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]](#footnote-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]](#footnote-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 optimised 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]](#footnote-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-1)

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.

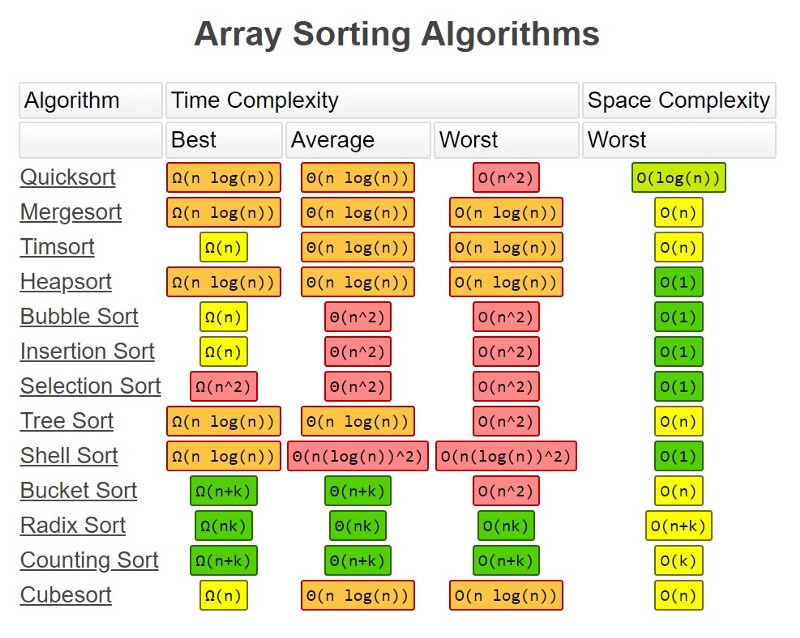


Figure 2.

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

As mentioned earlier sorting can be defined as arranging a collection of items according to some pre-defined ordering rules.[[5]](#footnote-4) Sorting algorithms should have the following properities;

* Stability
* Good run time efficiency
* In place sorting
* Suitability

It is also important to analyse each

* 1. Bubble Sort (a simple comparison based algorithm)[[6]](#footnote-5)

Bubble sort was first analysed in 1956. It is a comparison based algorithm, in that it uses comparison operations only to determine which of the twp elements should appear first in a sorted list. It keeps repeatedly swaps the adjacent element if they are in the wrong order.

Figure 3, shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are n−1n−1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete. At the start of the second pass, the largest value is now in place. There are n−1n−1 items left to sort, meaning that there will be n−2n−2 pairs. Since each pass places the next largest value in place, the total number of passes necessary will be n−1n−1. After completing the n−1n−1 passes, the smallest item must be in the correct position with no further processing required.[[7]](#footnote-6)

Sorting takes place by stepping through all the elements one by one and comparing with the adjacent element and swapping them if required. This algorithm gets is name from the way the larger values in a list “bubble up” to the end as the sorting takes place.

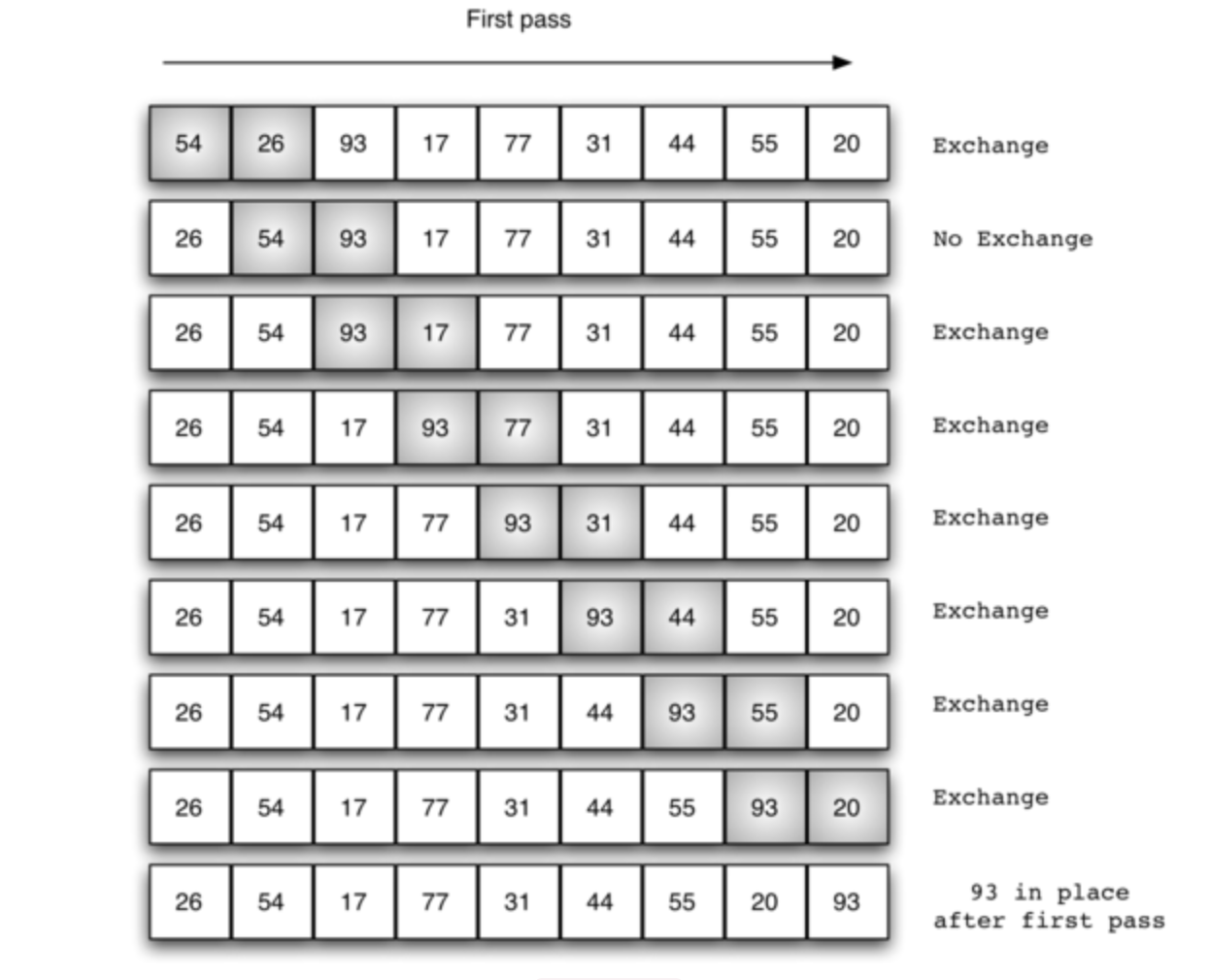


Figure 3. Bubble Sort: The First Pass

The Python code to implement this algorithm is shown below. There are two loops, the outer which runs from the second last item to the first in increments of one, and an inner loop, which from the first item in the list to the item in the list being currently run by the outer loop.

def bubbleSort(alist):

    for passnum in range(len(alist)-1,0,-1):

        for i in range(passnum):

            if alist[i]>alist[i+1]:

                temp = alist[i]

                alist[i] = alist[i+1]

                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]

bubbleSort(alist)

print(alist)

Output:

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

The Python code uses a nested loop to run the comparisons and sorts. It is possible to perform simultaneous assignment. The statement a,b=b,a will result in two assignment statements being done at the same time (see [Figure 4](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html#fig-pythonswap)). Using simultaneous assignment, the exchange operation can be done in one statement.[[8]](#footnote-7)

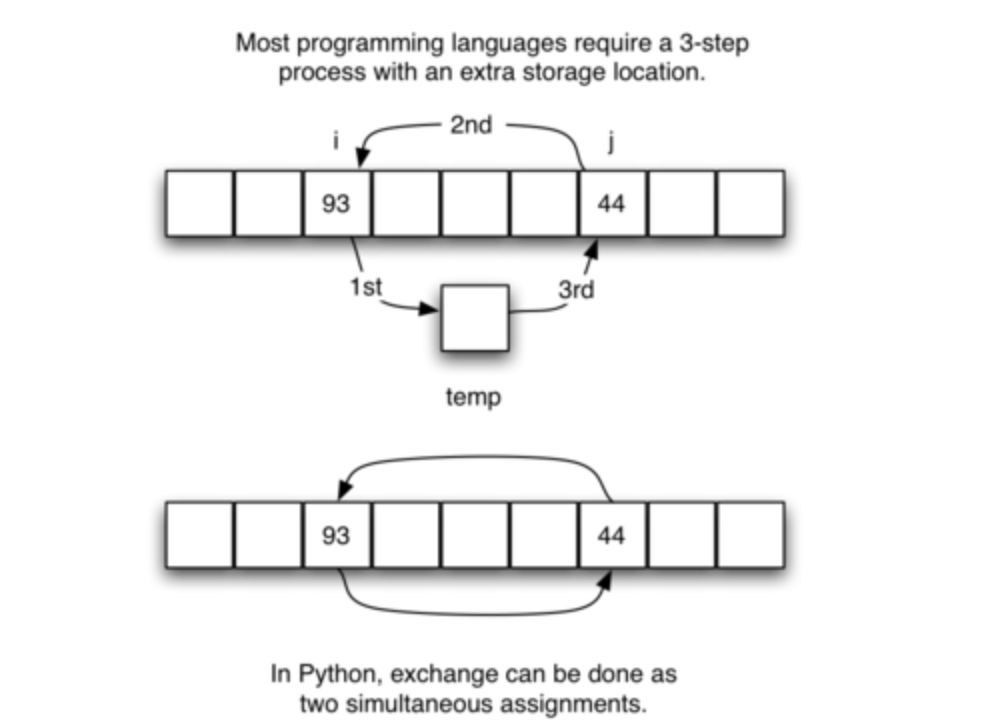


Figure 4. Exchanging Two Values in Python

* + 1. Complexity of Bubble Sort

Run time of Bubble Sort is very much dependent on the given input. If the given numbers are sorted, this algorithm runs in ***O(n)*** time. If the given numbers are in reverse order, the algorithm runs in ***O(n2)*** time

* **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

**Bubble Sort is easy to understand and therefore it is often use as a teaching tool.** It is simple to write, easy to understand and it only takes a few lines of code. The data is sorted in place so there is little memory overhead and, once sorted, the data is in memory, ready for processing.[[9]](#footnote-8) Bubble sort, if used when a data set is very small, can be an efficient algorithm to use.

The main disadvantage is the amount of time the algorithm takes to sort. The average time increases almost exponentially as the number of elements increases. In ten times the number of items, takes almost one hundred times as long to sort. Therefore, it will not deal well with large sets of data.

* 1. Merge Sort (an efficient comparsion based algorthim) [[10]](#footnote-9)

Merge sort was first proposed by John von Neumann in 1945 and is often described as a divide and conquer algorithm. In that 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. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.[[11]](#footnote-10)

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following diagram from shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

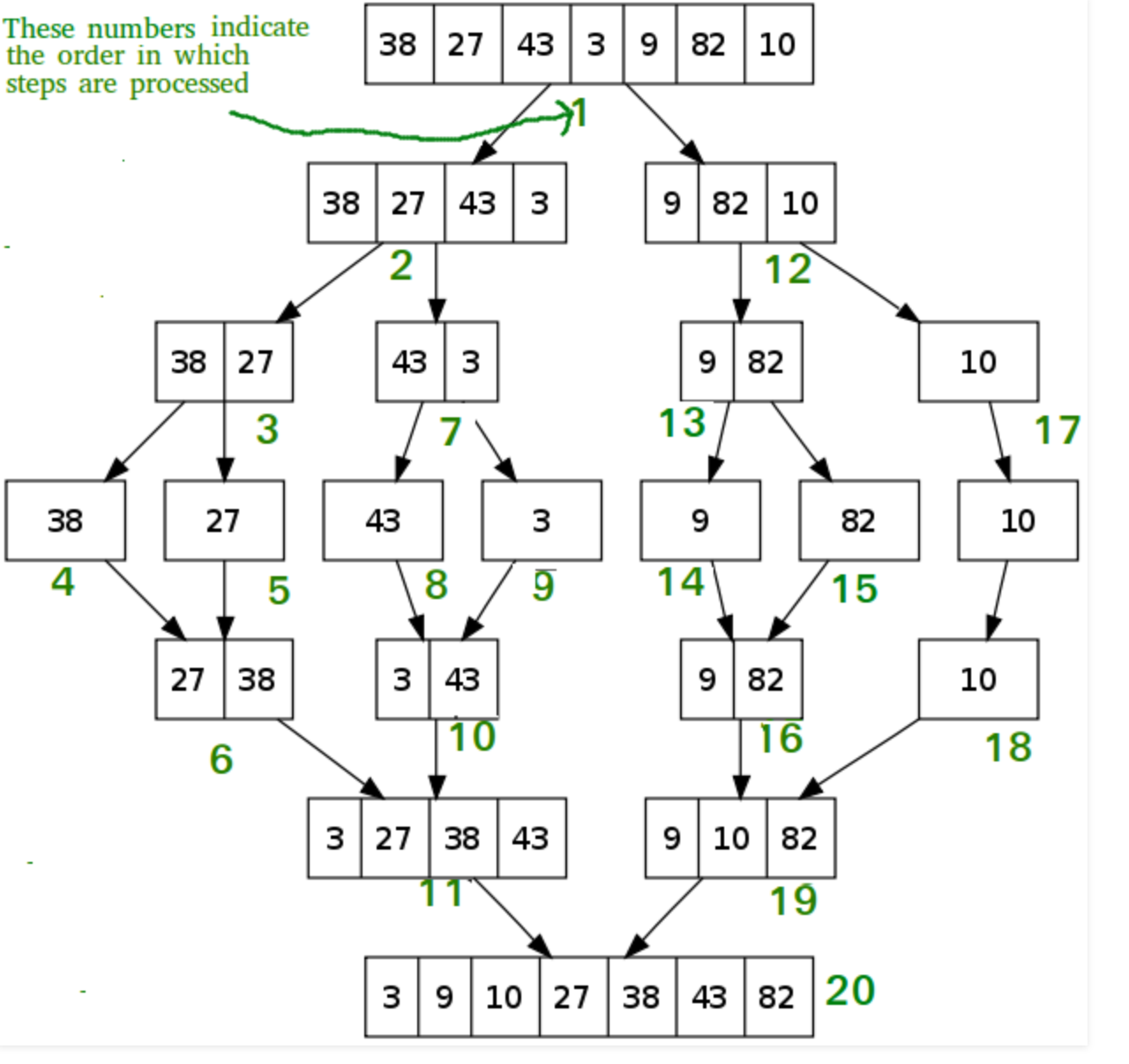


Figure 5. Merge Sort Process

The Python code to implement this algorithm is shown below. It begins by asking the base case question. If the length of the list is less than or equal to one, then we already have a sorted list and no more processing is necessary. If, on the other hand, the length is greater than one, then we use the Python slice operation to extract the left and right halves. It is important to note that the list may not have an even number of items. That does not matter, as the lengths will differ by at most one. Once the mergeSort function is invoked on the left half and the right half it is assumed they are sorted. The rest of the function is responsible for merging the two smaller sorted lists into a larger sorted list. Notice that the merge operation places the items back into the original list (alist) one at a time by repeatedly taking the smallest item from the sorted lists. Note that the statement lefthalf[i] <= righthalf[j] ensures that the algorithm is stable. A stable algorithm maintains the order of duplicate items in a list and is preferred in most cases.[[12]](#footnote-11)

The mergeSort function has been augmented with a print statement to show the contents of the list being sorted at the start of each invocation. There is also a print statement to show the merging process. The transcript shows the result of executing the function on our example list. Note that the list with 44, 55, and 20 will not divide evenly. The first split gives [44] and the second gives [55,20]. It is easy to see how the splitting process eventually yields a list that can be immediately merged with other sorted lists.

   def mergeSort(alist):

    print("Splitting ",alist)

    if len(alist)>1:

        mid = len(alist)//2

        lefthalf = alist[:mid]

        righthalf = alist[mid:]

        mergeSort(lefthalf)

        mergeSort(righthalf)

        i=0

        j=0

        k=0

        while i < len(lefthalf) and j < len(righthalf):

            if lefthalf[i] <= righthalf[j]:

                alist[k]=lefthalf[i]

                i=i+1

            else:

                alist[k]=righthalf[j]

                j=j+1

            k=k+1

        while i < len(lefthalf):

            alist[k]=lefthalf[i]

            i=i+1

            k=k+1

        while j < len(righthalf):

            alist[k]=righthalf[j]

            j=j+1

            k=k+1

    print("Merging ",alist)

alist = [54,26,93,17,77,31,44,55,20]

mergeSort(alist)

print(alist)

Output:

Splitting [54, 26, 93, 17, 77, 31, 44, 55, 20]

Splitting [54, 26, 93, 17]

Splitting [54, 26]

Splitting [54]

Merging [54]

Splitting [26]

Merging [26]

Merging [26, 54]

Splitting [93, 17]

Splitting [93]

Merging [93]

Splitting [17]

Merging [17]

Merging [17, 93]

Merging [17, 26, 54, 93]

Splitting [77, 31, 44, 55, 20]

Splitting [77, 31]

Splitting [77]

Merging [77]

Splitting [31]

Merging [31]

Merging [31, 77]

Splitting [44, 55, 20]

Splitting [44]

Merging [44]

Splitting [55, 20]

Splitting [55]

Merging [55]

Splitting [20]

Merging [20]

Merging [20, 55]

Merging [20, 44, 55]

Merging [20, 31, 44, 55, 77]

Merging [17, 20, 26, 31, 44, 54, 55, 77, 93]

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

The mergeSort function has been augmented with a print statement to show the contents of the list being sorted at the start of each invocation. There is also a print statement to show the merging process. The transcript shows the result of executing the function on our example list. Note that the list with 44, 55, and 20 will not divide evenly. The first split gives [44] and the second gives [55,20]. It is easy to see how the splitting process eventually yields a list that can be immediately merged with other sorted lists.[[13]](#footnote-12)

* + 1. Complexity of Merge Sort

As the worst, best and average case time complexity is O(n\*log n), it is a very efficient alogorithm. It starts with small subfiles and finshes with larger aubfiles, therefore it does not neet to stack. Merge sort is highly parrellelisable and can be used to impiment a stable sort.

**Time Complexity:**  Time complexity of Merge Sort is O(n\*log n) 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

Merge Sort is marginally slower that Quick Sort in practice, and it is not as efficient as Block Sort. Therefore

2.3 Counting Sort (a non-comparison sort)

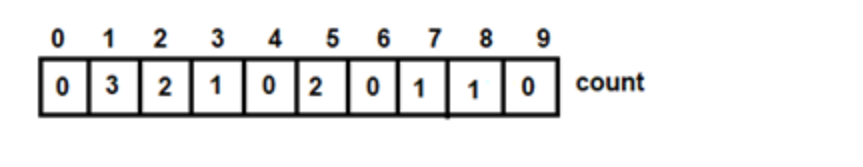
Counting sort was first proposed by Harold H.Seward in 1954. Counting sort allows us to do something which seems impossible, to sort a collection of items, in close to linear time. In essence Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.[[14]](#footnote-13)

There are a few assumptions when implementing Counting Sort. **it assumes that the input elements are**n**integers in the range [0,**k**].** When k = O(n), then the counting sort will run in O(n) time.

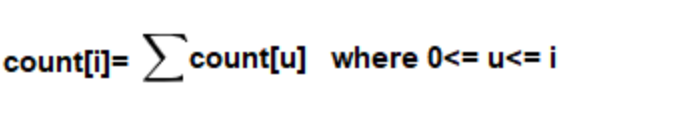
Consider the following input array A to be sorted. All the elements are in range 0 to 9

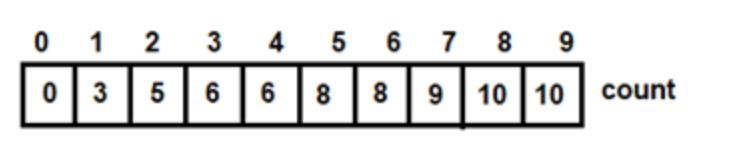


Step 1: Initialise an auxiliary array, say count and store the frequency of every distinct element. Size of count is 10 (k+1, such that range of elements in A is 0 to k)



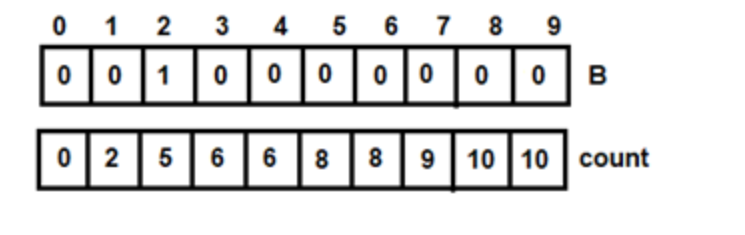
Step 2: Using the formula, updated count array is -



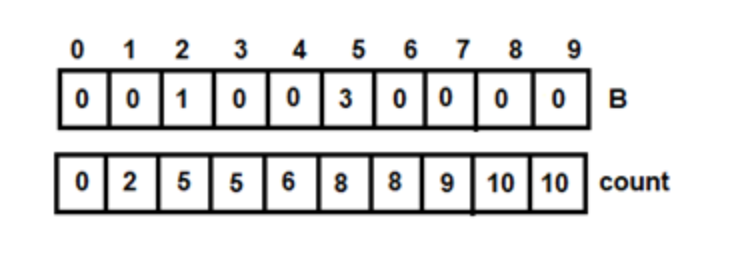


**Step 3**: Add elements of array A to resultant array B using the following steps:

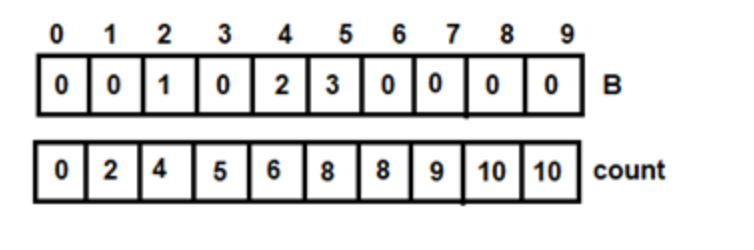
For, i=0, t=1, count[1]=3, v=2. After adding 1 to B[2], count[1]=2 and i=1



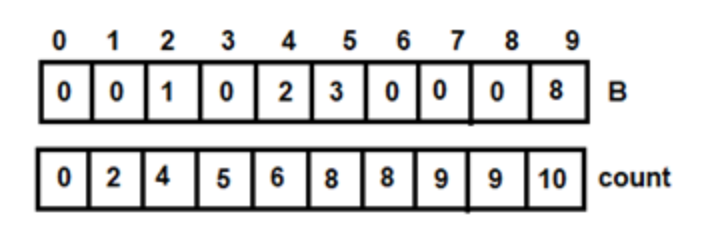
For i=1, t=3, count[3]=6, v=5. After adding 3 to B[5], count[3]=5 and i=2



For i=2, t=2, count[2]= 5, v=4. After adding 2 to B[4], count[2]=4 and i=3

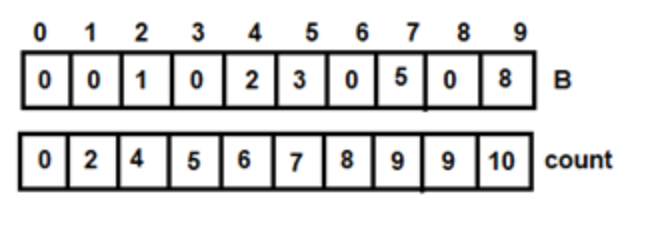


For i=3, t=8, count[8]= 10, v=9. After adding 8 to B[9], count[8]=9 and i=4

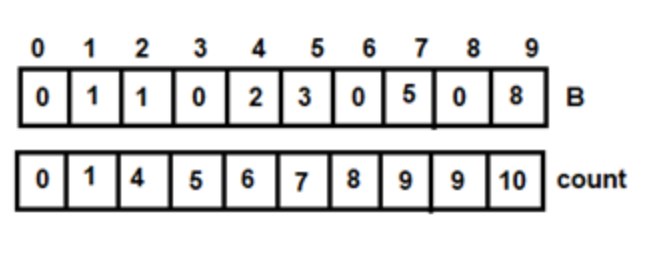


On similar lines, we have the following:

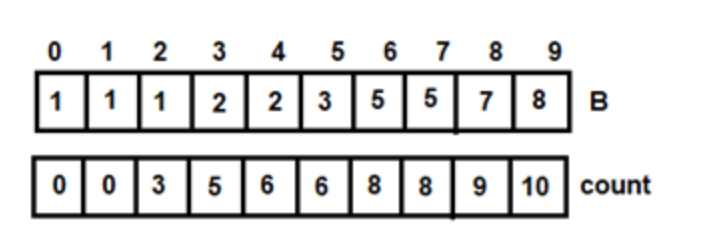
For i=4



For i=5



All the way until i=9



Thus, array B has the sorted list of elements.[[15]](#footnote-14)

The Python code to implement this algorithm is shown below.

def counting\_sort(array1, max\_val):

    m = max\_val + 1

    count = [0] \* m

    for a in array1:

    # count occurences

        count[a] += 1

    i = 0

    for a in range(m):

        for c in range(count[a]):

            array1[i] = a

            i += 1

    return array1

print(counting\_sort( [1, 2, 7, 3, 2, 1, 4, 2, 3, 2, 1], 7 ))

Output:

[1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 7]

* + 1. Complexity of Counting Sort

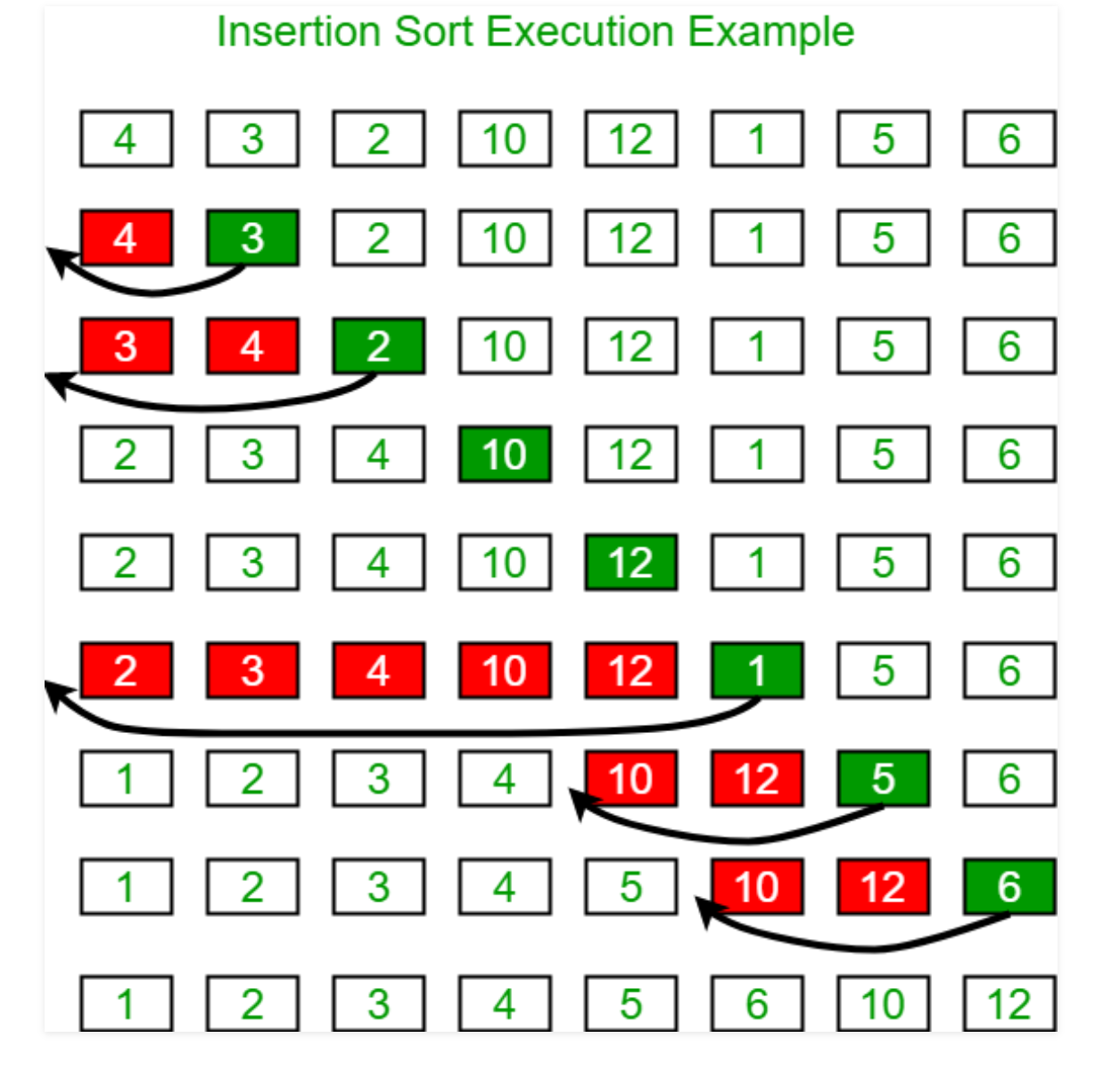
Counting sort has a O(k+n)*O*(*k*+*n*) running time. The first loop goes through A*A*, which has n*n* elements. This step has a O(n)*O*(*n*) running time. The second loop iterates over k*k*, so this step has a running time of O(k)*O*(*k*). The third loop iterates through A*A*, so again, this has a running time of O(n)*O*(*n*). Therefore, the counting sort algorithm has a running time of O(k+n)*O*(*k*+*n*). Counting sort is efficient if the range of input data, k*k*, is not significantly greater than the number of objects to be sorted, n*n*. Counting sort is a stable sort with a [space complexity](https://brilliant.org/wiki/space-complexity/) of O(k + n)*O*(*k*+*n*).[[16]](#footnote-15)

**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)

Counting Sort is both a fast and stable algorithm. However, is it not the most ideal for sorting large datasets or strings values.

* 1. Insertion Sort (a simple comparison-based sort)[[17]](#footnote-16)

Insertion sort is another example of a simple comparison based sorting algorthim. It always maintains a sorted sublist in the lower positions of the list. Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger. The image below shows the insertion sorting process. The shaded items represent the ordered sublists as the algorithm makes each pass. A real life example of Insertion Sort would be how card players sort a hand of cards. For example, a player has five cards, and they were already sorted from smallest to largest. The player gets a sixth card and inserts this card into the correct position. This is the idea behind **insertion sort**. Loop over positions in the array, starting with index 1. Each new position is like the new card handed to you by the dealer, and you need to insert it into the correct place in the sorted subarray to the left of that position.[[18]](#footnote-17)



The Python code to implement this algorithm is shown below. The implementation shows that there are again n−1n−1 passes to sort *n* items. The iteration starts at position 1 and moves through position n−1n−1, as these are the items that need to be inserted back into the sorted sublists. Line 8 performs the shift operation that moves a value up one position in the list, making room behind it for the insertion. Remember that this is not a complete exchange as was performed in the previous algorithms. The maximum number of comparisons for an insertion sort is the sum of the first n−1n−1 integers. Again, this is O(n2)O(n2). However, in the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted list. One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.[[19]](#footnote-18)

      def insertionSort(alist):

   for index in range(1,len(alist)):

     currentvalue = alist[index]

     position = index

     while position>0 and alist[position-1]>currentvalue:

         alist[position]=alist[position-1]

         position = position-1

     alist[position]=currentvalue

alist = [54,26,93,17,77,31,44,55,20]

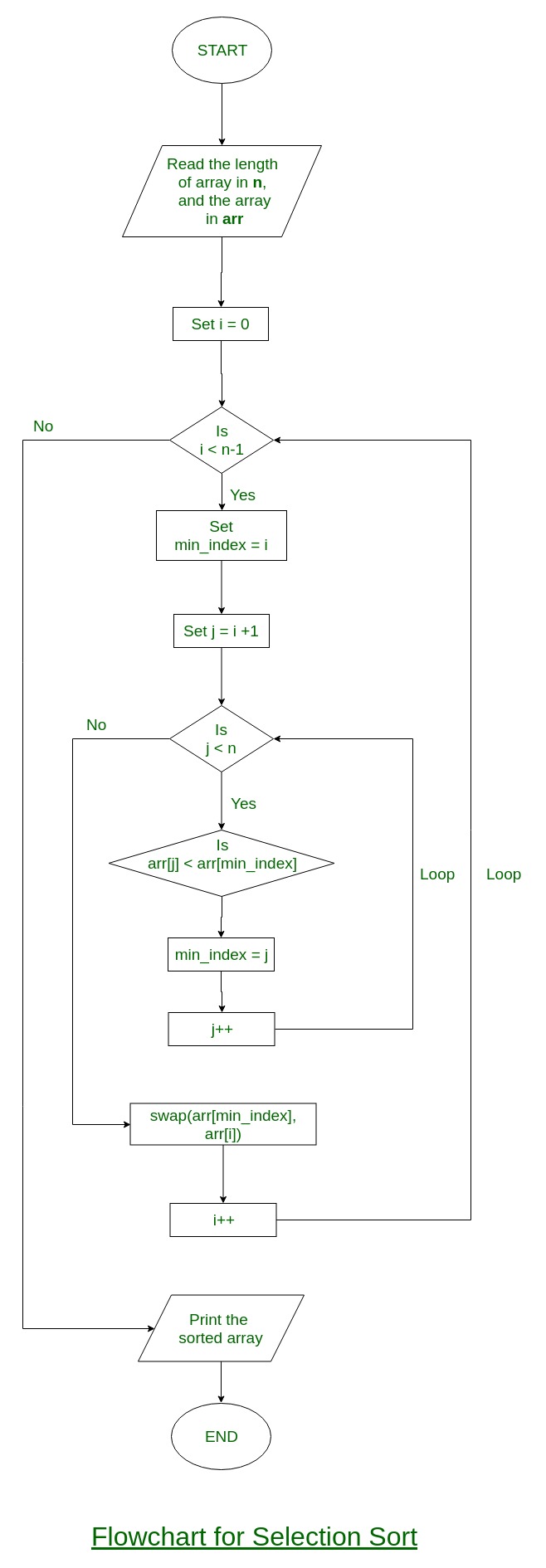
insertionSort(alist)

print(alist)

Output:

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

* + 1. Complexity 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 (a simple comparison-based sort)



def selectionSort(alist):

   for fillslot in range(len(alist)-1,0,-1):

       positionOfMax=0

       for location in range(1,fillslot+1):

           if alist[location]>alist[positionOfMax]:

               positionOfMax = location

       temp = alist[fillslot]

       alist[fillslot] = alist[positionOfMax]

       alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]

selectionSort(alist)

print(alist)

Output:

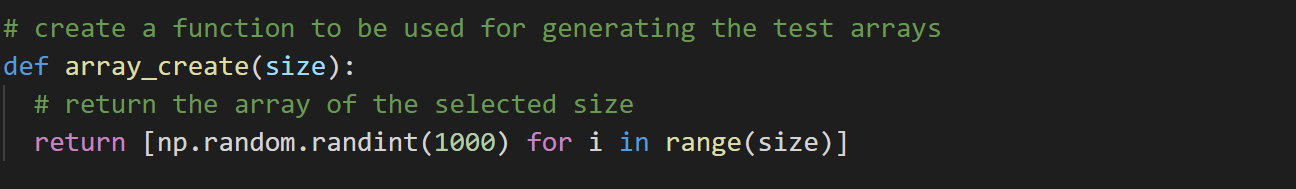
[17, 20, 26, 31, 44, 54, 55, 77, 93]

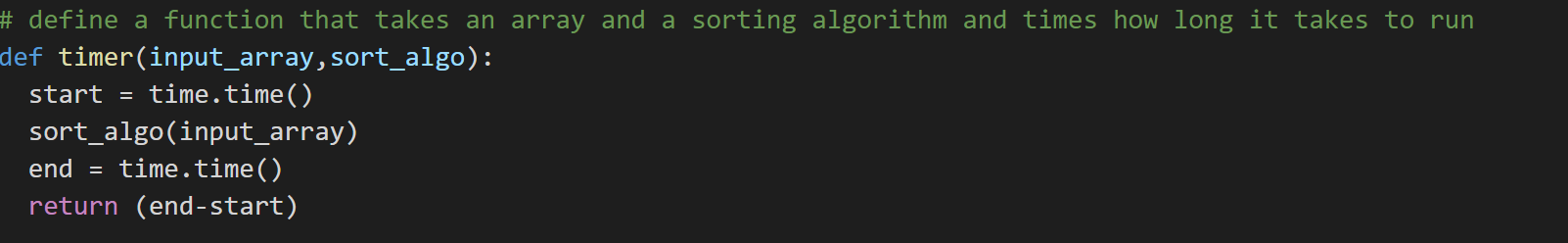
* + 1. Analysis of Selection Sort[[20]](#footnote-19)

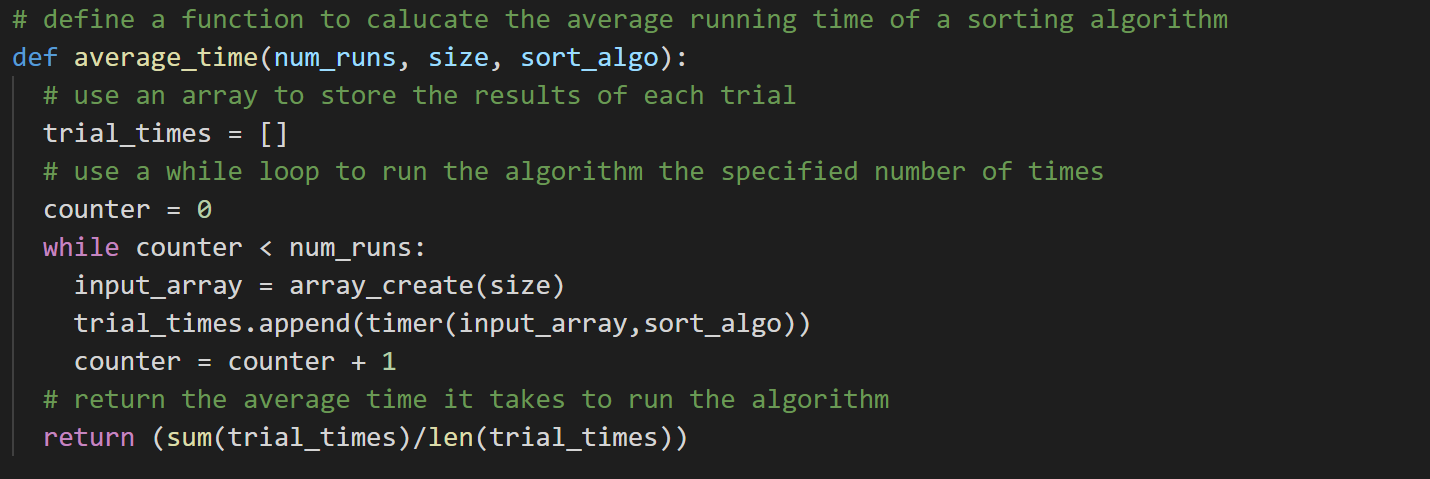
**Time Complexity:** O(n2) as there are two nested loops.

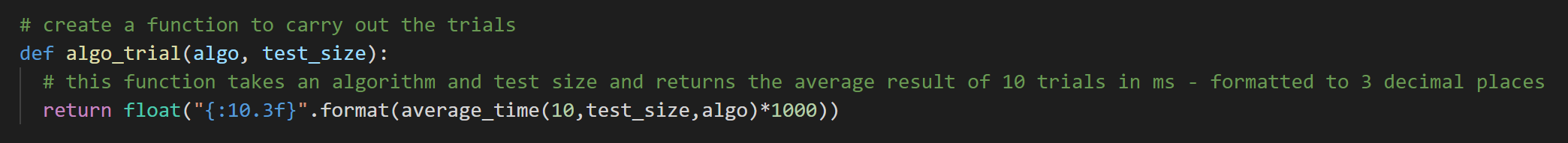
**Auxiliary Space:** O(1)  
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

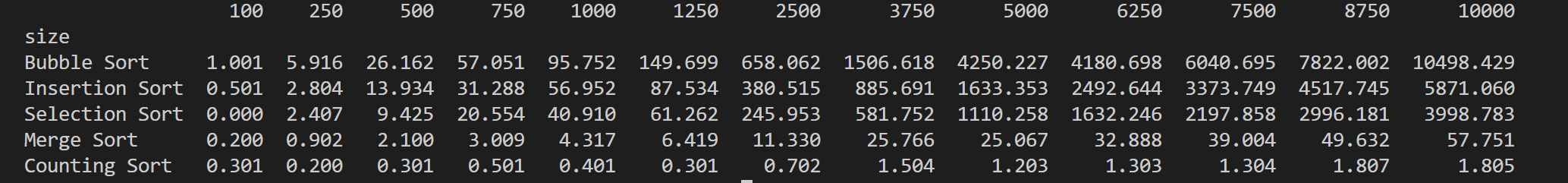
1. Implementation and Benchmarking











1. Conclusion

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

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

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

   Lecture notes

   <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>

   https://runestone.academy/runestone/books/published/pythonds/SortSearch/

   <https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-10.php>

   <https://www.techwalla.com/articles/advantages-disadvantages-of-bubble-sort> [↑](#endnote-ref-1)
5. Lecture notes – Sorting Algorithms Part II [↑](#footnote-ref-4)
6. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html> [↑](#footnote-ref-5)
7. [↑](#footnote-ref-6)
8. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheBubbleSort.html> [↑](#footnote-ref-7)
9. <https://www.techwalla.com/articles/advantages-disadvantages-of-bubble-sort> [↑](#footnote-ref-8)
10. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html> [↑](#footnote-ref-9)
11. <https://www.geeksforgeeks.org/merge-sort/> [↑](#footnote-ref-10)
12. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html> [↑](#footnote-ref-11)
13. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html> [↑](#footnote-ref-12)
14. <https://www.programiz.com/dsa/counting-sort> [↑](#footnote-ref-13)
15. <https://www.studytonight.com/data-structures/counting-sort> [↑](#footnote-ref-14)
16. <https://brilliant.org/wiki/counting-sort/> [↑](#footnote-ref-15)
17. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html> [↑](#footnote-ref-16)
18. <https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort> [↑](#footnote-ref-17)
19. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheInsertionSort.html#lst-insertion> [↑](#footnote-ref-18)
20. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheSelectionSort.html> [↑](#footnote-ref-19)