Course: Computational Thinking and Algorithms

Student Name: Niamh O’Leary

Student ID: G00376339

Project: Benchmarking Sorting Algorithms

1. Introduction

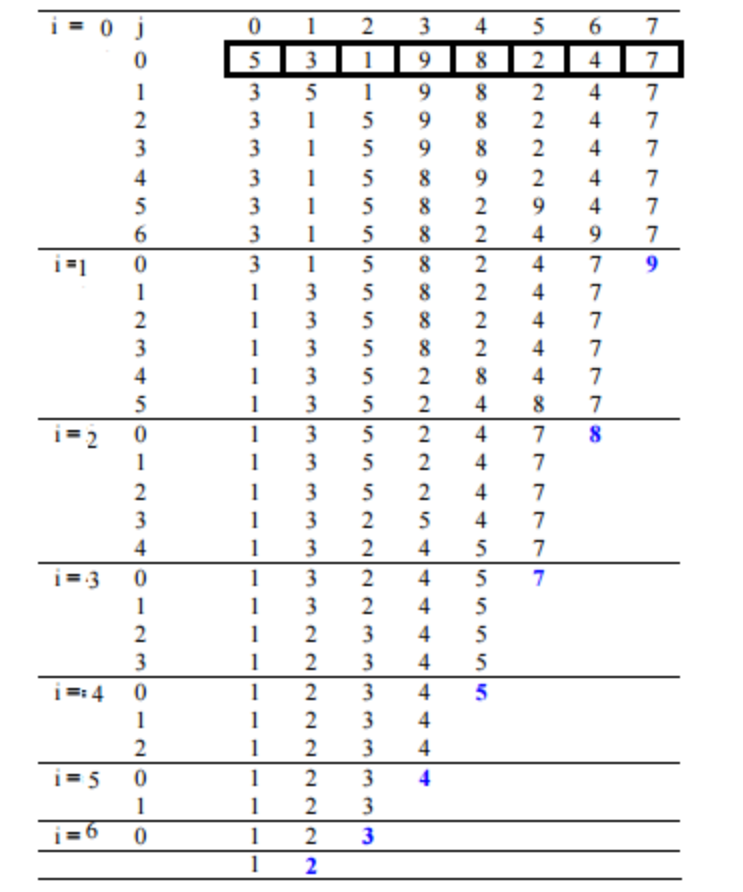
“Numerous computations and tasks become simple by properly sorting information in advance. The search for efficient sorting algorithms dominated the early days of computing. Indeed, much of the early research in algorithms focused on sorting collections of data that were too large for the computers of 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)

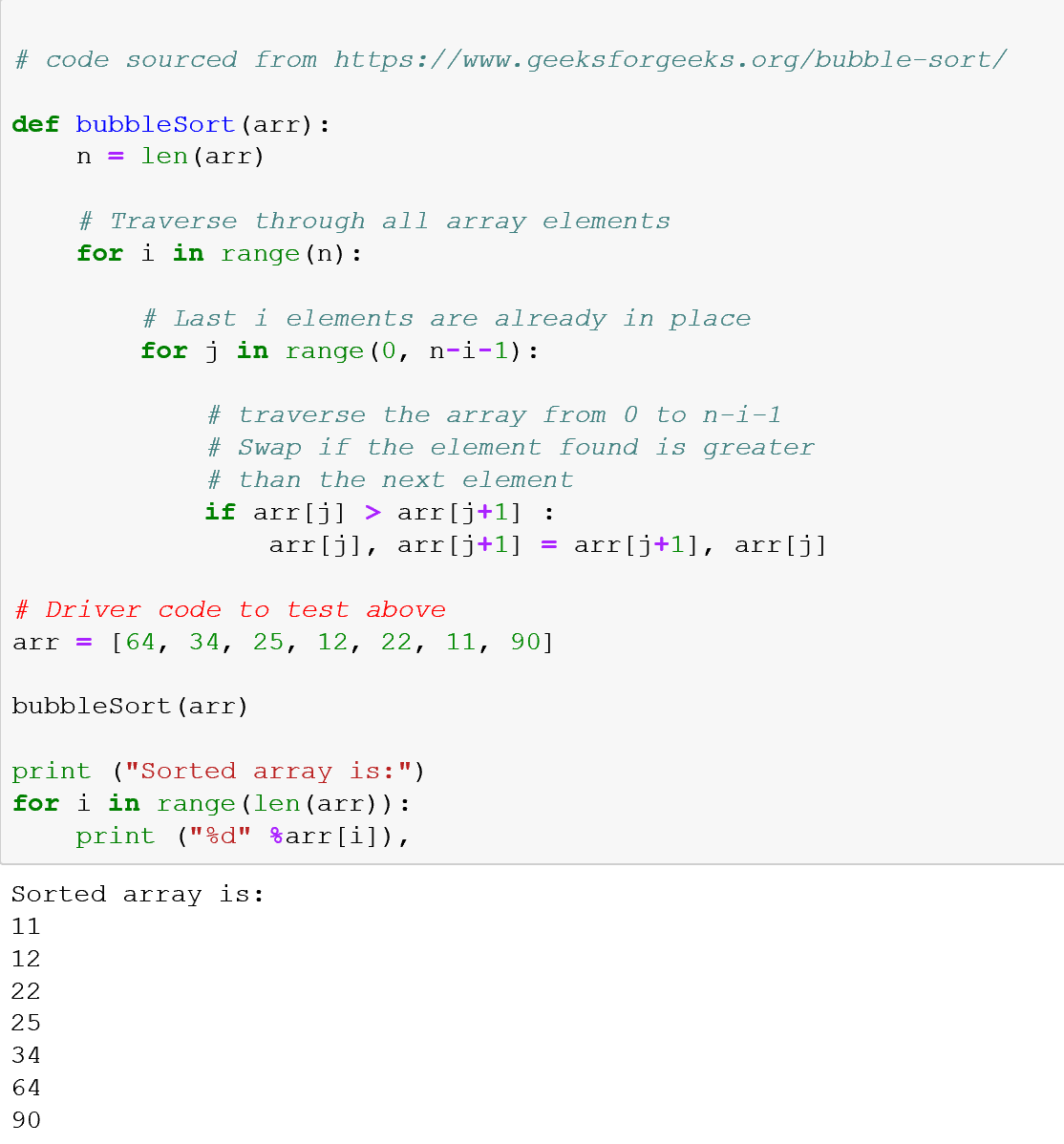
* 1. Sorting

Sorting is the most fundamental algorithmic problem that was faced in the early days of computing.

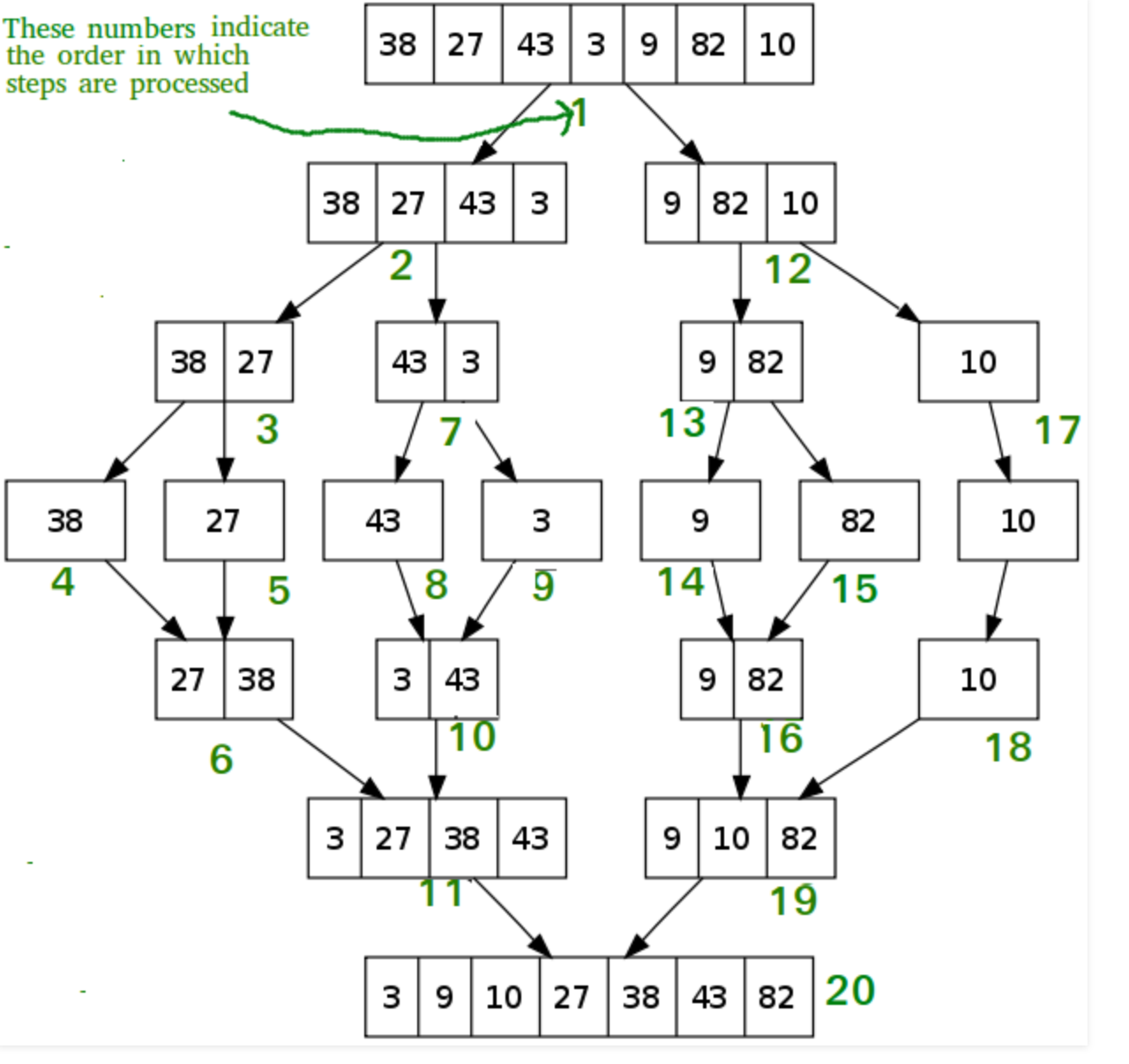
* 1. Complexity
  2. Performance
  3. In-place sorting
  4. Stable sorting
  5. Comparable elements and comparator functions
  6. Comparison based or non-comparision based sorting

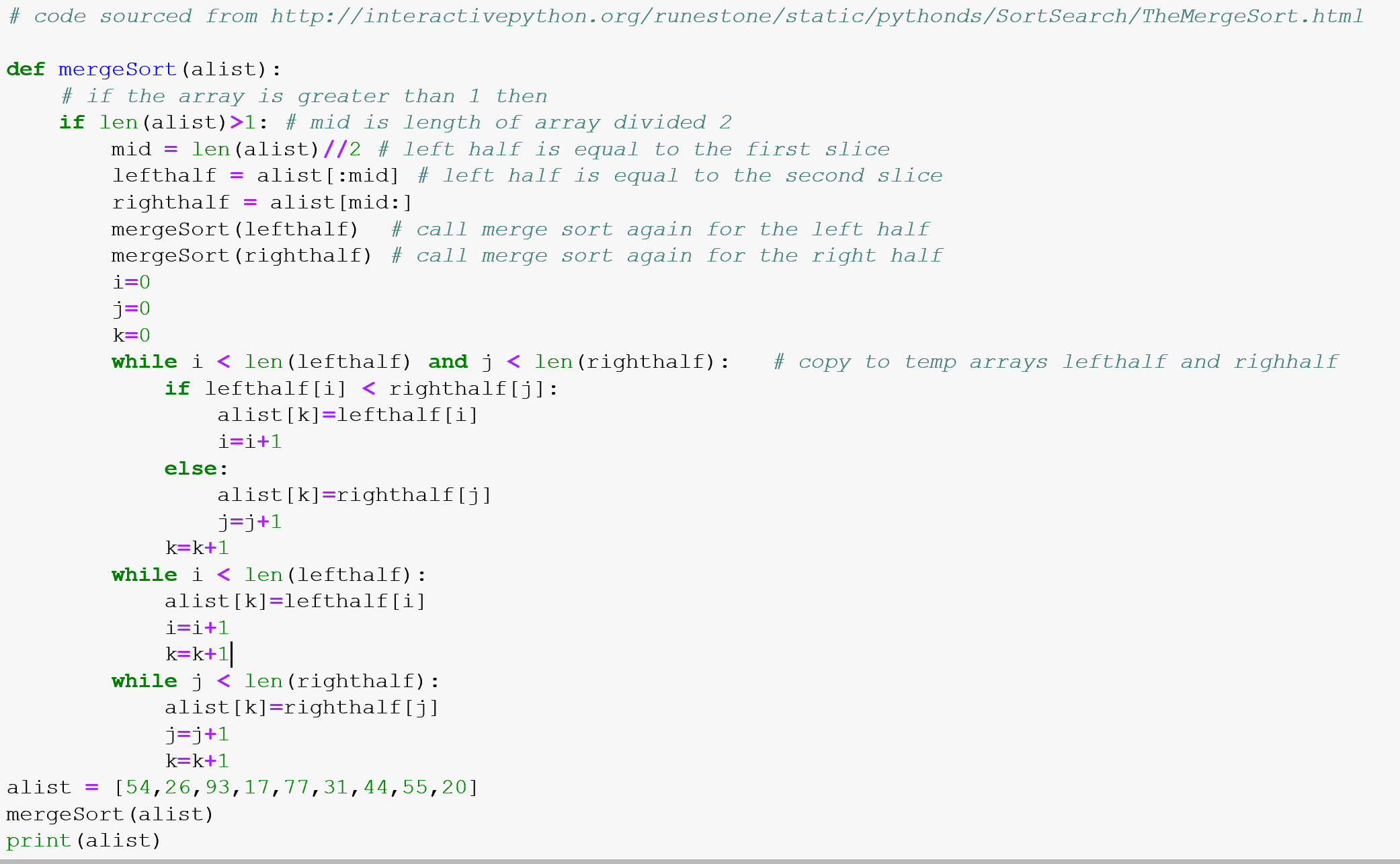
1. Sorting Algorithms
   1. Bubble Sort (a simple comparison based algorithm)





* + 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.

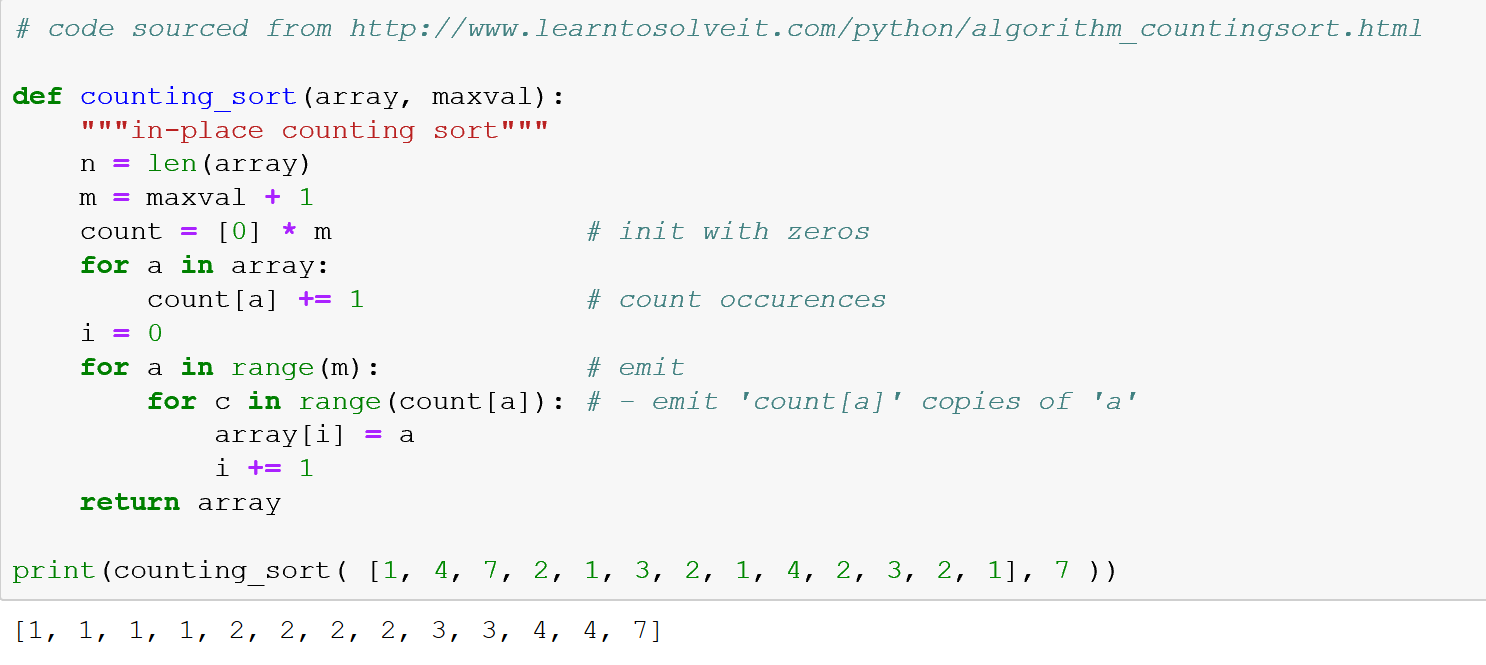
**Auxiliary Space:** O(n)

**Algorithmic Paradigm:**Divide and Conquer

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

**Stable:** Yes

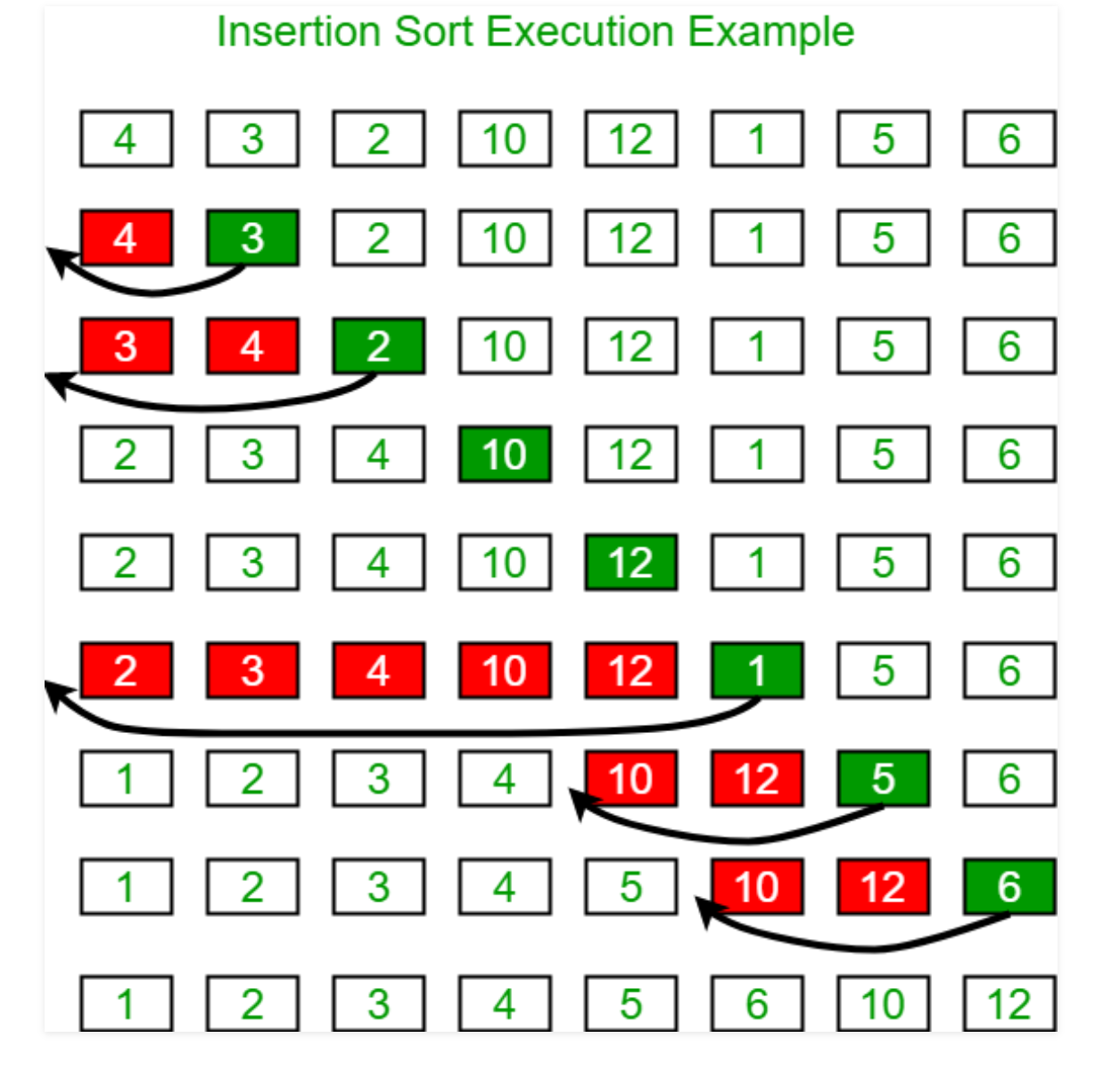
* 1. Counting Sort (a non-comparsionbased algorithm)

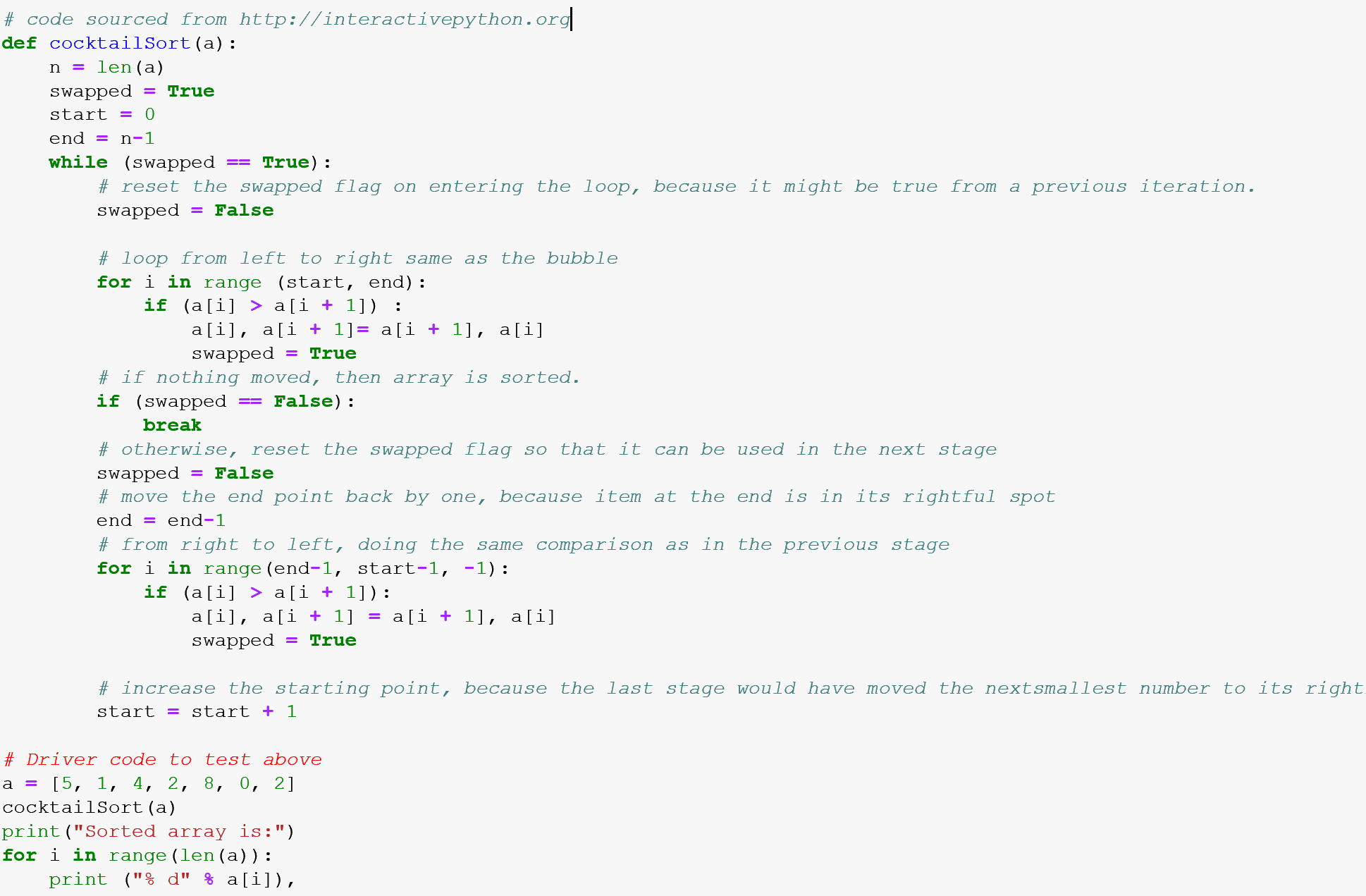


* + 1. Analysis of Counting Sort

**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

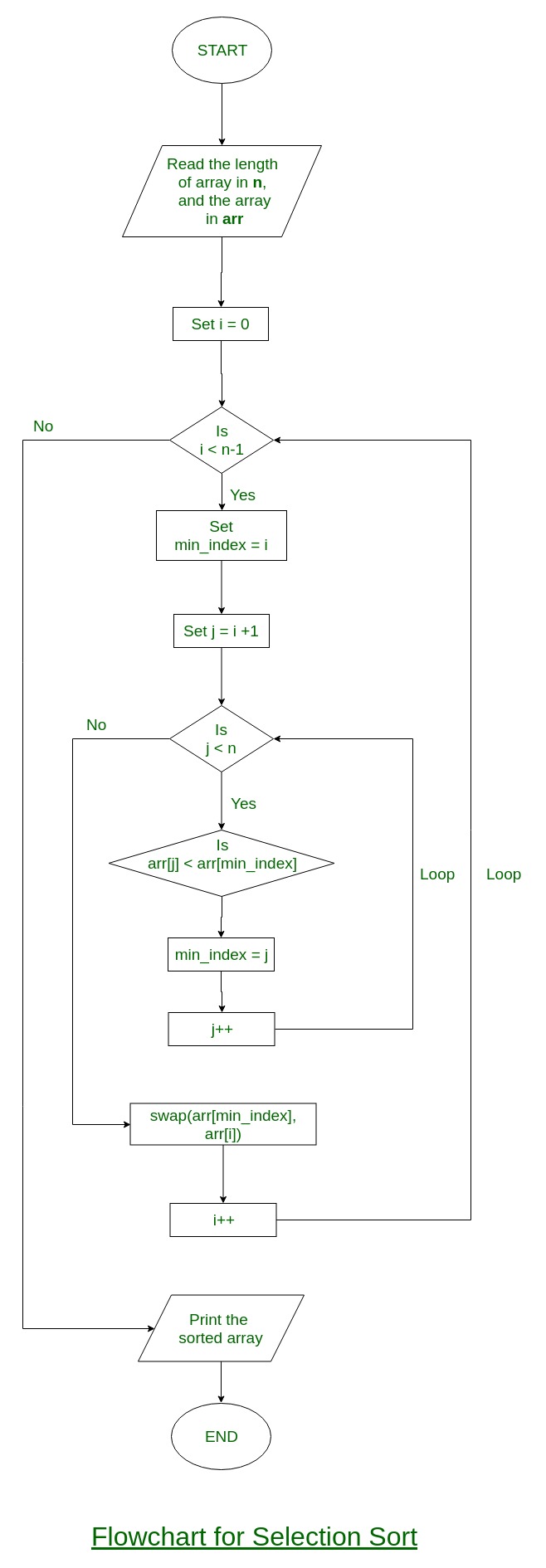
2

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
2. Conclusion

References.

1. Heineman, G., Pollice. G. & Selkow, S. (2015), *Algorithms in a Nutshell,* O’Reilly Media.

1. Heineman et al., (2015), Algorithms in a Nutshell. [↑](#endnote-ref-1)