Contents

[1.0 Introduction 1](#_Toc103527774)

[1.1 Sorting 1](#_Toc103527775)

[1.2 Real World Example 2](#_Toc103527776)

[1.3 Comparison sorts and Comparator Functions 2](#_Toc103527777)

[1.4 Partition-Based Sorting 4](#_Toc103527778)

[1.5 Properties for Sorting Algorithms 4](#_Toc103527779)

[1.5.1 Time and Space complexity 4](#_Toc103527780)

[1.5.2 Stability 6](#_Toc103527781)

[1.5.3 In-place Sorting 6](#_Toc103527782)

[2.0 Sorting Algorithms 7](#_Toc103527783)

[2.1 Bubble Sort 7](#_Toc103527784)

[2.2 Merge Sort 10](#_Toc103527785)

[2.3 Bucket Sort 14](#_Toc103527786)

[2.4 Selection Sort 17](#_Toc103527787)

[2.5 Quick Sort 19](#_Toc103527788)

[3.0 The Application and Results 22](#_Toc103527789)

[3.1 Building the application 22](#_Toc103527790)

[3.2 Analysing the results 22](#_Toc103527791)

[4.0 Bibliography 25](#_Toc103527792)

# 1.0 Introduction

## 1.1 Sorting

Sorting is the arrangement of data (values or items) according to pre-defined ordering rules[[1]](#footnote-1).

A collection of values or items are deemed to be sorted if each item in the collection is less than or equal to its successor. [[2]](#footnote-2)

Sorting is one of the most thoroughly studied algorithms in computer science[[3]](#footnote-3), mainly because it has been claimed that up to 25% of all CPU cycles are spent sorting. In the context of this course sorting data is one of the most fundamental parts of Data Analysis. To gain insights into the data and sometimes visualize it in a coherent way the data may require sorting in a specific manner. Sorting also helps with other processes on data such as searching, for a unique or duplicate element, these are sometimes far easier to find if the data is sorted using a particular method. Also analysing distributions from an array (something that often comes up in data analysis) is much less complicated if the data can be sorted in a comprehensible manner.

When I think of sorting, a list of jumbled numbers is the first thing that comes to my mind, but it is vital to understand that although this project will be using numeric data (integers and floats) to demonstrate the different types of sorting algorithms, many other data types such as strings and bytes can also be sorted.

It is a very human trait to put things in some type of order, so normally we have a good concept of basic sorting – however in computing and Data Analysis in particular, sorting the data by hand is often an impossible task, mainly because the volume of data is just so massive. A one-off sort of 10 numbers, even 50-100 numbers manually may sometimes be more efficient than writing an algorithm, however, what happens if you have 10,000 or 10,000,000 values, or you need to do that small sort 100 times a day – manual sorting is just not viable, automating the task makes much more sense.

Since sorting is such a cornerstone of computing, thousands and thousands of hours have gone into studying the most efficient (and fastest) ways to sort, while using the least amount of processing power. These two elements are often in contradiction as sometimes speed is the priority, and other times salvaging processing power is the greater need – an example of this would be on a production line.

## 1.2 Real World Example

If a machine is packing units of a product into a box it may want to capture and sort the serial numbers of each unit. We’ll say it does this sort once the box is full and there are 1000 units per box. If the cycle time on the line is tight (if that process is a bottleneck), then the time taking for the sorting algorithm to run would be the priority as you would not want to be waiting for the sorting to happen and adding to the cycle time of the process. However, if the previous process was the bottleneck – say it was doing a final test on the units before pack-out and this was slower than the packing process, then time would not be as significant a factor and you may decide that the processing power is more valuable as the machine interface may also be doing some other work that requires heavy processing power (e.g. using an automated robot or generating a live statistical data using graphics).

Because of the different requirements for different scenarios many sorting algorithms have been created over the years. Some focused on time, others focused on processing power, and some balanced as best as possible for the specified circumstances. It is important to know which algorithms suit your specific need. Here I will go through some of the factors that are taken into account when choosing a sorting algorithm.

## 1.3 Comparison sorts and Comparator Functions

In my opening paragraph, I defined sorting and in that definition the term “*less than”* is used. What do I mean by “*less than*” here? Well, that depends on the data type. Numeric sorting is the easiest to wrap your head around as each number has a predefined set value that is clear to the observer – but what if it is a list of words, is the value of the item the number of characters in the word? Is the ordering required lexicographical (alphabetical)? How will it know to sort the words in a specific order? Well, that really depends on the items in the collection and the application in question. [[4]](#footnote-4) Some items may be given a specific value by the application if so instructed – this would be considered a custom ordering scheme.

A sorting algorithm is called comparison-based if the only way to gather information about the total order is by comparing two of the elements at a time using the comparator ≤ (less than or equal to). These algorithms are popular and used in many cases as they don't care what type of data is being sorted, all they are doing is comparing each element of the data using the (X, Y) method.[[5]](#footnote-5)

e.g.

$\displaystyle \ensuremath{\mathtt{compare(x,y)}}
\begin{cases}
{}<0 & \text{...
...{}=0 & \text{if $\ensuremath{\mathtt{x}}=\ensuremath{\mathtt{y}}$}
\end{cases}$

Some examples of comparison sorts are:

* **Bubble Sort:** A comparison-based sorting algorithm that compares two neighbouring items in a list and switches their positions until they are sorted.[[6]](#footnote-6)
* **Insertion Sort:** An algorithm that places one unsorted element at a sorted position during eachiteration.[[7]](#footnote-7)
* **Selection Sort:** An algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list. **[[8]](#footnote-8)**
* **Quick Sort:** An array is divided into subarrays by selecting a pivot element (element selected from the array). While dividing the array, the pivot element should be positioned in such a way that elements less than the pivot are kept on the left side, and elements greater than the pivot are on the right side of the pivot. Those arrays are then sub-divided using the same method until esub-arrayrray has a single element – they are then combined and output as a sorted list.[[9]](#footnote-9)
* **Merge Sort:** Similar to the quicksort this is a divide and conquer recursive algorithm. The array is recursively divided into two halves till the size becomes 1. Once this has happened (base case is reached), the merge processes come into action and start merging arrays back until the complete sorted array is output.[[10]](#footnote-10)

## 1.4 Partition-Based Sorting

This is also a type of divide and conquer method of sorting but without comparisons. For example in a bucket sorting algorithm the values are divided into “buckets” and then the method for sorting each bucket is called recursively until all the elements have been sorted in each bucket, then the final sorted buckets are combined to output the sorted array.[[11]](#footnote-11) We do this sometimes in life, if I have a big jar of coins and I get 2 of my friends to help me sort the coins into groups of their value, we will each sort our coins and then combine – this is an ok example, but for a true bucket sort we would have to sort each pile of change using the same method and then combine our sorted piles and sort the entire pile once again using the same method.

## 1.5 Properties for Sorting Algorithms

#### 1.5.1 Time and Space complexity

The time complexity of an algorithm states the amount of time an algorithm takes to run in terms of the characteristics of the input data. This is an estimate of the total number of operations performed in the sort program. It is understood that each sorting operation takes a set time to be completed.

There are several notations used for expressing time complexity

* Big O Notation: O – The notation O(n) expresses the upper bound of an algorithm's running time.
* Omega Notation: Ω - The notation Ω (n) expresses the lower bound of an algorithm's running time.
* Theta Notation: θ - The notation θ(n) lies between O(n) and Ω(n) and is used to express the exact asymptotic behavior of an algorithm’s running time.[[12]](#footnote-12)

For clarity, upper bound is the maximum time a program will take to produce an output, depending on the input i.e. the “Worst Case Scenario”, inversely, lower bound is the minimum time a program will take to produce an output, depending on the input i.e. the “Best Case Scenario”. So upper bound is the worst possible input for the algorithm to try to sort, and lower bound is the best case possible for the algorithm to sort.

These cases can be tested using a sorted input to see best case, multiple random inputs analyzed for average case, and reverse sorted input (Worst case, where every element has to be sorted).

This was a little hard to get my head around initially, however, this graph (similar to the one shown in the lectures) gave me a better understanding of Big O Notation.

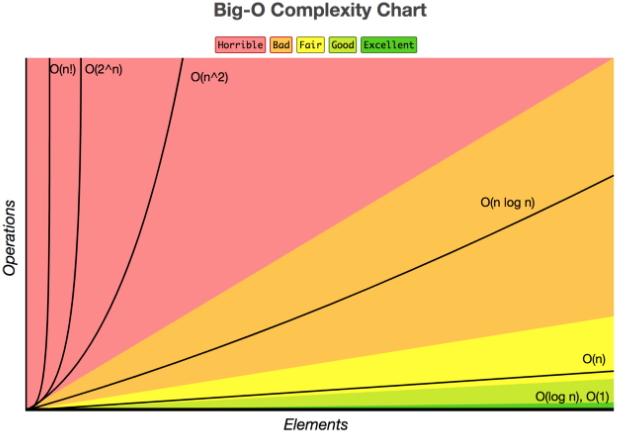


Image Source: <https://betterprogramming.pub/big-o-notation-for-dummies-like-me-98ac2d141f9f>

For different input data sizes n (e.g. the number of elements in the array) there will be different resource consumption N (e.g. time, memory). See a table below that outlines the upper, lower, and average bounds for the different sorting algorithms, **NB** – *the Selection sort, Quick sort and Heap sort would be considered unstable by definition but it is possible to get stable versions for some of those sorting algorithms*. [[13]](#footnote-13)

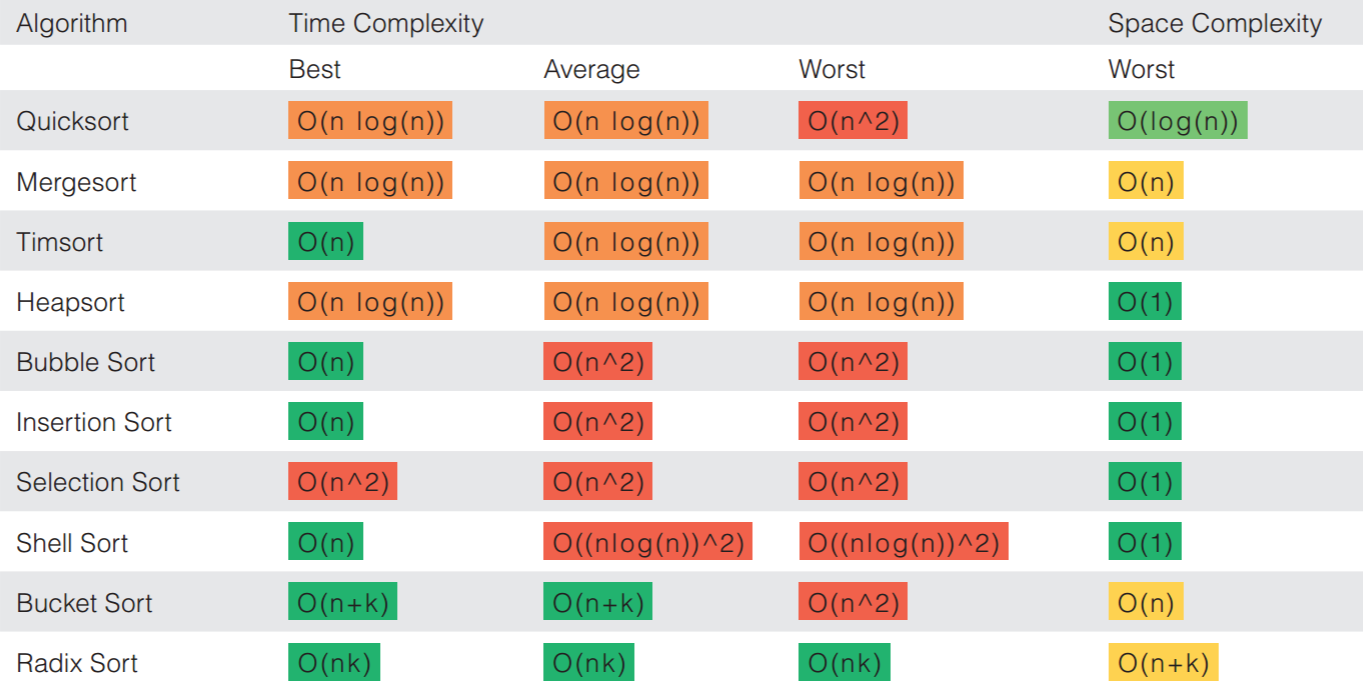


Image Source: <https://zaxrosenberg.com/must-know-sorting-algorithms-in-python/>

O(n) or linear time usually represents an algorithm that iterates through each item in the list and uses them to complete the sort task either by comparison or some other method – these are usually considered brute force algorithms as every single item is sorted, even if it was already sorted at input.

O(log n) or logarithmic time uses a binary type sorting method where the list is halved and one half is sorted, then half that and repeat the process until the data is sorted in the specified manner.

O(n²) or quadratic time can be very effective with small data sets but it can get slow extremely quickly as this normally deals with nested data (lists within lists) so the growth can be exponential and very time consuming with unsuitable inputs.[[14]](#footnote-14)

#### 1.5.2 Stability

Stability in this context is not focused on the integrity of the actual algorithm, it is related to the sorting of equal elements in a list. A stable sorting algorithm is when the relative order of elements with equal value is maintained, while unstable sorting algorithms do not. An unstable algorithm will sort the data but the integrity of the relative order of the values may not be preserved. It is important that if the relative order of the input needs to be maintained, the correct algorithm type is chosen. [[15]](#footnote-15)

#### 1.5.3 In-place Sorting

This is extremely relevant if you have concerns for the memory requirements of a machine when running the algorithm. A sorting algorithm is considered “I*n-Place*” if it only uses a set amount of memory, autonomous of the input size. A broad definition I found helpful was:

“*In-place means that the algorithm does not use extra space for manipulating the input but may require a small though non-constant extra space for its operation. Usually, this space is O(log n), though sometimes anything in O(n) (Smaller than linear) is allowed.*”[[16]](#footnote-16)

# 2.0 Sorting Algorithms

## 2.1 Bubble Sort

**Definition:** Bubble Sort **–** “*A sorting algorithm that compares two adjacent elements and swaps them until they are in the intended order.”[[17]](#footnote-17)*

The first algorithm I am going to introduce is the bubble sort. It is so named as the concept is that it will sort the list by allowing either the lower or higher values “*bubble*” to the top.

**Key Properties:**

1. **Time Complexity(Upper Bound):** *O(n²)*
2. **Time Complexity(Lower Bound):** O(n)
3. **Time Complexity(Average):** *O(n²)*
4. **Space Complexity:** *O(1)*
5. **Stable:** Yes
6. **In-Place:** Yes

Real life examples of bubble sort are hard to find an it is rarely implemented in complex programs as the inputs have to be relatively small and it helps if the values are somewhat sorted, or else it will become inefficient.

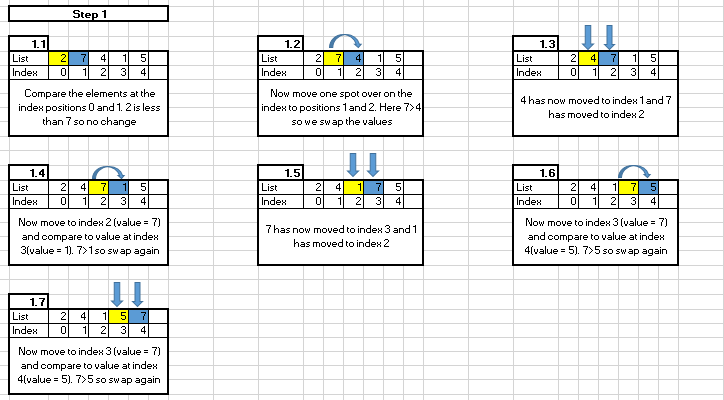
The below diagrams show a simplified example of how a bubble sort method is processed.

It shows the iteration over the elements in a list.

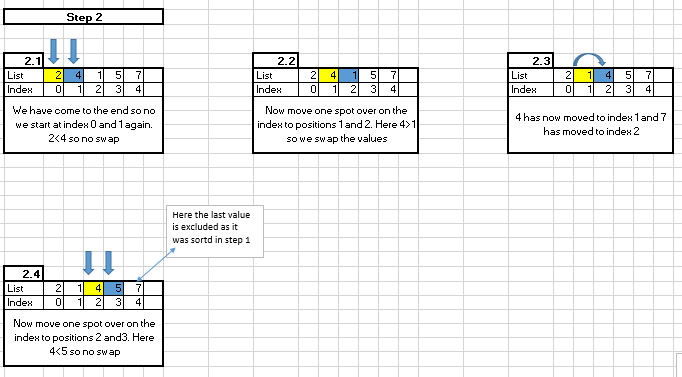
In step 1 the first two elements are compared if element at index 0 > element at index 1 then they are swapped, if not they stay in place. Then 1 step over on the index and repeat the process until the end of the array. This “*pushs*” the highest value element to its final sorted position at the end of the array.

**Input:** [2,7,4,1,5]

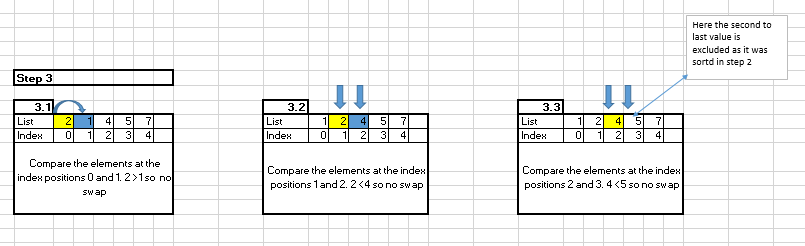
**Desired output:** [1,2,4,5,7]



In Step 2 the same process is followed, but the last element can be ignored as we know it is already in the sorted position.



In step 3 there is now more swap and the rest of the array is sorted, it has been sorted iteratively in each time the method pass over the list.



**Example of Algorithm Implementation With Code:**

**Code and Input:**

**Output:**

## 2.2 Merge Sort

**Definition:** “*A sorting technique that sequences data by continuously merging items in the list. Every single item in the original unordered list is merged with another, creating groups of two. Every two-item group is merged, creating groups of four and so on until there is one ordered list.”[[18]](#footnote-18)*

**Key Properties:**

1. **Time Complexity(Upper Bound):** *O(nlogn)*
2. **Time Complexity(Lower Bound):** *O(nlogn)*
3. **Time Complexity(Average):** *O(nlogn)*
4. **Space Complexity:** *O(n)*
5. **Stable:** Yes
6. **In-Place:** No

Merge sort is used in some computing, often considered in database scenarios – because they are stable and can handle some distributed scenarios where supplementary data may arrive after the initial sort.

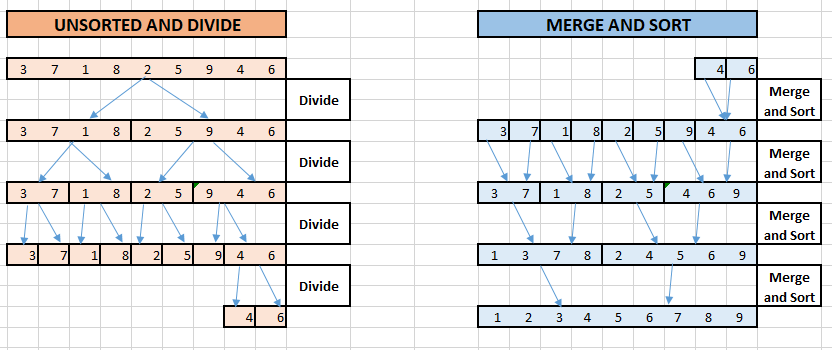
Merge Sort is similar to quicksort and similarly operates using the divide and conquer principle.

It is an efficient sort and one of the preferred algorithms used in linked lists as it is a stable algorithm.[[19]](#footnote-19)

The array of size *n* is split in to half and this step happens recursively until subarrays of a length of 1 are created.

Both of these subarrays are merged so a sorted array is generated from each of the sub arrays. This step happens repeatedly until we are left with one sorted array.

**See excel diagram, inspired by** [**https://www.happycoders.eu/algorithms/merge-sort/**](https://www.happycoders.eu/algorithms/merge-sort/)**, that shows the steps:**



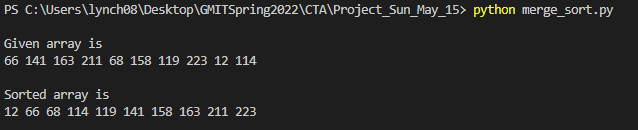
**Example of Algorithm Implementation With Code:**

**Code and Input:**





**Output:**



## 2.3 Bucket Sort

**Definition:** “*Bucket Sort is a non-comparison sorting algorithm that divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm. The sorted buckets are then combined to form the sorted list”[[20]](#footnote-20)*

**Key Properties:**

1. **Time Complexity(Upper Bound):** *O(n²)*
2. **Time Complexity(Lower Bound):** O(n+k)
3. **Time Complexity(Average):** O(n+k)
4. **Space Complexity:** *O(nk)*
5. **Stable:** Yes
6. **In-Place:** No

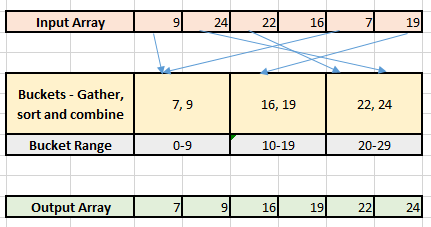
Steps to creating a bucket sort algorithm.

1. Create an empty array
2. Loop through the original input array and put each item in a “bucket”.
3. Sort the buckets that are not empty
4. Check the buckets in order to make sure they are sorted and then combine the buckets to generate sorted array.[[21]](#footnote-21)

Those steps seem simple but the bucket sort can be one of the more complex sorting algorithms as some scenarios there can be an argument to use different sorting algorithms, one to sort the elements *into* buckets and another to sort the elements *in the* buckets. A recursive function, a function that calls itself until its base case is reached, can also be used to achieve this.

Bucket sorts usually work best over a known range of elements as the algorithm can then be optimised if the range is known. The elements are then assigned to the appropriate buckets (uniform data is preferred), sorted and combined.

**See simple diagram to show a bucket sort method below:**



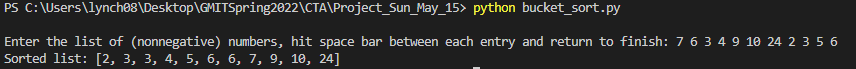
**Example of Algorithm Implementation With Code:**

**Code – Input Positive numbers**





**Output:**



## 2.4 Selection Sort

Definition: “*The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.”*

**Key Properties:**

1. **Time Complexity(Upper Bound):** *O(n²)*
2. **Time Complexity(Lower Bound):** *O(n²)*
3. **Time Complexity(Average):** *O(n²)*
4. **Space Complexity:** *O(1)*
5. **Stable:** No
6. **In-Place:** Yes - *O(n²)*

Similar to Bubble sort selection sort is a comparison based sorting algorithm. It is still impractical for real world uses if the data input size is anyway significant as it is an in-place algorithm with *O(n²)* for all time complexity calculations.

The algorithm by design will have 2 sub arrays:

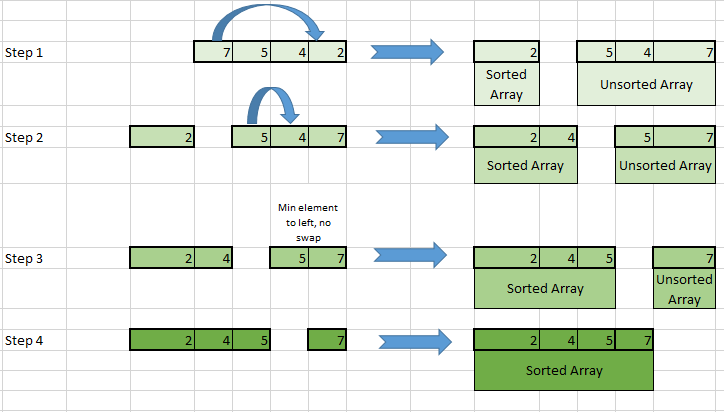
1. The sorted subarray

2. The remaining unsorted original array

How the sorting algorithm works is it sets the first element to a variable, lets call it *Min. Min* is then compared to the second element and if the second element is less-then, the second element becomes *min,* otherwise do nothing and first element stays as *min.* Then move on to the third element and so on so forth to the end of the array.

After each iteration the *min* variable is placed to the front of the unsorted array and indexing starts from the unsorted element. These steps are repeated until all the elements are placed in the correct positions.

**See diagram to show method below:**



**Example of Algorithm Implementation With Code:**

**Code and Input:**



**Output:**



## 2.5 Quick Sort

Definition: *“Quicksort picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of QuickSort that pick pivot in different ways*.”[[22]](#footnote-22)

**Key Properties:**

1. **Time Complexity(Upper Bound):** *O(n²)*
2. **Time Complexity(Lower Bound):** *O(n\*logn)*
3. **Time Complexity(Average):** *O(n\*logn)*
4. **Space Complexity:** *O(logn)*
5. **Stable:** No
6. **In-Place:** No

Quicksort algorithms are often used in different computing scenarios and have traditionally been used for software that requires repeated runtimes of the algorithm. However it is important when using on an application where security is a priority as its upper bound time complexity could consume required memory which can lead to Denial of Service crash – this is when the machine is flooded by requests and the system is overloaded causing system malfunctions.[[23]](#footnote-23)

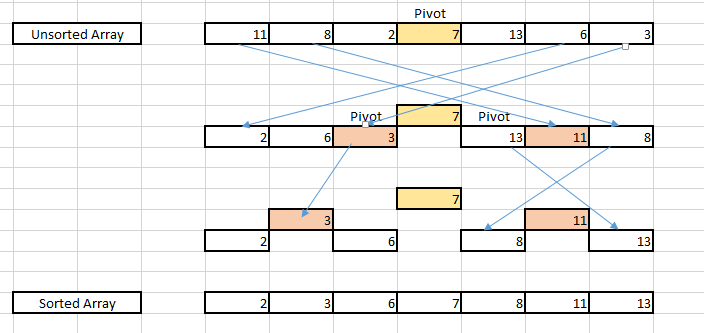
The two key decisions to be made when creating a quicksort algorithm is the pivot point and the partition.

The structure of the input data should help the algorithm creator determine the best position of the pivot point.

* Nearly sorted – Pivot Point at the beginning(first element)
* Nearly reverse sorted – Pivot Point at the end(last element)
* Unknown/Random data – Use middle or random element for pivot for best chance to avoid upper bound (*O(n²)*).

In quicksort, we select a pivot item from the array and immediately compare everything to it, putting the smaller elements at the beginning and larger elements at the end of the array. Quicksort then recursively sorts the beginning of the array and the end of the array, while the random binary search tree recursively inserts the smaller elements in the left subtree of the root and larger elements in the right subtree of the root.[[24]](#footnote-24)

**Here is a simplified diagram using the middle value element as the pivot:**



**Example of Algorithm Implementation With Code:**

**Code – Input Positive numbers**





**Output:**



# 3.0 The Application and Results

## 3.1 Building the application

Building the application was by far the hardest part of this project. I did a lot of google research including researching former students’ projects on Github such as [Slawak1](https://github.com/Slawak1/GMIT-CTA-Project-2020), [SeanOhAileasa](https://github.com/SeanOhAileasa/cta-sorting/blob/main/cta-sorting.ipynb) and [andkoc001](https://github.com/andkoc001/CTA_sorting_project). These were a great help to get me started and I used the layout of their code to help me get an idea of how to structure my own code.

I began by creating all my sort functions in individual files ensuring they would run correctly by hardcoding multiple arrays into the inputs so I could see that they were sorting the way I was expecting them to. Then, using *Numpy*, I created arrays of different lengths with random integers to do the same – printing out the arrays, before and after the sorting functions had run to ensure they were working. Finally I added the code to benchmark the time average time of each of the sorting algorithms individually, depending on input size. This took the average of 10 sorts (each sort had different values) of 9 different array sizes and output the results in table format on the command line – the code for this was put together from multiple *stackoverflow* entries that are referenced in the code comments.

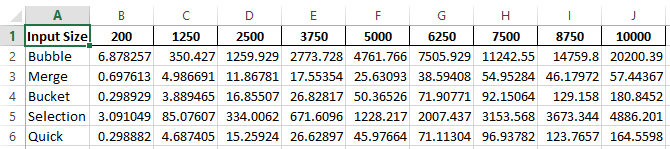
This worked well and I was able to then combine all of my individual files into one file in order to output the timing of each of my sorts. One thing that caused me endless consternation was trying to create the benchmark in a way that would be dynamic so I could just call any of the sort functions to use it, however this did not go as planned as the inputs required for some of the functions such as the quick sort, required the length of the array to be passed to the function where as other functions only wanted the array itself. This, plus the added benefit of being able to copy and paste the benchmark code from the main file and being able to run it in an individual file for debug (only took a fraction of the time to run) when I ran into any issues were the main reasons I did not pursue that route any further, even though the code would have been more efficient.

Outputting the data in table format on the command line took a little research, but again some investigation through google and *stackoverflow*, helped me find the relevant method to do this. The application will also output the data and the graph to the local folder it is run from.

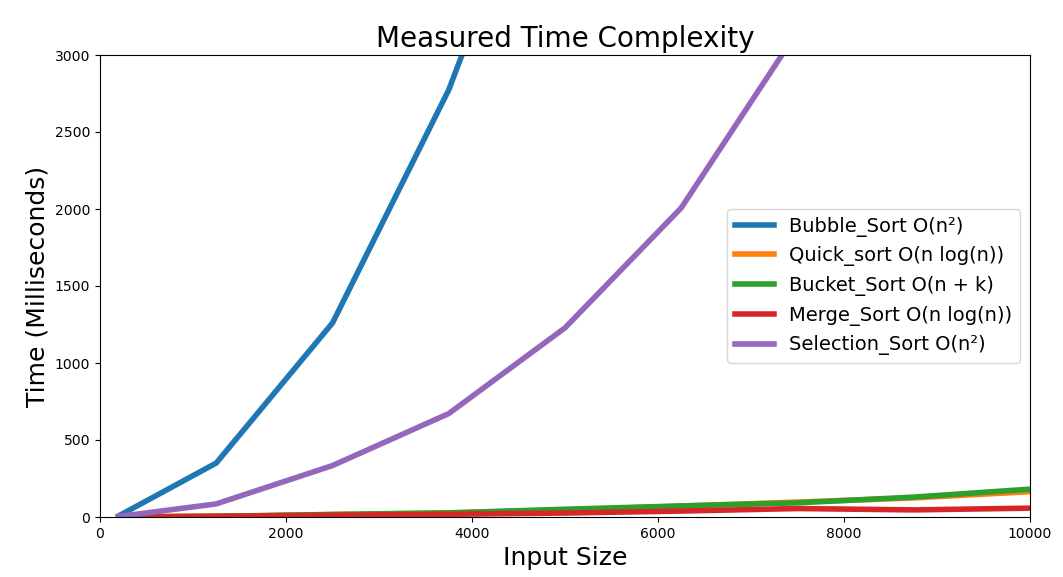
## 3.2 Analysing the results

Below is a screen shot of the table (output to excel) and the graph generated from my application. As mentioned above the application will also output these to local files (table to output.xlxs and plot to Benchmark\_Sort\_Times.png) the application is run from.

***From excel output – generated by application***



***Graph representing data from excel output:***



The headings horizontally on the table are the input sizes of the arrays with the headings on left the types of sorting algorithms that sorted the data values. The figures represent the average (of 10 sorts) amount of time in milliseconds it took for that particular sorting algorithm to be completed of that array size. As you can see, as you would expect, in general, the bigger the array size, the longer average time that is being returned to do the sort.

From both the graph and the table we can see that the time taken to do the sort gets exponentially longer for both the Selection sort and especially the Bubble sort when the size of the sorts become larger. From my earlier research of these algorithms this was expected and it is an excellent representation of the reason why these sorts are rarely used in real life computing.

The Merge sort as you can see, from the red line in the plot, seems to be the fastest – surprisingly the average sort time was slightly faster for 8750 values than it was for 7500 values in this run. This is because the time complexity for the Merge sort is (n (log(n)) for the upper bound, lower bound **and** average cases. One reason for this may be that the random data that was generated was easier to sort, maybe the values for the 7500 size run was more unsorted than it was for the 8750 size run.

The Bucket and Quick sorts take roughly the same amount of time, however from the graph and the inputs to the table it looks like that if I was to do analysis with some bigger size arrays we would begin to see a larger discrepancy between their efficiency, further research would have to be carried out here to definitively say which is the better sorting algorithm for the random data being generated.

3.3 Conclusion

In general I enjoyed doing this project. I had never really considered the vast amount of thought that has gone into sorting data, however between the video lectures, the notes, and my own research on Google and YouTube it is clear that if sorting is required, the correct algorithm for your input is vital, for both the efficiency of the process and memory consumption required of your machine. One thing that I would note is that although the graph that was generated came out with the timing data pretty representative of the description I had done of each of the sorts in section 2, I would like to highlight that if this analysis was being done for an application for a machine that was running other applications, the analysis should be done on a replica environment as your algorithm may interfere with the machines resources and impede the efficacy of the other processes. CPU design and the System Architecture of a machine may influence the efficiency of your algorithm.

# 4.0 Bibliography

* Mannion, P., n.d. Computational Thinking with Algorithms lecture Notes – Sorting Algorithms Part 1,2 and 3. GMIT
* Python, R., 2022. Sorting Algorithms in Python – Real Python. [online] Realpython.com. Available at: ><https://realpython.com/sorting-algorithms-python>/<
* Opendatastructures.org. 2022. 11.1 Comparison-Based Sorting. [online] Available at: >[https://www.opendatastructures.org/ods-java/11\_1\_Comparison\_Based\_Sorti.html<](https://www.opendatastructures.org/ods-java/11_1_Comparison_Based_Sorti.html%3c)
* Programiz.com. 2022. Bubble Sort (With Code in Python/C++/Java/C). [online] Available at: ><https://www.programiz.com/dsa/bubble-sort#:~:text=Bubble%20sort%20is%20a%20sorting,is%20called%20a%20bubble%20sort.<>
* Programiz.com. 2022. Insertion Sort (With Code in Python/C++/Java/C). [online] Available at: >[https://www.programiz.com/dsa/insertion-sort<](https://www.programiz.com/dsa/insertion-sort%3c)
* GeeksforGeeks. 2022. Merge Sort - GeeksforGeeks. [online] Available at: >[https://www.geeksforgeeks.org/merge-sort/<](https://www.geeksforgeeks.org/merge-sort/%3c)
* Kocielski, A., 2022. CTA\_sorting\_project/Benchmark\_sorting\_algorithms.ipynb at master · andkoc001/CTA\_sorting\_project. [online] GitHub. Available at: >[https://github.com/andkoc001/CTA\_sorting\_project/blob/master/Benchmark\_sorting\_algorithms.ipynb<](https://github.com/andkoc001/CTA_sorting_project/blob/master/Benchmark_sorting_algorithms.ipynb%3c)
* Medium. 2022. Big O Notation for Dummies. [online] Available at: >[https://betterprogramming.pub/big-o-notation-for-dummies-like-me-98ac2d141f9f<](https://betterprogramming.pub/big-o-notation-for-dummies-like-me-98ac2d141f9f%3c)
* Stable Sorting Algorithms. 2022. Baeldung on Computer Science. [online]Available at: ><https://www.baeldung.com/cs/stable-sorting-algorithms#:~:text=Stable%20sorting%20algorithms%20preserve%20the,elements%20relative%20to%20one%20another.<>
* GeeksforGeeks. 2022.In-Place Algorithm - GeeksforGeeks. [online] Available at: >[https://www.geeksforgeeks.org/in-place-algorithm/<](https://www.geeksforgeeks.org/in-place-algorithm/%3c)
* GeeksforGeeks. 2022. Bubble Sort - GeeksforGeeks. [online] Available at: >[https://www.geeksforgeeks.org/bubble-sort/<](https://www.geeksforgeeks.org/bubble-sort/%3c)
* PCMAG. 2022. Definition of merge sort. [online] Available at: >[https://www.pcmag.com/encyclopedia/term/merge-sort<](https://www.pcmag.com/encyclopedia/term/merge-sort%3c)
* Educative: Interactive Courses for Software Developers. 2022. Algorithms 101: how to use Merge Sort and Quicksort in JavaScript. [online] Available at: ><https://www.educative.io/blog/algorithms-101-merge-sort-quicksort#merge<>
* Programiz.com. 2022. Bucket Sort Algorithm. [online] Available at: > [https://www.programiz.com/dsa/bucket-sort<](https://www.programiz.com/dsa/bucket-sort%3c)
* Sehgal, K., 2022. An Introduction to Bucket Sort. [online] Medium. Available at: >[https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124<](https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124%3c)
* GeeksforGeeks. 2022. QuickSort - GeeksforGeeks. [online] Available at: >[https://www.geeksforgeeks.org/quick-sort/<](https://www.geeksforgeeks.org/quick-sort/%3c)
* En.wikipedia.org. 2022.
* Denial-of-service attack - Wikipedia. [online] Available at: ><https://en.wikipedia.org/wiki/Denial-of-service_attack#cite_note-1<>
* Morin, P., 2022. 11.1 Comparison-Based Sorting. [online] Opendatastructures.org. Available at: ><https://www.opendatastructures.org/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001411000000000000000<>

1. Mannion, P., n.d. *Computational Thinking with Algorithms – Sorting Algorithms Part 1*. GMIT. [↑](#footnote-ref-1)
2. Mannion, P., n.d. *Computational Thinking with Algorithms – Sorting Algorithms Part 1.*GMIT. [↑](#footnote-ref-2)
3. Python, R., 2022. *Sorting Algorithms in Python – Real Python*. [online] Realpython.com. Available at: <https://realpython.com/sorting-algorithms-python/> [Accessed 4 May 2022]. [↑](#footnote-ref-3)
4. G Mannion, P., n.d. *Computational Thinking with Algorithms – Sorting Algorithms Part 1*. GMIT [↑](#footnote-ref-4)
5. Opendatastructures.org. 2022. *11.1 Comparison-Based Sorting*. [online] Available at: <https://www.opendatastructures.org/ods-java/11\_1\_Comparison\_Based\_Sorti.html> [Accessed 4 May 2022]. [↑](#footnote-ref-5)
6. Programiz.com. 2022. *Bubble Sort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/bubble-sort#:~:text=Bubble%20sort%20is%20a%20sorting,is%20called%20a%20bubble%20sort.> [Accessed 4 May 2022]. [↑](#footnote-ref-6)
7. Programiz.com. 2022. *Insertion Sort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/insertion-sort> [Accessed 4 May 2022]. [↑](#footnote-ref-7)
8. Programiz.com. 2022. *Insertion Sort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/ selection-sort> [Accessed 4 May 2022]. [↑](#footnote-ref-8)
9. Programiz.com. 2022. *Insertion Sort (With Code in Python/C++/Java/C)*. [online] Available at: <https://www.programiz.com/dsa/ quick-sort> [Accessed 4 May 2022]. [↑](#footnote-ref-9)
10. GeeksforGeeks. 2022. *Merge Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/merge-sort/> [Accessed 5 May 2022]. [↑](#footnote-ref-10)
11. Kocielski, A., 2022. *CTA\_sorting\_project/Benchmark\_sorting\_algorithms.ipynb at master · andkoc001/CTA\_sorting\_project*. [online] GitHub. Available at: <https://github.com/andkoc001/CTA\_sorting\_project/blob/master/Benchmark\_sorting\_algorithms.ipynb> [Accessed 4 May 2022]. [↑](#footnote-ref-11)
12. [↑](#footnote-ref-12)
13. G Mannion, P., n.d. *Computational Thinking with Algorithms – Sorting Algorithms Part 2*. GMIT [↑](#footnote-ref-13)
14. Medium. 2022. *Big O Notation for Dummies*. [online] Available at: <https://betterprogramming.pub/big-o-notation-for-dummies-like-me-98ac2d141f9f> [Accessed 5 May 2022]. [↑](#footnote-ref-14)
15. Stable Sorting Algorithms. 2022. *Baeldung on Computer Science*. [online] Available at: <https://www.baeldung.com/cs/stable-sorting-algorithms#:~:text=Stable%20sorting%20algorithms%20preserve%20the,elements%20relative%20to%20one%20another.> [Accessed 5 May 2022]. [↑](#footnote-ref-15)
16. GeeksforGeeks. 2022. *In-Place Algorithm - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/in-place-algorithm/> [Accessed 5 May 2022]. [↑](#footnote-ref-16)
17. GeeksforGeeks. 2022. *Bubble Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/bubble-sort/> [Accessed 5 May 2022]. [↑](#footnote-ref-17)
18. PCMAG. 2022. *Definition of merge sort*. [online] Available at: <https://www.pcmag.com/encyclopedia/term/merge-sort> [Accessed 6 May 2022]. [↑](#footnote-ref-18)
19. Educative: Interactive Courses for Software Developers. 2022. *Algorithms 101: how to use Merge Sort and Quicksort in JavaScript*. [online] Available at: <https://www.educative.io/blog/algorithms-101-merge-sort-quicksort#merge> [Accessed 7 May 2022]. [↑](#footnote-ref-19)
20. Programiz.com. 2022. *Bucket Sort Algorithm*. [online] Available at: < https://www.programiz.com/dsa/bucket-sort> [Accessed 6 May 2022]. [↑](#footnote-ref-20)
21. Sehgal, K., 2022. *An Introduction to Bucket Sort*. [online] Medium. Available at: <https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124> [Accessed 7 May 2022]. [↑](#footnote-ref-21)
22. GeeksforGeeks. 2022. *QuickSort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/quick-sort/> [Accessed 8 May 2022]. [↑](#footnote-ref-22)
23. En.wikipedia.org. 2022. *Denial-of-service attack - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Denial-of-service\_attack#cite\_note-1> [Accessed 10 May 2022]. [↑](#footnote-ref-23)
24. Morin, P., 2022. *11.1 Comparison-Based Sorting*. [online] Opendatastructures.org. Available at: <https://www.opendatastructures.org/ods-java/11\_1\_Comparison\_Based\_Sorti.html#SECTION001411000000000000000> [Accessed 8 May 2022]. [↑](#footnote-ref-24)