GMIT

Computational Thinking with Algorithms

Sean Ó hÁileasa

Abstract. Arranging a collection(s) (hereafter abbreviated C including arrays and lists) of items based on some set of ordering rules is termed sorting [1]. The sorting problem orders the C in the correct order using an algorithm, being the set of rules followed [1][2]. The search for efficient sorting algorithms was a significant part of early computer science history [1].

1. Introduction

Today, many computational tasks have been simplified by first sorting the information [1]. Searching for an element(s) (hereafter abbreviated E) in a sorted C, it is probable the E sought is accessed quicker than if the E were out of order [1]. Creating a histogram of distinct E in a C, counting the number of duplicate E in a C or obtaining order statistics is a trivial task when it's known that the E will appear as a contiguous block of increasing magnitude [1].

Mathmatically, the E in a C are sorted if each is less than or equal (or valued less than) its neighbour on the right [1]. If the E at index/position i (hereafter C index/position abbreviated using $[\ ]$) is less than the E at $[\ j\ ]$ then the $[\ i\ ]$ must also be less than the $[\ j\ ]$ (Eq. 1) [1].

$$[\ i\ ] < [\ j\ ]\ then\ i < j \tag{1}$$

A sorted C with duplicate E structured $[\ i\ ] = [\ j\ ]$ within a contiguous block there too should be no other different valued E interspersed within, no $[\ k\ ]$ such that the value of $E\ i$ is less than the value of $E\ k$ and less than the value of $E\ j$ (Eq. 2) [1].

$$\begin{gathered} i < k < j \\ [\ i\ ] \neq [\ k\ ] \end{gathered} \tag{2}$$

The concept of less than is obvious when referring to integers or floating-point numbers [1]. The lexicographical ordering of words in a dictionary or a country's flag colours (Dutch National Flag Problem - Dijkstra) relates to implementing the comparator function.

The function takes two E as parameters and determines if an E is less than or equal to or greater than another [1]. The classification of a sorting algorithm is based on the definition of less than and referred to as either simple (e.g. Insertion Sort algorithm) or efficient comparison-based [1].

Comparison-based sorting algorithms are the most known and fundamental sorting algorithm [1]. Simple comparison-based sorting algorithms include Bubble Sort, Selection Sort and Insertion Sort. Efficient comparison-based sorting algorithms include Merge Sort, Quicksort and Heapsort. All employ the sorting of E by comparing relative values [1], making no assumptions about the data [1].

Inversion is an essential concept in sorting algorithms such as Insertion Sort, whereby the running time is strongly related to the number of inversions that occur in the input instance [1]. It measures how far the E in a C are from being sorted [1]. If $E\,i$ and $E\,j$ are out of order, then an inversion means both are out of order to one another [1]. The greater E must appear later in the C (Eq. 3) [1].

$$[\,i\,] < [\,j\,] \equiv i < j \tag{3}$$

Non-comparison-based sorting algorithms (e.g. Counting Sort) require knowledge of a set of assumptions being the type of input instance deployed and have better time complexity performance than the comparison-based [1].

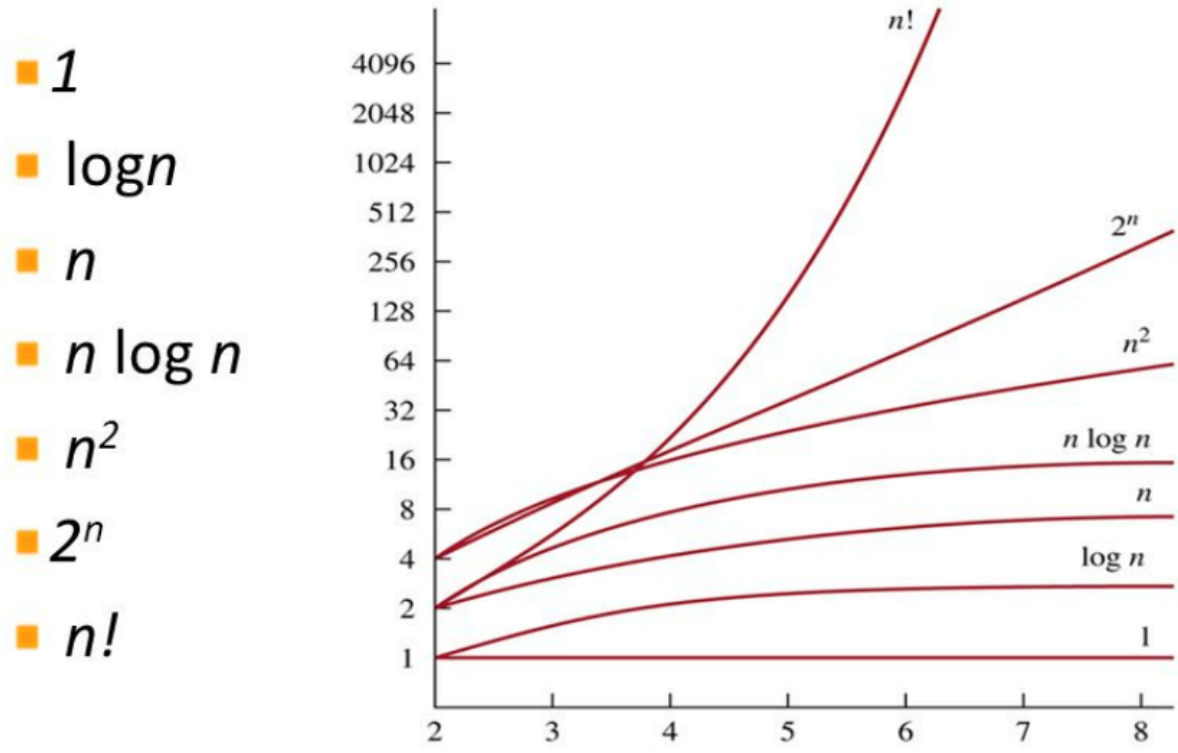
Important to note all algorithms are not created equally [2]. When choosing the best algorithm for a particular type of problem instance, one need to determine its efficiency; each has various levels [2]. The time efficiency of algorithms relates to the amount of time in terms of performance in a real-world implementation regarding time or number of operations required [2]. There are two options available for analysing algorithmic efficiency: i) a priori analysis or; ii) a posteriori analysis (empirical analysis) [2]. This project evaluates efficiency from a theoretical perspective or a priori analysis [2]. Analysing sorting algorithms by comparing the order of growth for the size of the input instance (hereafter abbreviated n) [2].

Time complexity relates to the time needed to run an algorithm or the number of operations performed [2]. It is not associated with the n or speed of the machine on which it's run [2]. It is only concerned with how the algorithm scales as n increases [2]. Crucial to consider when searching or sorting is how the algorithm execution time may vary with an increasing n to determine how the algorithm scales [2]. This mathematical analysis will provide a platform-neutral/agnostic way of comparing the expected performance/complexity of the algorithm [2].

Running Time $T(n)$ is proportional to	Complexity:
$T(n) \propto log\, n$	logarithmic
$T(n) \propto n$	linear
$T(n) \propto n\, log\, n$	linearithmic
$T(n) \propto n^2$	quadratic (polynomial)
$T(n) \propto n^3$	cubic (polynomial)
$T(n) \propto n^k$	polynomial (k any integer)
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n; k > 1$	exponential

Growth Functions used in Complexity Analysis (4)

It's mathematically proven that comparison-based sorting algorithms cannot do better than $O(n\, log\, n)$ performance in the average or worst case [1]. The performance of the non-comparison-based sorting algorithms time complexities in all instances (best/worst/average), when compared with the non-comparison based sorting algorithms (simple/efficient), is going to exceed the upper bound $O(n\, log\, n)$ [2]. The plot represents the different complexity families used for algorithmic analysis [2].



Z. Relph, "Chapter Algorithms 3.2 The Growth of Functions," slideplayer, June 2018.

Hybrid sorting algorithms such as Introsort and Timsort combine the properties of two or more different algorithms to achieve an overall algorithm that is better than the sum of its parts [3], leveraging the best properties of both or all algorithms combined [1]. The performance of hybrid sorting algorithms when compared with the efficient comparison-based sorting algorithms (Merge Sort/Quicksort/Heapsort) can achieve $O(n\ log\ n)$ time complexity in the worst case and average cases [3].

There are several desirable properties to judge a sorting algorithm and the suitability for a particular problem [1]. Stability preserves the order of already sorted input [1]. Considered to have the same value and appear in the same sequence order in the sorted output [4]. Run-time efficiency ensures the algorithm will run in an acceptable amount of time, indicating how efficient the algorithm will be in practice (best/average/worst-case time complexity) [1]. The orders of growth of the number of operations required for an algorithm to complete will be proportional to one of the functions listed (Eqs. 4) [1]. In-place sorting uses memory efficiently while sorting if memory requirements are a concern [1]. Suitability takes all these considerations in aggregate to consider any specific sorting algorithm [1].

It's essential to know various sorting algorithms' different strengths and weaknesses and the input instances when a particular sorting algorithm is implemented [1].

2. Sorting Algorithms

The report introduces five sorting algorithms consisting of three simple comparison-based (2.1 Bubble Sort, 2.2 Selection Sort, 2.3 Insertion Sort), an efficient comparison-based (2.4 Merge Sort) and a non-comparison-based (2.5 Counting Sort). Space and time complexity and bespoke diagrams with different input instances included.

2.1. Bubble Sort

The stable, simple comparison-based sorting algorithm, Bubble Sort, was formally analysed as early as 1956, so-called given that the larger E in a C bubble up to the end as sorting occurs [4].

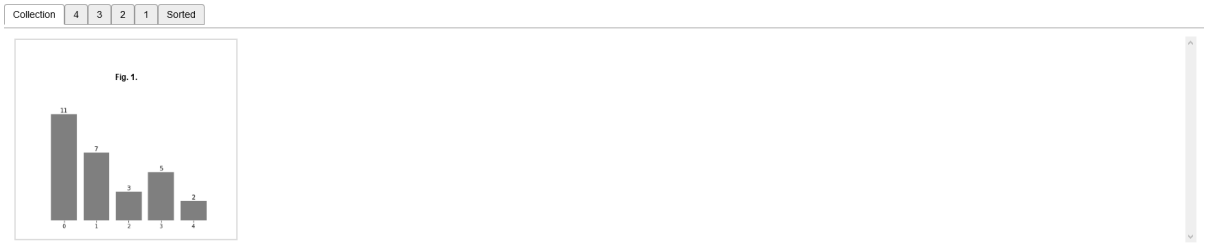
Constant space complexity of 1 [4]. Sorting is in place, meaning it uses only n plus a constant amount of additional working space in addition to the memory required for the input instance. This extra working space requires a variable(s) to make the swaps between E in different places. If an E is identified that's out of order, then an additional variable(s) are used as working space by way of a temporary variable to achieve the swap [4].

Performance achieved is linear n time complexity in the best case and quadratic (polynomial) n^2 time complexity in the worst and average cases [4].

It uses an iterative implementation and is easy to implement and understand but is slow and impractical for most problems. The only practical use case is on data that is nearly sorted or very small n [4].

2.1.1. Walkthrough

Sorting a C (Fig. 1) of the first five prime numbers $(\ 11,\ 7,\ 3,\ 5,\ 2\ )$ with out of order E or inversions (E state shown in gray) [4].

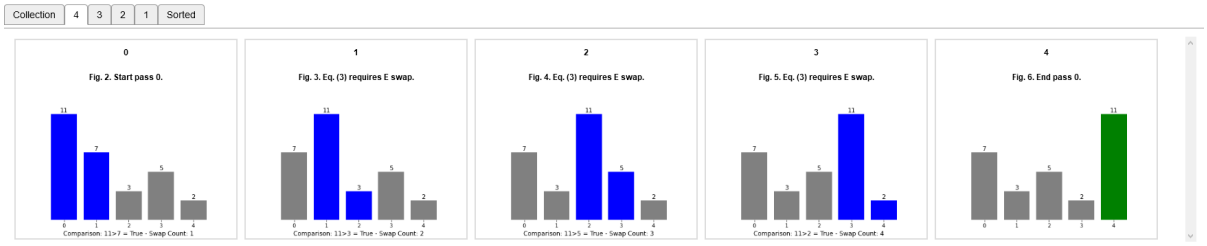


Traversing the C from right to left, each outer loop iteration (loop counter shown as a separate tab for each pass) swaps adjacent E as needed resulting in the largest E being in its final position (swap operation shown in blue) [4].

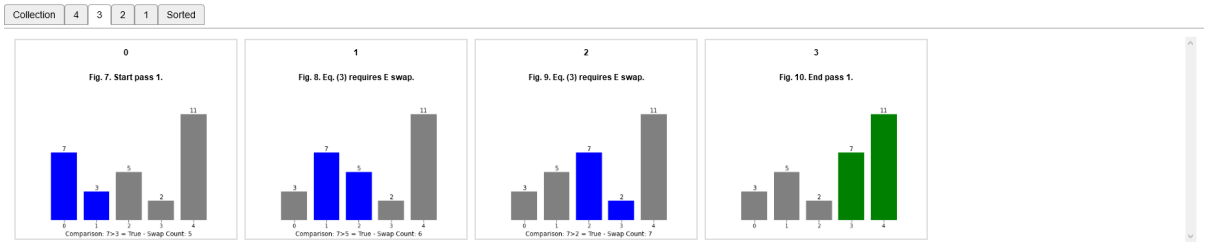
The outer loop iteration continuously counts down, so the resulting sorted partition populates from the right-hand side (hereafter abbreviated RHS) [4]. The E within the inner loop iteration (loop counter shown within the plot box for each pass afloat) must always be one less than the outer loop iteration [4]. Ensures the swap operation does not go out of bounds by comparing E outside the actual length of the C. Assumes that everything to the left is unsorted and as it goes through each iteration, the E which is on the right become sorted (Fig. 16) [4].

Starting at the leftmost index/position the inner loop $E\ i$ is compared with its neighbour to its right $E\ j$ exclusive of the last E [1]. This comparison checks for inversion [4][6] subject to Eq. (3) [1].

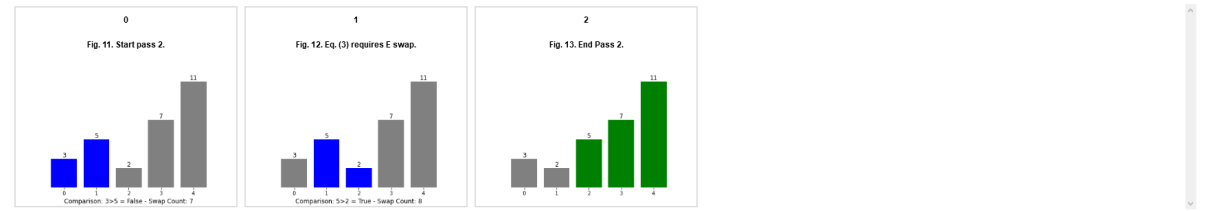
Fig. (2) shows $E\ i$ is greater than $E\ j$ [4]. To satisfy Eq. (3) requires $E\ i$ moving one position to the right into $[\ j\ ]$ and $E\ j$ moving one position to the left into $[\ i\ ]$ (Fig. 3) [4]. The process continues for adjacent E (Fig. 4 to Fig. 5) [4]. When the first outer loop iteration is complete (Fig. 6) the last E is now in its correct and final position (sorted partition shown in green) [4].



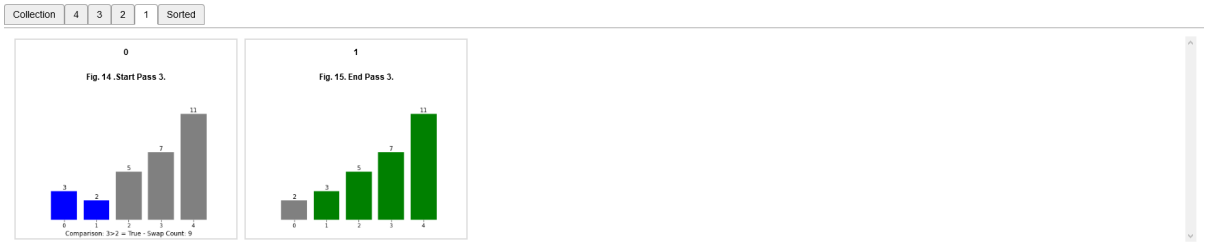
The outer loop iteration restarts (Fig. 7), repeating the comparison/swap procedure following Eq. (3) exclusive of the last two E [6]. When the second outer loop iteration is complete (Fig. 10), the last two E are now in the correct and final position [6]. The sorted partition now has two E (Fig. 10) [6].



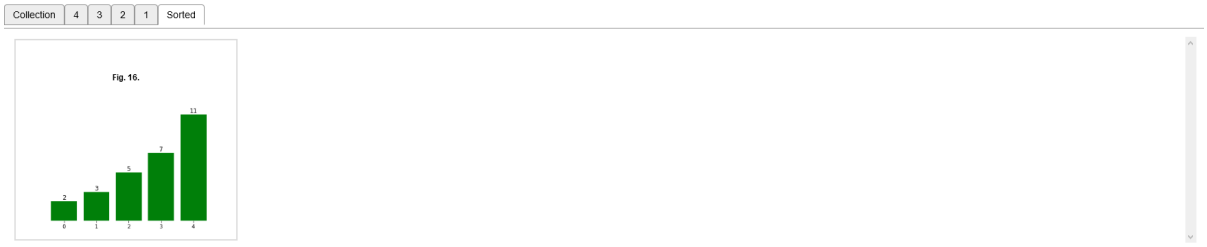
The outer loop iteration restarts (Fig. 11), and in this instance, the leftmost E already satisfies Eq. (3) therefore requiring no action [6]. Fig. 12 shows the comparison/swap procedure following Eq. (3) exclusive of the last three E [6]. When the third outer loop iteration is complete (Fig. 13), the last three E are correct and final [6]. The sorted partition now has three E (Fig. 13) [6].



The outer loop iteration restarts (Fig. 14), repeating the comparison/swap procedure following Eq. (3) exclusive of the last four E [4]. The Bubble Sort algorithm is complete when there is no unsorted E remaining on the left (Fig. 15) [4]. The fourth outer loop iteration is now complete (Fig. 15). Assume everything to the left is unsorted, and as it goes through each iteration, the E which is on the right become sorted [4].



The C becomes partitioned (Fig. 16) into two different parts whereby all the larger values have bubbled up to the top (sorted up to the RHS) [4]. The size of the sorted partition grows down to the left of the C as the algorithm does its work [4].



2.2. Selection Sort

The simple comparison-based sorting algorithm, Selection Sort, is unstable in contrast to Bubble Sort [4]. It does not guarantee E, which have the same value according to a comparator function, will appear in the exact ordering in the sorted output as they do in the unsorted input, therefore considered unstable [4].

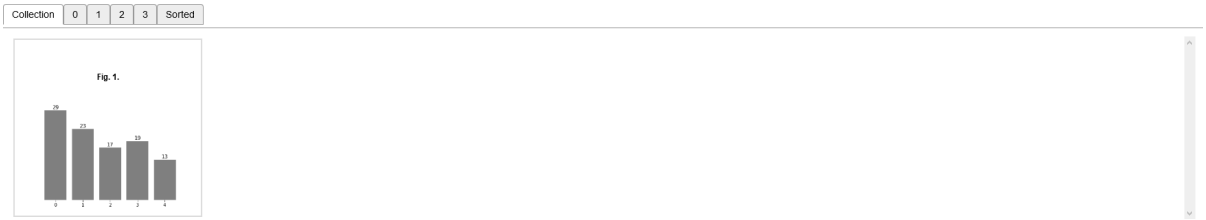
Constant space complexity of 1 [4]. Similar to Bubble Sort, it achieves in-place sorting [4].

Performance achieved is quadratic (polynomial) n^2 time complexity in the best/worst /average cases. It is not as good as Bubble Sort in the best case, although it is comparable with worst and average cases [4].

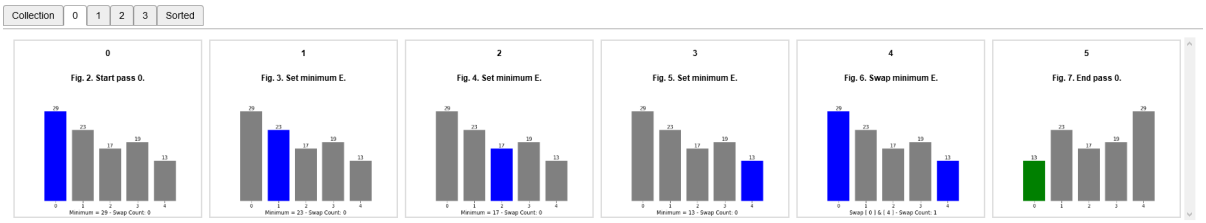
It uses an iterative implementation. On average, it can give better performance but is still impractical for any real-world task with a significant n given its n^2 performance in the worst case and the average case [4].

2.2.1. Walkthrough

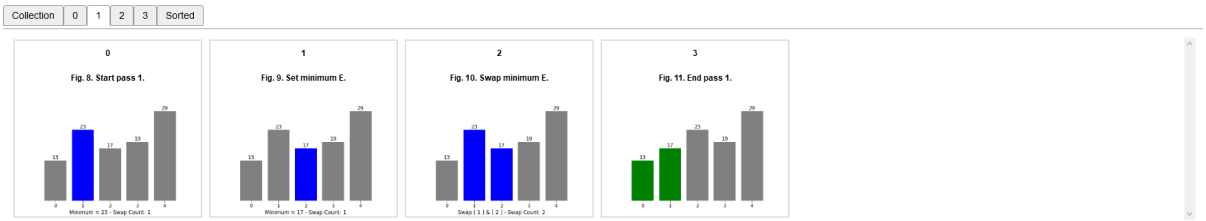
Sorting a C (Fig. 1) of the next five prime numbers ($29,\ 23,\ 17,\ 19,\ 13$) with inversions [4].

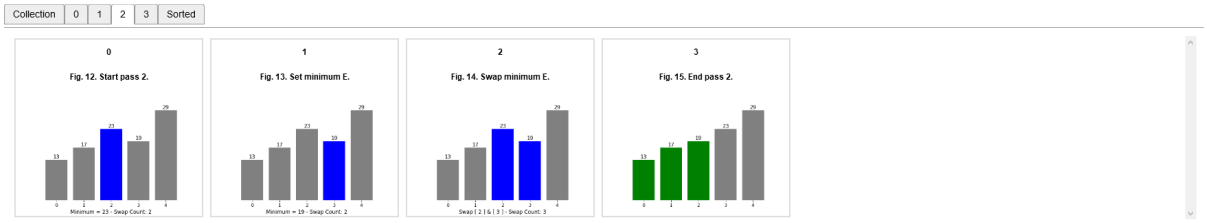


Traversing every E from $[\ 0\ ]$ through to $[\ n-1\ ]$, selecting the smallest value E found (Fig. 2 to Fig. 5) [4]. Swap the smallest valued E found with the E at $[\ 0\ ]$ (Fig. 6) in accordance with Eq. (3) [4]. The size of the sorted partition on the left-hand side (hereafter abbreviated LHS) has grown by one E (Fig. 7) [4].

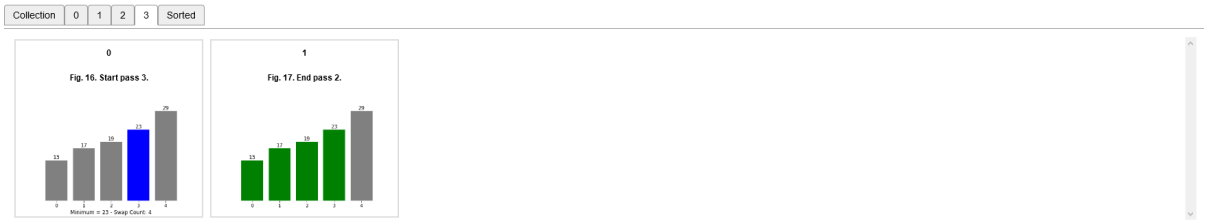


Continue the process, moving to the next index/position until there is nothing left to search [4]. With each outer loop iteration, the size of the sorted partition on the LHS continues to grow by one E each time (Fig. 11 and Fig. 15 and Fig. 17) [4].

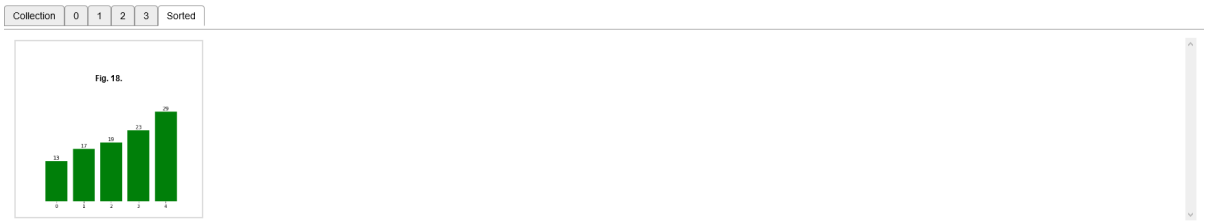




The swapping of an E with itself (Fig. 16), although harmless, is not worth considering [4].



Traversing the C (Fig. 18) using ascending order, each iteration of the minimum E from the unsorted partition on the right is picked and moved to a sorted partition on the left [4].



2.3. Insertion Sort

The stable, simple comparison-based sorting algorithm, Insertion Sort, uses a similar method usually used by card players when sorting cards in their hands. It is intuitive and used to do many day-to-day sorting tasks [4].

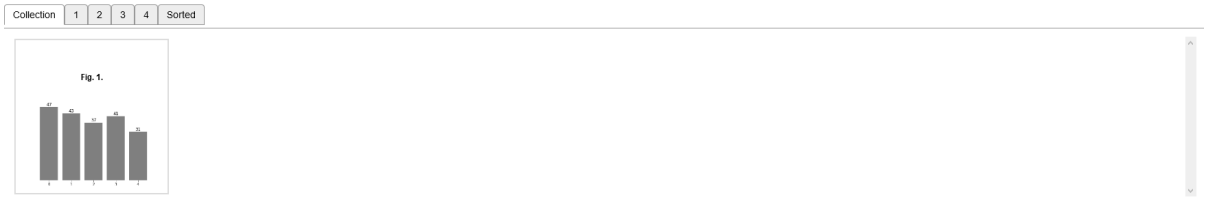
Constant space complexity of 1 [4]. Similar to Bubble Sort and Selection Sort, it achieves in-place sorting [4].

Performance achieved is $n + d$ time complexity, where d is the number of inversions in the input instance [4]. It is highly inefficient on large random C [4].

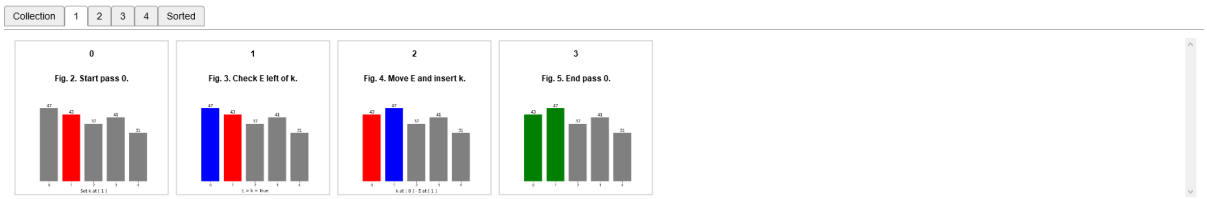
It uses an iterative implementation and is perfect for small or nearly sorted C [4] or mostly sorted (degree of inversions) [1]. Ideal if a compact and small algorithm that will require very little code is required. [1]

2.3.1. Walkthrough

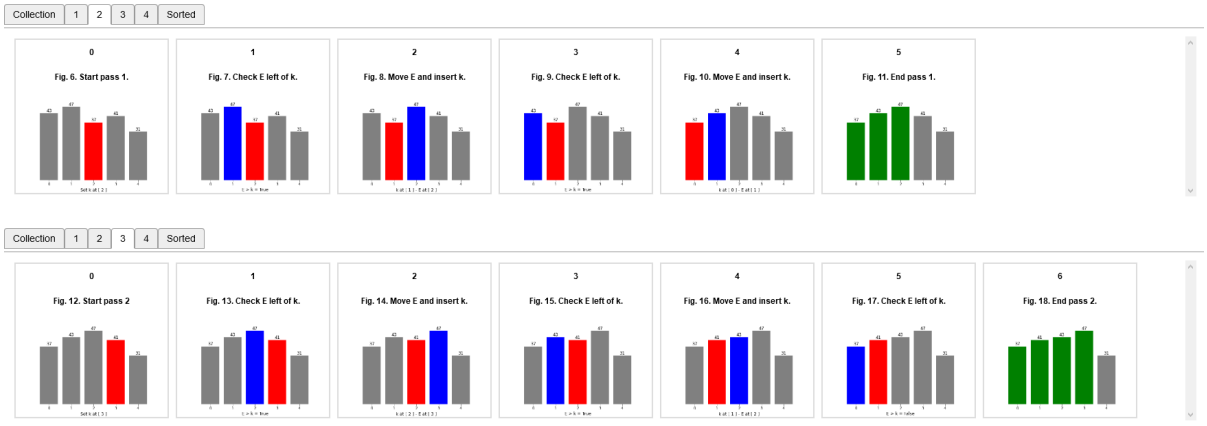
Sorting a C (Fig. 1) of the next five prime numbers ($47,\ 43,\ 37,\ 41,\ 31$) with inversions (Fig. 1) [4].



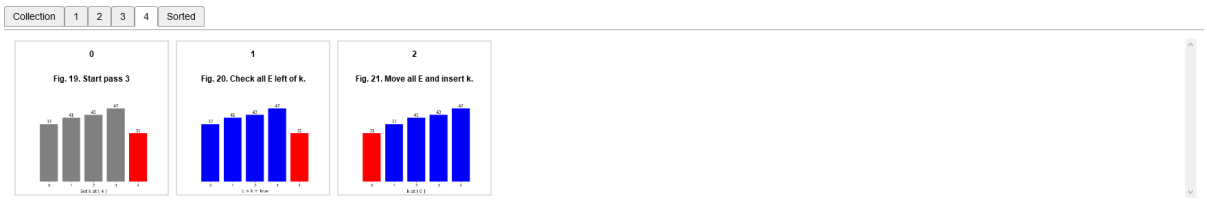
Starting at $[\ 0\ ]$, set the key (hereafter abbreviated k) as the E at $[\ 1\ ]$ (k state shown in red) (Fig. 2) [4]. Moving any E (shown in blue) to the left of the k which are greater to the right by one index/position and then inserting the k (Fig. 4) being the sorted partition (shown in green) [4].



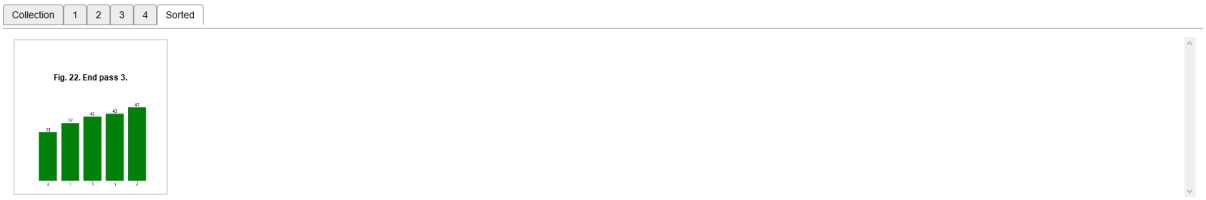
Continue the process, moving to the next index/position until the last index/position $[\ n-1\ ]$ is reached [4].



E to the left if greater than k move to the right by one index/position and insert the k on the left being the sorted partition [4].



At each step of the process, the sorted partition increases in size on the left (termed the head), and the unsorted partition decreases in size on the right (termed the tail) (Fig. 22) [4].



2.4. Merge Sort

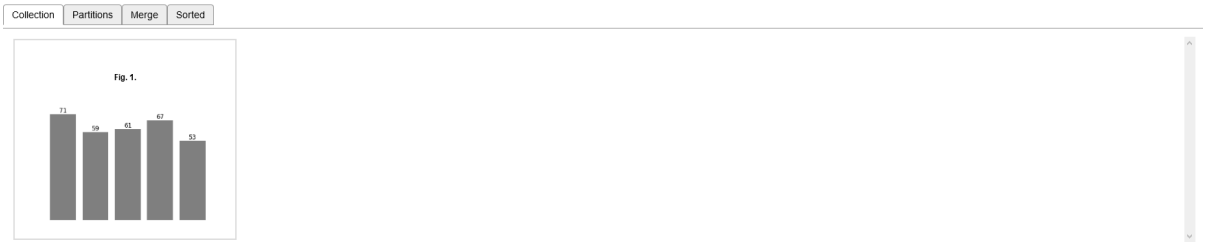
The guaranteed stable, in-place efficient comparison-based sorting algorithm, Merge Sort, was developed by the renowned computer scientist John Von Neumann in 1945 [1]. Over 70 years old, it is still in use today, whether in its own right or as part of a hybrid algorithm (in combination with some other sorting algorithms) [1].

The three simple comparison-based sorting algorithms (Bubble Sort/Selection Sort/Insertion Sort) use an iterative process (for or while loops) [3]. Merge Sort is a recursive algorithm that adheres to the divide-and-conquer paradigm [3]. The divide step indicates the problem broken down into smaller sub-instances of the same problem type. The conquer step means solving these broken sub-instances until some base case is reached with the solution propagated back up the recursion trace [4].

The performance achieved is $O(n\ log\ n)$ time complexity in the worst case. It is the best asymptotic behaviour compared with the three simple comparison-based sorting algorithms (Bubble Sort/Selection Sort/Insertion Sort). Can achieve $O(n\ log\ n)$ time complexity in the best case and average case, so its an excellent choice if a predictable runtime is required and not concerned about getting the absolute best performance every instance [3].

2.4.1. Walkthrough

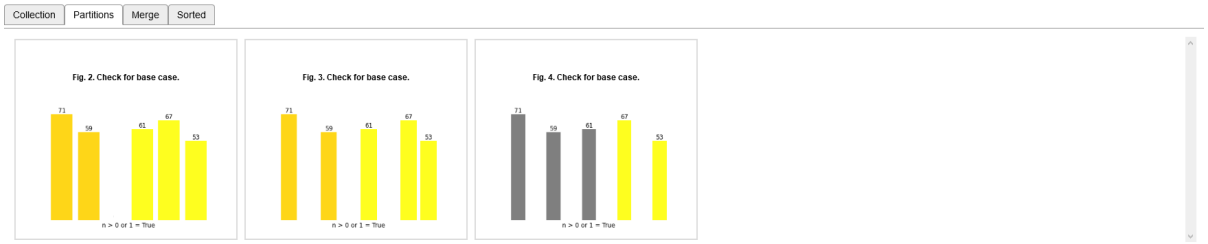
Sorting a C (Fig. 1) of the next five prime numbers ($71,\ 59,\ 61,\ 67,\ 53$) with inversions (Fig. 1) [4].



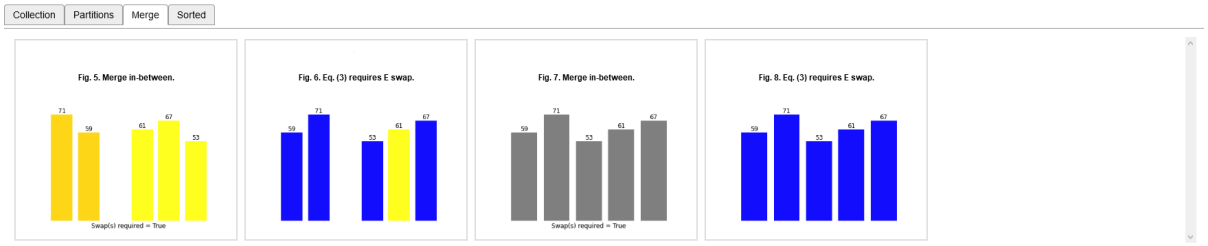
Halved the problem each time to get a case where $n = 0$ or $n = 1$, this being the base case of the recursion [3]. A partition of $n = 0$ or $n = 1$ can act as a base case because, by definition, no need to be sorted (separated sections shown in gold and yellow) (Fig. 2) [3].

Additional calls are made and if either $n = 0$ or $n = 1$ (left/right partition) then the recursion is applied again otherwise the C is split into additional partitions (Fig. 3) [3].

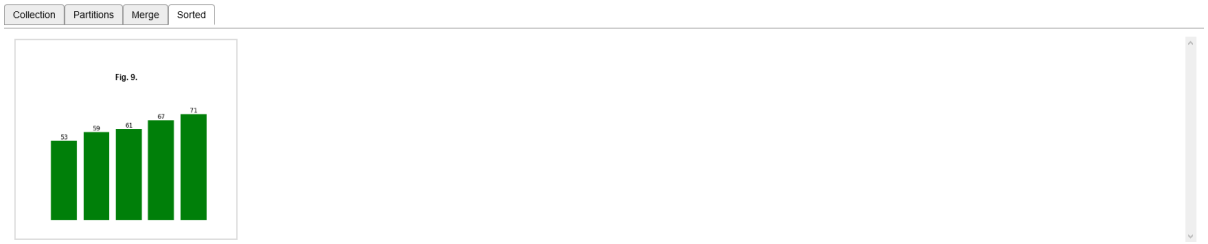
Additional calls are made to get to the base case of the remaining partitions; therefore, the partition for the moment is put on hold as it will be combined later (shown in gray) (Fig. 4) [3].



When all the single E have reached the base cases, merge in-between (Fig. 5) [3]. Sort according to Eq. (3), the E relative to one another (Fig. 6) [3]. Repeating the merge step (Fig. 7) and sorting according to Eq. (3) (Fig. 8) [3].



Finally, merge the two sorted halves into one sorted C (Fig. 9) [3].



2.5. Counting Sort

The non-comparison based sorting algorithm, Counting Sort, was proposed by Harold H. Seaward in 1954 [3]. Implemented correctly, it is a stable sorting algorithm [3]. Non-comparison-based sorting algorithms require a set of assumptions on the type of input instance [3]. To use Counting Sort, one needs to assume an n where each E has a non-negative integer key and has a range of k being the number of discrete integers possible (hereafter possible key values are abbreviated k) [3].

Space complexity is $n + k$ consisting of k (counting) and n (results) [3]. The n stores the results of the sorted output hence achieving $n + k$ space complexity [6].

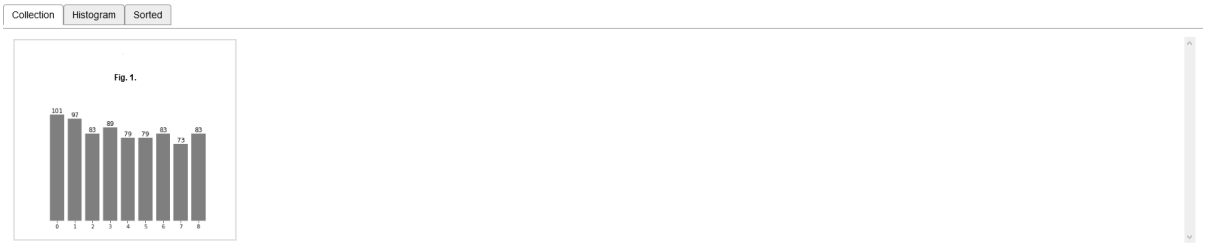
Performance, appearing impossible, is very close to linear n time complexity - achieving $n + k$ time complexity in the best/worst/average cases [3]. A significant improvement upon the efficient comparison-based algorithm, Merge Sort [3].

Probably the most straightforward non-comparison-based sorting algorithm [3]. However, the potential running time advantage does come at the cost not be as widely applicable as the comparison-based [3].

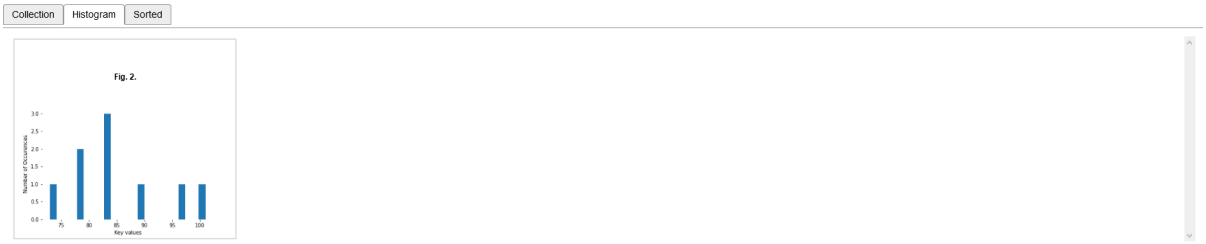
The type of input instance is assumed ahead of time [3]. There are workarounds such as assigning labels to integer values and using those proxy sort keys instead of the integer value [3]. The mapping between non-negative integer keys and strings would allow Counting Sort to apply to other types of problems, but this mapping must be determined beforehand [3]. The algorithm is a good choice if there is a lot of duplicate keys [3].

2.5.1. Walkthrough

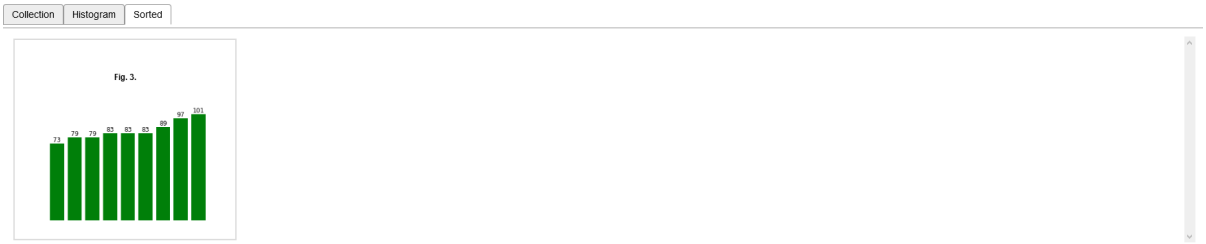
Sorting a C (Fig. 1) of the next five prime numbers $(\ 101,\ 97,\ 83,\ 89,\ 79,\ 79,\ 83,\ 73,\ 83)$ with inversions along with duplicate E (Fig. 1) [4].



Internally a C of k is initialised as discrete values [3]. Counting the number of times each k appears in the C given the possibility of duplicate E [3]. Traversing the C a record of the number of times each distinct k occurs in the input instance is captured similar to a histogram of frequencies (Fig. 2) [3]. The histogram x-axis (Fig. 2) represents the count of all k stored in the C [3].



Another C of n is used to build the sorted output based on the histogram of k frequencies stored in the C [3]. The ordering of k in the input instance allows the placement of the largest E at the end of the resulting C [3]. Continuing to work backwards looking for the subsequent largest k instance, which occurs counting down from $[\ n-1\ ]$ [3]. The placement of E builds the resulting C (Fig. 3) [3]. The ordering of the k in the original input instance ensures that stability is preserved [3]. If two different E have the same k, then knowing that if one appears on the right of the other, then that ordering would be preserved [3].



3. Implementation & Benchmarking

The report describes the process followed when implementing the application and presents the benchmarking result, using a plot (Fig. 1) and table (Fig. 2) created from within the application.

3.1. Implementation

The application requires the installation of Python, loaded on your local machine. This can be facilitated by downloading and installing Anaconda - anaconda.com/download. An additional package, although not required, dataframe_image, is used to output the console display as an image - pypi.org/project/dataframe-image.

Maximum console output headings are set to thirteen, starting at the low $n = 78$, increasing 50% each time. The code is compartmentalised, with the core functionality within *main()*. Using the standard library function *randint*, as per the Project Specification, to generate random integers in the interval [0, 100). Function *fRandomCollection* creates C of random integer of n . Function *tTotalRuns()* initialises the number of runs, as per the Project Specification is ten. Function *fAverageTime()* calculates the average based on the number of runs.

The function *main()* uses the build-in standard library function, *time()*, to ade with benchmarking. Declare C to hold time recording of each sorting algorithm and an additional C to keep the average of the times recorded. Declared a dictionary of key:value pairs, being the sorting algorithm function name and a string version of the same name. Traversing each n, traversing the sorting algorithm function key values, traversing the ten runs used to calculate average, then make a copy of each sorting algorithm function. Each function copy will call the sorting algorithm passing the randomly generated integers of n—finally, wrapper around the build-in standard library function *time()* and the obtain the averages.

3.2. Benchmarking

The three simple comparison-based sorting algorithms (Bubble Sort, Selection Sort and Insertion Sort) have inferior performance in the worst case given the use of nested iterations through the C [4].

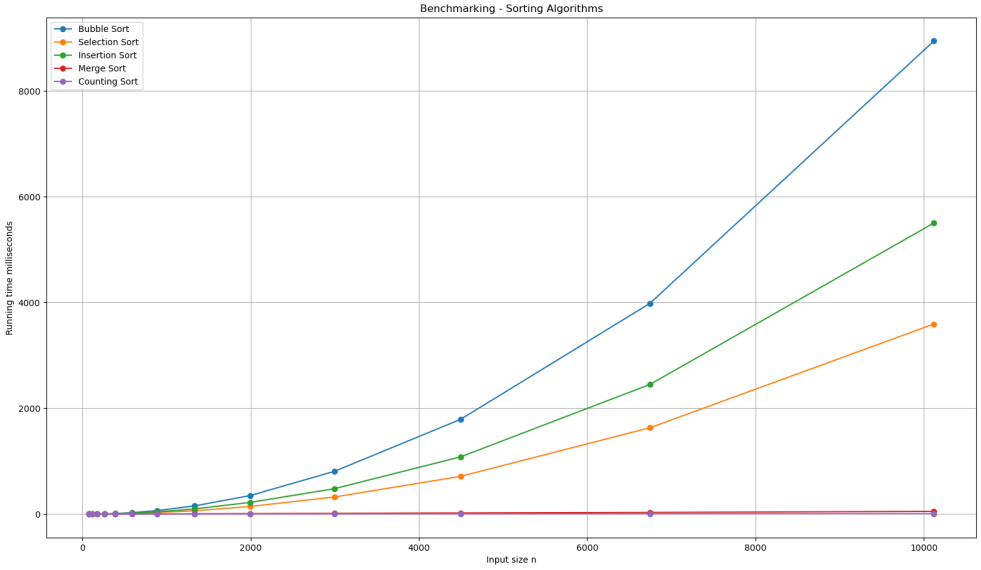
The iterative approach uses double for loops giving n^2 worst-case in most scenarios [4]. On average, the inner loop will execute about $\frac{n}{2}$ for each iteration of the outer loop [4]. The worst case would be an outer loop that has to execute n times and an inner loop running $\frac{n}{2}$ times for each iteration of the outer loop [4]. In addition, within the inner loop, the comparison and swap operations also take some constant time k [4].

$$\frac{n^2}{2} + k \approx O(n^2)$$

From a complexity analysis perspective, as n becomes more extensive, then n^2 is the dominant term [4]. The plot (Fig. 1) shows Bubble Sort graphically is the least optimal as n increases, followed by Selection Sort and Insertion Sort. The lower average n of each simple comparison-based sorting algorithm is trivial, but a marked increase (Fig. 2) is evident from $n = 263$, double and tripling etc., after that.

The performance of the efficient comparison-based sorting algorithm, Merge Sort, achieving $O(n\ log\ n)$ time complexity in the best case is the upper bound on the possible performance for a comparison-based sorting algorithm [4]. Evident (Fig. 1 and Fig 2.) Merge Sort is the top performer compared with the simple comparison-based algorithms. On average, we can expect it to be a lot faster than any of simple comparison-based sorting algorithms for any decent n [3]. Exercise caution when using a recursive approach; the algorithm can quickly become expensive given the number of active stack frames applied [3].

Counting Sort, the non-comparison-based sorting algorithm achieves better time complexities in all cases (best/worst/average) compared with the comparison-based sorting algorithms. Far exceeding the upper bound $O(n\ log\ n)$ [3]. Counting Sort is the best overall performer, followed closely by Merge Sort. The results (Fig. 1 and Fig 2.) align with expectations for all five algorithms analysed.



(Fig. 1.)

Size	78	117	175	263	394	592	888	1332	1999	2998	4497	6746	10120
Bubble Sort	0.000	1.562	3.124	4.687	10.935	28.119	65.610	154.651	351.480	809.182	1791.817	3983.440	8941.658
Selection Sort	1.560	1.562	1.562	3.124	6.249	14.059	29.680	62.485	145.278	323.676	713.895	1632.434	3592.913
Insertion Sort	0.000	0.000	1.562	3.124	6.248	17.183	40.615	98.414	221.986	481.141	1082.560	2450.986	5501.952
Merge Sort	0.000	0.000	0.000	0.000	1.562	1.562	3.124	6.248	9.373	14.059	21.870	32.805	51.550
Counting Sort	0.000	0.000	0.000	1.562	1.562	1.562	0.000	0.000	1.562	3.124	3.124	6.249	9.373

(Fig. 2.)

References

[1] P. Mannion, "07 Sorting Algorithms Part 1," GMIT, November 2017.

[2] P. Mannion, "03 Analysing Algorithms Part 1," GMIT, October 2017.

[3] P. Mannion, "09 Sorting Algorithms Part 3," GMIT, November 2017.

[4] P. Mannion, "08 Sorting Algorithms Part 2," GMIT, November 2017.

[5] K. Żak, "IPyPlot," GitHub, February 2021.

[6] w3, "Python Data Structures and Algorithms: Bubble sort," w3resource, January 2021.