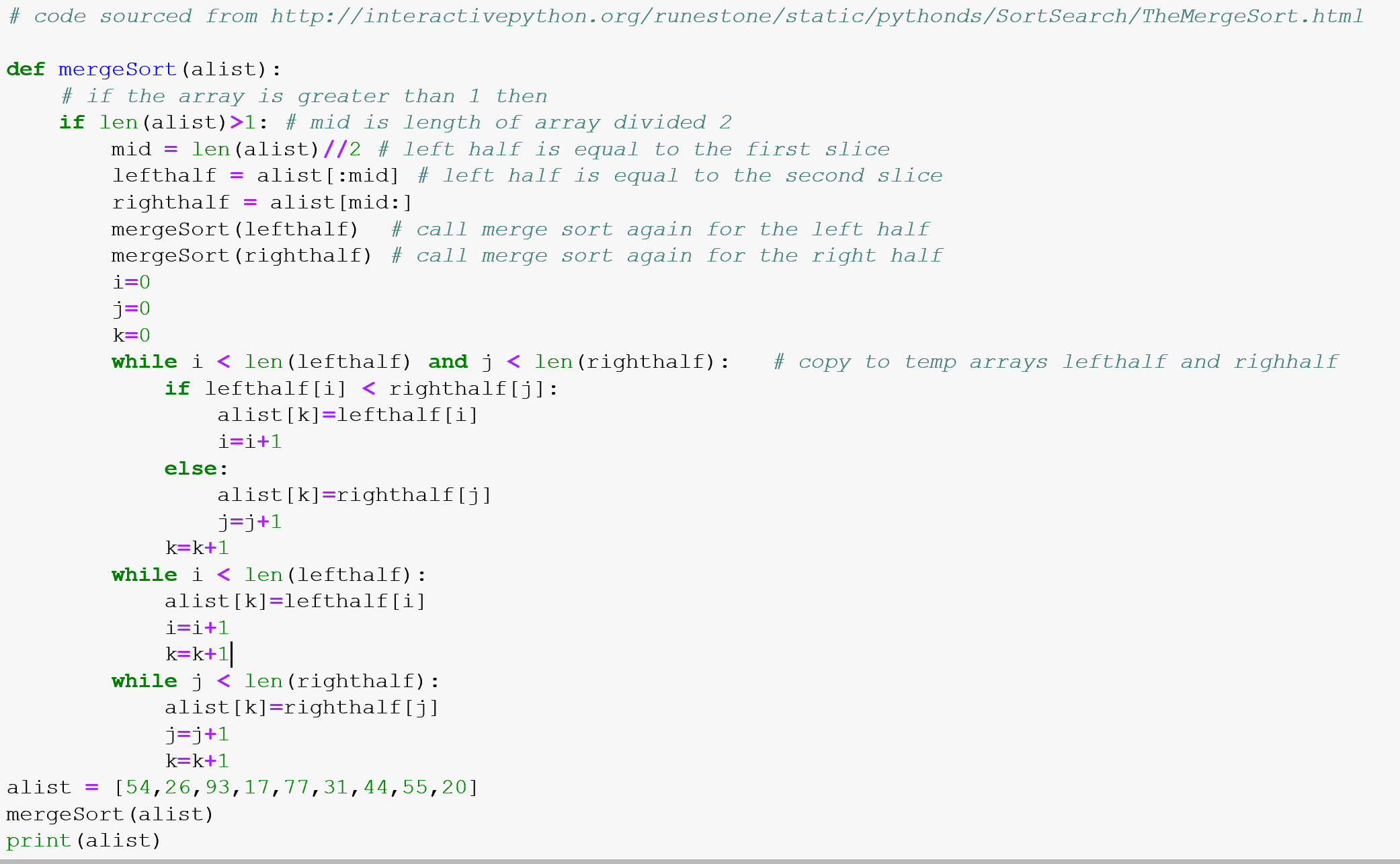
1. Sorting Algorithms
   1. Bubble Sort (a simple comparison based algorithm)





* + 1. Analysis of Bubble Sort
* **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)
* **Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.
* **Sorting In Place:**Yes
* **Stable:** Yes
  1. Merge Sort (an efficient comparsion based algorthim)





Sorted Array : [17, 20, 26, 31, 44, 54, 55, 77, 93]

* + 1. Analysis of Merge Sort

**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) +   
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.

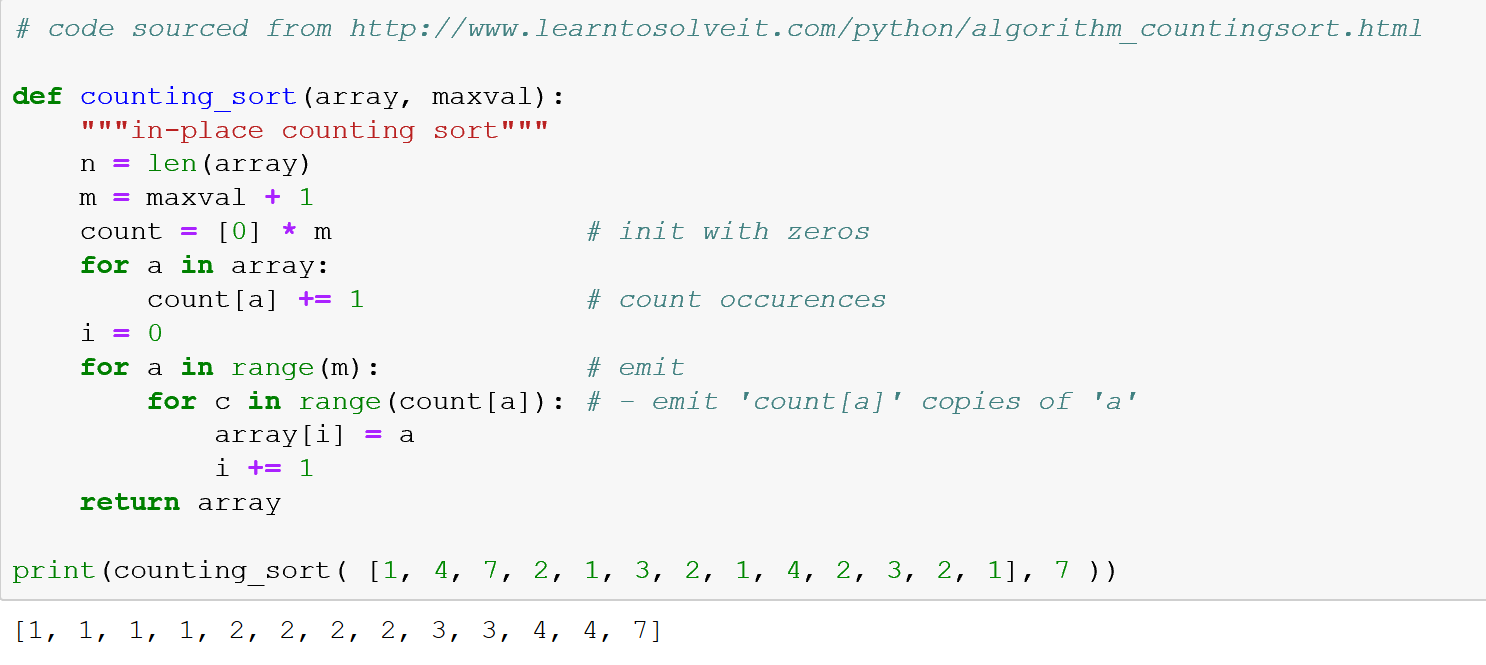
**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

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

**Stable:** Yes

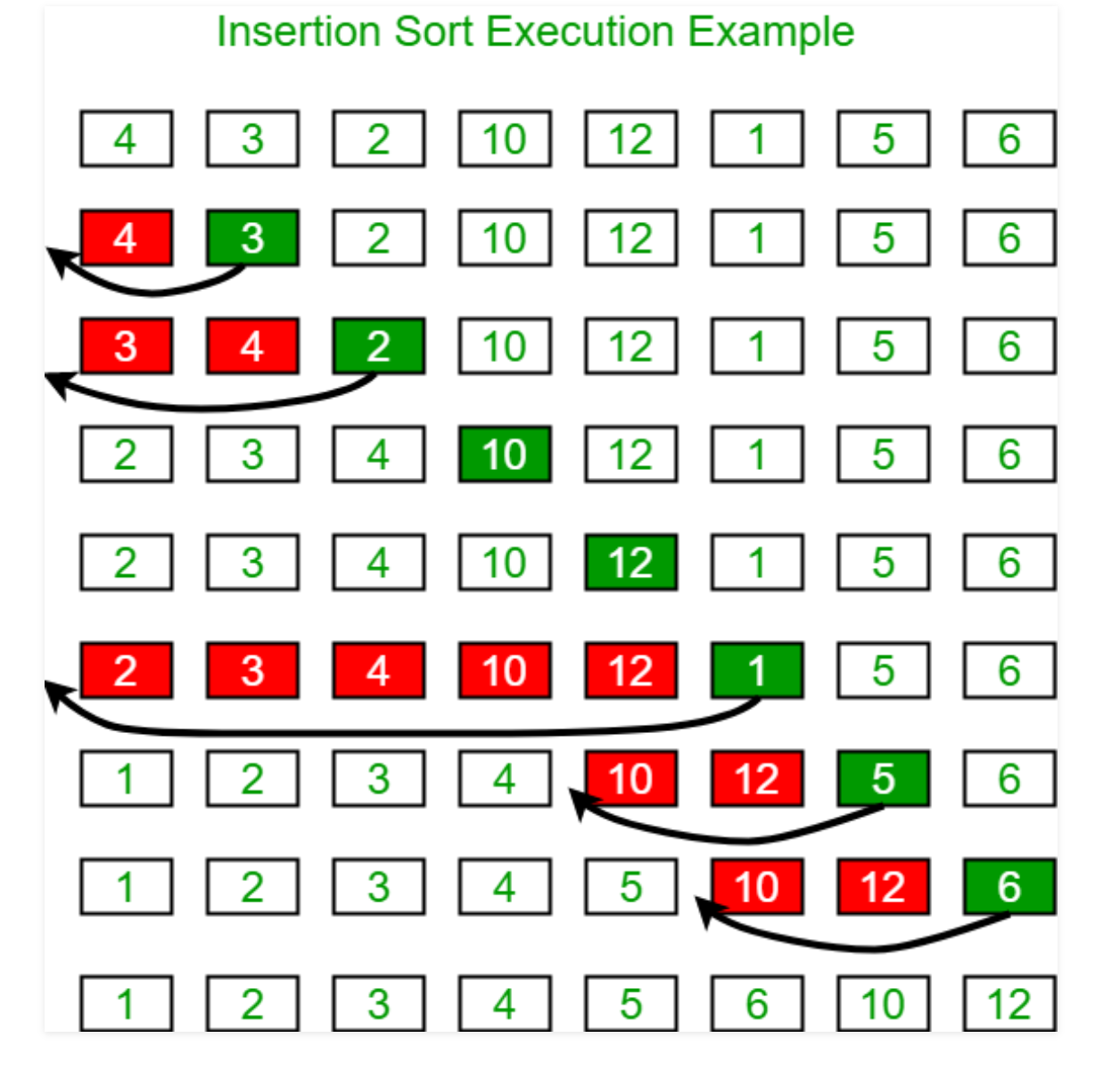
* 1. Counting Sort (a non-comparsionbased algorithm)



* + 1. Analysis of Counting Sort

**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)

* 1. Insertion Sort



Sorted array is:

0

1

2

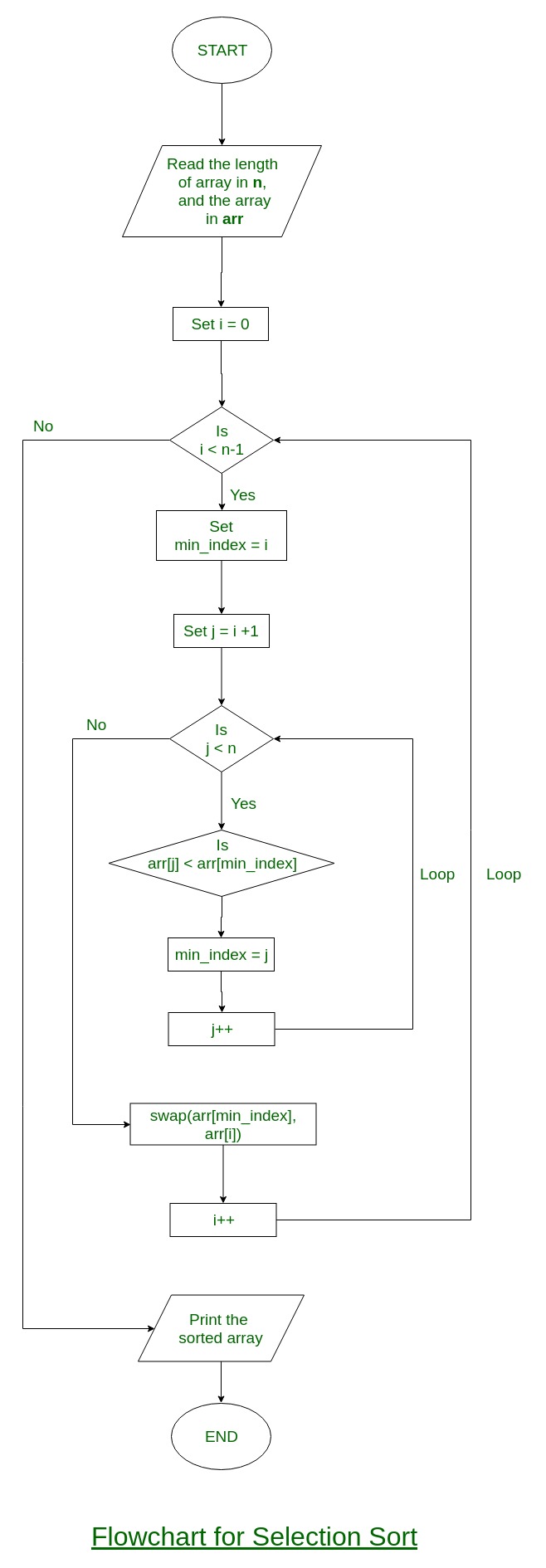
2

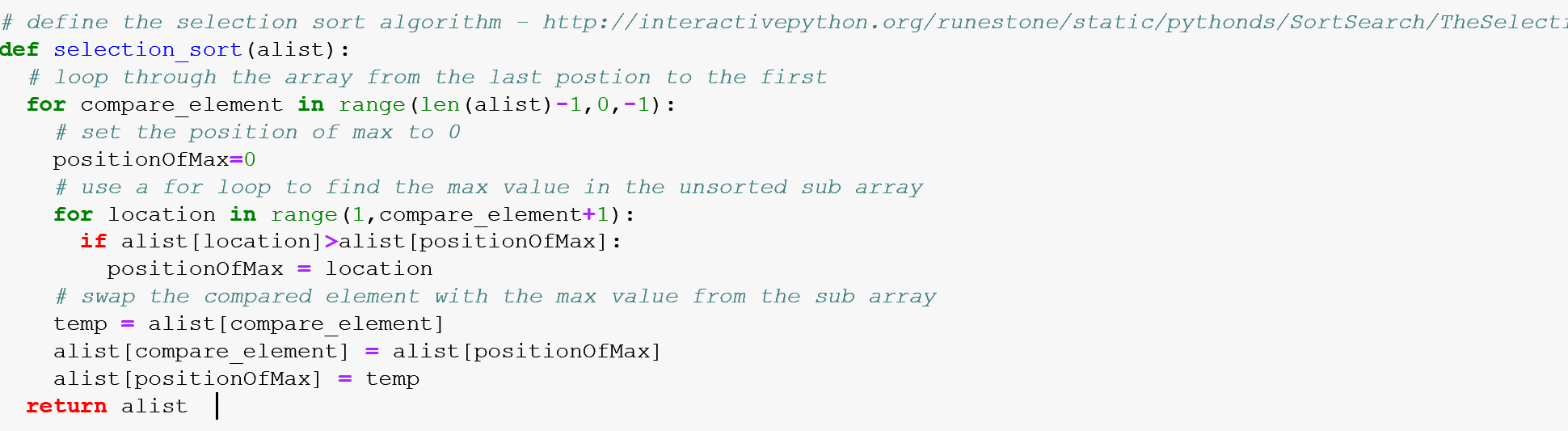
4

5

8

* + 1. Analysis of Insertion Sort
* **Time Complexity:** O(n\*2)
* **Auxiliary Space:**O(1)
* **Boundary Cases**: 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.
* **Algorithmic Paradigm:** Incremental Approach
* **Sorting In Place:** Yes
* **Stable:** Yes
* **Online:** Yes
* **Uses:** 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.
  1. Selection Sort



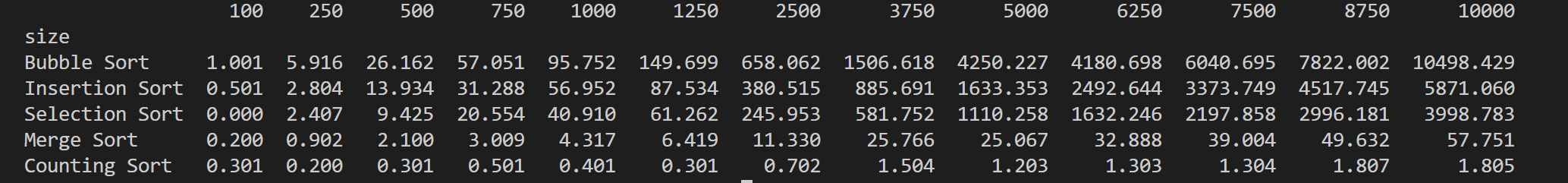


* + 1. Analysis of Selection Sort

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

**Auxiliary Space:** O(1)  
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

1. Implementation and Benchmarking



1. Conclusion

1. Heineman et al., (2015), Algorithms in a Nutshell [↑](#endnote-ref-1)
2. www.studytonight.com/data-structures/introduction -to-sorting [↑](#endnote-ref-2)
3. <https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm> [↑](#endnote-ref-3)
4. <https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structure-and-sorting-algorithms-588a57edf495>

   References:

   Heineman et al., (2015), Algorithms in a Nutshell

   www.studytonight.com/data-structures/introduction -to-sorting

   <https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm>

   <https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structure-and-sorting-algorithms-588a57edf495>

   <https://www.geeksforgeeks.org/sorting-algorithms/>

   <https://github.com/shkyler/gmit-cta-project>

   <https://github.com/RitRa>

   <https://github.com/G00364778> [↑](#endnote-ref-4)