Conceptual Thinking with Algorithms – Project

Benchmarking Sorting Algorithms – Report

Introduction

An algorithm is a sequence of logical instructions which are *"...precisely define a sequence of operations"* for carrying out a task [1]. An everyday example of an algorithm would be following a cooking recipe.

The features of a well-designed algorithm are [6]:

- **Input:** An algorithm has 0 or more well-defined inputs, i.e., data which is passed to the algorithm begins.
- **Output:** An algorithm has one or more well-defined outputs i.e., data which is produced after an algorithm has completed its task.
- **Defined Limit:** An algorithm has a defined boundary.
- **Explicit:** Each step is clearly defined.
- **Correctness:** The algorithm should consistently provide a correct solution (or a solution which is within an acceptable margin of error)
- **Feasibility:** It should be feasible to execute the algorithm using the available computational resources
- **Efficiency:** An algorithms execution should take place in a reasonable timeframe.

For this project we will be focusing on **sorting algorithms.**

What Is a Sorting Algorithm?

Sorting as defined by Collins English Dictionary is *"the process or operation of ordering items and data according to specific criteria"* [2]. Sorting often is one of the key steps in helping to simplify a task. It helps humans and computers search and find item quickly. For example, in my current line of work having employee names in alphabetical order helps streamline the checking of payments. Sorting has 2 common meanings [3]:

1. **Ordering:** This is organising items in a sequence by a criterion: For example, searching for a contact in your phone, searching for a record in a shop is made easier by the items being in alphabetical order. Another example is the Premier League table, we can straight away see which team has the most and least number of points as it is ordered.
2. **Categorising:** This is grouping items together which have comparable properties. For example, when building Lego grouping the same bricks together or in a shop displaying fruit and veg together etc.

In computing sorting algorithms order elements in a list or array, usually in ascending or descending order [4]. As sorting aids in reducing a problems complexity is in an important algorithm. Sorting algorithms help make computation tasks easier, organise unclean data thus making it easier to handle and work with. And often used with other types of algorithms such as searching algorithms [5].

The Complexity of Sorting Algorithms

"In choosing to sort an algorithm to solve a problem, you are trying to predict which algorithm will be fastest for a particular data set on a particular platform (or family of platforms)" [6]. A well-designed algorithm should use its computing resources efficiently and provide the correct output for a given input. Computer algorithms must have precise instructions so a computer can follow. This means inputs, steps to execute and outputs must be well defined and that it terminates after execution.

Just as there are many ways to bake a cake, there are many ways to design an algorithm. How do we know which is the correct algorithm to choose? The following are the main points to consider when deciding on which algorithm to choose:

- Efficiency.
- Performance.
- Complexity.
- The desired output.
- The input.

"An algorithm is considered efficient if its resource consumption, also known as computational cost, is at or below some acceptable level" [19]. An acceptable level is one which, means it will run in a reasonable amount of time or space on an available computer, usually as a function of the input size [19]. To reach an algorithms peak efficiency we must try to minimise its resource usage. To choose the correct algorithm for a task, we need to be able to measure how efficient it is. Several factors can affect the efficiency of an algorithm such as machine hardware, operating systems, compiler quality, etc. The main cornerstone of testing an algorithms performance are its time and space complexity i.e., how fast an algorithm can run and the amount of space it takes to run.

When discussing the analysis of an algorithms we consider the [16]:

- **Best Case:** When an algorithms performance takes the least time and space for a given input. It provides the lower bound of an algorithms complexity and thus is not consider the best option consider when analysing complexity.
- **Worst Case:** When an algorithms performance takes the most time and space for a given input. It provides the upper bound of an algorithms complexity and is often used as a measure of an algorithms efficiency as it allows us to see how badly an algorithm can perform.
- **Average Case:** When an algorithms performance is between the best and worst case for a given input The average case is widely used when analysing algorithms as it averages out both best and worst-case performances.

Space Complexity

Space Complexity is the total space which an algorithm takes up in relation to the input size passed. It is made up of both input space and auxiliary space. Auxiliary Space is temporary space used by an algorithm during its runtime [18].

Time Complexity

Time complexity provides information on an algorithm's runtime. It refers to the measure of time it takes an algorithm to carry out and complete its defined processes such as binary operations, control flow, etc. [15]. It can be affected by the number of processes an algorithm must perform [9] and is correlated to input size. When the input size increases so does the runtime [10].

Asymptotic Notations

Asymptotic Notations let us know how good one algorithm is compared to another. It is a method to describe an algorithms efficiency [13]. It takes into considerate how an algorithms performance changes depending on input size as differing sizes tend to effect performance; an increase in input size, will likely change performance. Asymptotic Notations are mathematical notations which help analyse an algorithms runtime complexity [13]. It is focused on what affect an algorithm's growth rate of (n).

There are three types of asymptotic notation:

1. Big-O Notation (O): upper bound of an algorithm's running time.
2. Omega Notation (Ω): lower bound of an algorithm's running time.
3. Theta Notation (Θ): exact asymptotic behaviour of an algorithm's running time (lies between the upper and lower bound).

Big O Notation

Big O Notation is a relative representation an algorithms complexity [11]. It describes the worst-case running time i.e., the longest time the algorithm could take to complete. Big O Notation represents the upper bound running time complexity. It is denoted as O(n) with *n* representing the input size. As Big O Natation provides the worst-case running time for an algorithm, it is the most used way to analyse an algorithm. To compute the Big-O of an algorithm, we take the number of iterations an algorithm makes in the worst-case scenario with an input of n [12].

Omega Notation

Omega Notation shows the lower bound of an algorithm's runtime i.e., the best-case/minimum amount of time and space resources needed for an algorithm to execute. Omega Notation is denoted as Ω(n) with *n* representing the input size.

Theta Notation

Theta Notation represents the average case of an algorithm. It shows the function from above and below i.e., lies between the upper bound (Big O) and lower bound (Big Omega). Theta Notation shows the average number of operations that an algorithm deploys to solve a problem over all inputs of a comparable size [17]. It is denoted as Θ(n) with *n* representing the input size.

Performance

When we talk about an algorithm's performance, we are talking about how much resources (memory, time, disk space etc.) are used when a programme runs. This is dependent on real-world factors such as, RAM, the complier used, the computer, disk space etc. Performance is a practical measurement of an algorithm whereas complexity can be thought of as a theoretical measurement which is not affected by real life factors such as the computer [19]. **An algorithms complexity affects performance but not visa-versa.** When it comes to analysing an algorithm, we only consider its space and time complexity as performance is limited when it comes to generally describing an algorithm.

When discussing an algorithm performance, we are trying to anticipate the resources required for an algorithm to execute successfully. [19]. By conducting a performance analysis, it enables us to choose the best algorithm to solve a problem. We perform a performance analysis by comparing the various algorithms which solve the same problem to each other. Factors to consider for an algorithm's performance [19]:

- Is it providing a precise solution?
- Simple to understand and implement?
- Is extra memory needed?
- Time constraints.

There are two ways to analyse an algorithms efficiency:

1. A Priori Analysis: This is a theoretical analysis of an algorithm. A Priori analysis independent of language compliers and hardware. It uses asymptotic notation and

provides an approximate answer of an algorithm’s complexity. It is done prior to an algorithm running on a system [7].

2. A Posteriori Analysis: This analysis is dependent on the compiler language and hardware. It provides statistics about an algorithm’s consumption of space and time. A Posterior analysis provides an exact answer and does not use asymptotic notation [7]. The posteriori analysis of the algorithm is done after it has run on a system and is system dependent, meaning analysis will vary between systems [8].

In-Place Sorting

No extra space is needed for an in-place algorithm. The same memory space is used for the input and output (note: minor extra space for variables is permitted) [20]. The idea is to produce an output in the same memory space that contains the input by successively transforming that data until the output is produced. This avoids the need to use twice the storage - one area for the input and an equal-sized area for the output. This means that the input and output occupy the same memory storage space. There is no copying of input to output, and the input ceases to exist unless you have made a backup copy. Selection Sort is an example of an in-place sort. Out of place sorting, is the opposite and these algorithms require extra space for sorting. Merge sort is an example of out of place sorting.

Stable Sorting

A stable sorting sorts matching elements (e.g., 1 and 1) in the same they appeared in the input. A stable sort is one that, for elements that compare equal, their relative position in the sorted output is guaranteed to be the same as in the source. Stable algorithms have higher big-O CPU and/or memory usage than unstable algorithms [21]. If you do not need stability, you can use a fast, memory-sipping algorithm. Merge Sort is an example of a stable sort.

Unstable Sort

An unstable sort does not guarantee to retain the relative position order of matching elements in the outputted sorted array. Quick Sort is an example of an unstable sort.

Adaptable Sorting

Adaptable sorting is when an algorithm takes advantage of instances where an input is already partially sorted, thus reducing *"…its requirements for computational resources as a function of the disorder in the input"* [43]. On the other hand, a non-adaptive sorting algorithm is one which passes no remarks to whether an input is partially sorted or not.

Comparator Functions

A comparator function is passed two arguments and returns the order they should be sorted in. when sorted [22]. For an algorithm to be able to sort, it must be able to compare elements to other elements in a group, to decide on its ordering. Many sorting algorithms rely on the correct comparison function being selected which in turn returns the ordering between two elements [23]. For example, comparing the elements of an array with each other to determine which should be ordered first in a returned array. E.g., element_1 > element_2.

Comparator functions are simple to use when the data is strings as the elements can be sorted lexicographically or numbers (integers and floats) which can be sorted numerically. Other data for example: sorting a menu by starters, mains, dessert, and drinks will need a custom comparator as the items are not being sorted in a numerical or lexicographic way.

Using comparator functions in an algorithm can increase their time efficiency and make them more effective.

Comparison Based Sorts

In a comparison-based sort, all elements are compared against each other, no assumptions about the data are made and they have the lower bound nlogn complexity [24]. The best-case space complexity is O(1) as it is possible to sort an array n place i.e., forgoing additional memory [25]. An example of a comparison-based sorting algorithm is quicksort.

Non-Comparison Based Sorts

On the other hand, non-comparison-based sorting algorithms perform a sort without comparing elements, this is because they make assumptions about the data [25]. They for the most part faster than comparison-based sorting algorithms as elements aren't compared. Non-Comparison sorting algorithms use the internal characters to sort the elements into the correct order. Non-comparison-based algorithms have a speed limit of O(n) i.e., linear time and the space complexity is always O(n) [25]. An example of a non-comparison sort is counting sort.

Recursive Based Sort

A recursive sorting algorithm calls on itself to sort a smaller part of the array, then combining the partially sorted results [26]. *"A recursive system is a system in which current output depends on previous output(s) and input(s)"* [26]. An example of a recursive sorting algorithm is quick sort.

Non-Recursive Based Sort

A non-recursive Based Sort is a sorting algorithm which does not use recursion i.e., it does all sorting once without calling itself [26]. In a *"…non-recursive system current output does not depend on previous output(s)"* [26]. An example of a non-recursive Based Sort is bubble sort.

Sorting Algorithms

1. Insertion Sort (A Simple Comparison Based Sort)

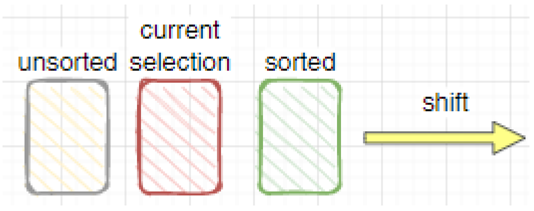
A popular way to visualise insertion sort is to imagine a card game where you are delt one card at a time. Every time you are dealt a card you insert it in the correct position [27]. In every iteration choose one element from the unsorted section of the array and insert it into the correct position in the sorted section. Insertion Sort differs from other algorithms in that it does not “swap” elements, rather it shifts elements which are greater to the right, leaving room on the left to insert the smaller element. This is done by copying to the right, overriding the current smallest element, which is why it is stored as a variable. Technically, the element is now stored in two positions but conceptually the position at that index is empty, meaning we can store the variable in the correct position [27]. It is an in-place sorting algorithm. As discussed earlier this means that, the output is stored in the same memory as the input.

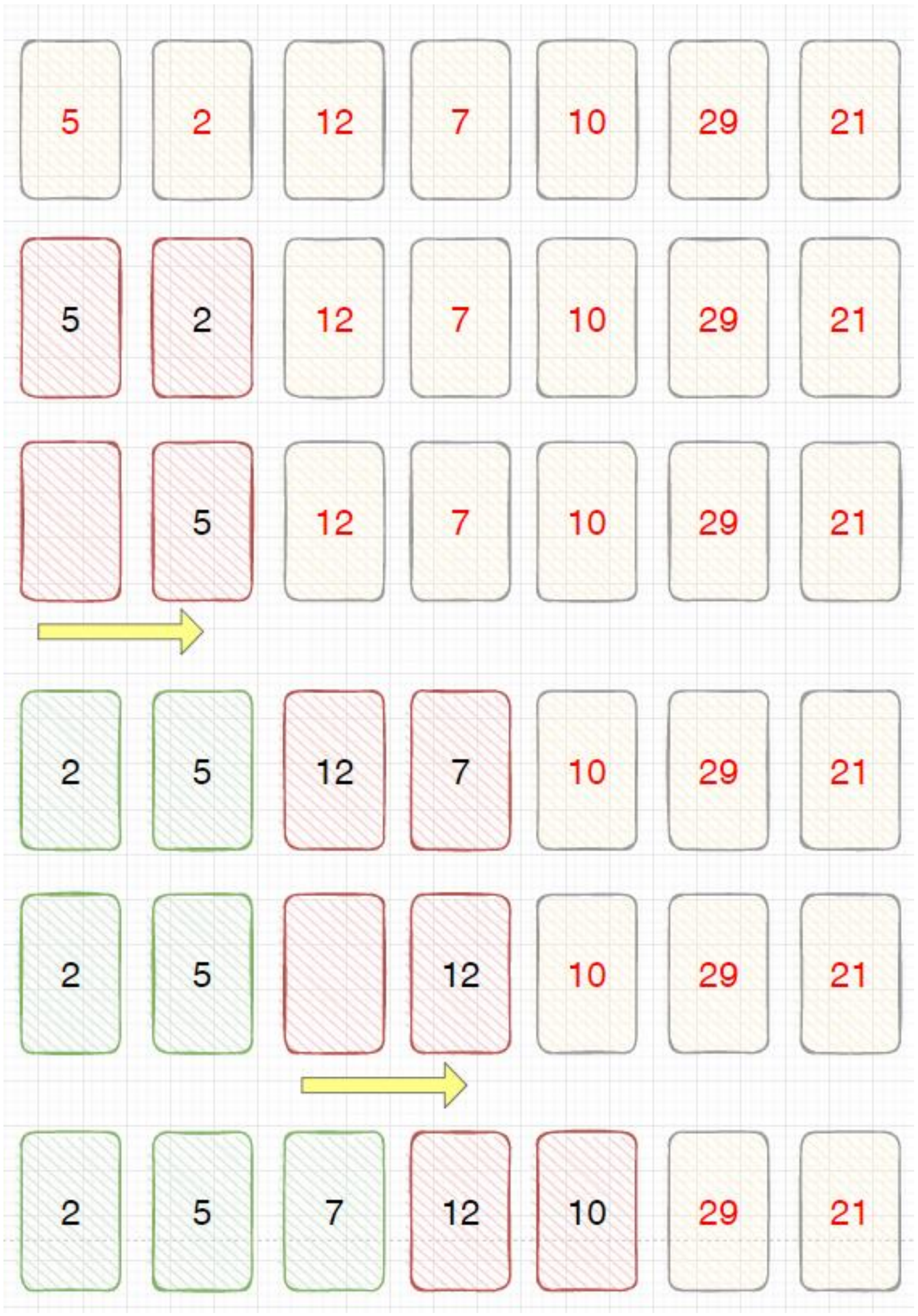
Working Principle

1. Assume the first element is in the correct position as it is the only one, we have seen.
2. Compare the element with its adjacent element.
3. If at every comparison, we could find a position in sorted array where the element can be inserted, then create space by shifting the elements to right and insert the element at the appropriate position.
4. Repeat the above steps until you place the last element of unsorted array to its correct position.

Diagram

Key:





Unsorted array.

Select the first element and assume it is sorted.

Compare 5 & 2.

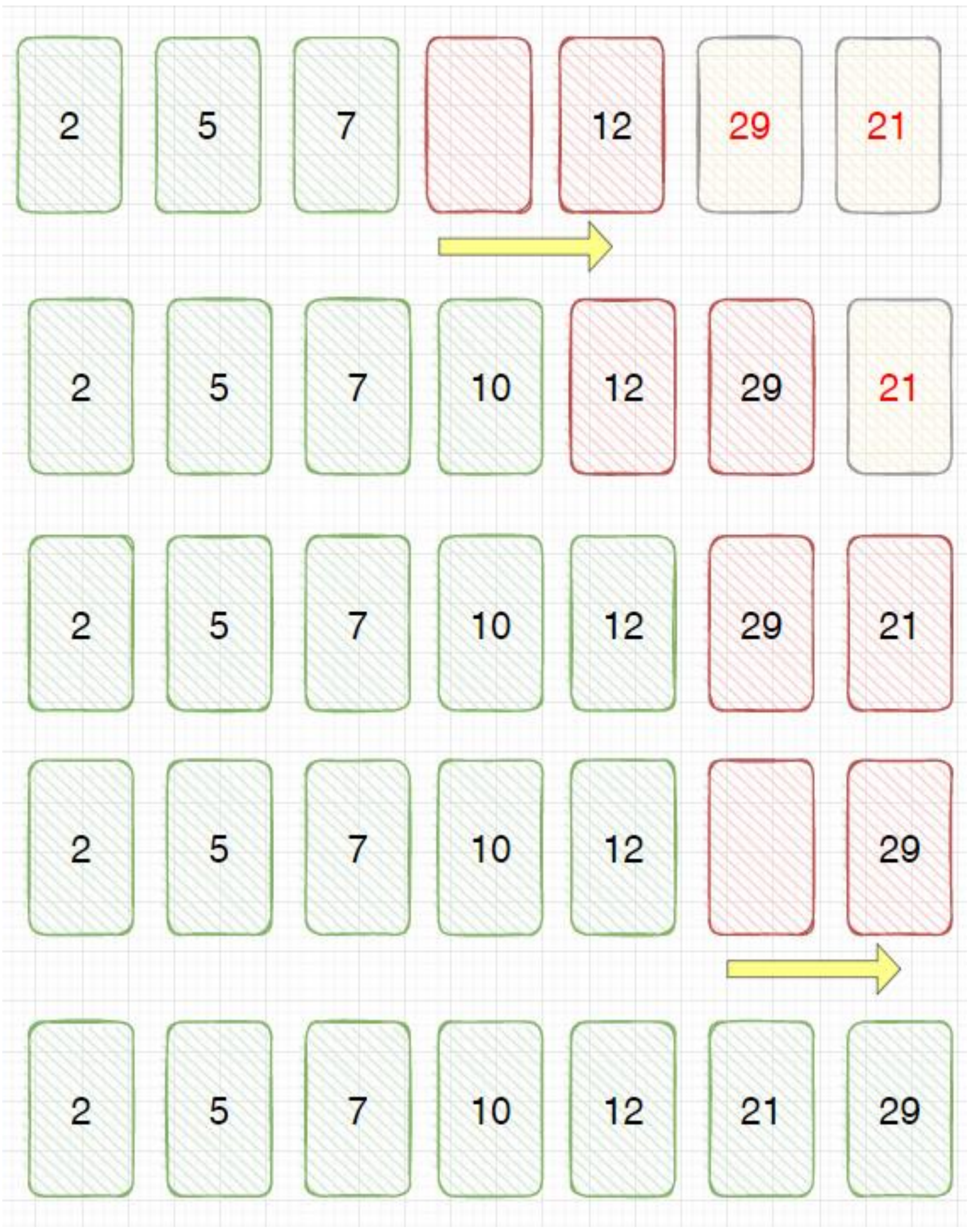
5 > 2. Shift 5 to the right and insert 2 to the space on the left.

5 < 12. No swap needed.

Next compare 12 & 7.

12 > 7. Shift 12 to the right and insert 7 to the space on the left.

Compare 12 & 10.



12 > 10. Shift 12 to the right & insert 10 to the space on the left.

Now compare 12 & 29.

12 < 29. No swap needed.

Next compare 29 & 21.

29 > 21. Shift 29 to the right and insert 21 to the space on the left.

Array is now sorted.

Complexity

Time Complexity	• <u>Best Case:</u> This would be when an array is already sorted. This means only the outer loop runs so they are only n number of comparisons. Ω(n) – Linear. • <u>Worst Case</u>: This would be when an array is sorted in descending order. As each element must be compared there must be n-1 comparisons. In this case the time complexity is O(n^2) - Quadratic. • <u>Average Case:</u> This is when the array is unsorted in descending or ascending. In this case the time complexity is Θ (n2) - Quadratic.
Space Complexity	As the array is sorted in place, no extra space is needed. The array is being simply being rearranged. Only a single additional memory space is required. O(1).
Stable	Yes. Elements are shifted to the correct position meaning the relative order of equal elements is kept.
Adaptable	Yes. It takes advantage if an input is partially sorted.

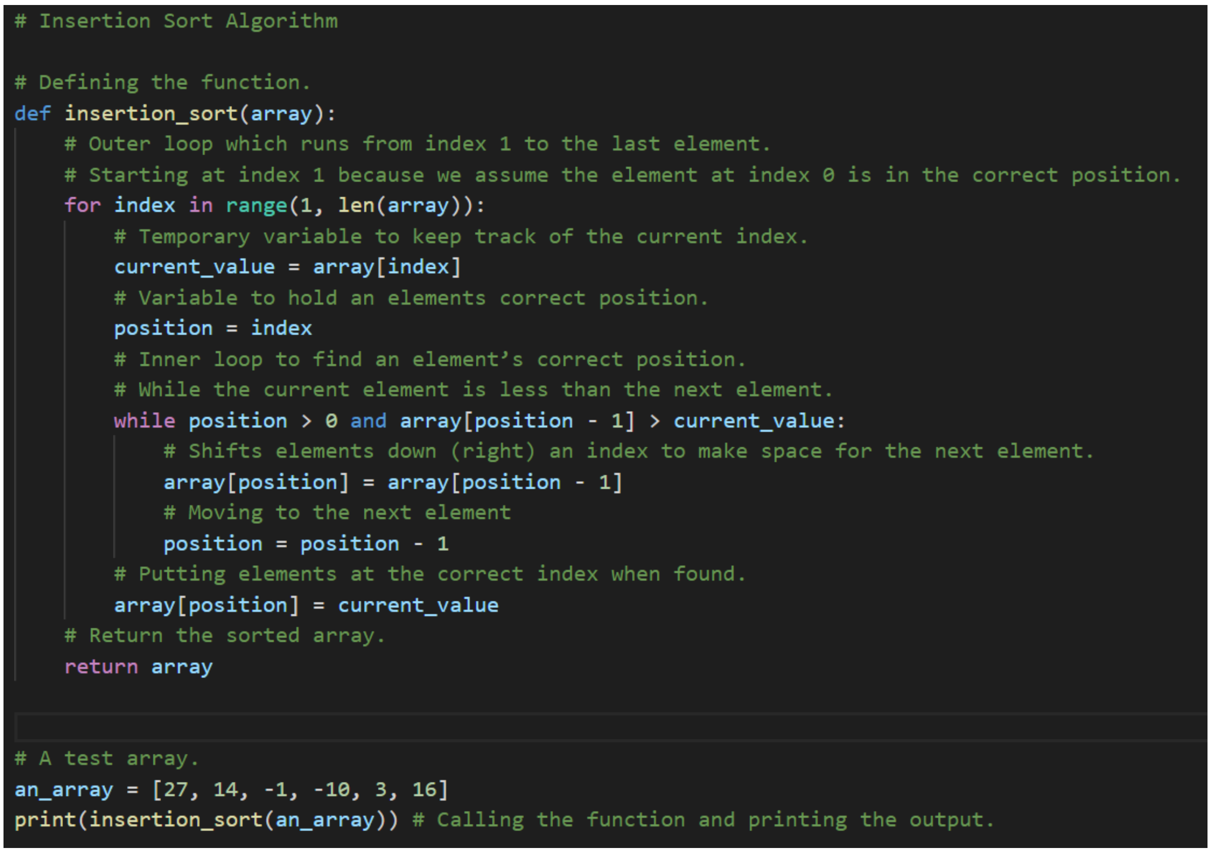
Use insertion sort when:

- When the array is nearly sorted - since insertion sort is adaptive.
- When we have memory usage constraints.
- When a simple sorting implementation is desired.
- When the array to be sorted is relatively small.

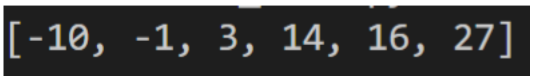
Avoid insertion sort when:

- The array to be sorted has many elements.
- The array is completely unsorted.
- You want a faster run time and memory is not a concern.

Code



```python
# Insertion Sort Algorithm

# Defining the function.
def insertion_sort(array):
    # Outer loop which runs from index 1 to the last element.
    # Starting at index 1 because we assume the element at index 0 is in the correct position.
    for index in range(1, len(array)):
        # Temporary variable to keep track of the current index.
        current_value = array[index]
        # Variable to hold an elements correct position.
        position = index
        # Inner loop to find an element's correct position.
        # While the current element is less than the next element.
        while position > 0 and array[position - 1] > current_value:
            # Shifts elements down (right) an index to make space for the next element.
            array[position] = array[position - 1]
            # Moving to the next element
            position = position - 1
        # Putting elements at the correct index when found.
        array[position] = current_value
    # Return the sorted array.
    return array


# A test array.
an_array = [27, 14, -1, -10, 3, 16]
print(insertion_sort(an_array)) # Calling the function and printing the output.
```

Output: 

```
[-10, -1, 3, 14, 16, 27]
```

2. Bubble Sort (A Non-Comparison Sort)

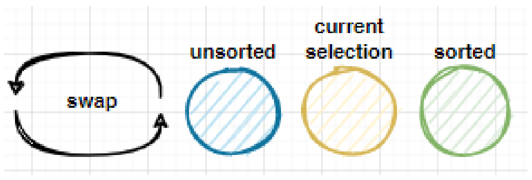
Bubble sort is mainly used as a teaching tool to understand the foundation of sorting [28]. It is an in-place algorithm which goes through an array comparing adjacent elements, swapping them if they are in the incorrect position. The highest number "bubbles" itself to the correct position like a *"…bubble rises up to the water surface"* [29]. We could visualise this by sorting people in a line from smallest to tallest or youngest to oldest. Elements are sorted by comparing them to each other one by one. Bubble Sort is one of the least efficient algorithms as it goes through each element in an input to find its correct position. It is a time and space consuming algorithm. Bubble sort will iterate though an array as many times as there are elements (n-1) until such a time there is no swaps. This happens when the array is sorted [28].

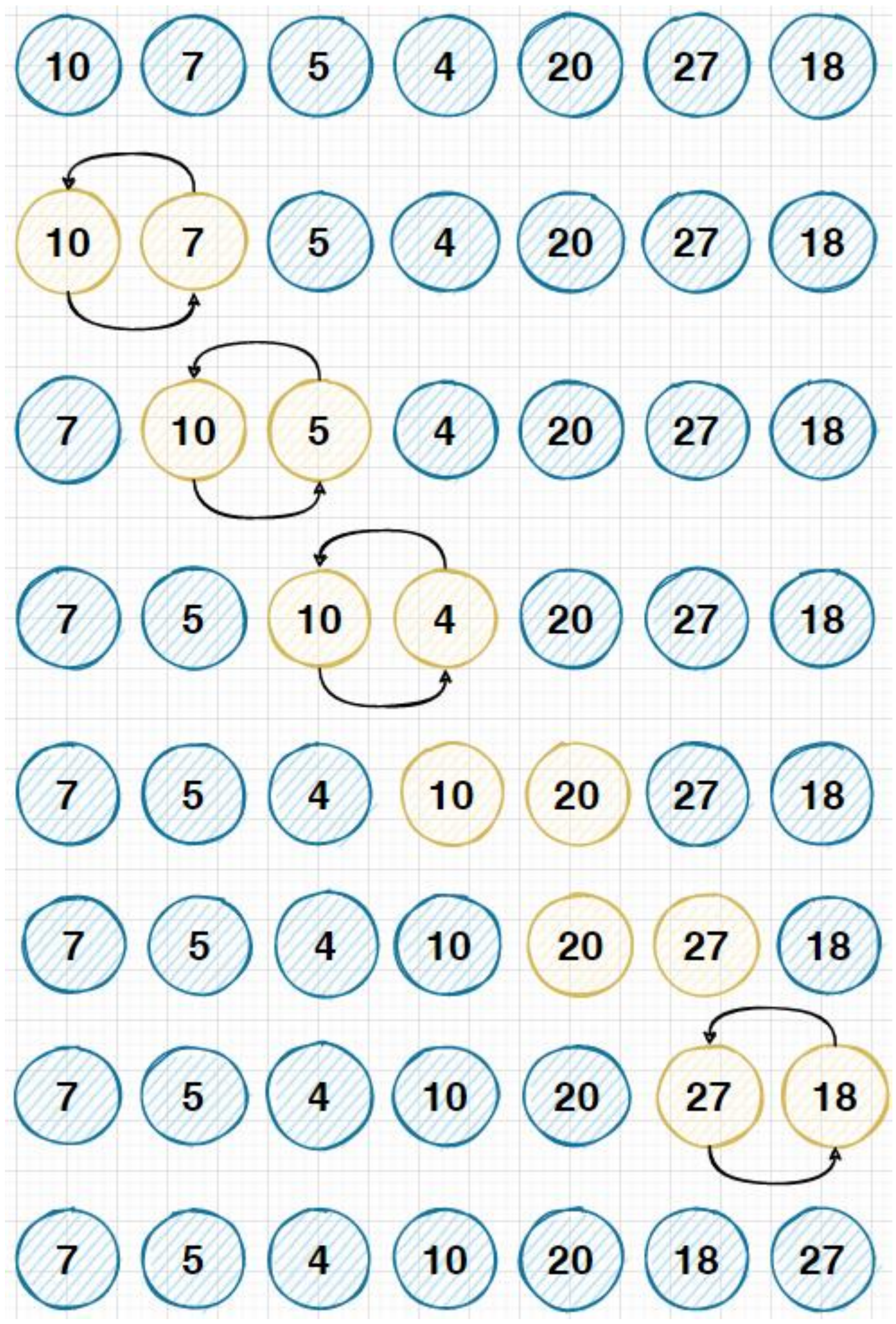
Working Principal

1. Iterate through the unsorted array comparing each pair to the next element.
2. If the adjacent elements are in the wrong order, swap them.
3. Repeat steps 1 and 2 from the beginning of the array till the entire array is sorted.

Diagram

Key:





Unsorted array.

Compare 10 & 7.
10 > 7, swap elements.

Compare 10 & 5.
10 > 5, swap elements.
Relative to 10, 5 is in the correct position.

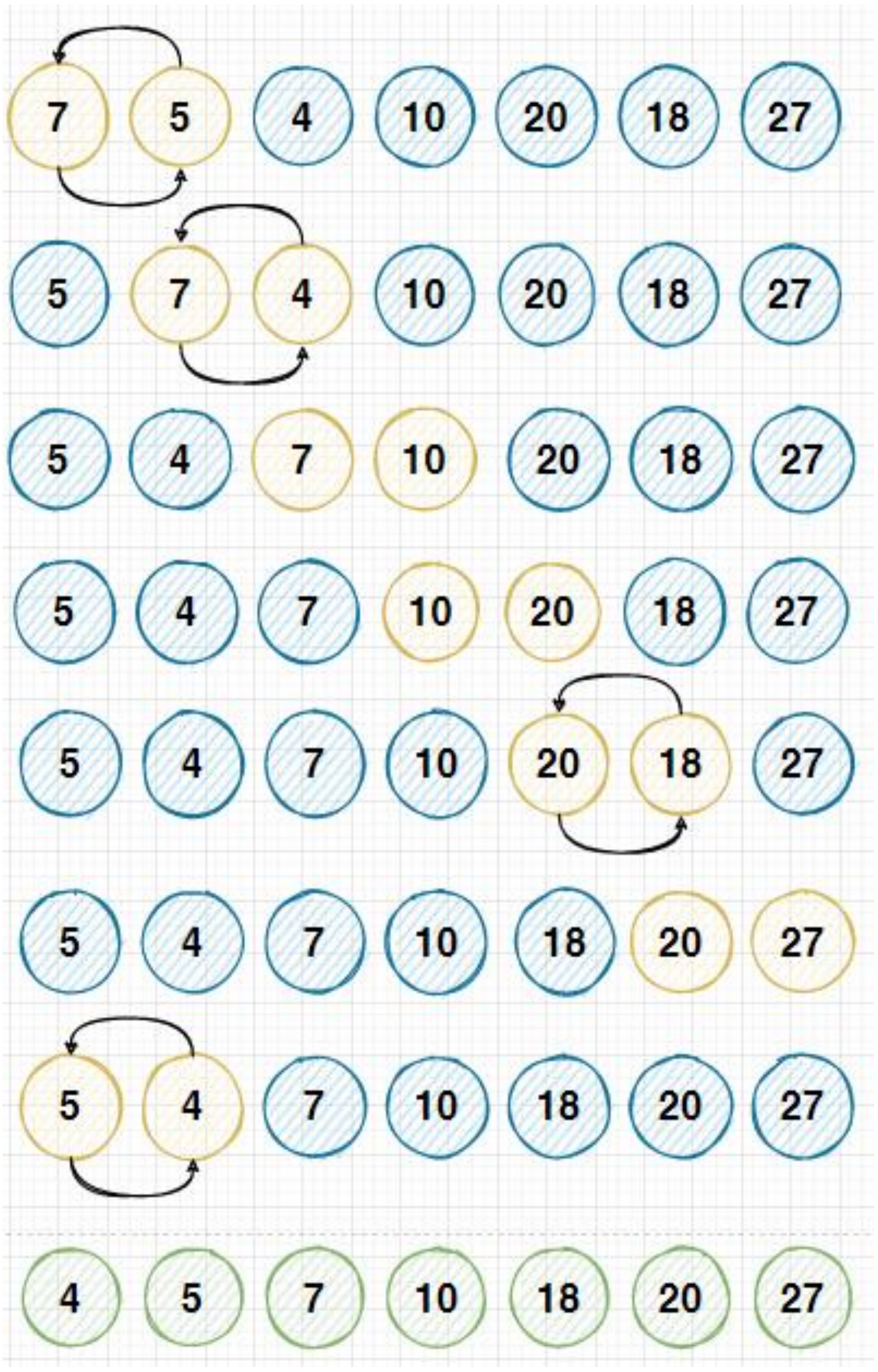
Compare 10 & 4.
10 > 4, swap elements.
Once again, relative to 10, 4 is in the correct position.

Compare 10 & 20.
10 < 20. No swap needed.

Compare 20 & 27.
20 < 27, no swap needed.

Compare 27 & 18.
27 > 18, swap elements.

Partially sorted array at the end of the first iteration.



Compare 7 & 5.
7 > 5, swap elements.

Compare 7 & 4.
7 > 4, swap elements. Relative to 7, 4 is in the correct position.

Compare 7 & 10.
7 < 10, no swap needed.

Compare 10 & 20.
10 < 20, no swap needed.

Compare 20 & 18.
20 > 18, swap elements.

Compare 20 & 27.
20 < 27, no swap needed. This is the end of the second iteration.

Start the third iteration.
Compare 5 & 4.
5 > 4, swap elements.

No other swapping of elements will occur for the rest of the third iteration. This is the final sorted array.

Complexity

Time Complexity	Best Case: This would be when a list is an array is already sorted. Meaning no swapping. Ω(n). Worse Case: This would be when the smallest element is in the last index. This means the sort will need to make the maximum number of passes. O(n^2). Average Case: The number of comparisons is constant. Irrespective of element order there is ϴ (n^2) comparisons.
Space Complexity	As the array is sorted in place, no extra space is needed. The array is being simply being rearranged. Only a single additional memory space is required. O(1).
Stable	Yes. Elements which are equal are NOT re-arranged in the final sort order relative to one another
Adaptable	Yes. It takes advantage if an input is partially sorted.

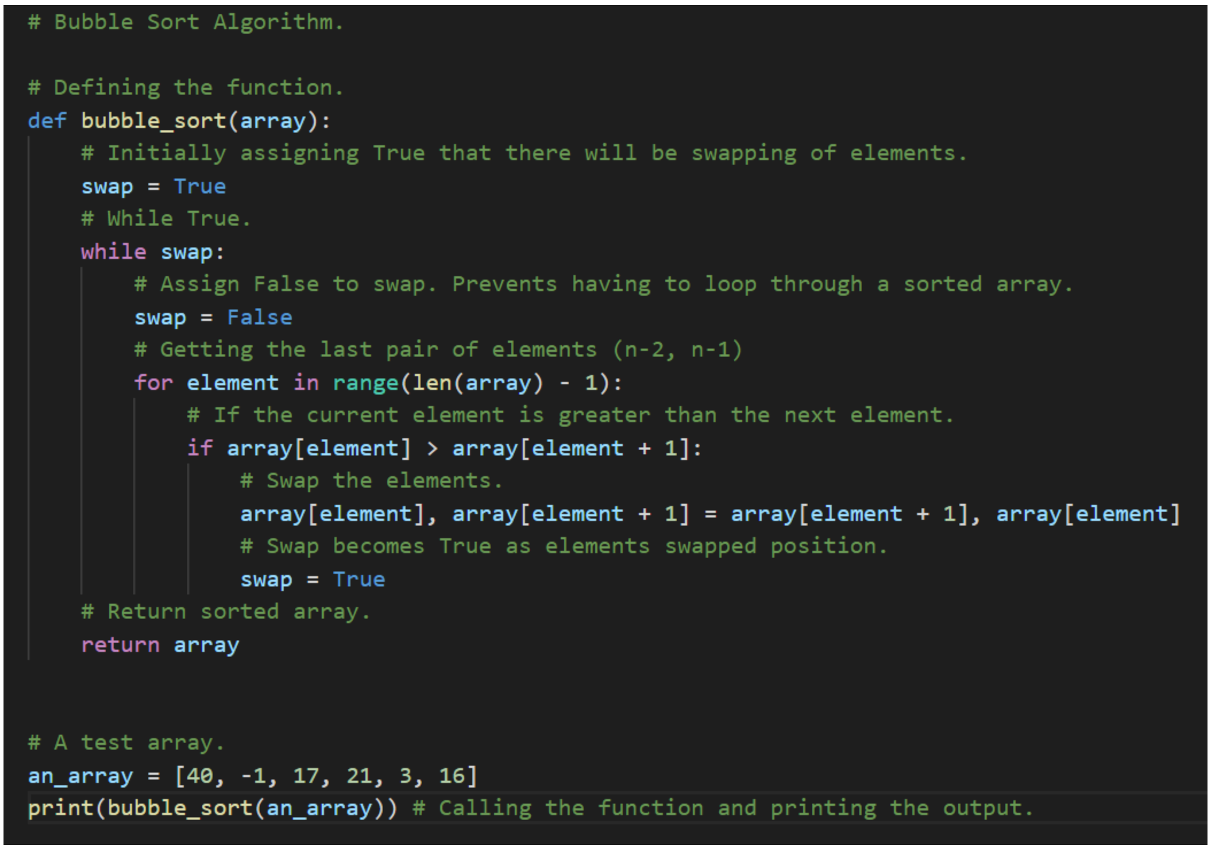
Use bubble sort when:

- When the array is partially sorted - since bubble sort is adaptive.
- When we have memory usage constraints.
- When a simple sorting implementation is desired.
- When the array to be sorted is relatively small.

Avoid bubble sort when:

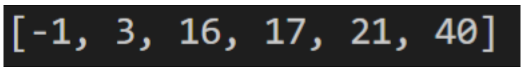
- The array to be sorted has a large number of elements.
- The array is completely unsorted.
- You want a faster run time and memory is not a concern.

Code



```python
# Bubble Sort Algorithm.

# Defining the function.
def bubble_sort(array):
    # Initially assigning True that there will be swapping of elements.
    swap = True
    # While True.
    while swap:
        # Assign False to swap. Prevents having to loop through a sorted array.
        swap = False
        # Getting the last pair of elements (n-2, n-1)
        for element in range(len(array) - 1):
            # If the current element is greater than the next element.
            if array[element] > array[element + 1]:
                # Swap the elements.
                array[element], array[element + 1] = array[element + 1], array[element]
                # Swap becomes True as elements swapped position.
                swap = True
    # Return sorted array.
    return array


# A test array.
an_array = [40, -1, 17, 21, 3, 16]
print(bubble_sort(an_array)) # Calling the function and printing the output.
```

Output:



```
[-1, 3, 16, 17, 21, 40]
```

3. Quick Sort (An Efficient Comparison Based Sort)

Quick Sort uses recursion and is an in-place algorithm that deploys a divide and conquer technique. It is a highly efficient algorithm which is based on breaking down the problem into smaller problems and combing the results together to solve the original problem [31]. The Quick Sort algorithm sorts an input by choosing a pivot point and partitioning the rest of the elements around the pivot. Elements smaller than the pivot are before it (array to the left) and elements larger than the pivot are after it (array to the right). This process continues as the algorithm calls itself recursively to sort the two resulting subarrays, repeating the process before combing the subarrays back together to form one sorted list. Quick Sort is quite efficient for large inputs as we never have to compare elements on the left side of the partition to elements on the right side as they are in the right spot in relation to the pivot [30].

The Pivot

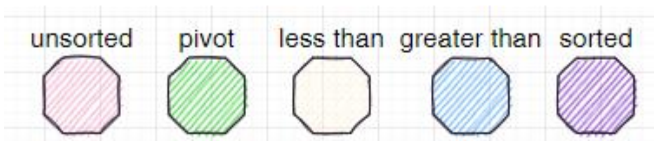
How do we choose a pivot? Randomly selecting the pivot is best practice as it lessens the odds of choosing either the biggest or smallest element – which depending on the desired output could leave us with the worse carse of $O(n^2)$ [32]. E.g., we want to sort an array in descending order but the elements are already partially sorted in ascending order. Many algorithms choose either the first or last element as the pivot. The best case would be to select the median value as in theory this would mean the partitioned arrays would be equal for the most part [32].

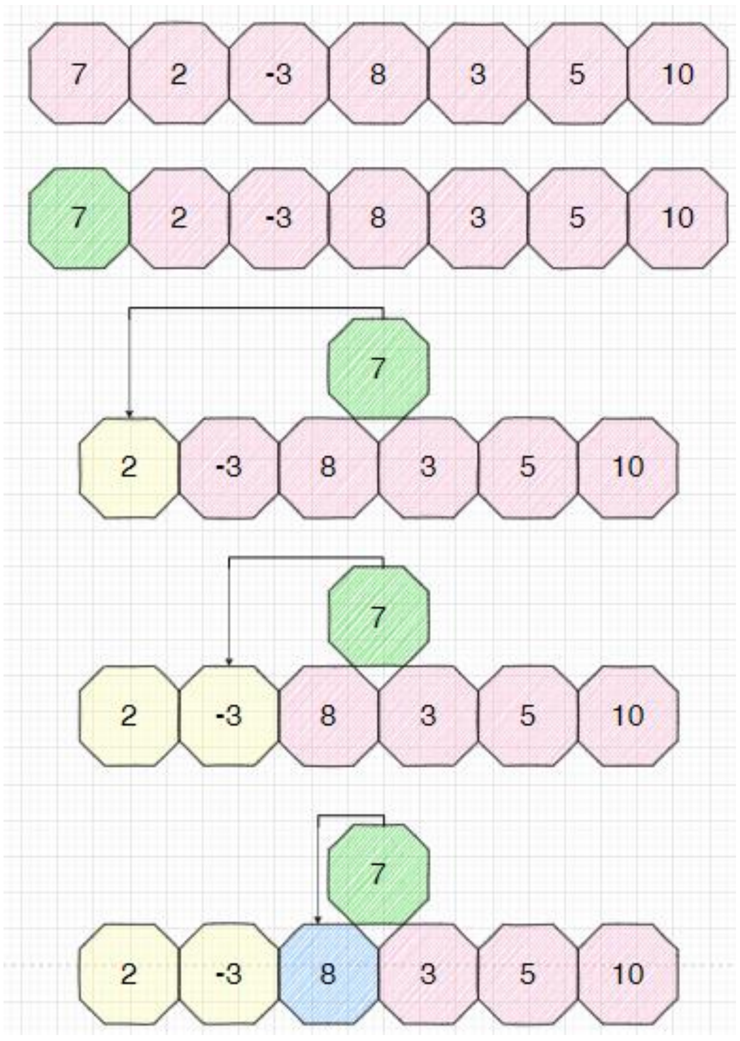
Working Principal

1. Partition array around a pivot element. Elements smaller than the pivot go into a sub array on the left of the pivot and elements bigger move to a sub array on the right.
2. Repeat the process by choosing a new pivot in the sub arrays and use recursion to further divide the new arrays into two parts. Continue until sorted.
3. Combine the sub-arrays back together to form the final sorted array.

Diagram

Key:





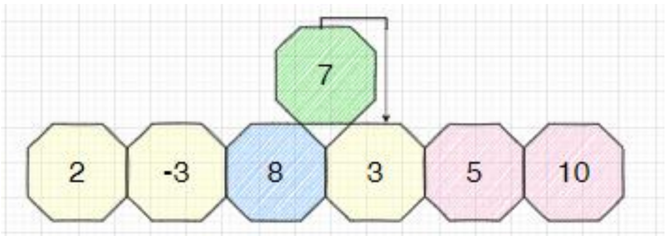
Unsorted array.

Selecting 7 as the pivot.

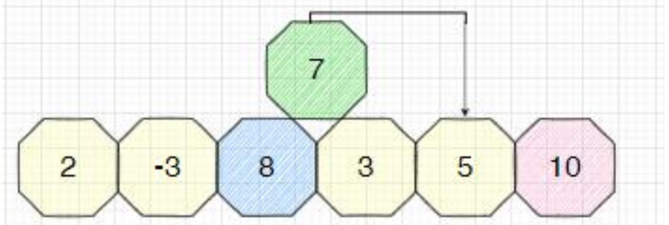
Compare 2 to the pivot (7).
2 < 7, it is sorted into to the less than array (left of pivot).

Compare -3 to the pivot (7).
-3 < 7, it is sorted into to the less than array (left of the pivot).

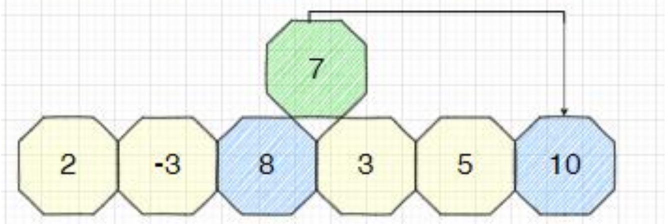
Compare 8 to the pivot (7).
8 > 7, it is sorted into to the greater than array (right of the pivot).



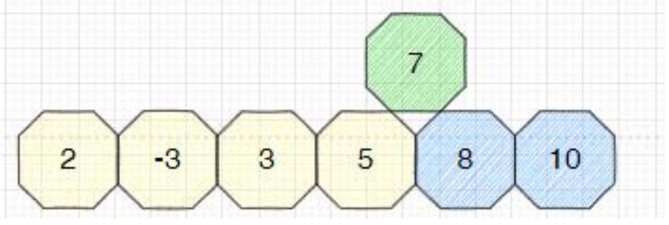
Compare 3 to the pivot (7).
3 < 7, it is sorted into to the less than array (left of the pivot).



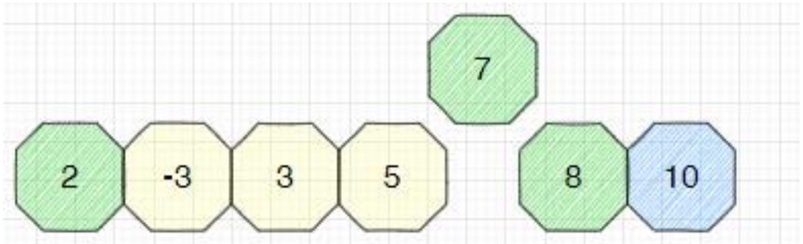
Compare 5 to the pivot (7).
5 < 7, it is sorted into to the less than array (left of the pivot).



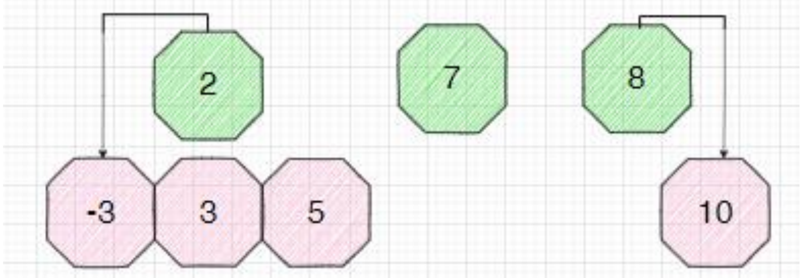
Compare 10 to the pivot (7).
10 > 7, it is sorted into to the greater than array (right of the pivot).



Now we have 2 sub arrays. Less & greater than the pivot. We further partition these arrays.

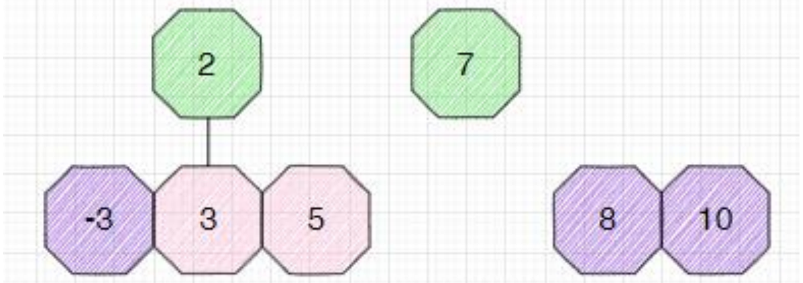


The Pivot is now 2 for the less than array and 8 for the greater than array.



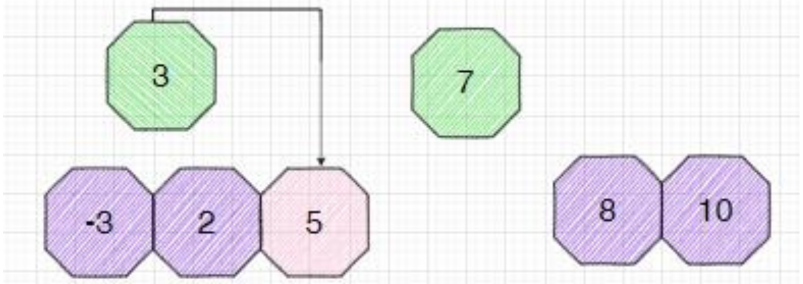
Compare -3 to pivot (2).
-3 < 2, swaps to the left.

Compare 10 to pivot (8).
10 > 8, swaps to the right.

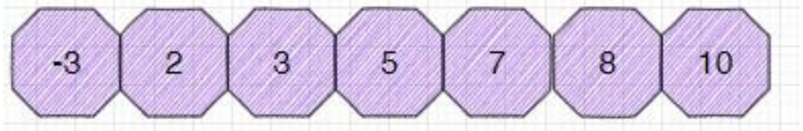


Compare 3 to the pivot (2).
3 > 2, moves to the right.

The greater than array is sorted.



The less than array is to be partitioned again, with 3 now becoming the pivot.
Compare 5 to the pivot (3).
5 > 3, it swaps to the right of the pivot.



After recursively dividing the unsorted array into (left & right of the pivot) and sorting them, we put join them back together and output the final sorted array.

Complexity

Time Complexity	• Best Case: When the pivot is/near the middle element. Ω(n log n). • Worse Case: If the pivot is the biggest of smallest element. O(n^2). • Average Case: The pivot is not the biggest/smallest element and is close to the middle element. Θ (n log n).
Space Complexity	The in-place partition logic uses O(log n) space, and the recursive quick sort algorithm uses O(1) space. So, the overall space complexity of quick sort if O(n log n) [25].
Stability	No. The relative order of equal elements is not preserved.
Adaptable	No. It does not take advantage if an input is partially sorted.

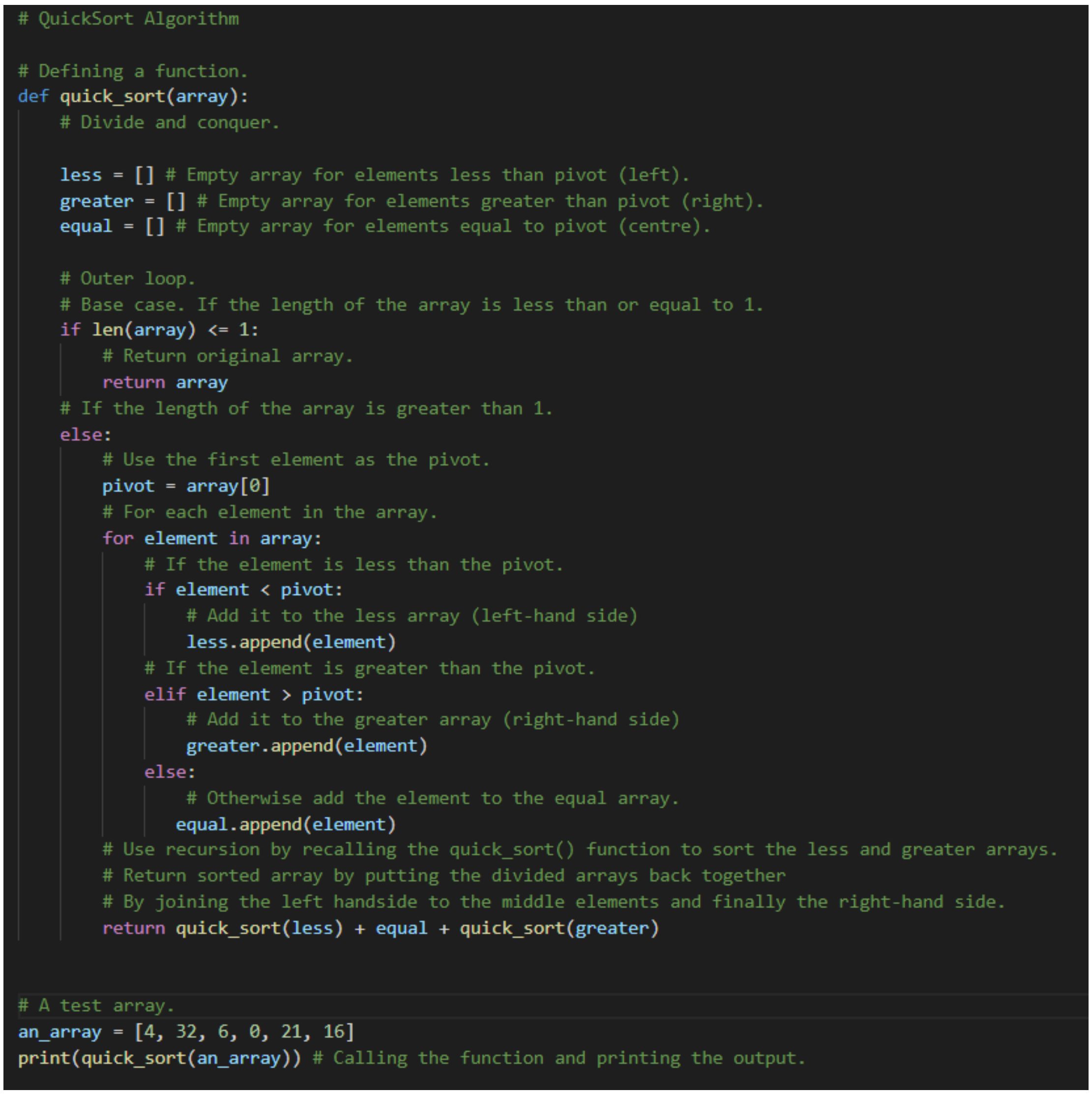
Use quick sort when:

- When fast sorting is desired since quicksort has an average case complexity of O(n log n) which is better than bubble or insertion sort.

Avoid quick sort when:

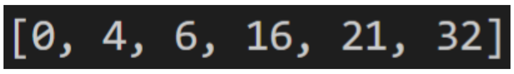
- When space is limited like in embedded systems.
- When ordering of elements matter in the final sorted list, i.e., stable sorting is desired.

Code



```
# QuickSort Algorithm

# Defining a function.
def quick_sort(array):
    # Divide and conquer.

    less = [] # Empty array for elements less than pivot (left).
    greater = [] # Empty array for elements greater than pivot (right).
    equal = [] # Empty array for elements equal to pivot (centre).

    # Outer loop.
    # Base case. If the length of the array is less than or equal to 1.
    if len(array) <= 1:
        # Return original array.
        return array
    # If the length of the array is greater than 1.
    else:
        # Use the first element as the pivot.
        pivot = array[0]
        # For each element in the array.
        for element in array:
            # If the element is less than the pivot.
            if element < pivot:
                # Add it to the less array (left-hand side)
                less.append(element)
            # If the element is greater than the pivot.
            elif element > pivot:
                # Add it to the greater array (right-hand side)
                greater.append(element)
            else:
                # Otherwise add the element to the equal array.
               equal.append(element)
        # Use recursion by recalling the quick_sort() function to sort the less and greater arrays.
        # Return sorted array by putting the divided arrays back together
        # By joining the left handside to the middle elements and finally the right-hand side.
        return quick_sort(less) + equal + quick_sort(greater)


# A test array.
an_array = [4, 32, 6, 0, 21, 16]
print(quick_sort(an_array)) # Calling the function and printing the output.
```

Output:



```
[0, 4, 6, 16, 21, 32]
```

4. Counting Sort (A Non-Comparison Sort)

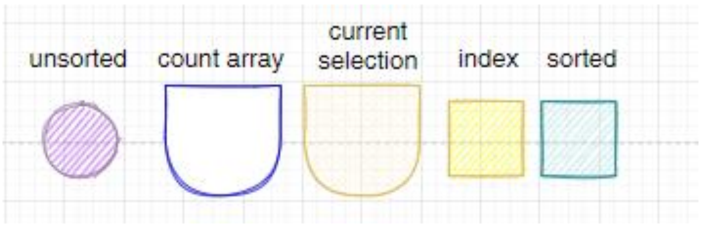
Counting sort sorts by taking in a range of integers to be sorted. It then iterates through the input, counting the number of times each unique element occurs i.e., it sums how many numbers appear in the unsorted input. It is an out of place sorting algorithm. The count is stored in a secondary array. The index of the count array is then used to sort elements index in the final, sorted array [33]. Counting Sort is a highly efficient non-comparison-based algorithm running in O(n) time (quicker than Quick Sort) but, its use is limited by the fact that it can only be with positive integers. Another limiting factor is if the range of the input is exceptionally large because it means an equally large array will need to be created [33]. When deployed effectively it will run linearly

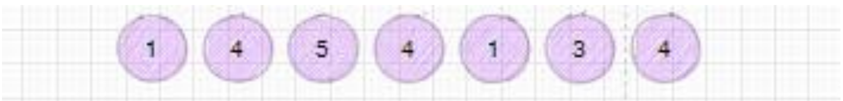
Working Principal

1. Get the max integer, counting the elements to create an auxiliary array whose length matches the range of the input, plus 1. Iterating once over the elements in the unsorted array and, for each element, incrementing the value at the index that maps to the corresponding element.
2. Store the cumulative sum of the auxiliary array at their respective index. Next, decrease its index by one.

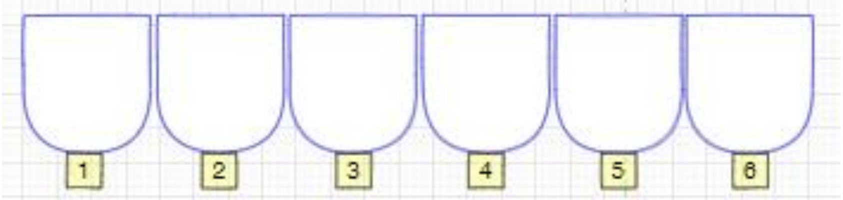
Diagram

Key:

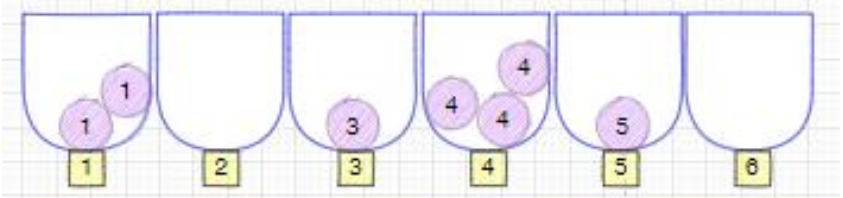




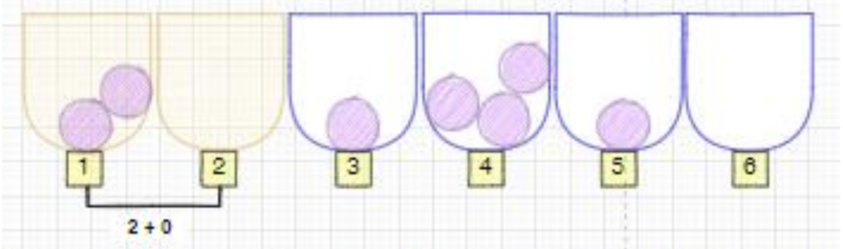
Unsorted array.



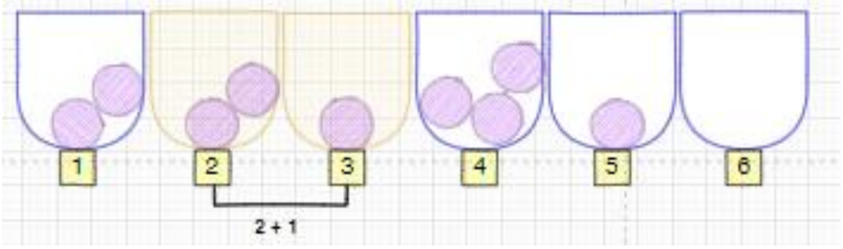
Empty count array, the length of the max element (5) plus 1. (i.e., 6).



The unsorted array is iterated over and the count of which a unique element appears is stored in its respective index in the count array. E.g., 1 appears twice so 2 is stored in index 1 and so on.

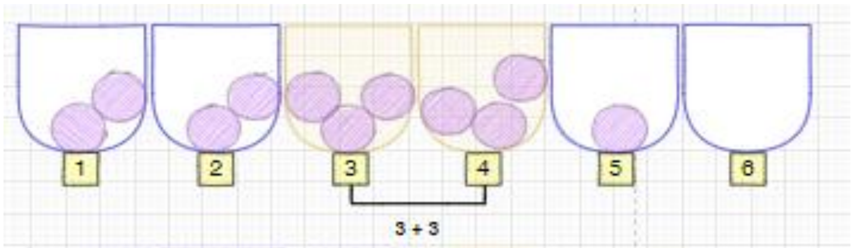


The next step is to get the cumulative sum of the elements in the count array.
The value of index 1 is 2.
the value of index 2 is 0.
2 + 0 = 2.

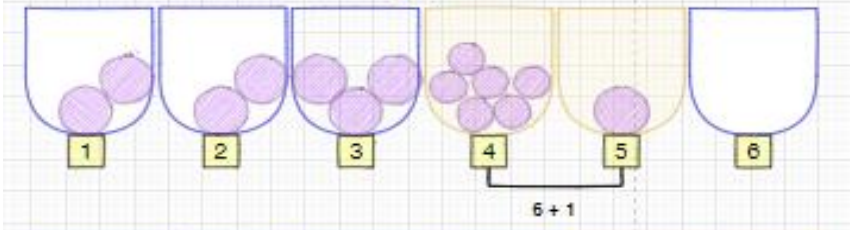


We continue this for the rest of the array.

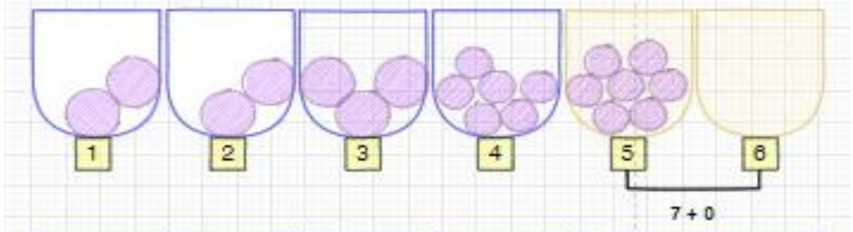
We add 2 to the value of index 3.
2 + 1 = 3.



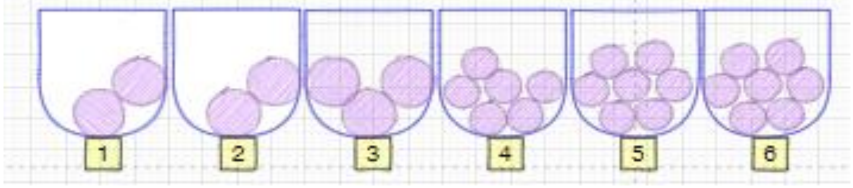
Next, we add 3 to the value of index 4.
3 + 3 = 6.



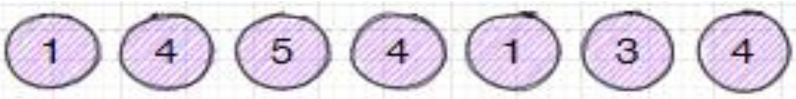
Add 6 to the value of index 5.
6 + 1 = 7.



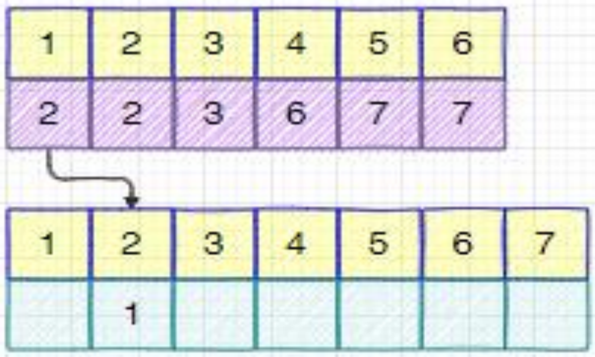
And finally, adding 7 to the value of index 6.
7 + 0 = 7.



Now we have the cumulative sum of the count array.

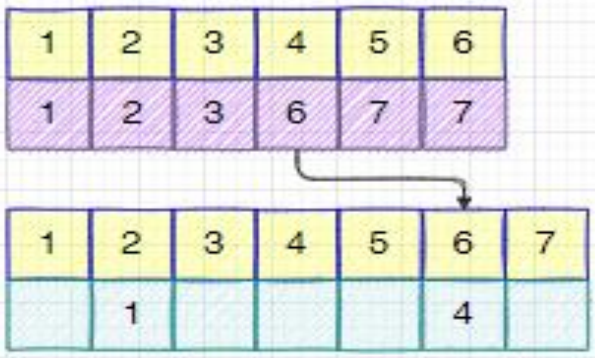


*Reminder of unsorted array.



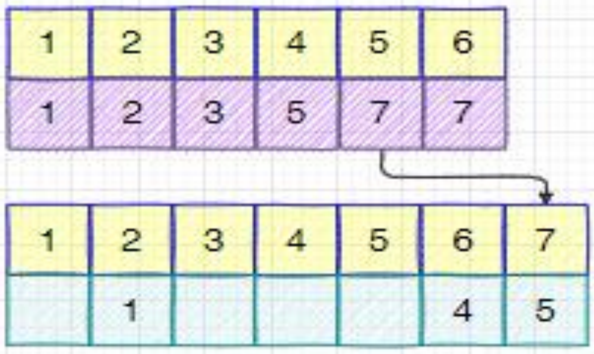
This our count array, which consists of the cumulative values we just worked out, and their index.
The first element in the unsorted array is 1. We go to index 1 in the count array and find its value which is 2.

We place the element 1 at index 2 in the sorted array and decrease the value used in the count array by 1 each time it is used, 2 – 1 = 1.
1 is the new value of index 1 in the count array.



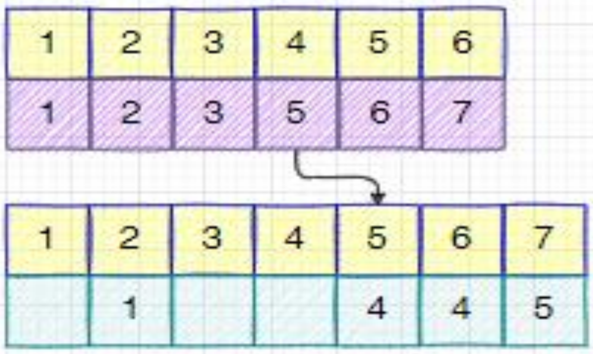
The next element to sort is 4. The value at index 4 in the count array is 6. We place 4 at index 6 in the sorted array.

Decrease the count value by 1.
6 – 1 = 5.



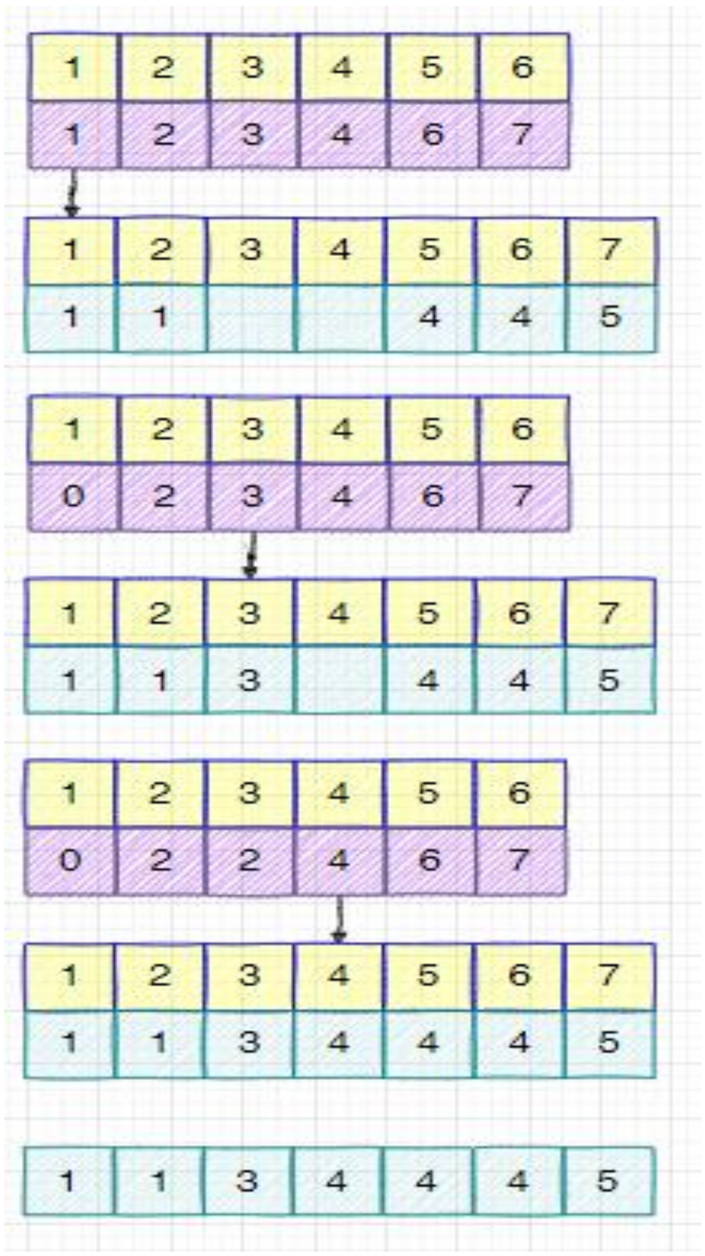
The next element to sort is 5. The value at index 5 in the count array is 7. We place 5 at index 7 in the sorted array.

Decrease the count value by 1.
7 – 1 = 6.



The next element to sort is 4. The value at index 4 in the count array is now 5. We place 4 at index 5 in the sorted array.

Decrease the count value by 1.
5 – 1 = 4.



The next element to sort is 1. The value at index 1 in the count array is now 1. We place 1 at index 1 in the sorted array.

Decrease the count value by 1.
1 – 1 = 0.

The next element to sort is 3. The value at index 3 in the count array 3. We place 3 at index 3 in the sorted array.

Decrease the count value by 1.
3 – 1 = 2.

The last element to sort is 4. The value at index 4 in the count array is now 4. We place 4 at index 4 in the sorted array.

Decrease the count value by 1.
4 – 1 = 3.

The array is now sorted.

Complexity

The time complexity for each case is the same as it must iterate n+k (k=input range) times, regardless of how elements are ordered.

Time Complexity
 	
O = number of elements
k = input range
	• <u>Best Case:</u> When no sorting is required, i.e., the array is already sorted. Ω(n + k).
	• <u>Worse Case:</u> This is when the elements are in reverse order of the desired output (ascending or descending). O(n + k).
	• <u>Average Case:</u> When the elements are neither in an ascending or descending order. ϴ (n + k).

Space Complexity	The bigger the range elements fall between, the greater the space complexity. O(max).
Stability	Yes. Though classified as an out-of-place algorithm the order of elements in the original array is maintained making it a stable.
Adaptable	No. The number of unique occurrences are counted regardless of input order.

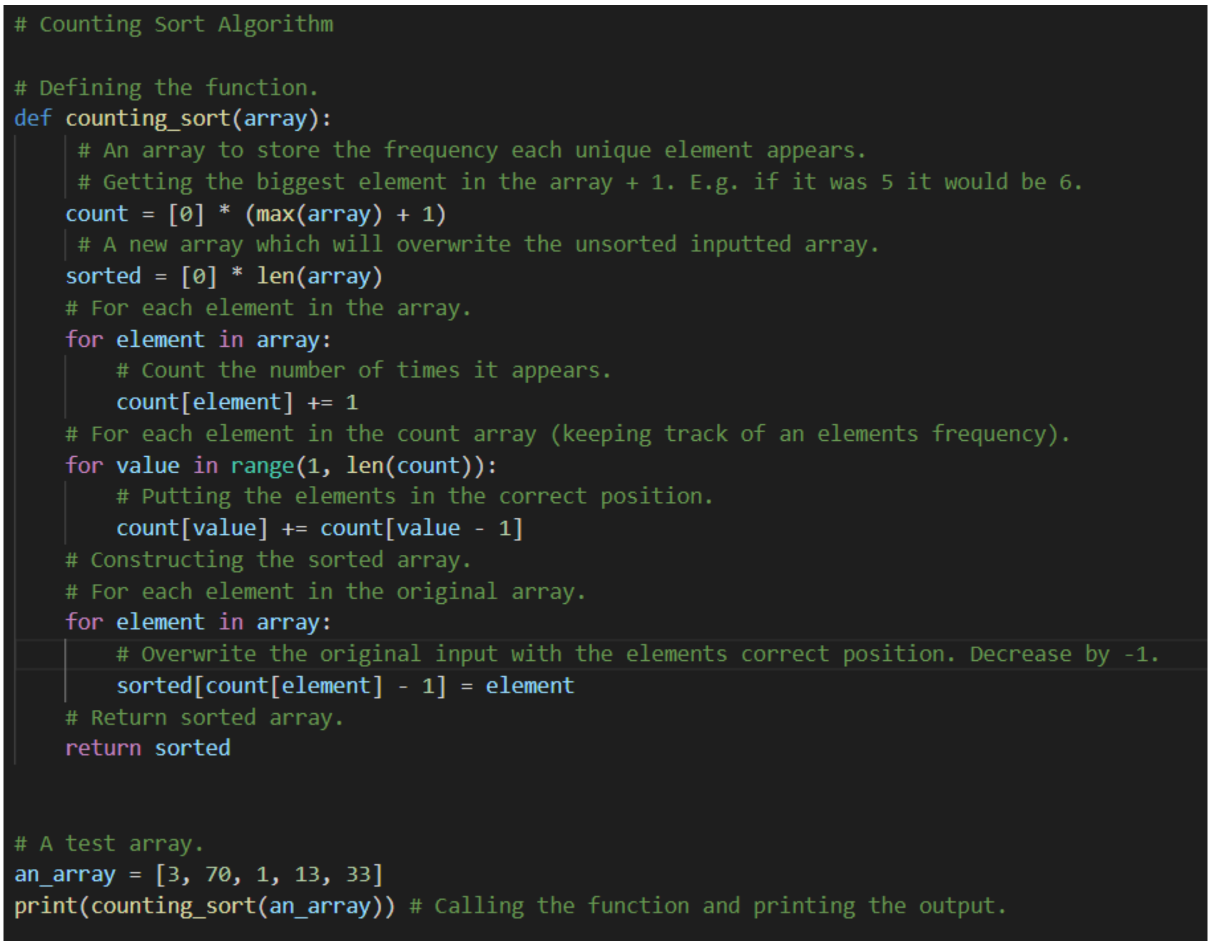
Use counting sort when:

- The data to be sorted consists of positive integers.
- You know the range of the input.
- The range to be sorted is not too large.
- Using it as a subroutine for another sorting algorithm.

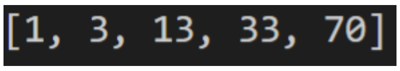
Avoid counting sort when:

- The input is not a positive integer.
- The range is too large.

Code



```python
# Counting Sort Algorithm

# Defining the function.
def counting_sort(array):
    # An array to store the frequency each unique element appears.
    # Getting the biggest element in the array + 1. E.g. if it was 5 it would be 6.
    count = [0] * (max(array) + 1)
    # A new array which will overwrite the unsorted inputted array.
    sorted = [0] * len(array)
    # For each element in the array.
    for element in array:
        # Count the number of times it appears.
        count[element] += 1
    # For each element in the count array (keeping track of an elements frequency).
    for value in range(1, len(count)):
        # Putting the elements in the correct position.
        count[value] += count[value - 1]
    # Constructing the sorted array.
    # For each element in the original array.
    for element in array:
        # Overwrite the original input with the elements correct position. Decrease by -1.
        sorted[count[element] - 1] = element
    # Return sorted array.
    return sorted


# A test array.
an_array = [3, 70, 1, 13, 33]
print(counting_sort(an_array)) # Calling the function and printing the output.
```

Output: 

```
[1, 3, 13, 33, 70]
```

5. Selection Sort (A Simple Comparison Based Sort)

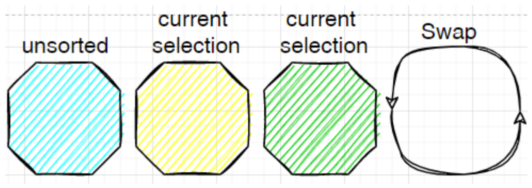
Selection Sort takes the smallest element in an unsorted array and brings it to the front. It's an in-place sorting algorithm which consists of two nested loops and starts by selecting the first element and iterates through an array (left to right) comparing it to each element until it finds the smallest one. The first item in the array is now sorted, while the rest of the array is unsorted [34]. This process is repeated until the array is sorted. Selection sort in the main, performs better than Bubble Sort but worse than Insertion sort. Note: the correct position for the current selected element is found before moving on to next element in the array. The time efficiency of selection sort is quadratic.

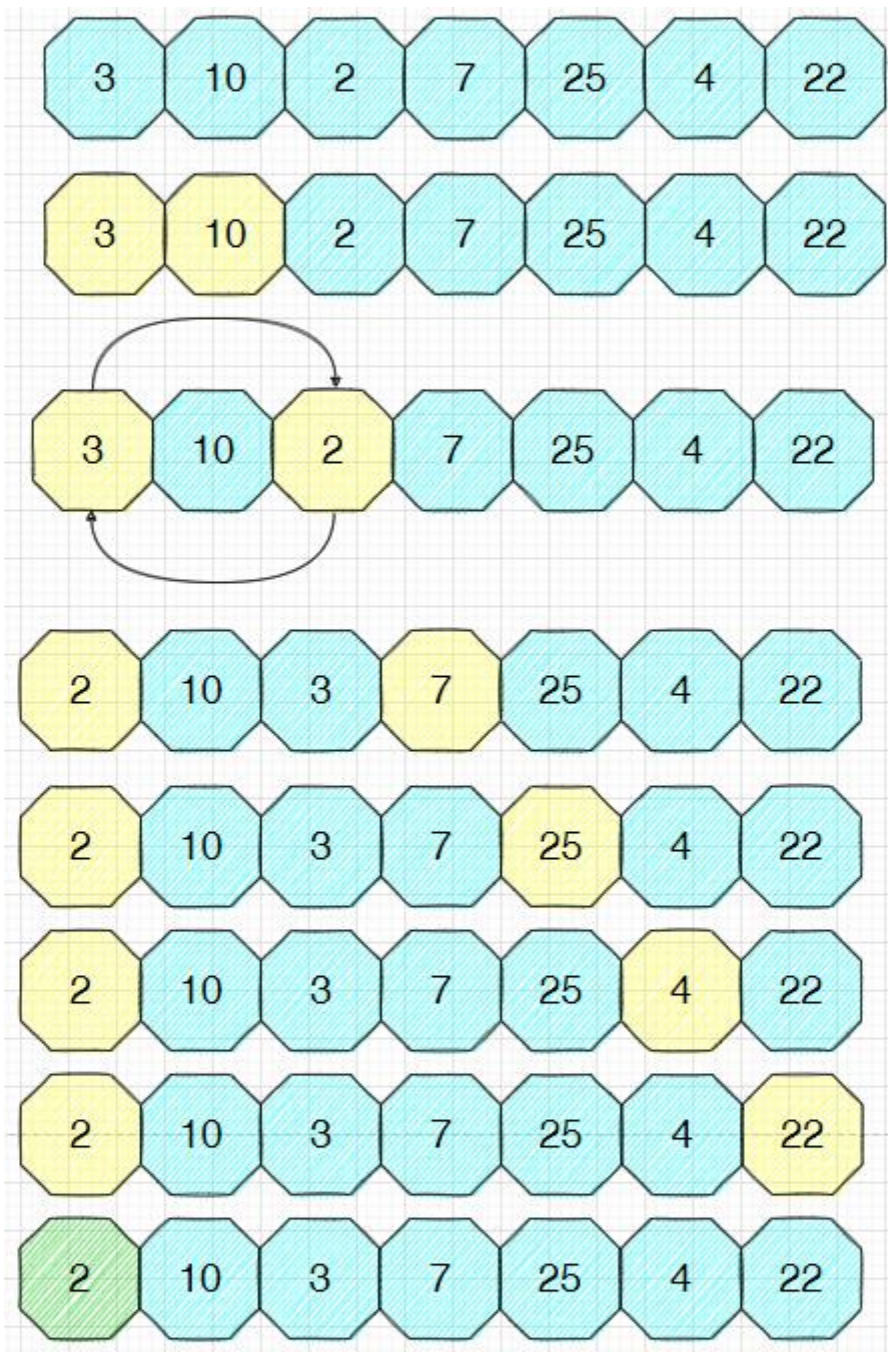
Working Principal

1. Select one element in the array, starting with the first element.
2. Compare it to all other elements in the array sequentially.
3. If an element is found to be smaller than the currently selected element, swap their positions.

Diagram

Key:





Unsorted array.

Compare 3 to 10.
3 < 10, no swap needed.

Compare 3 to 2.
3 > 2, swap elements.

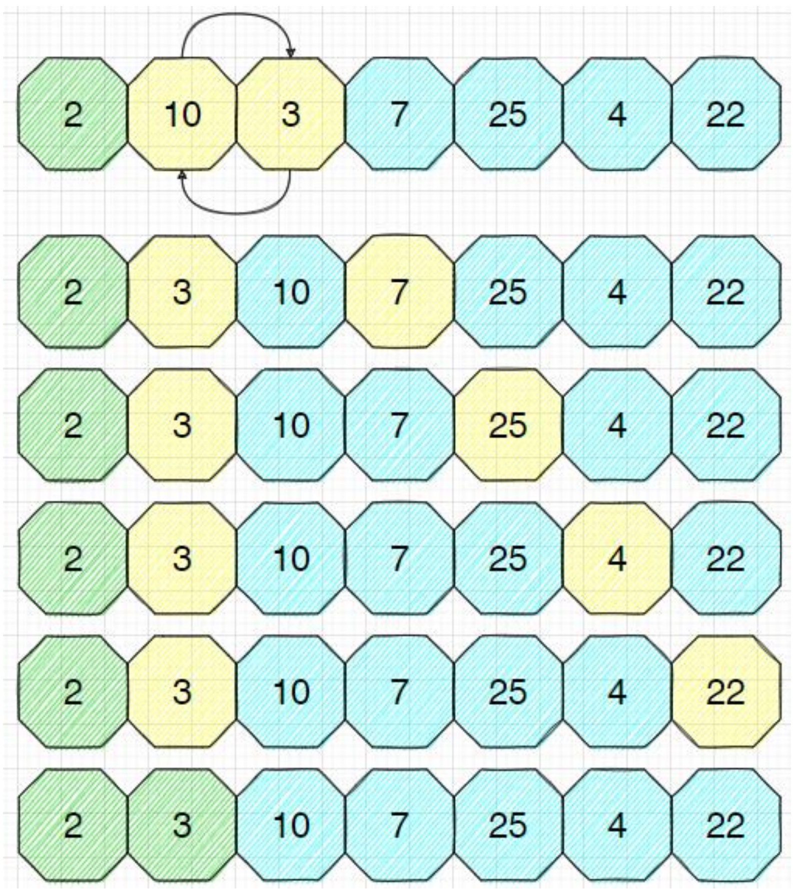
Compare 2 to 7.
2 < 7, no swap needed.

Compare 2 to 25.
2 < 25, no swap needed.

Compare 2 to 4.
2 < 4, no swap needed.

Compare 2 to 22.
2 <22, no swap needed.

Partially sorted array after the
first iteration. We now know that
2 is smallest element.



Second iteration.
Compare 10 & 3.
10 > 3, swap.

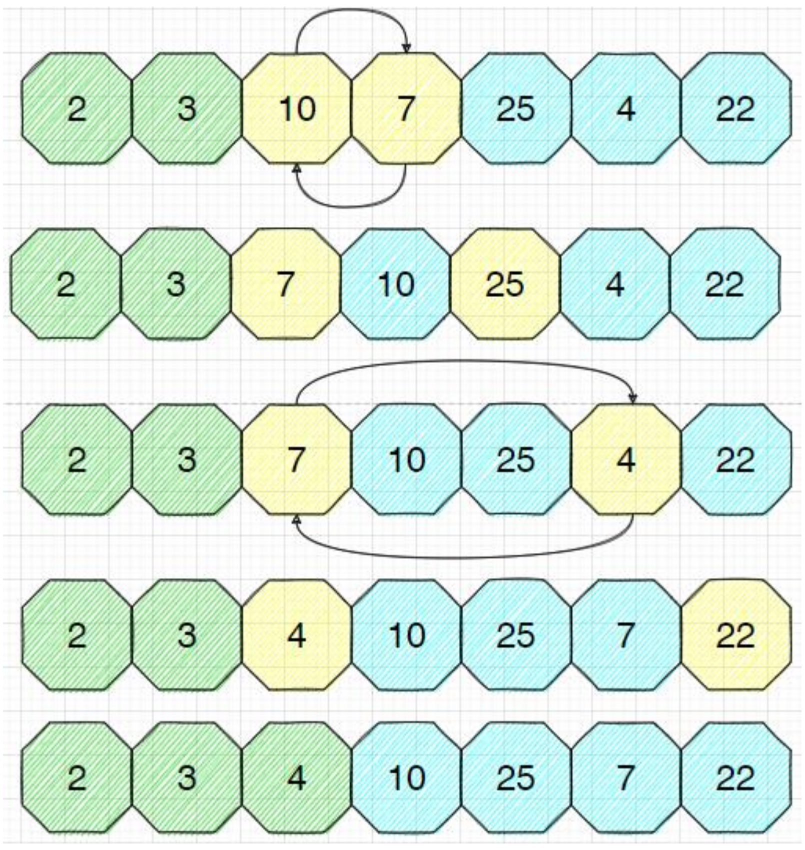
Compare 3 & 7.
3 < 7, no swap needed.

Compare 3 & 25.
3 < 25, no swap needed.

Compare 3 & 4.
3 < 4, no swap needed.

Compare 3 & 22.
3 < 22, no swap needed.

Partially sorted array after the second iteration.



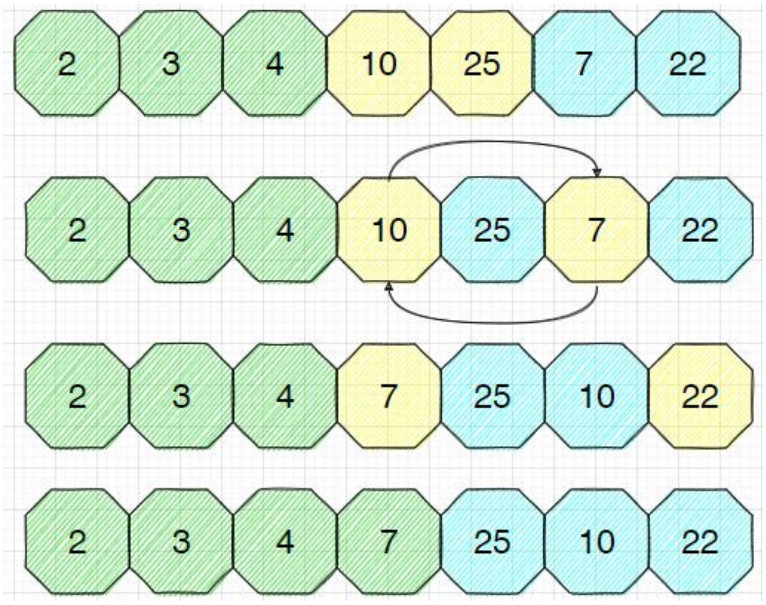
Third iteration.
Compare 10 & 7.
10 > 7, swap.

Compare 7 & 25.
7 < 25, no swap needed.

Compare 7 & 4.
7 > 4, swap.

Compare the new unsorted minimum value, 4 with 22.
4 < 22, no swap.

Partially sorted array after the third iteration.

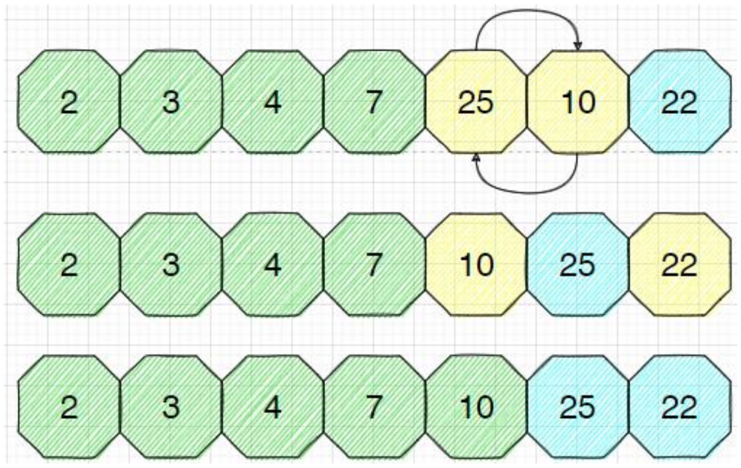


Fourth iteration.
Compare 10 & 25.
10 < 25, no swap needed.

Compare 10 & 7.
10 > 7, swap.

Compare the new unsorted minimum value, 7 with 22.
7 < 22, no swap.

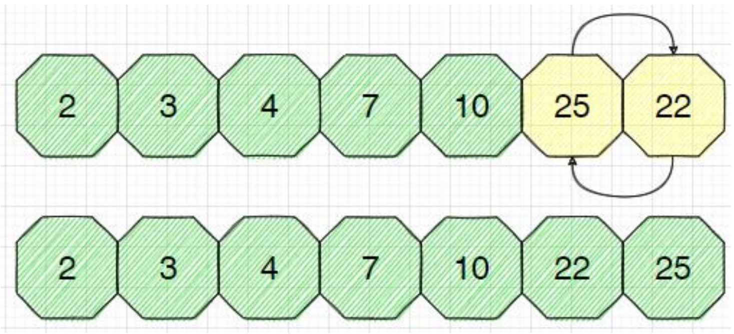
Partially sorted array after the fourth iteration.



Fifth iteration.
Compare 25 & 10.
25 > 10, swap.

Compare the new unsorted minimum value, 10 with 22.
10 < 22, no swap.

Partially sorted array after the fifth iteration.



Sixth iteration.
Compare 25 & 22.
25 > 22, swap.

Final sorted array.

Complexity

Time Complexity	• <u>Best Case:</u> No sorting required, i.e., array is already sorted. $\Omega(n^2)$. • <u>Worse Case:</u> When elements are in reverse order of the desired output. $O(n^2)$. • <u>Average Case:</u> When the elements are neither in an ascending or descending order. Θ (n^2).
Space Complexity	Since the array is sorted in place and no extra space is used, the space complexity is O(1)
Stability	No. Elements which are equal may be re-arranged in the final sort.
Adaptable	No. Each element must be compared, cannot break early.

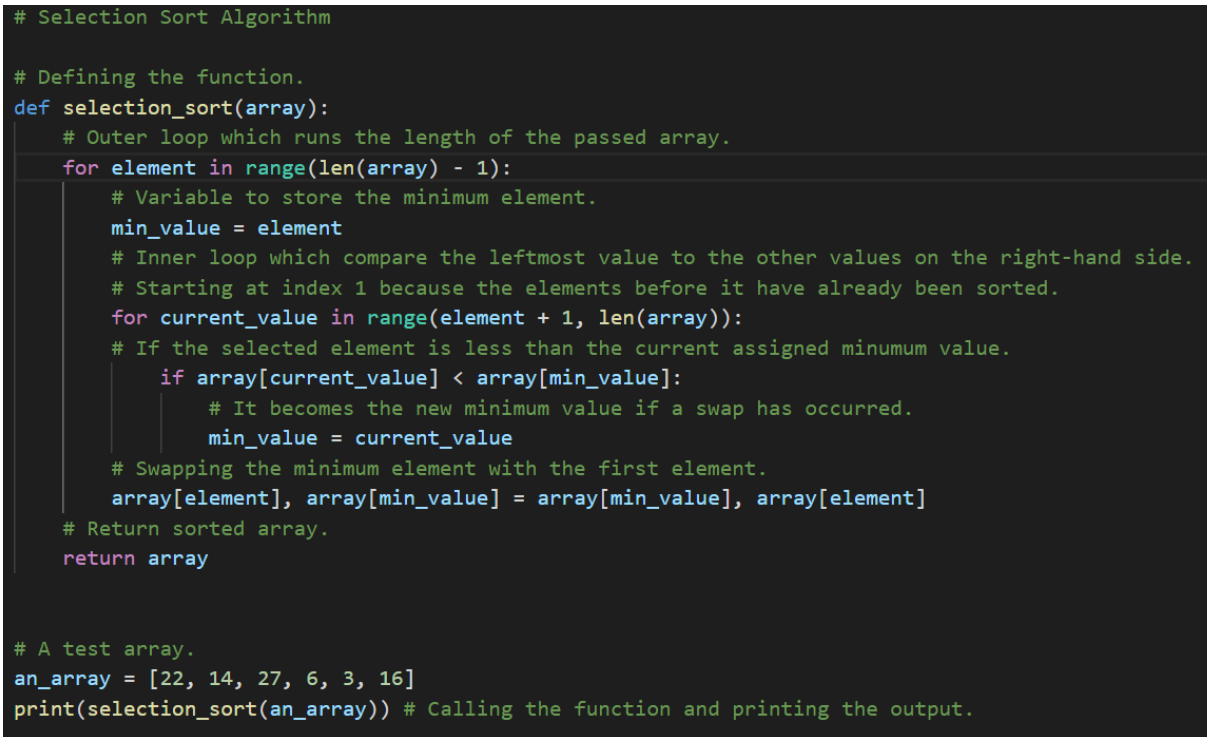
Use selection sort when:

- When the array is NOT partially sorted.
- When we have memory usage constraints.
- When a simple sorting implementation is desired.
- When the array to be sorted is relatively small.

Avoid selection sort when:

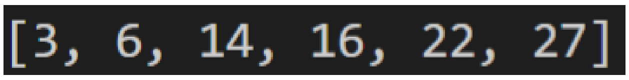
- The array to be sorted has a large number of elements.
- The array is nearly sorted.
- You want a faster run time and memory is not a concern.

Code



```python
# Selection Sort Algorithm

# Defining the function.
def selection_sort(array):
    # Outer loop which runs the length of the passed array.
    for element in range(len(array) - 1):
        # Variable to store the minimum element.
        min_value = element
        # Inner loop which compare the leftmost value to the other values on the right-hand side.
        # Starting at index 1 because the elements before it have already been sorted.
        for current_value in range(element + 1, len(array)):
        # If the selected element is less than the current assigned minumum value.
            if array[current_value] < array[min_value]:
                # It becomes the new minimum value if a swap has occurred.
                min_value = current_value
        # Swapping the minimum element with the first element.
        array[element], array[min_value] = array[min_value], array[element]
    # Return sorted array.
    return array


# A test array.
an_array = [22, 14, 27, 6, 3, 16]
print(selection_sort(an_array)) # Calling the function and printing the output.
```

Output:



```
[3, 6, 14, 16, 22, 27]
```

Implementation and Benchmarking

In computing, benchmarking is the process of comparing programmes with respect to relative performance measures which is conducted by running various standardised tests against it [35].

The script *benchmarking.py* included in the project submission is the application to carry out this testing. The purpose of *benchmarking.py* is to benchmark the 5 chosen sorting algorithms by using randomly generated arrays of integers between 0 and 100 of varying input sizes (n). This allows for the testing of input size on an algorithm's running time. The runtime of each algorithm is to be measured 10 times per input size using different randomly generated arrays each time. The average of the 10 runs is to be outputted to the console when the application finishes executing. The application will also generate a graph of the average runtime for each algorithm as well as create a *.csv* file to show the results of each independent run if needed for further investigated.

The first step in the process was to research and understand each of the chosen sorting algorithms. The next step was to write the benchmarking application.

Set up

The machine hardware and software used to carry out the benchmarking process:

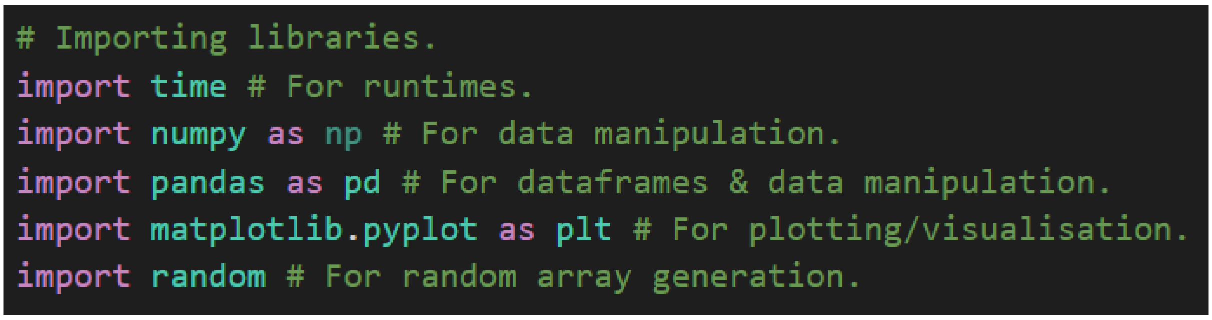
<u>Machine hardware:</u>

- Lenovo Thinkbook 14-IIL.
- Intel Core i7-1065G7 processor CPU @ 1.30GHz 1.50 GHz.
- 16GB RAM.
- 64-bit operating system.

<u>Machine software:</u>

- Windows 11 Home.
- Python 3.8.11.
- Visual Studio Code 1.66.2.

Libraries used:



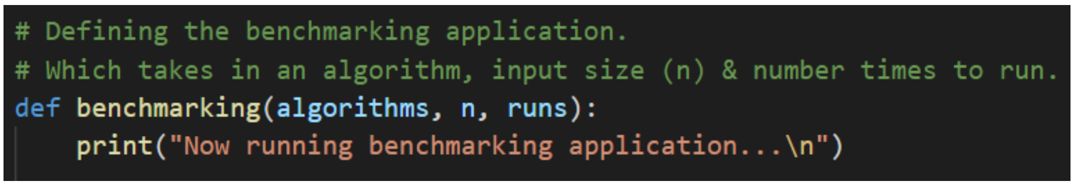
```
# Importing libraries.
import time # For runtimes.
import numpy as np # For data manipulation.
import pandas as pd # For dataframes & data manipulation.
import matplotlib.pyplot as plt # For plotting/visualisation.
import random # For random array generation.
```

- The time module was used for measuring the algorithms runtime.
- Numpy is used for the arrays, and it is also a Matplotlib dependency.
- Pandas is used to create a dataframe to hold the test results as well as data manipulation.
- Matplotlib is used for generating a graph of the results.
- The Random module is used to generate the arrays of random integers.

Sorting Algorithms:

The next part of the process was creating functions for each of the 5 sorting algorithms – Bubble Sort, Counting Sort, Insertion Sort, Quick Sort and Selection Sort. Each of the sorting algorithms take in one argument – an array and returns a sorted version of the array when it is called.

Benchmarking:



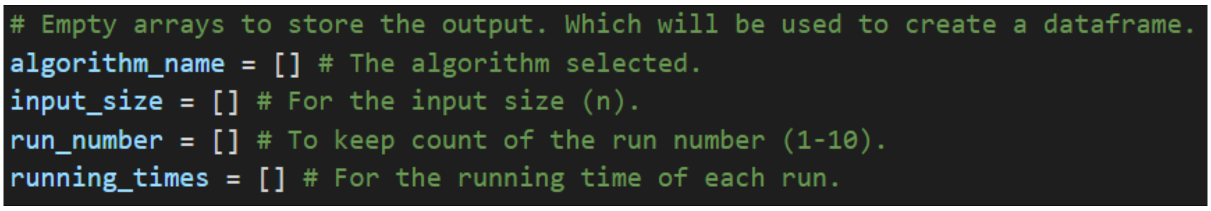
```
# Defining the benchmarking application.
# Which takes in an algorithm, input size (n) & number times to run.
def benchmarking(algorithms, n, runs):
    print("Now running benchmarking application...\n")
```

The *benchmarking()* function which carries out the testing takes in 3 arguments:

- An algorithm (i.e., Bubble Sort, Counting Sort, Insertion Sort, Quick Sort, Selection Sort).
- The input size (n) i.e., the size of the array of randomly generated numbers.

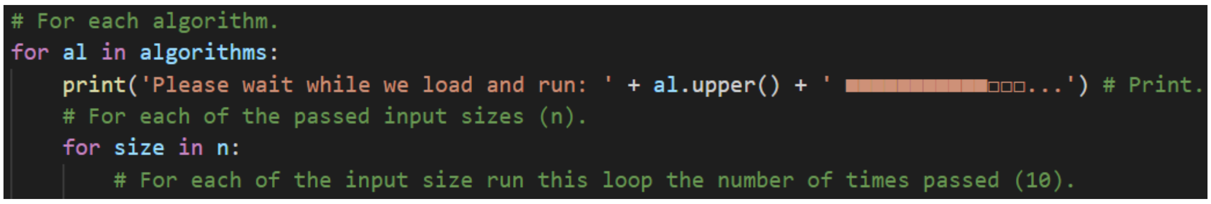
- Runs; the number of times to run each input size per algorithm. (10 in this case).

Within the function there are 4 empty arrays which hold the output of each run. These arrays will later be used to create a pandas dataframe which will be the final output.



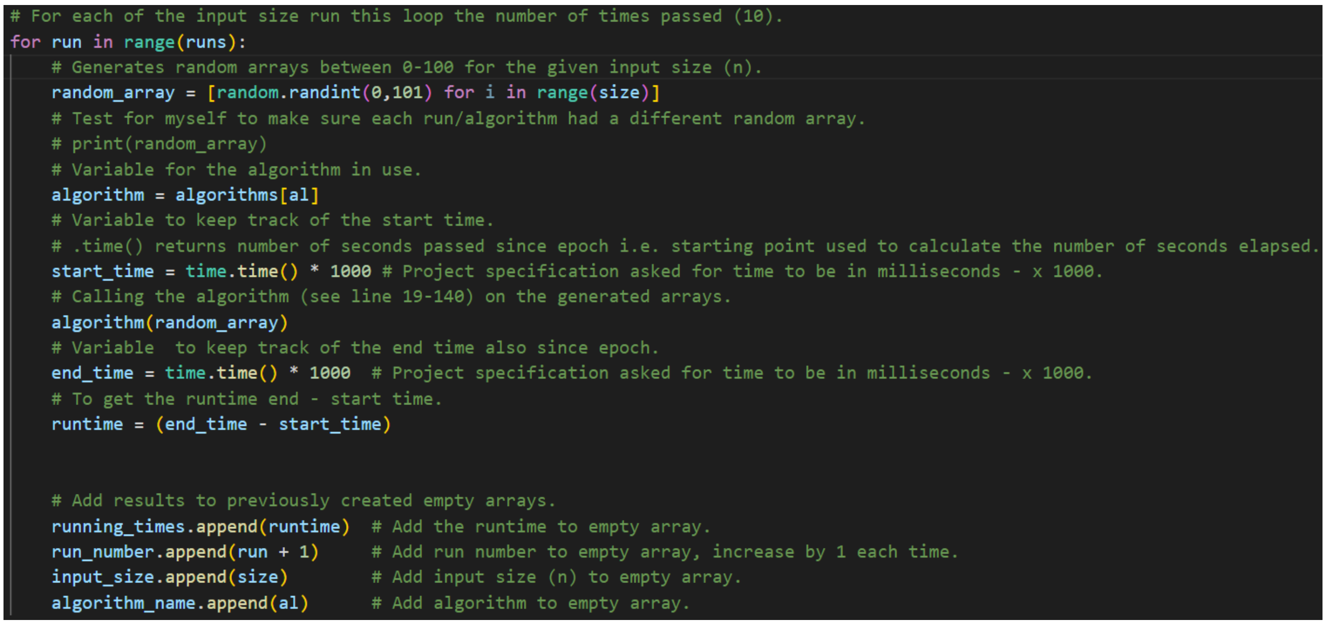
```
# Empty arrays to store the output. Which will be used to create a dataframe.
algorithm_name = [] # The algorithm selected.
input_size = [] # For the input size (n).
run_number = [] # To keep count of the run number (1-10).
running_times = [] # For the running time of each run.
```

The function is made up of 3 loops.



```
# For each algorithm.
for al in algorithms:
    print('Please wait while we load and run: ' + al.upper() + ' ■■■■■■■■■■■□□□...') # Print.
    # For each of the passed input sizes (n).
    for size in n:
        # For each of the input size run this loop the number of times passed (10).
```

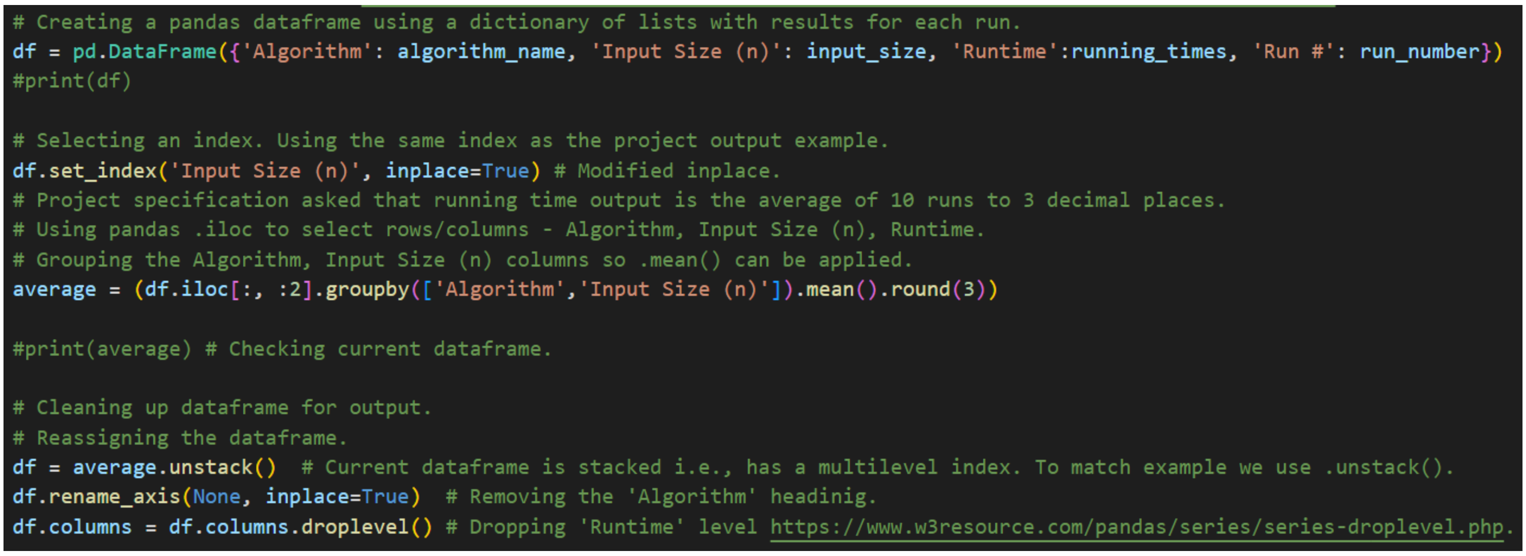
The outer loop iterates through each of the sorting algorithms. The first nested loop iterates through the different array input sizes.



```
# For each of the input size run this loop the number of times passed (10).
for run in range(runs):
    # Generates random arrays between 0-100 for the given input size (n).
    random_array = [random.randint(0,101) for i in range(size)]
    # Test for myself to make sure each run/algorithm had a different random array.
    # print(random_array)
    # Variable for the algorithm in use.
    algorithm = algorithms[al]
    # Variable to keep track of the start time.
    # .time() returns number of seconds passed since epoch i.e. starting point used to calculate the number of seconds elapsed.
    start_time = time.time() * 1000 # Project specification asked for time to be in milliseconds - x 1000.
    # Calling the algorithm (see line 19-140) on the generated arrays.
    algorithm(random_array)
    # Variable  to keep track of the end time also since epoch.
    end_time = time.time() * 1000  # Project specification asked for time to be in milliseconds - x 1000.
    # To get the runtime end - start time.
    runtime = (end_time - start_time)


    # Add results to previously created empty arrays.
    running_times.append(runtime)  # Add the runtime to empty array.
    run_number.append(run + 1)     # Add run number to empty array, increase by 1 each time.
    input_size.append(size)        # Add input size (n) to empty array.
    algorithm_name.append(al)      # Add algorithm to empty array.
```

The inner most loop executes the number of runs specified by the run's argument passed (10). Firstly, it generates a random array of numbers between 0-100 for each of the input sizes in the first nested loop. Each iteration yields a different randomly generated array each time. Next it assigns the algorithm currently being tested to the variable *algorithm*, so we know which algorithm the current results are for.

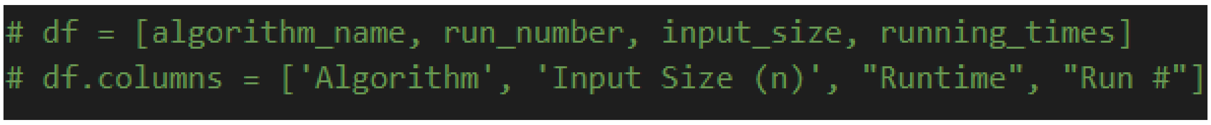
To calculate the runtime of an algorithm we need to get its starting and end times. The *time.time()* method returns the number of seconds since epoch [36] i.e., when a computer's date and time was set [37]. The start and end times are multiplied by 1000 because the project brief specified that the running time is in milliseconds. Each iteration of the loops adds – algorithm, runtime, input size and the run number (1-10) – the results/variables to the empty arrays previously created.



```python
# Creating a pandas dataframe using a dictionary of lists with results for each run.
df = pd.DataFrame({'Algorithm': algorithm_name, 'Input Size (n)': input_size, 'Runtime':running_times, 'Run #': run_number})
#print(df)

# Selecting an index. Using the same index as the project output example.
df.set_index('Input Size (n)', inplace=True) # Modified inplace.
# Project specification asked that running time output is the average of 10 runs to 3 decimal places.
# Using pandas .iloc to select rows/columns - Algorithm, Input Size (n), Runtime.
# Grouping the Algorithm, Input Size (n) columns so .mean() can be applied.
average = (df.iloc[:, :2].groupby(['Algorithm','Input Size (n)']).mean().round(3))

#print(average) # Checking current dataframe.

# Cleaning up dataframe for output.
# Reassigning the dataframe.
df = average.unstack()  # Current dataframe is stacked i.e., has a multilevel index. To match example we use .unstack().
df.rename_axis(None, inplace=True)  # Removing the 'Algorithm' headinig.
df.columns = df.columns.droplevel() # Dropping 'Runtime' level https://www.w3resource.com/pandas/series/series-droplevel.php.
```

After the inner most loop has executed 10 times per input size (n) for each algorithm, we switch back to the outer loop, which will create a dataframe from the newly populated arrays holding the results.

The results of the arrays are transformed into a pandas dataframe by using a dictionary of lists. I originally tried created the dataframe using the following:

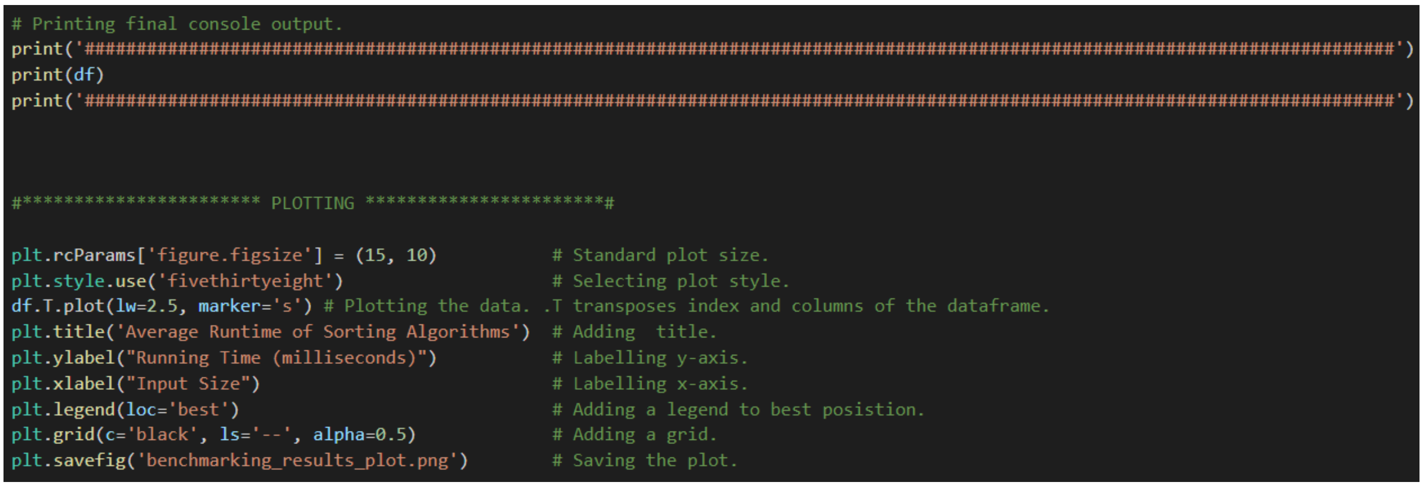


```python
# df = [algorithm_name, run_number, input_size, running_times]
# df.columns = ['Algorithm', 'Input Size (n)', "Runtime", "Run #"]
```

But it kept throwing an error after realising I was using the wrong type and searching for a solution, I found educative.io which described using a dictionary of lists to create a dataframe [38]. The input size (n) is set as the index to match the project output example.

The output is to display the average of 10 runs per input size for each algorithm, rounded to 3 decimal places. To do this we access the first 3 columns in the dataframe – algorithm, Input Size (n) and Runtime – by using *.iloc* and then using *.groupby()* method to group by category which allows for the aggregate function *.mean()* to be applied to the data to get the average runtimes.

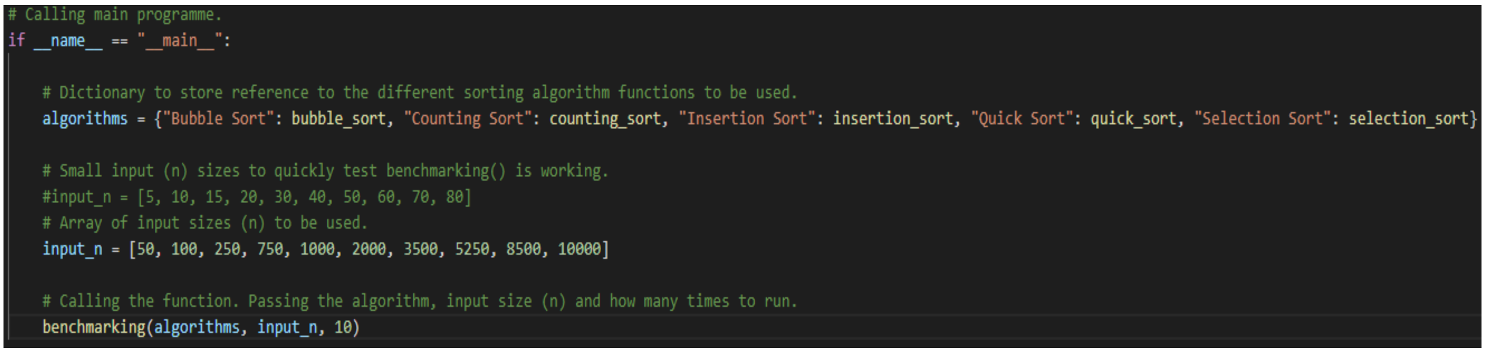
After checking what the current dataframe looked like, to match the brief output example I unstacked the dataframe as it currently had a multileveled index [39]. To fully match the output example, the *Algorithm* and *Runtime* column headings are removed.



```
# Printing final console output.
print('###############################################################################################################################')
print(df)
print('###############################################################################################################################')



#*********************** PLOTTING ***********************#

plt.rcParams['figure.figsize'] = (15, 10)           # Standard plot size.
plt.style.use('fivethirtyeight')                    # Selecting plot style.
df.T.plot(lw=2.5, marker='s') # Plotting the data. .T transposes index and columns of the dataframe.
plt.title('Average Runtime of Sorting Algorithms')  # Adding  title.
plt.ylabel("Running Time (milliseconds)")           # Labelling y-axis.
plt.xlabel("Input Size")                            # Labelling x-axis.
plt.legend(loc='best')                              # Adding a legend to best posistion.
plt.grid(c='black', ls='--', alpha=0.5)             # Adding a grid.
plt.savefig('benchmarking_results_plot.png')        # Saving the plot.
```

Once the dataframe is created, it is printed to the console and a graph of the results is generated before the application terminates. *df.T.plot()* transposes the data and allows for the plotting of multiple series [40]. The plot is saved as *"benchmarking_results_plot.png"* and can be found in the main project folder.

Main Programme:



```
# Calling main programme.
if __name__ == "__main__":

    # Dictionary to store reference to the different sorting algorithm functions to be used.
    algorithms = {"Bubble Sort": bubble_sort, "Counting Sort": counting_sort, "Insertion Sort": insertion_sort, "Quick Sort": quick_sort, "Selection Sort": selection_sort}

    # Small input (n) sizes to quickly test benchmarking() is working.
    #input_n = [5, 10, 15, 20, 30, 40, 50, 60, 70, 80]
    # Array of input sizes (n) to be used.
    input_n = [50, 100, 250, 750, 1000, 2000, 3500, 5250, 8500, 10000]

    # Calling the function. Passing the algorithm, input size (n) and how many times to run.
    benchmarking(algorithms, input_n, 10)
```

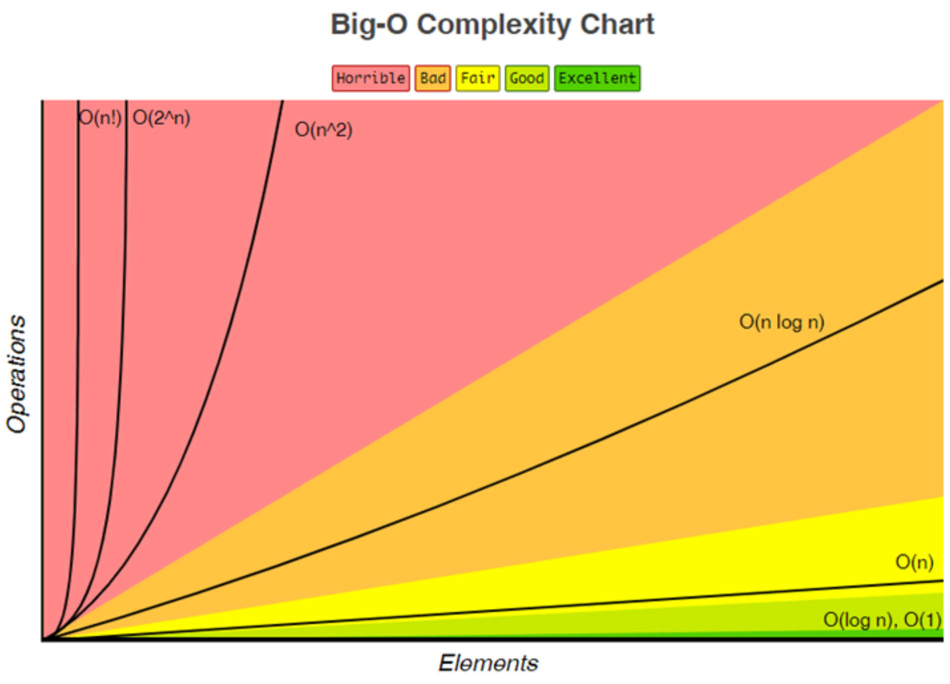
For the *benchmarking()* function to call each algorithm, they are stored in a dictionary called *algorithms* under the main function. The dictionary is used to store a reference to each of the sorting algorithm functions to be used. The is no reasoning behind their order other than being in lexicographical (alphabetical) order.

The input sizes to be used when generating random arrays for each algorithm is stored in a variable called *n_input.*

And finally, the *benchmarking()* function is called with 10 (runs) being the last of the arguments passed as per the brief description.

Results

The below diagram [41] demonstrates the growth rate of an algorithm. The growth rate is the growth of an algorithm's time complexity as the input increases i.e., the runtime [42].

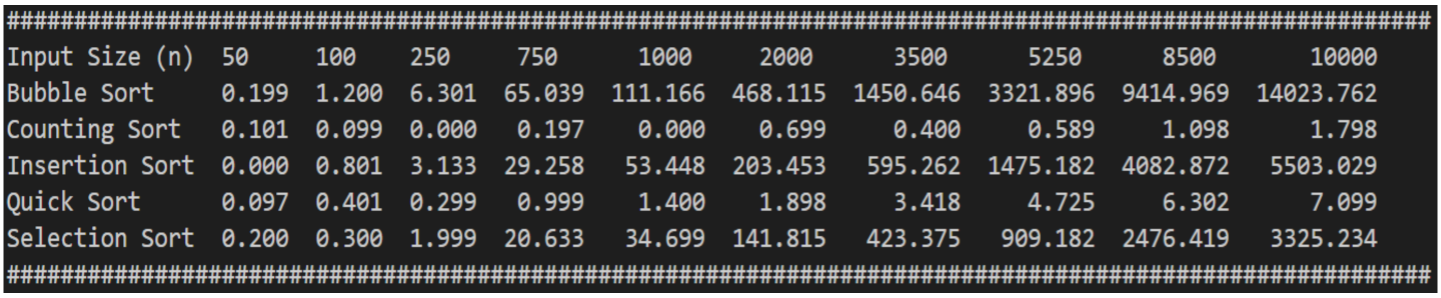


The input sizes for which to fill the arrays with are *50, 100, 250, 750, 1,000, 2,000, 3,500, 5,250, 8,500, 10,000*. Before comparing the results of the application let us remind ourselves of the average case results for each of the chosen algorithms to see if expected tends and results are met.

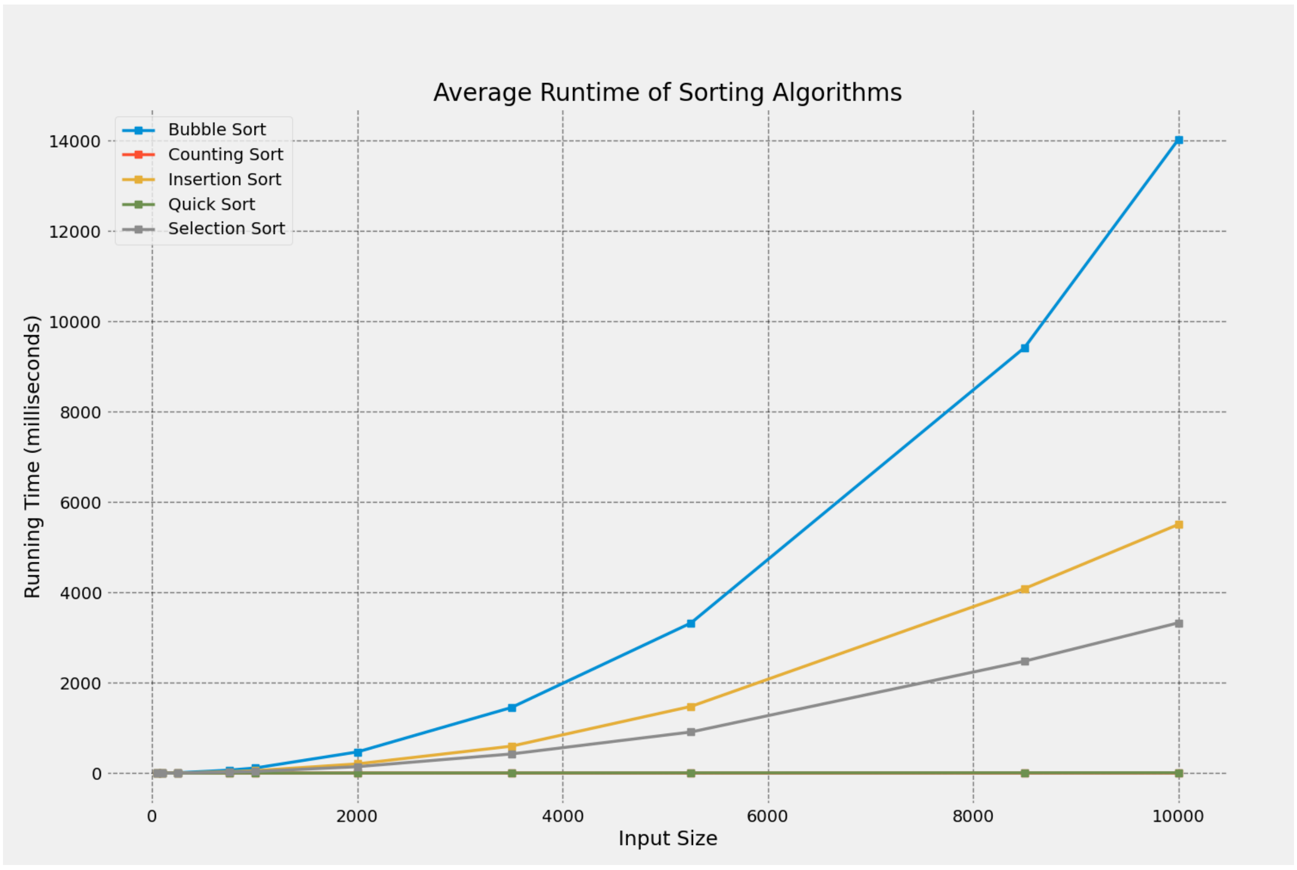
As the results are displaying the average runtimes, we will be looking at each algorithms average case. I am expecting bubble sort to be the worst of the algorithms and counting sort to be the clear leader with Quick Sort coming in as second most efficient. In general, Insertion Sort is thought to perform better than selection sort, those results will be of interest.

Bubble Sort	n2
Counting Sort	n + k
Insertion Sort	n2
Quick Sort	n log n
Selection Sort	n2

benchmark.py output:



```
############################################################################################
Input Size (n)  50     100    250    750     1000     2000      3500      5250      8500       10000
Bubble Sort     0.199  1.200  6.301  65.039  111.166  468.115   1450.646  3321.896  9414.969  14023.762
Counting Sort   0.101  0.099  0.000   0.197    0.000    0.699      0.400     0.589     1.098      1.798
Insertion Sort  0.000  0.801  3.133  29.258   53.448  203.453    595.262  1475.182  4082.872   5503.029
Quick Sort      0.097  0.401  0.299   0.999    1.400    1.898      3.418     4.725     6.302      7.099
Selection Sort  0.200  0.300  1.999  20.633   34.699  141.815    423.375   909.182  2476.419   3325.234
############################################################################################
```



<u>Observations:</u>

- The graph generated from *benchmarking.py* follows the trajectory of what would be expected as seen in the Big O-Complexity chart [41].
- Each algorithms perform very well and relatively identically for the smaller input sizes of *50* and *100*. There is minor change in the growth rate.
- Once the input size goes beyond *n=750* we can start to see the difference in time efficiency.
- As expected, bubble sort preformed the worst of the 5 algorithms. It took over *14 seconds* to sort an array of *10,000* values compared to the best performing – Counting Sort – which took just *.001* of a second.
- Bubble sort preformed worse than the other algorithms for all input sizes. Its inefficiency begins to show after *n=50*. Once the input to be sorted is over *750*, bubble sort should be avoided.
- Counting sort blows the other algorithms out of the water. In fact, there is almost a *200%* difference in its ability to sort an input size of *10,000* when compared to bubble sort.
- Although Quick sort uses recursion it held its own and was very efficient. It was the second quickest algorithm. Although again there is a clear difference between it and counting sort. This begins to kick in when *n=750*. At this point Counting Sort runs at *.197ms* and quicksort runs at *.900ms*. Although not a vast difference this trend continues for the succeeding runtimes. By the time the 10th run happens, counting sort is preforming *74.67%* more efficiently than Quick Sort.
- Insertion and Selection Sort where quite closely matched until the *n= 5,200*. After which Selection Sort began to pull away, running at *909.182ms* to Insertion Sorts *1475.182ms*, which almost *50%* faster. In general, insertion sort would be expected to perform better than selection sort. I believe the reasoning why it did not in this instance is because insertion sort works best when the array is already almost sorted whereas in this benchmarking test the arrays are randomly generated.

Conclusion

- The benchmarking process generated no major shocks. In the main, the results are as expected; bubble sort performed the by far the worst and counting sort the best. The outlier of the group was Selection sort outperforming Insertion Sort, but this can be explained by the nature of the testing. Insertion Sort works best when the input is already partially sorted whereas with this testing method each array was random.
- It is best to avoid using simple comparison-based sorting algorithms (bubble, selection, and insertion) in instances outside of learning. When compared against efficient (quicksort) and non-comparison based (counting sort) sorting algorithms they are far too slow, especially when the input array is of any great significance.
- Although not the best option for real world situations, bubble, selection, and insertion sort where a viable choice for getting to grips with the basics of sorting.
- If it is a choice between insertion and selection sort, if the input is order is totally unsorted, choose selection sort. On the other hand, if the array is partially sorted go with insertion sort.
- Counting Sort is a highly efficient sorting method and results in rapidly sorting of large arrays. Although, there may be a danger of its efficiency being thwarted if the range in which the integers fall between is quite spread out and additionally it only supports positive integers.
- Another important factor when selecting and designing an algorithm is correctness. If I had incorrectly wrote an algorithm, it would render the results invalid.
- It is important to consider the input and end goal of sorting. For example, although it was the best preforming in this test, if I were to use Counting Sort on an input with a vast range it would more than likely be outperformed.
- I was impressed with Quick Sort. It is a very efficient algorithm. The only drawback to using it as it is unstable would be if the retention of the original input is important.
- If I were to do this project again, I would veer away from the simple comparison sorts but I think as this was my first foray into sorting algorithms it was important to ground myself with a basic understanding of their workings so I could move on to more complex algorithms.

References

[1] Stone HS. Introduction to Data Structures and Computer Organization. Maidenhead, England: McGraw Hill Higher Education; 1972.

[2] Collinsdictionary.com. [cited 2022 Mar 20]. Available from: https://www.collinsdictionary.com/dictionary/english/sorting

[3] Wikipedia contributors. Sorting [Internet]. Wikipedia, The Free Encyclopedia. 2022. Available from: https://en.wikipedia.org/w/index.php?title=Sorting&oldid=1068875397

[4] Wikipedia contributors. Sorting algorithm [Internet]. Wikipedia, The Free Encyclopedia. 2022. Available from: https://en.wikipedia.org/w/index.php?title=Sorting_algorithm&oldid=1077718096

[5] Sorting algorithms explained with examples in python, java, and C++ [Internet]. freeCodeCamp.org. 2019 [cited 2022 Mar 20]. Available from: https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/

[6] Heineman GT, Pollice G, Selkow S. Algorithms in a Nutshell [Internet]. O'Reilly Media; 2008. Available from: https://books.google.at/books?id=qhaOxkQANEgC

[7] Difference between posteriori and Priori analysis [Internet]. GeeksforGeeks. 2019 [cited 2022 Mar 21]. Available from: https://www.geeksforgeeks.org/difference-between-posteriori-and-priori-analysis/

[8] Why is Time Complexity Essential and What is Time Complexity? [Internet]. GreatLearning Blog: Free Resources what Matters to shape your Career! 2022 [cited 2022 Mar 21]. Available from: https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/

[9] What are a posteriori and a priori analyses of algorithm operations? [Internet]. Stack Overflow. [cited 2022 Mar 21]. Available from: https://stackoverflow.com/questions/16052457/what-are-a-posteriori-and-a-priori-analyses-of-algorithm-operations

[10] Merriam A. Space and time complexity in computer algorithms [Internet]. Towards Data Science. 2021 [cited 2022 Mar 21]. Available from: https://towardsdatascience.com/space-and-time-complexity-in-computer-algorithms-a7fffe9e4683

[11] Wikipedia contributors. Big O notation [Internet]. Wikipedia, The Free Encyclopedia. 2022. Available from: https://en.wikipedia.org/w/index.php?title=Big_O_notation&oldid=1076386956

[12] Khalid S, Al-Kharabsheh IM, Alturani A, Mahmoud I, Nabeel I, Alturani IM, et al. Review on sorting algorithms A comparative study [Internet]. Psu.edu. [cited 2022 Mar 21]. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.736.3357&rep=rep1&type=pdf

[13] Erickson J. Hacking: The art of exploitation, 2nd edition: The art of exploitation. San Francisco, CA: No Starch Press; 2007.

[14] Big-O notation, Omega notation and Big-O notation (asymptotic analysis) [Internet]. Programiz.com. [cited 2022 Mar 22]. Available from: https://www.programiz.com/dsa/asymptotic-notations

[15] Time and Space Complexities of all Sorting Algorithms [Internet]. Interviewkickstart.com. [cited 2022 Mar 22]. Available from: https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms

[16] Singh C. Data structure asymptotic notation [Internet]. beginnersbook.com. 2018 [cited 2022 Mar 22]. Available from: https://beginnersbook.com/2018/10/ds-asymptotic-notation/

[17] Lundy A. An overview of asymptotic notation [Internet]. Level Up Coding. 2021 [cited 2022 Mar 23]. Available from: https://levelup.gitconnected.com/an-overview-of-asymptotic-notation-2086cef60172

[18] Space complexity of algorithms [Internet]. Studytonight.com. [cited 2022 Mar 23]. Available from: https://www.studytonight.com/data-structures/space-complexity-of-algorithms

[19] Rajinikanth. Data Structures Tutorials - Performance Analysis with examples [Internet]. Btechsmartclass.com. [cited 2022 Mar 28]. Available from: http://www.btechsmartclass.com/data_structures/performance-analysis.html

[20] In-place algorithm [Internet]. GeeksforGeeks. 2018 [cited 2022 Mar 28]. Available from: https://www.geeksforgeeks.org/in-place-algorithm/

[21] Difference between stable and unstable sorting algorithm - MergeSort vs QuickSort [Internet]. Blogspot.com. [cited 2022 Mar 28]. Available from: https://javarevisited.blogspot.com/2017/06/difference-between-stable-and-unstable-algorithm.html

[22] Comparators guide [Internet]. Clojure.org. [cited 2022 Mar 28]. Available from: https://clojure.org/guides/comparators

[23] Wikipedia contributors. Comparison sort [Internet]. Wikipedia, The Free Encyclopedia. 2022. Available from: https://en.wikipedia.org/w/index.php?title=Comparison_sort&oldid=1069809832

[24] Comparison of sorting algorithms [Internet]. Afteracademy.com. [cited 2022 Mar 28]. Available from: https://afteracademy.com/blog/comparison-of-sorting-algorithms

[25] Difference between comparison (QuickSort) and non-comparison (counting sort) based sorting Algorithms? Example [Internet]. Blogspot.com. [cited 2022 Mar 28]. Available from: https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html

[26] mxm. What is recursive and non-recursive algorithm? [Internet]. faq-ans.com. Anonim; 2021 [cited 2022 Mar 28]. Available from: https://faq-ans.com/en/Q%26A/page=240b54d4ed41b455b9a2aa02108f7c2e

[27] Programming with Mosh. Insertion Sort algorithm made simple [sorting algorithms] [Internet]. Youtube; 2020 [cited 2022 Mar 29]. Available from: https://www.youtube.com/watch?v=nKzEJWbkPbQ

[28] Sambol M. Bubble sort in 2 minutes [Internet]. Youtube; 2016 [cited 2022 Mar 30]. Available from: https://www.youtube.com/watch?v=xli_FI7CuzA

[29] Bubble Sort algorithm [Internet]. Studytonight.com. [cited 2022 Mar 30]. Available from: https://www.studytonight.com/data-structures/bubble-sort

[30] Data Structure and Algorithms - Quick Sort [Internet]. Tutorialspoint.com. [cited 2022 Mar 30]. Available from: https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm

[31] Quick Sort - javatpoint [Internet]. www.javatpoint.com. [cited 2022 Mar 30]. Available from: https://www.javatpoint.com/quick-sort

[32] Quicksort: Choosing the pivot [Internet]. Stack Overflow. [cited 2022 Mar 30]. Available from: https://stackoverflow.com/questions/164163/quicksort-choosing-the-pivot

[33] Counting sort (with code in Python/C++/Java/C) [Internet]. Programiz.com. [cited 2022 Apr 1]. Available from: https://www.programiz.com/dsa/counting-sort

[34] Selection sort (with code in Python/C++/Java/C) [Internet]. Programiz.com. [cited 2022 Apr 1]. Available from: https://www.programiz.com/dsa/selection-sort

[35] Wikipedia contributors. Benchmark (computing) [Internet]. Wikipedia, The Free Encyclopedia. 2022. Available from: https://en.wikipedia.org/w/index.php?title=Benchmark_(computing)&oldid=1077164968

[36] Python time module [Internet]. Programiz.com. [cited 2022 Apr 17]. Available from: https://www.programiz.com/python-programming/time

[37] Contributor T. What is epoch? [Internet]. SearchDataCenter. TechTarget; 2021 [cited 2022 Apr 17]. Available from: https://www.techtarget.com/searchdatacenter/definition/epoch

[38] Creating a DataFrame from arrays and lists - data analysis & processing with pandas [Internet]. Educative: Interactive Courses for Software Developers. [cited 2022 Apr 17]. Available from: https://www.educative.io/courses/data-analysis-processing-with-pandas/q2pYRjnxlGR

[39] Pandas DataFrame: stack() function [Internet]. w3resource. [cited 2022 Apr 17]. Available from: https://www.w3resource.com/pandas/dataframe/dataframe-stack.php

[40] Plotting [Internet]. Github.io. [cited 2022 Apr 18]. Available from: https://swcarpentry.github.io/python-novice-gapminder/09-plotting/index.html

[41] Know thy complexities! [Internet]. Bigocheatsheet.com. [cited 2022 Apr 18]. Available from: https://www.bigocheatsheet.com/

[42] 8.3. Comparing algorithms — OpenDSA data structures and algorithms modules collection [Internet]. Cs.vt.edu. [cited 2022 Apr 18]. Available from: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/AnalIntro.html

[43] Atallah M. Algorithms and theory of computation handbook [Internet]. Atallah MJ, editor. Boca Raton, FL: CRC Press; 1998. Available from: http://dx.doi.org/10.1201/9781420049503