CTA Project - Benchmarking 5 Sorting Algorithms in Python

Table of Contents

Introduction

In this report I have choses 5 different sorting algorithms in Python to benchmark. I will run each algorithm 10 times with increasing size arrays and record the average running time for each algorithm. The sorting algorithms that I have decided to benchmark are:

1. Insertion Sort
2. Quicksort
3. Counting Sort
4. Merge Sort
5. Bubble Sort

Sorting is a component that lots of other types of algorithms are based off (Sorting Algorithms in Python, 2020). Sorting is defined as an arrangement of data in a certain order. Sorting algorithms are an important area of Computer Science as sorting large amounts of data can take a substantial amount of computing resources if the algorithms are inefficient (GeeksforGeeks, 2022). In this report, we will use the above algorithms to arrange numerical data in ascending order.

A sorting algorithms efficiency is proportional to the number of items it transfers. For small datasets, complex sorting algorithms are inefficient. For large datasets, the efficiency and speed of the algorithms must be increased as much as possible (GeeksforGeeks, 2022).

Time and Space Complexity

The efficiency of an algorithm depends on two parameters. These are Time Complexity and Space Complexity. The definition of Time Complexity is the number of times a certain set of instructions (arithmetic/bitwise instructions, memory referencing, control flow, etc) is executed rather than the total time that is taken. The time complexity is not the total time taken because this depends on external factors such as the compiler used, processor's speed, etc (Gupta, N/A). Space Complexity is defined as the total memory consumed by the program during execution (Time Complexity of Sorting Algorithms, N/A).

Big O Notation

Big O notation is also very important when it comes to sorting algorithms. The formal definition for Big O is (Wikipedia, N/A):

$$f(x) = O\big(g(x)\big)\ as\ x \to \infty$$

So, what is Big O notation? Big O notation is a simplified analysis of an algorithm's efficiency. It is a way to compare algorithms and it gives us an idea of how long an algorithm will take to run. Big O gives us an algorithms complexity in terms of the input size n. It gives us a way to abstract the efficiency of algorithms from the machines that they run on. As mentioned above, we don't want to consider the stats of a computer, rather we examine the basic computational steps of the code. Big O is used to analyse both time and space. There are a few ways to look at an algorithm's efficiency. In this report I will examine (Khan Academy, N/A):

- Worst-case
- Best-case
- Average-case

For Big O notation, we typically consider worst-case. There are two rules in Big O notation.

1. Ignore constants.

For example, if we had a function that is has a running time of 5n, we can say that it runs on the order of O(n) (Big O of n). This is because as n gets large, the 5 no longer matters.

2. Certain terms "dominate" others

As n grows, certain terms "dominate" others. We ignore or drop low order terms when they're dominated by high order terms. Consider the below code:

```
x = 5 + (15 * 20)
```

This basic computer statement computes x and notice that it does not depend on the input size in any way. We can say that this is O(1) (Big O of 1) or constant time. What happens when we have a sequence of statements?

```
x = 5 + (15 * 20)
y = 15 -2
print(x + y)
```

Notice that the 3 lines of code are constant time. So to compute Big O for this block of code, we simply add each of their times and we get 3 multiplied by big O of 1:

Total time = O(1) + O(1) + O(1) = O(1)

3*(1)

But remember that we drop constants so it's actually still just O(1).

Now, let's look at linear time. Consider the following for loop:

```
for x in range(0, n):
        print x
```

This prints the numbers from 0 to n. We know that the print statement in O(1). This means that the block of code is N*O(1), in other words O(N).

Here's another sequence:

```
y = 5 + (15 * 20)
for x in range(0, n):
    print x
```

Again, we know that the first line of code is O(1) and the for loop is O(N). The total time is the summation of these 2 blocks but remember that we drop low order terms.

Total time = O(1) + O(N) = O(N)

When N gets large, the time it takes to compute y is meaningless as the for loop dominates the run time.

Now let's look at quadratic time.

```
for x in range(0, n):
    for y in range(0, n):
        print(x*y)
```

The print statement will be executed n times n (N*N) which gives us O(N^2) (Big O of N squared).

To really understand this, let's look at 2 more examples.

```
x = 5 + (15 * 20)                # O(1)
for x in range(0, n):
    print(x)                     # O(N)
for x in range(0, n):
    for y in range(0, n):
        print (x * y)            # O(N squared)
```

The nested for loop dominates here so we get O(N^2).

Consider this if/else code:

```
if x > 0:
    // O(1)
elif x > 0:
    // O(logn)
else:
    // O(N squared)
```

Let’s pretend that the sequence of statements has already been deduced to the Big O’s shown. We said earlier that when we’re discussing Big O usually look at the worst-case scenario. So, for this situation we choose the largest runtime which in this case is O(N^2) (Sambol, 2017).

In-Place Sorting

he next thing to consider for sorting algorithms is whether the algorithm is in **in-place sorting** algorithm or a **not-in place sorting** algorithm. An **in-place sorting** algorithm requires a miniscule amount of extra space (which was defined earlier) besides the initial array holding the elements that must be sorted. Typically for sorting *n* elements, O(log*n*) extra space is required (planetmath.org, N/A). A **not-in place sorting** algorithm requires space that is greater than or equal to the elements that must be sorted.

Stable Vs Unstable Sorting Algorithms

When referring to the stability of a sorting algorithm, this really means how it deals with or treats repeated or equal elements. A stable sorting algorithm preserves the relative order of equal elements. Unstable sorting algorithms do not. Another way to put this is that stable sorting algorithms maintain the position of two equals elements relative to one another. Consider the below code and diagrams:

```
i < j and A[i] = A[j] and m < n
```

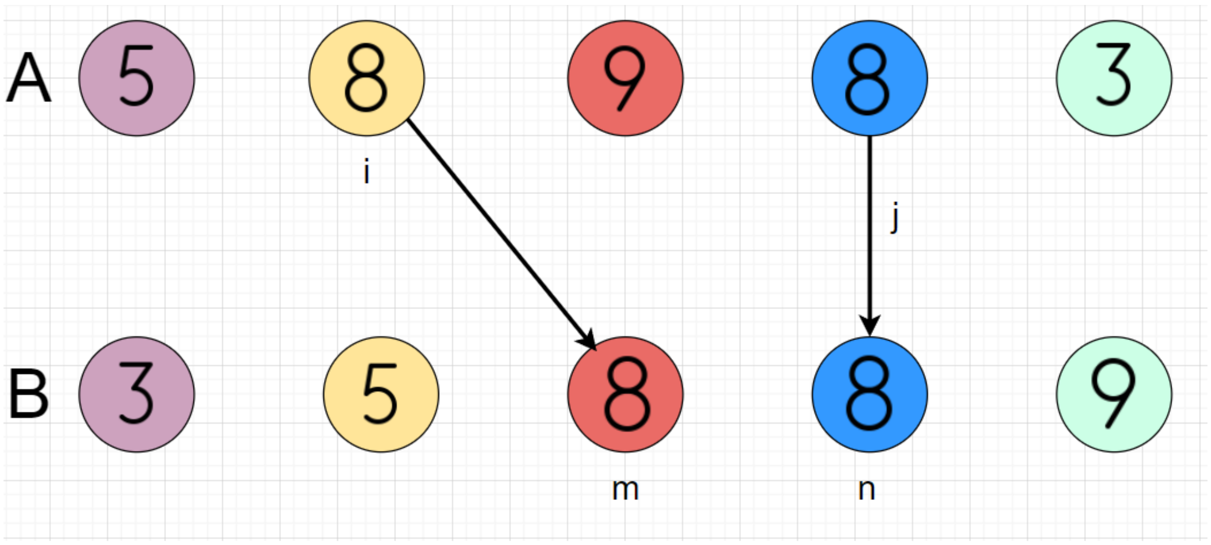


Table 1: Stable Sorting

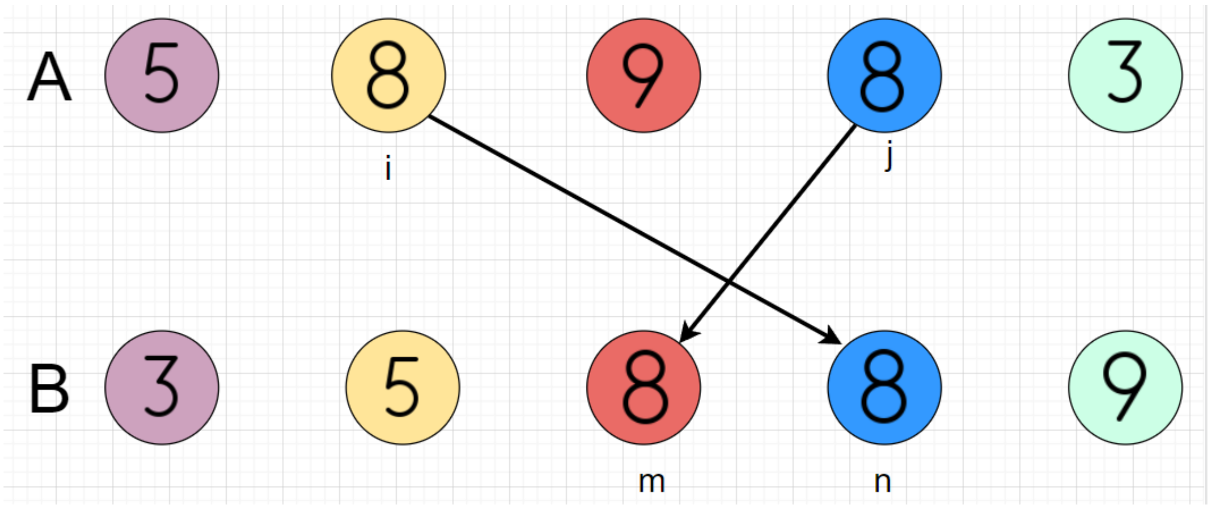


Table 2: Unstable Sorting

A is a collection of elements(array). < means that that i comes before j in this case and that m comes before n. B is the sorted version of the array A. In the unsorted array A, there are two equal elements I and j (A[i] and A[j]). The sorting is stable if the above equation is true.

Looking at the diagrams, stable sorting maintains the order of the two equal 8 balls, but unstable sorting might invert the relative order of the two 8 balls (Srivastava, 2021).

Comparison Based Sorting Algorithms

A sorting algorithm which makes a comparison between two elements(items) at a time during the process of sorting through the dataset is known as a **comparison sort**. These types of sorting algorithms always use a comparison operator while sorting. In python, these are things like ==, !=, <>, >, < etc.

Non-comparison sorting algorithms do not use any comparison operators during the sorting process (Joshi, 2017).

Sorting Algorithms

Bubble Sort

The Bubble Sort algorithm takes an unordered list and orders it in ascending values. It is a comparison sorting algorithm which has been discussed in the introduction of this report. The largest elements "bubble" to the end of the sequence. It is the simplest sorting algorithm conceptually and it works by repeatedly swapping the adjacent elements if they are in the wrong order (GeeksforGeeks, 2022). Bubble Sort goes through the sequence(array) multiple times and swaps the pairs of elements so that the largest element ends up at the end of the sequence and the smallest element ends up at the beginning the sequence (askpython.com, N/A).

Let's take an example array:

$\mathrm{arr} = [5, 3, 8, 6, 7, 2]$

First Iteration:

$[\mathbf{5}, \mathbf{3}, 8, 6, 7, 2]$

Bubble Sort makes its first comparison which is to compare the first two elements. 5 > 3 so they are swapped with each other. The array now becomes:

$[\mathbf{3}, \mathbf{5}, 8, 6, 7, 2]$

For the second comparison, 5 < 8 so they are **not** swapped with each other. The array now becomes:

$[3, \mathbf{5}, \mathbf{8}, 6, 7, 2]$

For the third comparison, 8 > 6 so they are swapped with each other. The array now becomes:

$[3, 5, \mathbf{6}, \mathbf{8}, 7, 2]$

For the fourth comparison, 8 > 7 so they are swapped with each other. The array now becomes:

$[3, 5, 6, \mathbf{7}, \mathbf{8}, 2]$

For the fifth comparison, 8 > 2 so they are swapped with each other. The array now becomes:

$[3, 5, 6, 7, \mathbf{2}, \mathbf{8}]$

The first iteration is now complete. The largest element "bubbled" to the end of the array.

Second Iteration:

$[\mathbf{3}, \mathbf{5}, 6, 7, 2, 8] \rightarrow [\mathbf{3}, \mathbf{5}, 6, 7, 2, 8]$ 3 < 5 so they are **not** swapped with each other

$[3, \mathbf{5}, \mathbf{6}, 7, 2, 8] \rightarrow [3, \mathbf{5}, \mathbf{6}, 7, 2, 8]$ 5 < 6 so they are **not** swapped with each other

$[3,5,\mathbf{6},\mathbf{7},2,8] \rightarrow [3,5,\mathbf{6},\mathbf{7},2,8]$	6 < 7 so they are **not** swapped with each other
$[3,5,6,\mathbf{7},\mathbf{2},8] \rightarrow [3,5,6,\mathbf{2},\mathbf{7},8]$	7 > 2 so they **are** swapped with each other
$[3,5,6,2,\mathbf{7},\mathbf{8}] \rightarrow [3,5,6,2,\mathbf{7},\mathbf{8}]$	7 < 8 so they are **not** swapped with each other

Third Iteration:

$[\mathbf{3},\mathbf{5},6,2,7,8] \rightarrow [\mathbf{3},\mathbf{5},6,2,7,8]$	3 < 5 so they are **not** swapped with each other
$[3,\mathbf{5},\mathbf{6},2,7,8] \rightarrow [3,\mathbf{5},\mathbf{6},2,7,8]$	5 < 6 so they are **not** swapped with each other
$[3,5,\mathbf{6},\mathbf{2},7,8] \rightarrow [3,5,\mathbf{2},\mathbf{6},7,8]$	6 > 2 so they **are** swapped with each other
$[3,5,2,\mathbf{6},\mathbf{7},8] \rightarrow [3,5,2,\mathbf{6},\mathbf{7},8]$	6 < 7 so they are **not** swapped with each other
$[3,5,2,6,\mathbf{7},\mathbf{8}] \rightarrow [3,5,2,6,\mathbf{7},\mathbf{8}]$	7 < 8 so they are **not** swapped with each other

Fourth Iteration:

$[\mathbf{3},\mathbf{5},2,6,7,8] \rightarrow [\mathbf{3},\mathbf{5},2,6,7,8]$	3 < 5 so they are **not** swapped with each other
$[3,\mathbf{5},\mathbf{2},6,7,8] \rightarrow [3,\mathbf{2},\mathbf{5},6,7,8]$	5 > 2 so they **are** swapped with each other
$[3,2,\mathbf{5},\mathbf{6},7,8] \rightarrow [3,2,\mathbf{5},\mathbf{6},7,8]$	5 < 6 so they are **not** swapped with each other
$[3,2,5,\mathbf{6},\mathbf{7},8] \rightarrow [3,2,5,\mathbf{6},\mathbf{7},8]$	6 < 7 so they are **not** swapped with each other
$[3,2,5,6,\mathbf{7},\mathbf{8}] \rightarrow [3,2,5,6,\mathbf{7},\mathbf{8}]$	7 < 8 so they are **not** swapped with each other

Fifth Iteration:

$[\mathbf{3},\mathbf{2},5,6,7,8] \rightarrow [\mathbf{2},\mathbf{3},5,6,7,8]$	3 > 2 so they **are** swapped with each other

The program now ends as there no unsorted elements to the left (Pedamkar, N/A).

Let's look at the Python code now that we have a high-level understanding of how Bubble Sort works:

```python
def bubble(arr):
    indexing_length = len(arr) - 1
    sorted = False

    while not sorted:
        sorted = True
        for i in range(0, indexing_length):
            if arr[i] > arr[i+1]:
                sorted = False
                arr[i], arr[i+1] = arr[i+1], arr[i]
    return arr
```

The input to this function is the unsorted array(arr). The first thing that we need is the indexing length of where we're going to make these comparisons. The reason that we say len(arr) – 1 here is because there is no number after the last number in the array so we can't do a comparison. The sorted variable is inside the control flow so that we can break out of the while loop when the array is sorted. The start of the while loop means that if the sorted variable is false, we'll perform the following actions. We will come back to the next line sorted = True later. The for loop means that we want to iterate through the list from element 0 to the second last element. The nested if statement is a comparison. If the value to the left is greater than the value to the right, we need to say that sorted = False, and then we need to flip those 2 values in our list. So, what we're telling our algorithm to do is if there is an item to the left is greater than the item to its right, say that the sorted variable is false, and then flip those two items. When we sort all the items, the if statement won't activate, so the sorted variable will remain true, and that will break us out of the while loop. Finally, we need to return the now sorted list (Sherrill, 2019).

Let's now discuss the time and space complexity of Bubble Sort. Bubble sort has an inner and an outer loop as seen in the code above. The inner loop does O(n) comparisons. In the worst-case, the outer loop runs O(n) times. This is when the array is reversed (in descending order). The results in:

$$O(n \times n) = O(n^2)$$

The best-case scenario is when the array is already sorted. However, Bubble Sort still performs O(n) comparisons.

The average-case is when Bubble Sort requires n/2 iterations and O(n) comparisons for each iteration (Upadhyay, 2022):

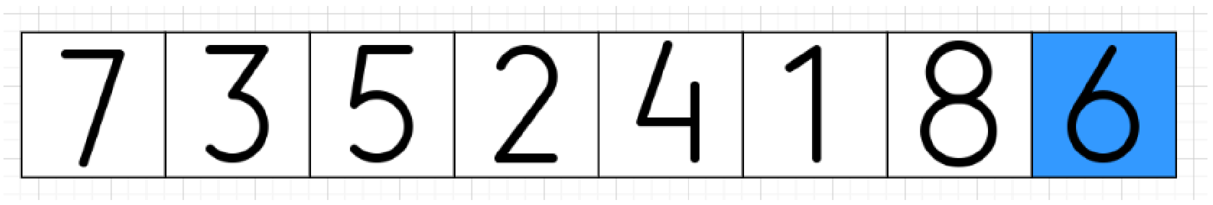
$$O\left(\frac{n}{2} \times n\right) = O(n^2)$$

Bubble Sort has a space complexity of O(1). This is because it uses a constant amount of extra space for the flag(variable) (gatevidyalay.com, N/A).

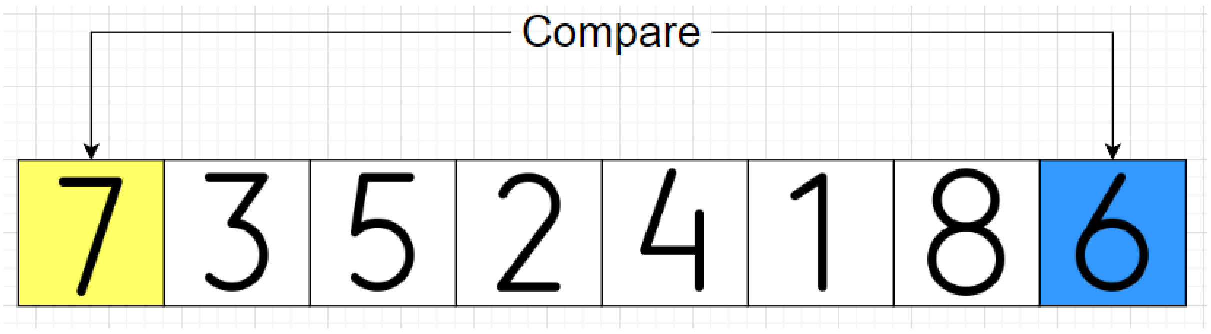
Bubble sort is an **in-place** sorting algorithm since it swaps adjacent pairs without the use of large data structure (GeeksforGeeks, 2022). It is a **stable** sorting algorithm. See the introduction of this report for an explanation of stable algorithms.

Quicksort

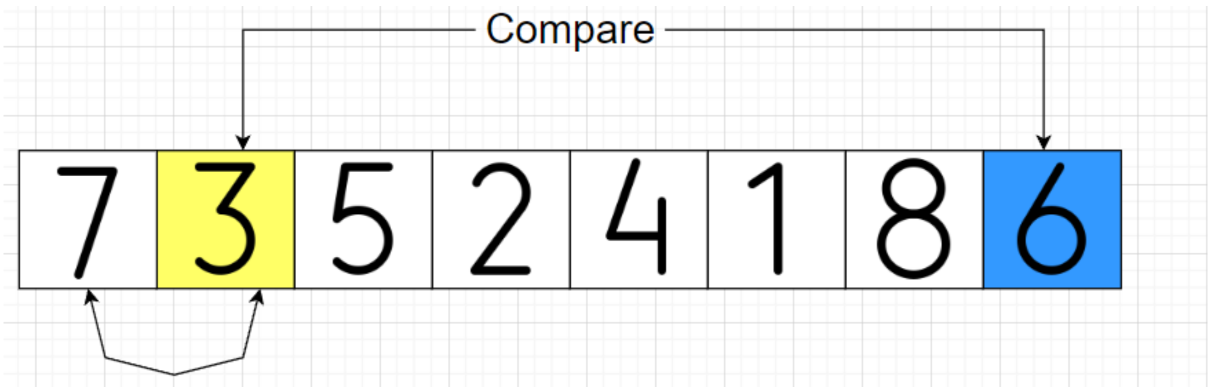
The Quicksort algorithm takes an unsorted array and sorts it in either ascending or descending order. In the example below, we'll be working in ascending order. We start by taking one element in our unsorted array and making it out pivot point. In this example, we'll take the last element in the array as our pivot. The pivot is the number that we want to base comparisons of all the other numbers with. We begin by iterating through the rest of the values in our unsorted sequence (algotree.org, N/A).



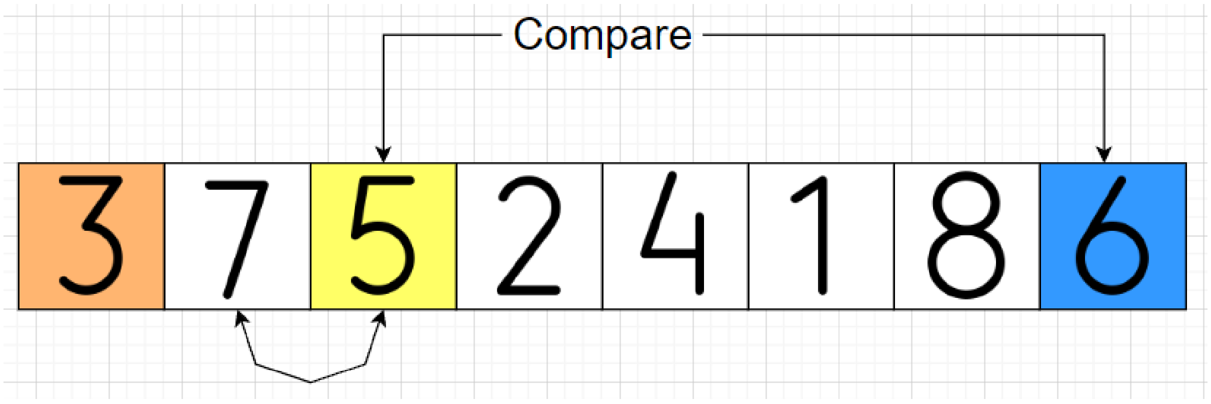
Set pivot to the last element in the array (Pivot = 6)



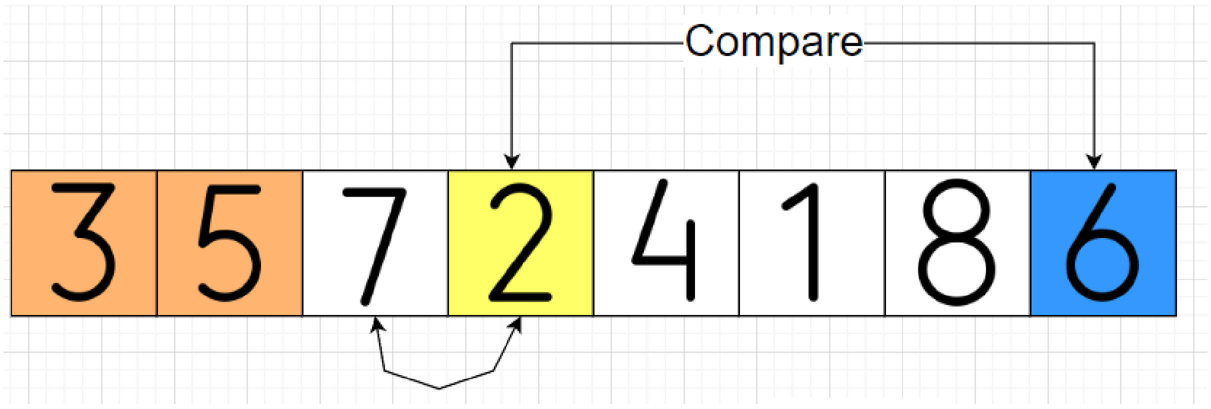
Since 7 > 6, continue and compare the next number



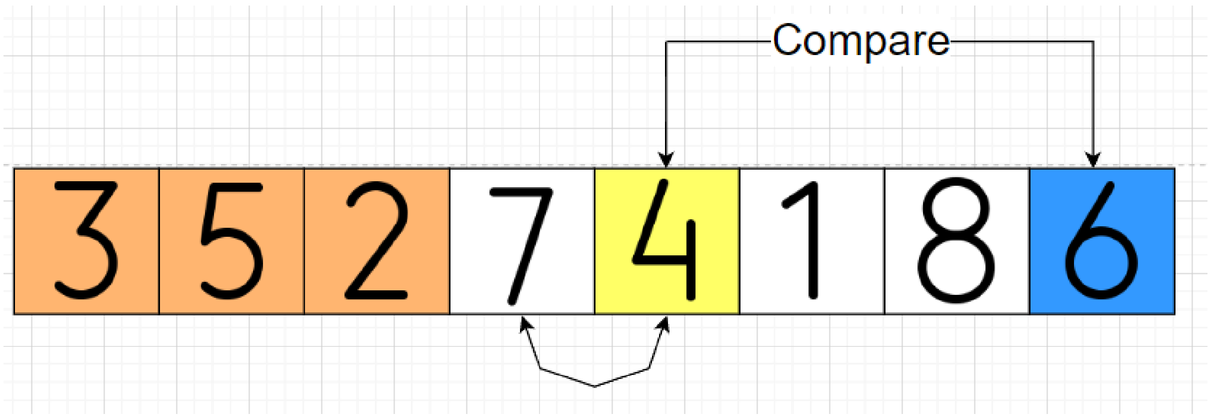
Since 3 < 6, put it at the beginning by swapping 3 with 7.



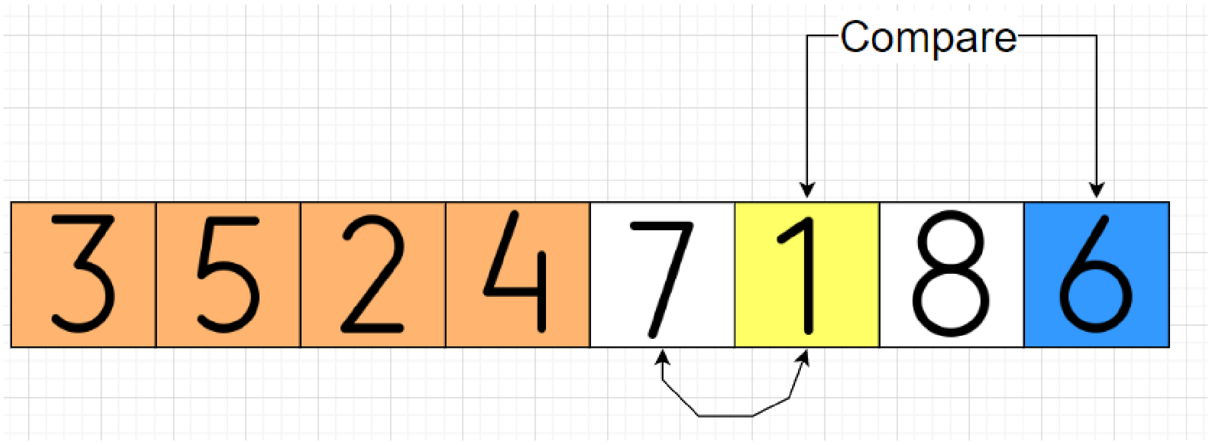
Since 5 < 6, put 5 before 7(swap them)



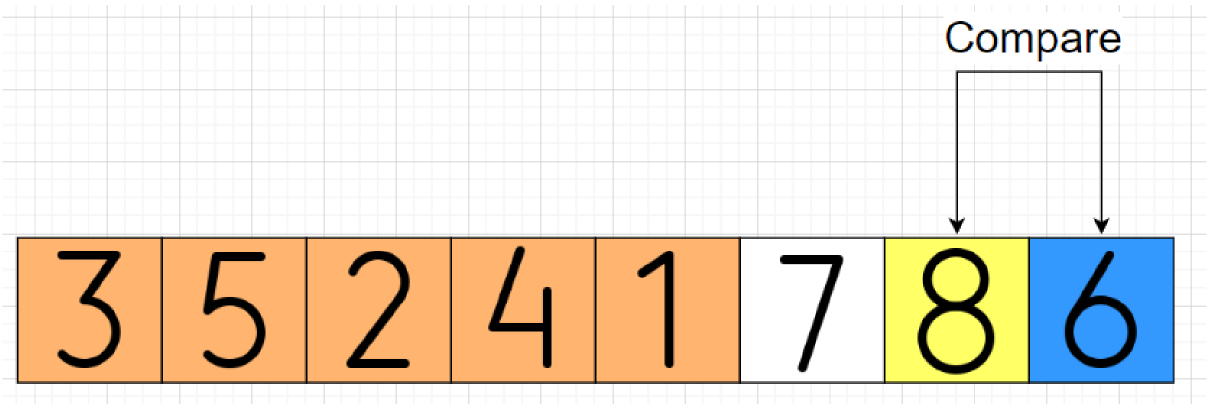
Since 2 < 6, put 2 before 7(swap them)



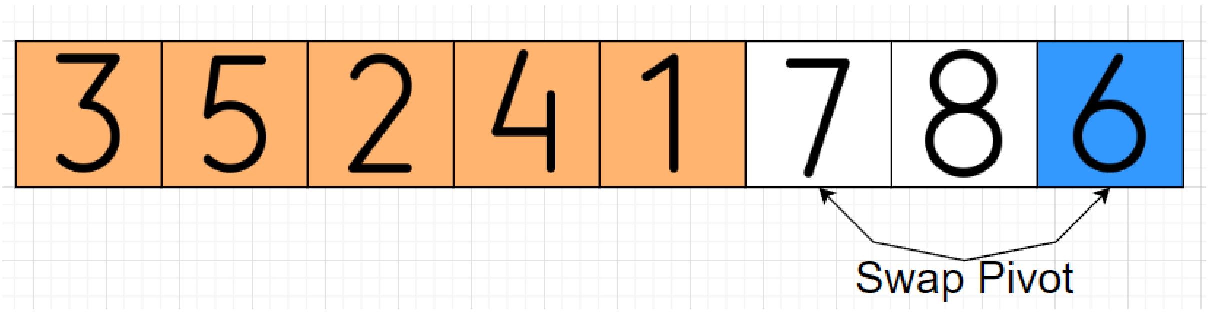
Since 4 < 6, put 4 before 7(swap them)



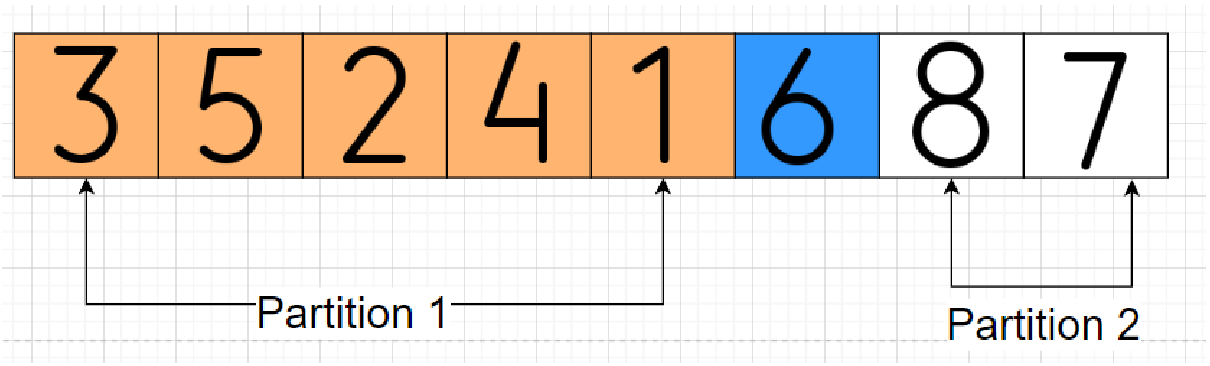
Since 1 < 6, put 1 before 7(swap them)



Since 8 > 6, continue and compare the next number



Swap pivot (6) with 7 to place it at its final position and create 2 new lists

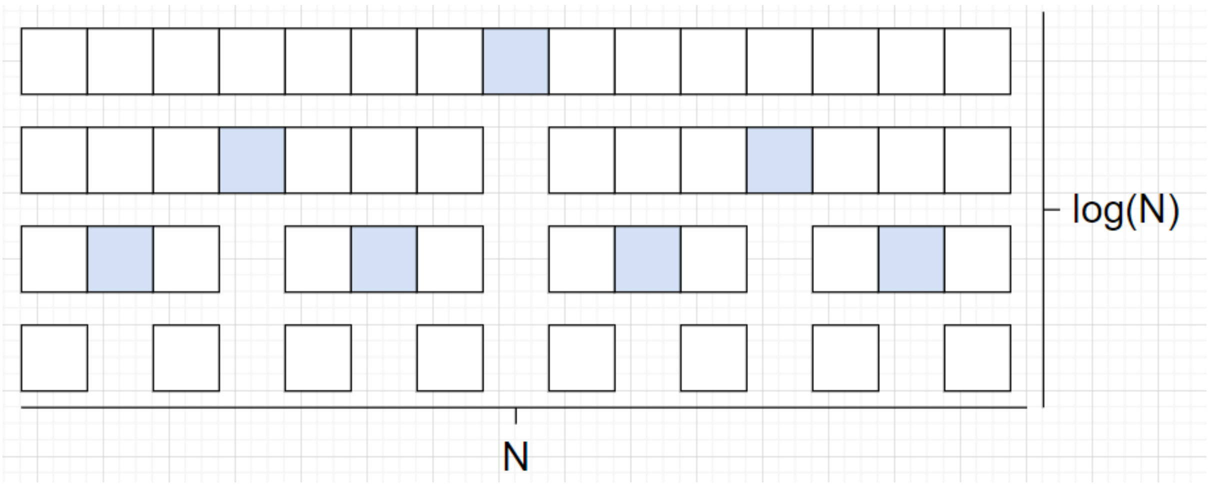


In other words, if a number is less than our pivot number (6) then it goes into partition 1, and if it's higher it goes into partition 2 as seen above.

The pivot does not have to be the last number in the array. It could be the first number, the middle number, or the median of all three of those numbers. Once we reach the end of our array, we know that our initial pivot point (6) has been sorted. Now we need to sort the numbers in the two partitions that we created. To do this, we used the same method again. In this case, we would use 1 and 7 and repeat the whole process recursively until the array is sorted (Serrill, 2019).

In terms of time complexity, Quicksort is difficult analyse. This is because the pivot selection is random and can heavily affect the performance of this algorithm.

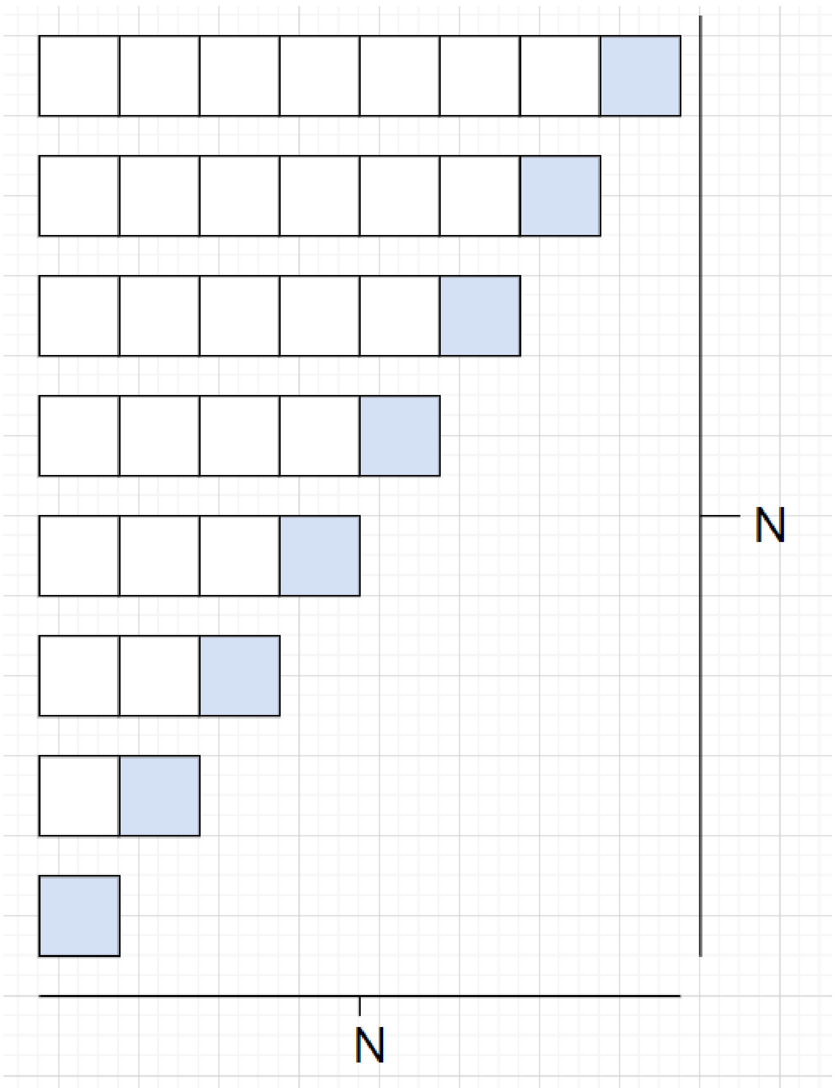
For the average-case, we must assume an equally distributed array. In other words, that the numbers are evenly spread between the lowest number and the highest number and that there are no big bands of similar numbers. We also need to assume that the pivot is close to the average number in our array. So, if the array is evenly distributed and we pick a pivot number at random, we would have a 50% chance that the number would be close to the average rather than the minimum or maximum value in the array. Interestingly, if the pivot number is close to the average, the array will be partitioned into two equally sized halves. Therefore, each recursive call to the function will be working with data that is half the initial array. Let's look at an example. The array below contains 15 elements. If we pick the average number as our pivot value, we end up with a tree of recursive calls:



Each level of the diagram looks at around N elements. There are four levels to the trees. This is approximated to log(N). We can therefor say that the time complexity for the average case for Quicksort is O(Nlog(N)) (Computational Core, N/A).

The best-case for Quicksort is if the pivot-point selected is exactly the mean. This is of course extremely rare in practice, so the average-case is the most common case (Bais, N/A).

Now let's take the worst-case. Contrary to the average case, if the pivot point chosen is the maximum value, each recursive call to the function would generate one empty partition. The other partition would of course be only one less that the size of the original array. If the array had 8 numbers, the recursive tree diagram would look like the following:



We are reducing the size of the array by only one at every level. This means that it would take N recursive calls to finish. This boils down to a mathematical series. The series is:

$$N + (N-1) + (N-2) + \ldots + 2 + 1$$

This approximates to N^2. Quicksort runs at $O(N^2)$ in the worst-case. In practice though it is very rare that the pivot point is the max or min value, so Quicksort is usually one of the faster sorting algorithms (Computational Core, N/A).

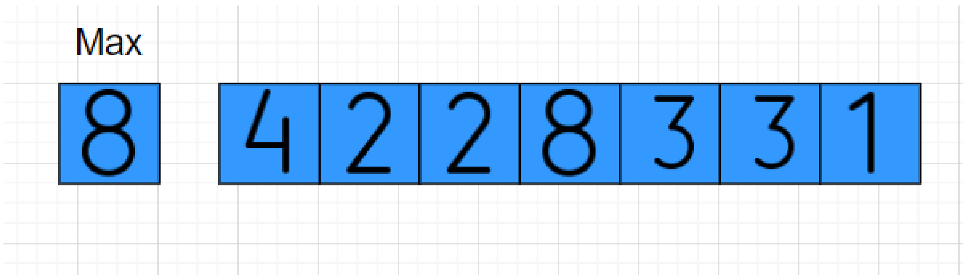
The Space Complexity for Quicksort is O(N). This is because no other container is created other than the original array (Bais, N/A).

Quicksort is not a stable sorting algorithm as per the definition in the introduction of this report. This is because it swaps elements according to pivot's position (without considering their original positions) (askinglot.com, 2020).

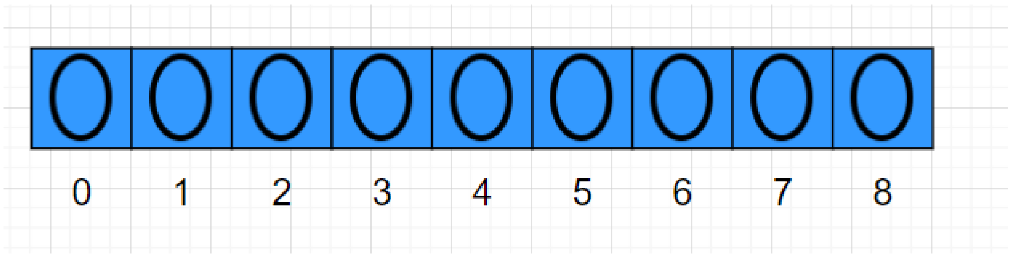
Quicksort is an in-place sorting algorithm because it only uses extra space for storing the recursive function calls but not for manipulating the input (GeeksforGeeks, N/A).

Counting Sort

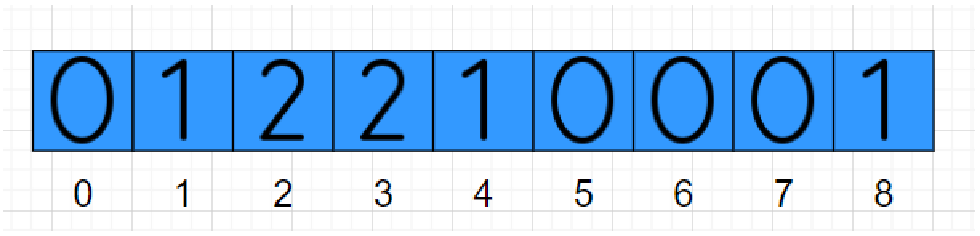
Counting Sort is an integer sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is achieved by mapping the count as an index of the auxiliary array (programiz.com, N/A). Let's consider the below array.



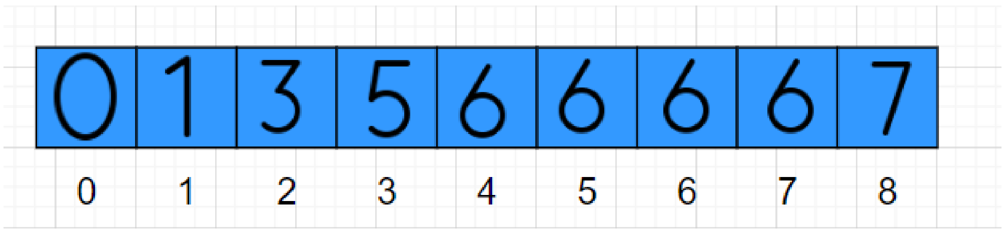
First, we need to find the maximum value(element) from in the array. 8 in this case.



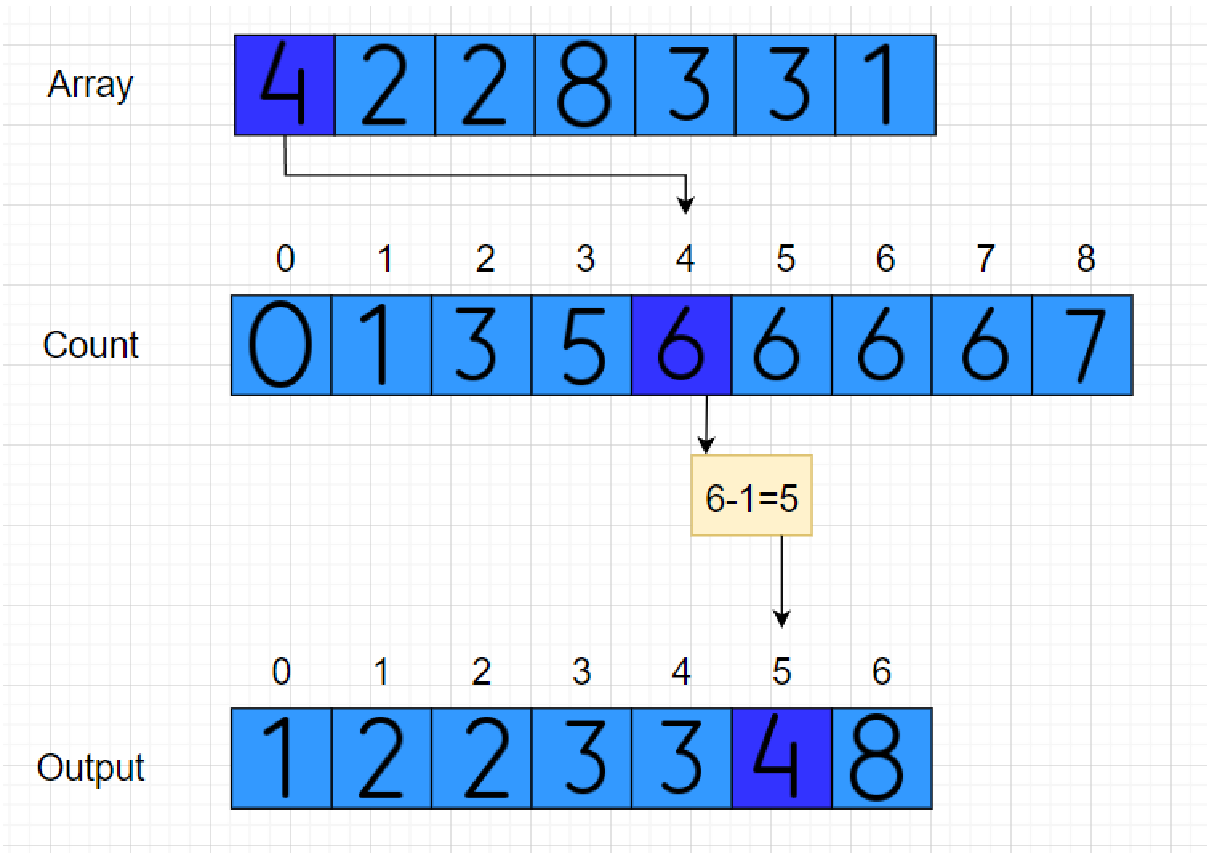
Next, we need to generate a new array of size (or length) max + 1. All the elements (values) are set to 0. So, in the above example, new auxiliary array has 9 elements. The auxiliary array is used to store the count of the elements in the array. In other words, all the numbers up to the max value are accounted for in the auxiliary array.



Next, we need to store the count of each element at their corresponding index in the auxiliary array. In this example, the count of element 3 is 2. So, 2 is stored in the third position of auxiliary array. Element 5 is not present in the auxiliary array so, 0 is stored in 5th position.



Next, starting at index 1, we need to add the previous index to change the count or auxiliary array to the above. This is the cumulative sum of the elements.



Next, match each element of the original array to the index of the auxiliary or count array. This gives the cumulative count. Place the element at the index calculated as shown above. Finally, decrease the count by 1 and this result is the new index of the output array. The array is now sorted (programiz.com, N/A).

In terms of time complexity for Counting Sort, let's have a look at the code for the algorithm:

```
def counting_sort(ary):
    size = len(ary)
    output = [0] * size
    count = [0] * 101
    for no in range(0, size):
        count[ary[no]] += 1
    for no in range(1, 101):
        count[no] += count[no - 1]
    no = size - 1
    while no >= 0:
        output[count[ary[no]] - 1] = ary[no]
        count[ary[no]] -= 1
        no -= 1
    for no in range(0, size):
        ary[no] = output[no]
```

Let k be the size of the array to be sorted and max is the value of the maximum element. There are 4 loops in the code (programiz.com, N/A). The best-case is when all elements are within the same range or when k is 1. Counting the occurrence of each number in the input array takes constant time. It takes n time to find the correct index value of each element in the sorted output array. This results in linear time complexity O(1 + N), i.e. O(N) (simplilearn.com, 2021).

The worst case is when max is much bigger than k because this broadens the range of k. This is also linear time complexity O(N). This average case is also linear O(N). In all cases, the time complexity is O(N+K). This is because the algorithm iterates n+k times, regardless of how the elements are in the array (javatpoint.com, N/A).

The Space Complexity of Count Sort is O(k) because when max is much bigger than k it broadens the range of k. The larger the range of elements in an array, the larger the space complexity becomes (simplilearn.com, 2021).

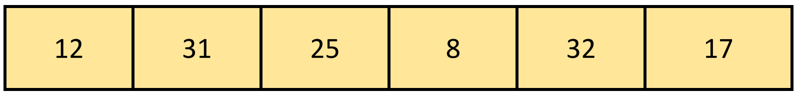
Counting Sort is a non-comparison-based algorithm that maintains its stability by making sure that the output array is filled in a reverse order. This ensures that all elements with equivalent values maintain the same relative position (geeksforgeeks.org, 2019). Since Count Sort uses a temporary auxiliary array, it is not an in-place sorting algorithm.

Insertion Sort

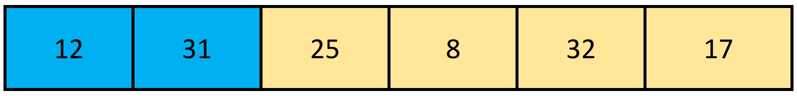
Insertion Sort is a simple sorting algorithm, it has a simple implementation, and it is easy to understand. Some downsides are that it has a large runtime. It is slower compared to other more complex sorting algorithms.

We can think of Insertion Sort like sorting a hand of playing cards. If we were to do this in real life, we would assume that the first card is already sorted. We would then pick an unsorted card, compare it to the first card, and if it is larger than the first card it gets put to the right of the first card. If it's smaller, it gets put to the left of the first card. For Insertion Sort, we one element and iterate it through the already sorted array (FelixTechTips, Insertion Sort In Python Explained, 202).

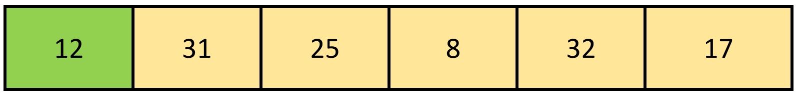
Let's look at an example below to see how it works:

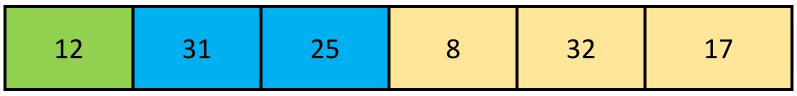


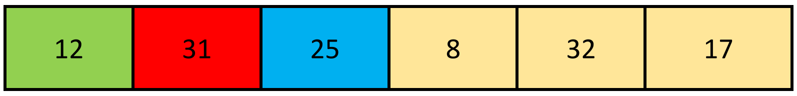
The first thing it does is compare the first two elements.



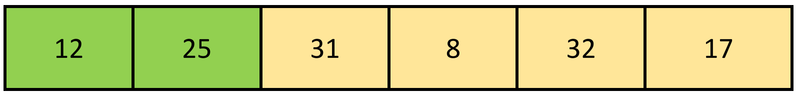
31 is greater than 12. The two elements are already in ascending order. The 12 is stored in a sorted sub-array.



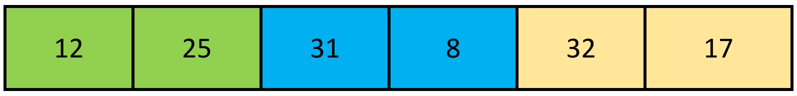




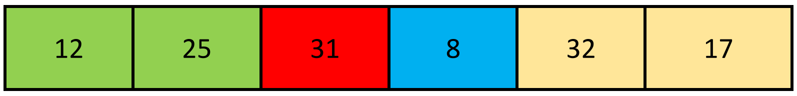
25 is less than 31. 31 is swapped with 25. 25 is now compared with the elements in the sorted array.

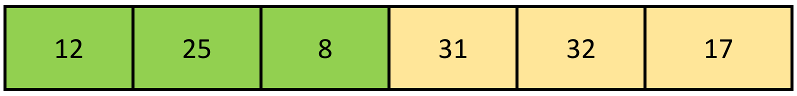


At the moment, there is only one element in the sorted array which is 12. 25 is greater than 12. The sorted array remains the same. The sorted array now consists of 12 and 25. We must now move onto the next elements 31 and 8.

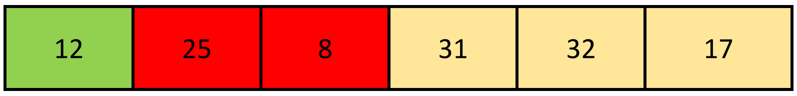


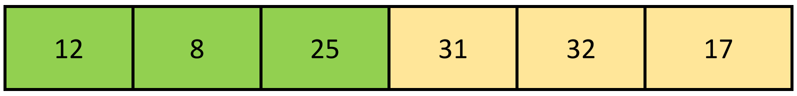
31 and 8 are unsorted. They get swapped.



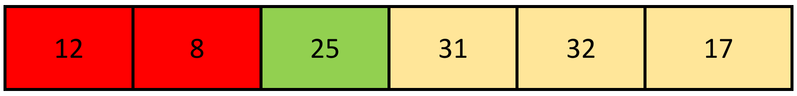


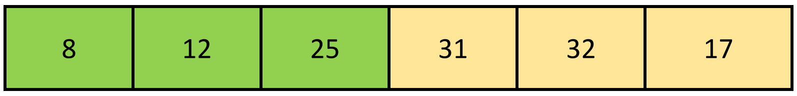
But now 25 and 8 are unsorted in the sorted array. They also get swapped.



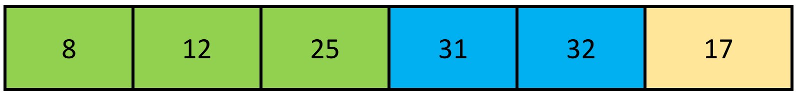


Elements 12 and 8 are now unsorted. They also get swapped.

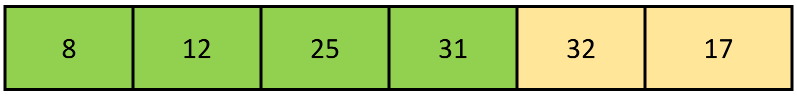




The sorted array in green now consists of 8, 12 and 25. We must now move onto the next elements 31 and 32.



They are already sorted.



The sorted array in green now consists of 8, 12, 25 and 31. We must now move onto the next elements 32 and 17.

8	12	25	31	32	17

17 is less than 32. 17 is swapped with 32. 17 is now compared with the elements in the sorted array.

8	12	25	31	17	32

8	12	25	31	17	32

Elements 31 and 17 are now unsorted. They get swapped

8	12	25	17	31	32

8	12	25	17	31	32

17 is less than 25. 17 is swapped with 25. 17 is now compared with the elements in the sorted array.

8	12	17	25	31	32

The array is now sorted and Insertion Sort ends (javatpoint.com, N/A).

Let's now discuss the time complexity for Insertion Sort. The worst-case for Insertion Sort is when it is used on a reverse-sorted array. This is because it will insert each element at the beginning of the sorted subarray. This is $O(N^2)$.

The average-case is when the elements are in a random order. The run time is:

$$O\left(\frac{N^2}{4}\right) = O(N^2)$$

The best-case is when the array is already sorted. It only compares each element to its predecessor, thereby requiring n steps to sort the already sorted array of n elements (interviewkickstart.com, N/A).

The Space Complexity is linear, O(1). This is because uses a constant amount of additional memory apart from the input array (interviewkickstart.com, N/A).

Insertion Sort is a comparison based stable algorithm per the definition in the introduction of this report. It maintains relative order (GeeksforGeeks, 2022).

Quicksort is an in-place sorting algorithm as it required very little extra space apart from the original array (geeksforgeeks.org, 2022).

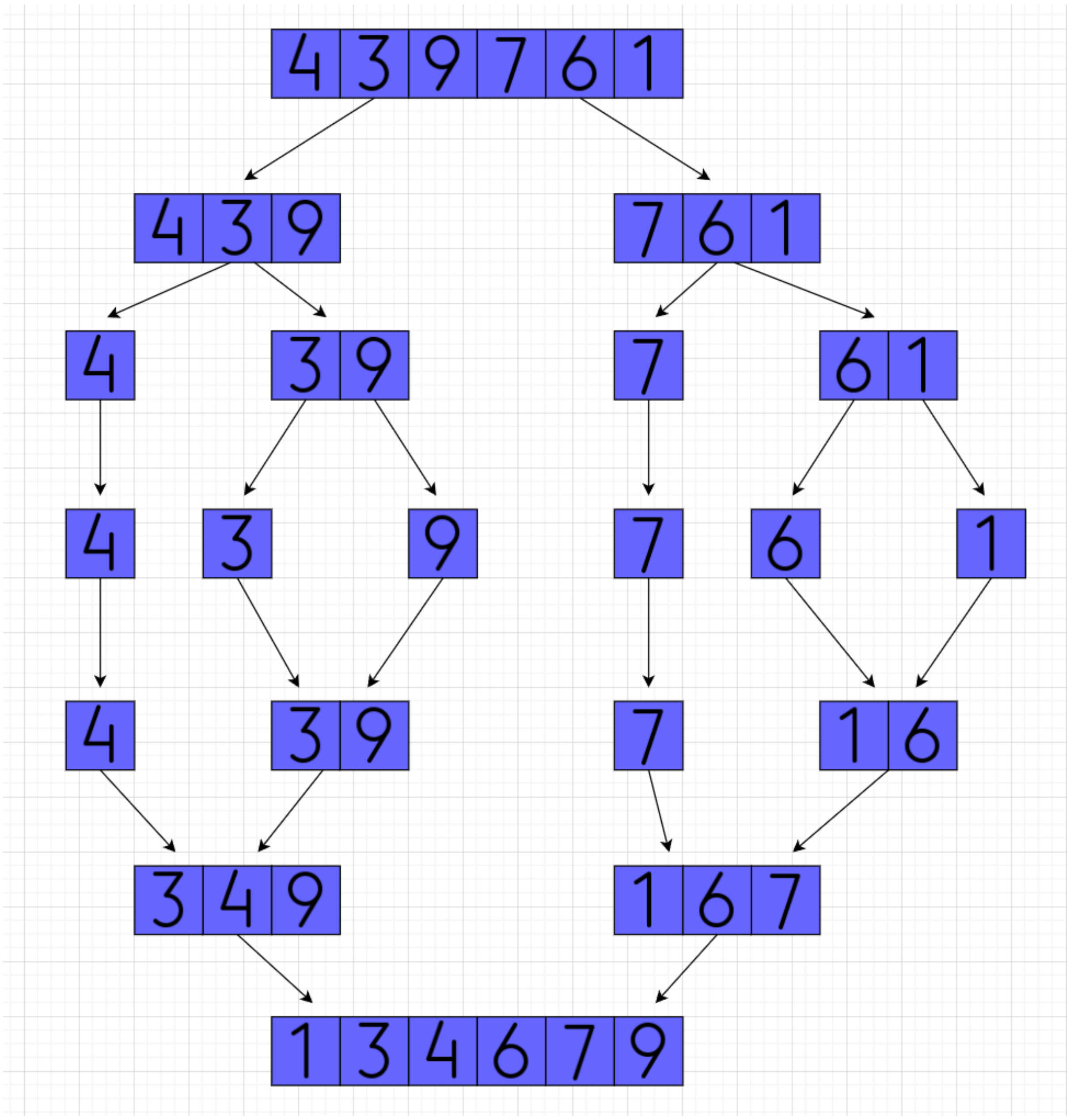
Merge Sort

Merge Sort is a divide and conquer comparison-based sorting algorithm. The array to be sorted is divided into multiple sub-problems and those sub-problems are then divided by the same algorithm into more sub-problems. This procedure goes on recursively until the sub-problems become very easy to solve. At this point, the sub-problems are then combined to solve the original problem (FelixTechTips, Merge Sort In Python Explained, 2020).

The general principles for Merge Sort are:

1. Split array in half.
2. Call Merge Sort on each half to sort them recursively.
3. Merge both sorted halves into one sorted array.

Let’s look at an example of Merge Sort.



The array first split into 2 parts at the mid-point. In this case we have 6 numbers but if there was an odd number, one side would be slightly bigger. The dividing continues recursively until all sub-arrays have just one entry left. This is because when there is just one entry left, the array is already sorted. Now we can start combining them. This is the merging step. When the array size is greater than 1, the leftmost elements are compared in both arrays (programiz.com, N/A).

Merge Sort has O(Nlog(N)) time complexity which is the optimal running time for comparison-based algorithms. This is for all cases (programiz.com, N/A).

The space complexity of merge sort is linear O(n) and it is a stable algorithm (programiz.com, N/A).

Merge Sort is an in-place sorting algorithm because it processes the input and produces an output in the same memory location that contains the input without using any extra space (interviewkickstart.com, N/A).

Implementation & Benchmarking

Bubble Sort

The implementation for the Bubble Sort code has already been described in the introduction section of this report as it was necessary when describing the time complexity for Bubble Sort. Please refer to this section for the implantation of Bubble Sort for this project.

Quicksort

Let's look at the Python code.

```python
def quicksort(arr):
    quicksort_algo(arr, 0, len(arr) - 1)


def quicksort_algo(arr, left, right):
    if left < right:
        partition_pos = partition(arr, left, right)
        quicksort_algo(arr, left, partition_pos -1)
        quicksort_algo(arr, partition_pos + 1, right)

def partition(arr, left, right):
    i = left
    j= right - 1
    pivot = arr[right]

    while i < j:
        while i < right and arr[i] < pivot:
            i += 1
        while j > left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]

    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]

    return i
```

The quicksort function takes in an unsorted sequence. The quicksort_algo function takes in three parameters. The unsorted sequence(array) and the left and right indexes that determine the part of the array that we want to sort. At first, we want to sort the whole array from index 0, so left will be 0 and right will be the length of the array (index -1). If the lengths of the sub-array to sort is just 1, quicksort should do nothing. This is the base case of the recursive function; it checks if left < right. This means that sub-array contains at least 2 elements for Quicksort to activate. Next, we need to call another function called partition. The partition function is declared below quicksort_algo and it takes the same parameters. The partition function returns the index of the pivot element after the first step of Quicksort. When we have this index position saved in the variable partition_pos, we can call Quicksort on the original array from index left to the index partition position -1. This means that we call Quicksort on all elements that are less than the pivot element and we can call Quicksort on the array from partition position +1 to the right. We call Quicksort on the sub-array to the right contains all elements that are greater than the pivot element.

Now we need to define the partition function. First, we have our known indexes. So, i is the left point of the array to sort and j is the element to the right of the pivot. The pivot itself is at index right. As seen in the explanation of how Quicksort works from the Sorting Algorithms section of this report, we need to remember that i moves to the right and j moves to the left until i and j cross. The first while loop checks the condition where i and j cross, simply by checking if i is less than j. Inside the while loop, we move i to the right and j to the left. The second while loop states while i is not at the end of the array and the element at index i is less than the pivot, we increase i. Similarly with j we check if j is greater than left and if the element at index j is greater than the pivot. Now we decrease j because it moved to the left. When both while loops finish, we check if i and j have crossed yet. If they did not cross, we implement a swap. We swap the element at index i with the element at index j.

Not we need to consider what happens when i and j cross. This is the if statement under the inner while loop. After i and j cross, we have a case where the array index i is greater than the pivot, and in this case, we need to swap those elements.

Now we just need to return i because the quicksort_algo function defined earlier needs this i to determine where to split the array to call Quicksort recursively (FelixTechTips, Quicksort In Python Explained, 2020).

Counting Sort

Let's look at the Python code

The

```python
def counting_sort(ary):
    size = len(ary)
    output = [0] * size
    count = [0] * 101
    for no in range(0, size):
        count[ary[no]] += 1
    for no in range(1, 101):
        count[no] += count[no - 1]
    no = size - 1
    while no >= 0:
        output[count[ary[no]] - 1] = ary[no]
        count[ary[no]] -= 1
        no -= 1
    for no in range(0, size):
        ary[no] = output[no]
```

First, we define the size variable which is the length of the list. The output variable stores the sorted array. The count variable counts the number if elements. All the arrays in this project had numbers 0-100. The for loop stores the elements in the count array. The index variable used here is no.

Now we need a second for loop to count the 100 elements. This stores the cumulative count.

Now we need to find the Find the index of each element of the original array in count array and place the elements in output array. The final for loop copies the sorted elements into the original array (programiz.com, N/A).

Insertion Sort

Let’s look at the Python code.

```python
def insertion_sort(arr):
    indexing_length = range(1, len(arr))
    for i in indexing_length:
        value_to_sort = arr[i]

        while arr[i-1] > value_to_sort and i > 0:
            arr[i], arr[i-1] = arr[i-1], arr[i]
            i = i -1
```

There is an outer loop that iterates from the second element to the last element. There is an inner loop that starts at the index of the outer loop and iterates the list to the left and checks if we need to do any swaps and stops if we don’t need to do anymore swaps. This is the while loop. The first condition is our swapping condition. It’s checking if the left neighbour is bigger than the current element. The other condition is that i must be greater than 0 as there is no -1th entry in the list.

Merge Sort

Let’s look at the python code:

```python
def merge(a1,a2):
    c=[]
    x=0
    y=0
    while(x<len(a1) and y<len(a2)):
        if(a1[x]<a2[y]):
            c.append(a1[x])
            x+=1
        else:
            c.append(a2[y])
            y+=1
    while(x<len(a1)):
        c.append(a1[x])
        x+=1
    while(y<len(a2)):
        c.append(a2[y])
        y+=1
    return c

def mergesort(array):
    if(len(array)==1):
      return array
    mid=(len(array))//2
    a1=mergesort(array[:mid])
    a2=mergesort(array[mid:])
    return merge(a1,a2)
```

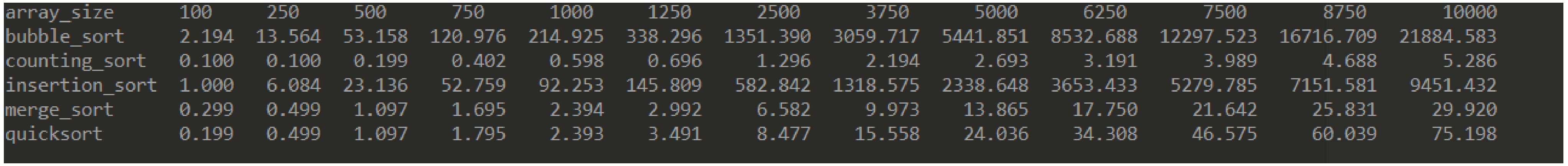
The base case for Merge Sort is that the length of the array to sort is greater than 1. If the array is not greater than 1, the array is already sorted. The recursive part of the algorithm is 2 sub-arrays. One that goes from the beginning of the original array to the mid-point, and one that goes from the mid-point to the end of the array. We have used slicing indexes in the variables a1 and a2 to do this. The // means that we want to divide and get an integer value. The other function merge is used for the merge step as described in the Sorting Algorithms section of this report. We want to compare the leftmost element of one array to the leftmost element of the other array. We will use index x to keep track of the leftmost element in the left array and index y to keep track of the leftmost element in the right array. Now we need to use a while loop. Inside the while loop we will do the comparison. This compares the left index at index x with the right array at index y. When the left array is smaller than the right array at our current indexes, we can save the value of our left array inside our merged array c. Now we must increase x. The other cases are that the right array is smaller than the left array at the current indexes, or they are equal. In this case, we just do the same thing but save the right index at index y in the merged array. We must now increment y (FelixTechTips, Merge Sort In Python Explained, 2020).

If we have looked at every element in the right array and we have transferred every element from the right array to the merged array, there's nothing to compare the left-over elements on the left array with. We must consider the case where we want to transfer every element from the left array to the merged array without considering the right array. In this case, x is still smaller than the length of the left array because there are still elements missing from the left array to transfer to the merged array. We need to transfer them by assigning the left array at index x to the merged array. We then need to increase both indexes (FelixTechTips, Merge Sort In Python Explained, 2020).

The final case is where every element from the left array is already sorted but there are some missing elements of the right array. We check if we are not at the end of the right array and then we just assign every element of the right array to our merged array (FelixTechTips, Merge Sort In Python Explained, 2020).

Benchmarking

To implement the benchmarking code for this project, I created a benchmarking application in Python to time the average runtime for each of the 5 algorithms. I tested each algorithm with randomly generated arrays with numbers in the range from 0-100 of different sizes using NumPy's random package. The array sizes were 100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000 and each array size was tested 10 times. An explanation of how the code works can be seen in the Scrips section of this project. The results of the runtimes were then printed to the console and recorded as seen on the next page.



```
array_size       100     250     500      750      1000     1250      2500      3750      5000      6250       7500       8750      10000
bubble_sort      2.194  13.564  53.158  120.976  214.925  338.296  1351.390  3059.717  5441.851  8532.688  12297.523  16716.709  21884.583
counting_sort    0.100   0.100   0.199    0.402    0.598    0.696     1.296     2.194     2.693     3.191      3.989      4.688      5.286
insertion_sort   1.000   6.084  23.136   52.759   92.253  145.809   582.842  1318.575  2338.648  3653.433   5279.785   7151.581   9451.432
merge_sort       0.299   0.499   1.097    1.695    2.394    2.992     6.582     9.973    13.865    17.750     21.642     25.831     29.920
quicksort        0.199   0.499   1.097    1.795    2.393    3.491     8.477    15.558    24.036    34.308     46.575     60.039     75.198
```

array_size	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**3750**	**5000**	**6250**	**7500**	**8750**	**10000**
bubble_sort	2.194	13.564	53.158	120.976	214.925	338.296	1351.390	3059.717	5441.851	8532.688	12297.523	16716.709	21884.583
counting_sort	0.100	0.100	0.199	0.402	0.598	0.696	1.296	2.194	2.693	3.191	3.989	4.688	5.286
insertion_sort	1.000	6.084	23.136	52.759	92.253	145.809	582.842	1318.575	2338.648	3653.433	5279.785	7151.581	9451.432
merge_sort	0.299	0.499	1.097	1.695	2.394	2.992	6.582	9.973	13.865	17.750	21.642	25.831	29.920
quicksort	0.199	0.499	1.097	1.795	2.393	3.491	8.477	15.558	24.036	34.308	46.575	60.039	75.198

Table 3: Results table – Average of 10 runs with increasing size arrays

The running time and array sizes were then plot on a graph using the python package Matplotlib as seen below.

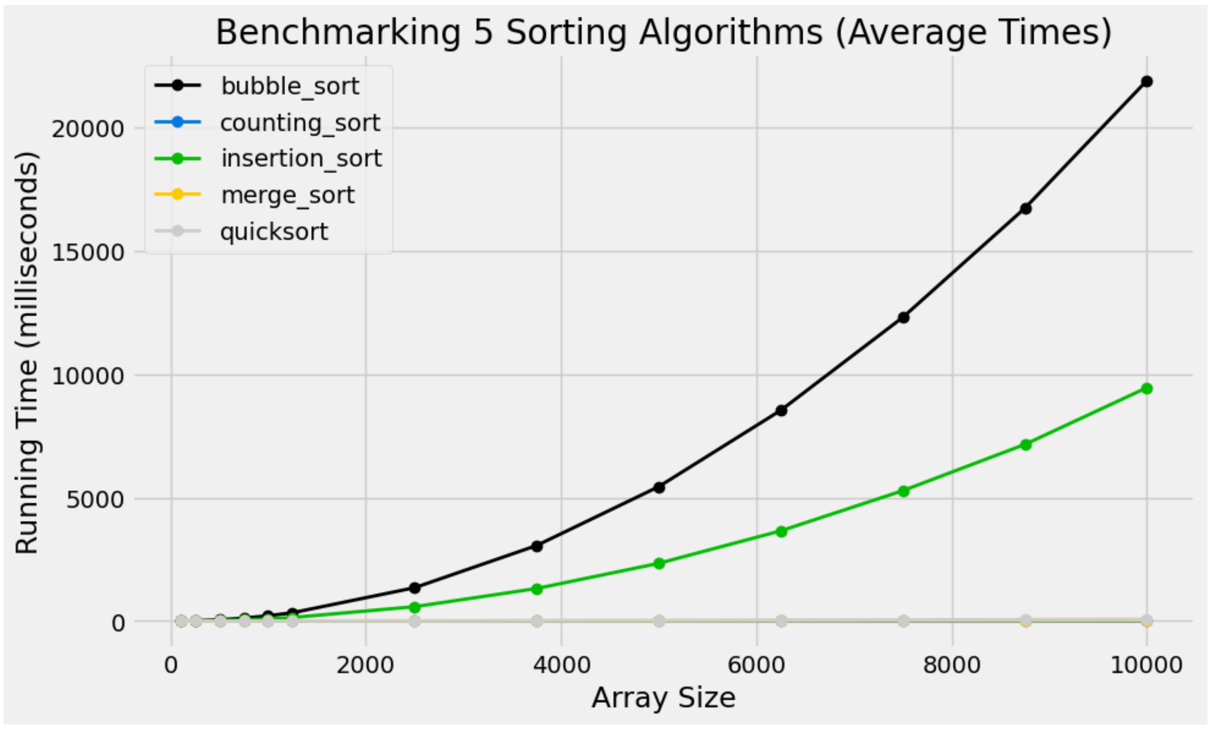


Table 4: Graph of the average runtimes in milliseconds

From these results we can see that Bubble sort took over 20 seconds to sort an array with 10,000 integers. Counting Sort did this in 5 milliseconds. As the array sizes increased, Bubble and Insertion Sort's runtime went up exponentially compared to the other three. Insertion Sort's runtime was about half that of Bubble Sort. Merge Sort, Quicksort and Counting Sort took about the same time overall. The plot illustrates that Bubble and Insertion Sort have an n^2 rate of growth. Since Bubble Sort has a much slower runtime than the rest of the algorithms, it is difficult to distinguish Merge Sort, Quicksort and Counting Sort. However, we can tell from the results table (table 3) that Merge Sort and Quicksort are growing logarithmically and counting sort is growing linearly.

These results are not surprising as they are in-line with what was discussed in terms of time complexity earlier in this report. An interesting result was that Quicksort had a logarithmic growth. This is because the worst case was $O(n^2)$ which is extremely rare. The best case is also extremely rare and so this seemed to average out to a O(nlogn) runtime.

References

algotree.org. (N/A, N/A N/A). *QUICKSORT ALGORITHM*. Retrieved from algotree.org: https://algotree.org/algorithms/sorting/quicksort/

askinglot.com. (2020, March 4th). *Why quick sort is not stable?* Retrieved from askinglot.com: https://askinglot.com/why-quick-sort-is-not-stable

askpython.com. (N/A, N/A N/A). *Bubble Sort in Python*. Retrieved from askpython.com: https://www.askpython.com/python/examples/bubble-sort-in-python

Bais, V. S. (N/A, N/A N/A). *Time and Space complexity of Quick Sort*. Retrieved from iq.opengenus.org: https://iq.opengenus.org/time-and-space-complexity-of-quick-sort/

Computational Core. (N/A, N/A N/A). *QUICKSORT TIME COMPLEXITY*. Retrieved from textbooks.cs.ksu.edu: https://textbooks.cs.ksu.edu/cc310/7-searching-and-sorting/20-quicksort-time-complexity/

FelixTechTips. (202, June 12). *Insertion Sort In Python Explained*. Retrieved from FelixTechTips-YouTube: https://www.youtube.com/watch?v=R_wDA-PmGE4

FelixTechTips. (2020, June 20). *Merge Sort In Python Explained*. Retrieved from FelixTechTips - YouTube: https://www.youtube.com/watch?v=cVZMah9kEjI

FelixTechTips. (2020, June 27). *Quicksort In Python Explained*. Retrieved from FelixTechTips - YouTube: https://www.youtube.com/watch?v=9KBwdDEwal8

gatevidyalay.com. (N/A, N/A N/A). *Bubble Sort Algorithm | Example | Time Complexity*. Retrieved from gatevidyalay.com: https://www.gatevidyalay.com/tag/bubble-sort-space-complexity/

GeeksforGeeks. (2022, May 11). *Bubble Sort*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/bubble-sort/

GeeksforGeeks. (2022, Jan 19). *Python Program for Bubble Sort*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/python-program-for-bubble-sort/

GeeksforGeeks. (2022, Mar 02). *Sorting Algorithms in Python*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/sorting-algorithms-in-python/

GeeksforGeeks. (N/A, N/A N/A). *QuickSort*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/quick-sort/

geeksforgeeks.org. (2019, September 26). *Stability in sorting algorithms*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

geeksforgeeks.org. (2022, May 11). *Insertion Sort*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/insertion-sort/

Gupta, S. (N/A, N/A N/A). *Time and Space Complexities of Sorting Algorithms Explained*. Retrieved from Interview Kickstart: https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms

interviewkickstart.com. (N/A, N/A N/A). *Learn In-Place Merge Sort*. Retrieved from interviewkickstart.com: https://www.interviewkickstart.com/learn/in-place-merge-sort interviewkickstart.com. (N/A, N/A N/A). *Time and Space Complexities of Sorting Algorithms Explained*. Retrieved from interviewkickstart.com: https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms

javatpoint.com. (N/A, N/A N/A). *Counting Sort Algorithm*. Retrieved from javatpoint.com: https://www.javatpoint.com/counting-sort

javatpoint.com. (N/A, N/A N/A). *Insertion Sort Algorithm*. Retrieved from javatpoint.com: https://www.javatpoint.com/insertion-sort

Joshi, V. (2017, May 8). *Sorting Out The Basics Behind Sorting Algorithms*. Retrieved from medium.com: https://medium.com/basecs/sorting-out-the-basics-behind-sorting-algorithms-b0a032873add#:~:text=There%20are%20many%20ways%20to,sort%20or%20non-comparison%20sort

Khan Academy. (N/A, N/A N/A). *Big-O notation*. Retrieved from khanacademy.org: https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation

Pedamkar, P. (N/A, N/A N/A). *Bubble Sort in Python*. Retrieved from educba.com: https://www.educba.com/bubble-sort-in-python/

planetmath.org. (N/A, N/A N/A). *in-place sorting algorithm*. Retrieved from planetmath.org: https://planetmath.org/inplacesortingalgorithm

programiz.com. (N/A, N/A N/A). *Counting Sort Algorithm*. Retrieved from programiz.com: https://www.programiz.com/dsa/counting-sort

programiz.com. (N/A, N/A N/A). *Merge Sort Algorithm*. Retrieved from programiz.com: https://www.programiz.com/dsa/merge-sort

Sambol, M. (2017, January 21). *Big-O notation in 5 minutes — The basics*. Retrieved from Michael Sambol: https://www.youtube.com/watch?v=__vX2sjlpXU

Serrill, D. (2019, September 9). *Quick Sort Algorithm Explained*. Retrieved from Derrick Serrill - YouTube: https://www.youtube.com/watch?v=kFeXwkgnQ9U&t=204s

Sherrill, D. (2019, September 4). *Python-Algorithms-YouTube-Series*. Retrieved from github.com: https://github.com/Derrick-Sherrill/Python-Algorithms-YouTube-Series/blob/master/1-bubblesort.py

simplilearn.com. (2021, October 19). *Counting Sort Algorithm: Overview, Time Complexity, Implementation in C & More*. Retrieved from simplilearn.com: https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm

Sorting Algorithms in Python. (2020, April 15). Retrieved from Real Python: https://realpython.com/sorting-algorithms-python/#the-importance-of-sorting-algorithms-in-python

Srivastava, P. (2021, May 21). *Stable Sorting Algorithms*. Retrieved from baeldung.com: https://www.baeldung.com/cs/stable-sorting-algorithms

Time Complexity of Sorting Algorithms. (N/A, N/A N/A). Retrieved from javaTpoint: https://www.javatpoint.com/time-complexity-of-sorting-algorithms

Upadhyay, S. (2022, February 21). *Bubble Sort Algorithm: Overview, Time Complexity, Pseudocode and Applications*. Retrieved from www.simplilearn.com: https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm

Wikipedia. (N/A, N/A N/A). *Big O notation*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Big_O_notation