Sorting Algorithms

Isabella Doyle

G00398800

Computational Thinking with Algorithms

Lecturer: Dominic Carr

Submission: Aug 2021

Table of Contents

1. Introduction

A significant number of tasks performed on computers require sorting in advance. In fact, it is estimated that over 25 percent of CPU cycles are spent sorting.[1] For this reason, the creation of efficient sorting algorithms is paramount. The earliest sorting algorithm, the Radix sort, dates back to 1887 and today efforts are ongoing to improve algorithms of this type.[2] There are many examples in everyday life where sorting algorithms play an important role, for example, bank transactions, computer graphics and web search results, which are sorted by relevance to the specified query.

Sorting allows us to organise information in such a way that the output is meaningfully ordered. The definition of sorting is the arrangement of items within a collection in accordance with a pre-defined set of rules.[3] Thus, the output of any sorting algorithm must conform to two conditions. First, each item must be less than or equal its successor in a collection i.e. $A[i] < A[j]$, then $i < j$. Second, the output must be a permutation of the original input. That is, the initial items in the collection must remain the same in the reordered output.[4]

Some data types are naturally comparable such as numbers and characters (lexicographical ordering). However, many items require customised rules that determine how they are sorted such is the case with the Dutch National Flag problem where sorting relies on comparator functions and assigning values to colours (red < white < blue).[5] It follows that, sorting relies on a function that compares two elements (a, b) and returns the output; -1 (a is not less than b) if $a < b$, 0 (a is equal to b) if $a = b$, 1 (a is greater than b) if $a > b$.

An important consideration when choosing an algorithm is the input array. One needs to assess the size of the array and also to what degree the array is sorted. The inversion count in an array indicates how far away the array is from being sorted. If

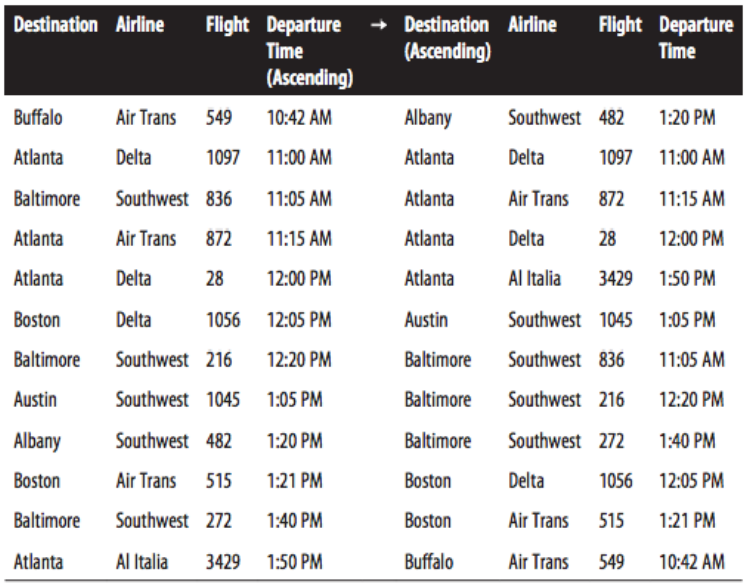
[1] Mu, Qi, and Song (2015) pp. 1
[2] Knuth (1997) pp. 105
[3] Collins English Dictionary (Online)
[4] Mannion (2018)
[5] Dijkstra (1976) pp. 111

the array is already sorted then there are 0 inversions. However, if the array were to be sorted in the reverse there would be a maximum number of inversions in that collection. In that case, two items constitute an inversion if a[i] > a[j] and i < j.[6] For example, the array [6, 1, 4] contains 2 inversions (6, 1) and (6, 4). The inversion count of an input will have a significant impact on the run-time of a sorting algorithm.

While the input is an important consideration when choosing a sorting algorithm, other equally important and desirable properties for sorting algorithms that are fundamental to its efficiency include stability, good run-time efficiency, in-place sorting and suitability.

Stability

When two elements in an array are equal in the original unordered collection, lets call them *a(i) and a(j),* the preservation of that order may be necessary in terms of their relation to one another.[7] In such case, if *i* < *j*, the ending location for *a(i)* must come before the ending location for *a(j).* An algorithm that preserves this order is known as stable. One real world application of a stable sorting algorithm is the organisation of flight information. The first ordering arranges flights by departure time, for instance. However, a subsequent ordering may require ordering by destination, see figure.[8] Note how the departure times on the right preserve the relative ordering of the original collection.

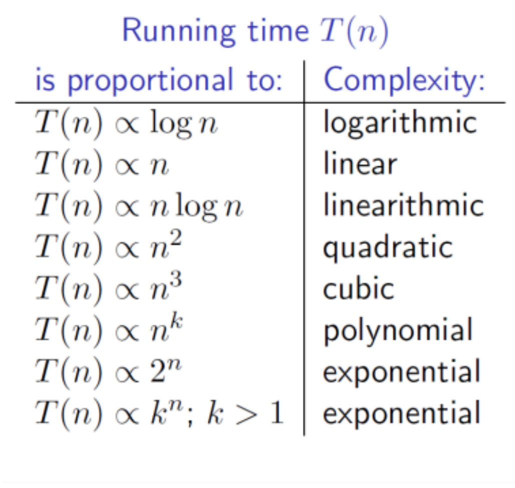


Destination	Airline	Flight	Departure Time (Ascending)	→	Destination (Ascending)	Airline	Flight	Departure Time
Buffalo	Air Trans	549	10:42 AM		Albany	Southwest	482	1:20 PM
Atlanta	Delta	1097	11:00 AM		Atlanta	Delta	1097	11:00 AM
Baltimore	Southwest	836	11:05 AM		Atlanta	Air Trans	872	11:15 AM
Atlanta	Air Trans	872	11:15 AM		Atlanta	Delta	28	12:00 PM
Atlanta	Delta	28	12:00 PM		Atlanta	Al Italia	3429	1:50 PM
Boston	Delta	1056	12:05 PM		Austin	Southwest	1045	1:05 PM
Baltimore	Southwest	216	12:20 PM		Baltimore	Southwest	836	11:05 AM
Austin	Southwest	1045	1:05 PM		Baltimore	Southwest	216	12:20 PM
Albany	Southwest	482	1:20 PM		Baltimore	Southwest	272	1:40 PM
Boston	Air Trans	515	1:21 PM		Boston	Delta	1056	12:05 PM
Baltimore	Southwest	272	1:40 PM		Boston	Air Trans	515	1:21 PM
Atlanta	Al Italia	3429	1:50 PM		Buffalo	Air Trans	549	10:42 AM

[6] Count Inversions - Geeks for Geeks (2020)
[7] Heinemen (2016) pp. 55
[8] *ibid*. pp. 56

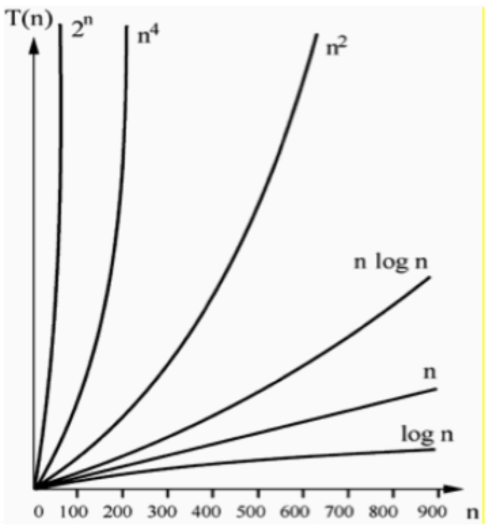
Complexity of a Sorting Algorithm

When analysing a sorting algorithm it must be discussed in terms of best, average and worst-case time complexity. The average-case is difficult to accurately determine and demands advanced mathematics and approximation and will not be a focus of this report. Best-case refers to that which requires the least operations to complete the task i.e., when the input array is already sorted (there are no inversions). However, a sorting algorithm should not be chosen solely based on its best-case. This is because factors such as input size largely affect its runtime. Perhaps the most important consideration in terms of efficiency is the worst-case, which refers to the maximum number of operations needed to complete a sort. Worst-case is useful in determining a task's upper-threshold completion time. Like time complexity, space complexity is commonly demonstrated asymptotically in Big O notation. This notation represents the growth rate of a function as it tends towards infinity, describing the upper bound of growth (worst-case) such as *O(n), O(n log n)* etc. See image below.



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
$T(n) \propto n^3$	cubic
$T(n) \propto n^k$	polynomial
$T(n) \propto 2^n$	exponential
$T(n) \propto k^n;\ k > 1$	exponential



Thus far, we have outlined that the size of array and the extent to which the items in an array are pre-sorted influence run-time. Another important factor to be considered is the varying memory requirements.

In-place and out-of-place Sorting

In-place sorting algorithms are those that do not require external memory in order to sort the input array. Hence, the name *in-place* – the data is manipulated, which may require some working memory to complete the task and the input is overwritten by the output.[9] In-place sorting is advantageous when memory is restricted. Out-of-place sorting requires additional space and therefore, more operations are needed to complete a task.

Suitability

The input type must be assessed in relation to an algorithm's properties in order to determine their suitability. As such, each task must be assessed individually prior to choosing an algorithm.

Comparison and Non-Comparison Sorting Algorithms

Comparison sorts use comparison operators alone to decide which of two items should come first in a list. For a sorting algorithm to be considered comparison-based, it must only gain information about the ordering of an array by comparing two items at a time with the order $\leq$ (less than or equal to).[10] Fundamental to algorithm analysis is the fact that any algorithm which sorts by comparing items cannot performs better than $n \log n$ in the average or worst-cases.[11] This limitation does not apply to non-comparison sorts. Non-comparison sorts are concerned with the actual data contained in the array, which is unlike comparison sorts that only compare items against each other. $O(n)$ runtime is possible with non-comparison sorts as we will see when discussing bucket sort.

To further understand how sorting algorithms can vary in efficiency in relation to various factors including those mentioned above, we will outline and explore the following five sorting algorithms:

1. Insertion Sort (A simple comparison-based algorithm)
2. Quick Sort (An efficient comparison-based sort)
3. Bucket Sort (A non-comparison-based sort)
4. Bubble sort (A simple comparison-based sort)
5. Merge Sort (An efficient comparison-based sort)

[9] In-place Sorting – Baeldung (2020)
[10] Mannion (2018)
[11] *ibid.*

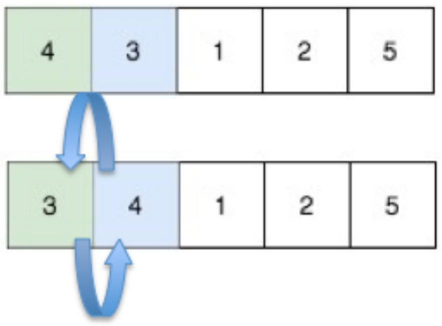
2. Sorting Algorithms

1. Insertion Sort

Insertion sort is a simple comparison-based algorithm that sorts an array one item at a time. While very efficient with smaller arrays, insertion sort is much less efficient with larger sets of data. This sorting technique uses iteration by selecting one item to sort per repetition, with the left-most item in the array considered the partition. The partition splits the array into an ordered array (to the left) and unordered array (to the right).

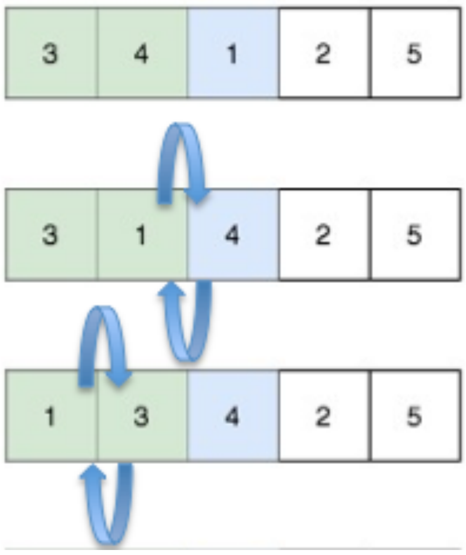
Iteration 1

The item at index 1 is compared to the item to its left (index 0). If value in index 1 is < item at index 0, they will be swapped.

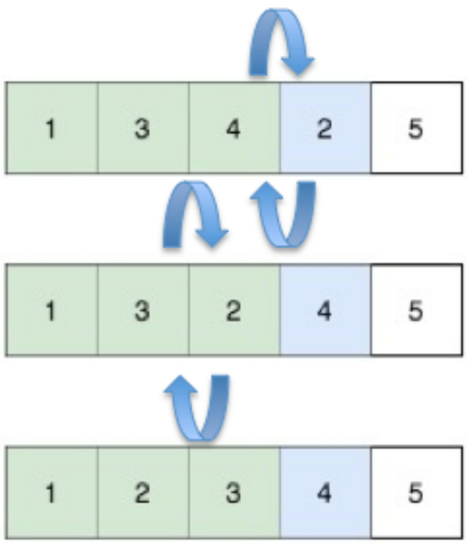


Iteration 2

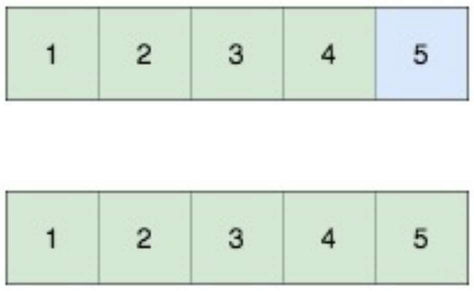
The next iteration selects the item at index 2. This item is compared to the item to its left (at index 1). If it is bigger, it is swapped and the item that was at index 2 in the array is now at index 1 and must be compared to the item to its left of the array (index 0). If it is bigger than the value at index 0, it will be swapped (if it is not bigger, the for-loop breaks and the next iteration is executed) and because there are no more elements to the left to compare to, the for-loop breaks and the next iteration executes. At this point, the items to the left of index 2 are part of the sorted array, while the items to the right are in the unsorted array.



Iteration 3



Iteration 4



NO changes were necessary on the final iteration as the final two values were in the correct positions.

Complexity of Insertion Sort

Insertion sort running time is *O(n)* in its best case, meaning that the program would iterate through the array (*n*) performing no swaps as the array is already sorted. The worst case runs in *O(n²),* which occurs when the input list is in decreasing order, for example, [5, 4, 3, 2, 1]. The amount of operations needed to sort the array in this case is: $2\times(1+2+\cdots+n-2+n-1)$.[12]

Is it stable and in-place?

Insertion sort is **stable** and typically **in-place**.

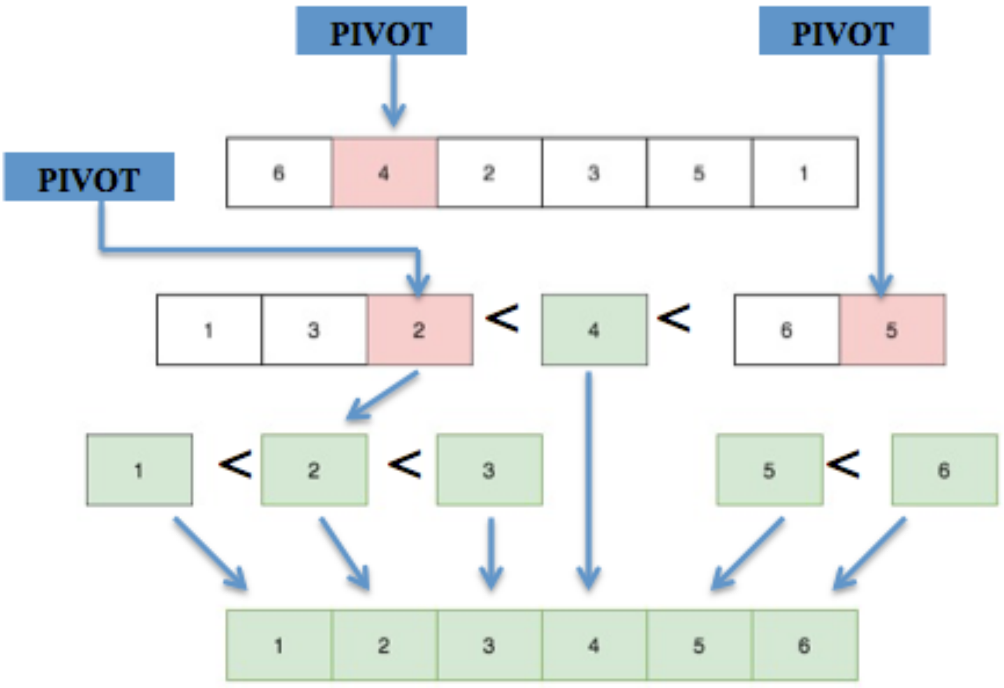
Why use this algorithm?

Insertion sort is very fast when sorting a small collection of elements or when the input array is almost sorted.

[12] Insertion Sort – Brilliant (2021)

2. Quick Sort

Quick sort, like insertion sort, is a comparison-based sorting algorithm. However, it is more complex and generally uses recursion. Although an iterative implementation is possible, it is generally not recommended due to being unintuitive and more complex. Quick sort is based on the divide and conquer strategy whereby an array is split into sub-arrays based on the location of the pivot.[13] The elements with values less than the pivot should be kept to its left and the bigger values to the pivot's right. The sub-arrays follow the same process until the arrays contain just one item each. The items are then combined to create a sorted array.



Method:

1. Randomly select a pivot & place items smaller than pivot in one array & items bigger in another. Items that are the same as the pivot remain in the same array as the pivot.

2. There are now three arrays. Step 1 is repeated on each sub-array containing 2 or more items until each array contains just one item.

3. Merge the arrays into one sorted list.

[13] The pivot may be chosen in various ways, however, random selection was chosen in the implementation in main.py.

Complexity of Quick Sort

Quick sort has a time complexity of *O(n log n)* in the best-case and $O(n^2)$ in the worst-case. The worst-case occurs when the pivot is the smallest or biggest element in the array while the best case is achieved when the pivot is the middle element in the array.

Is it stable and in-place?

Quick sort is **not** a **stable** sorting algorithm and is considered **in-place**.

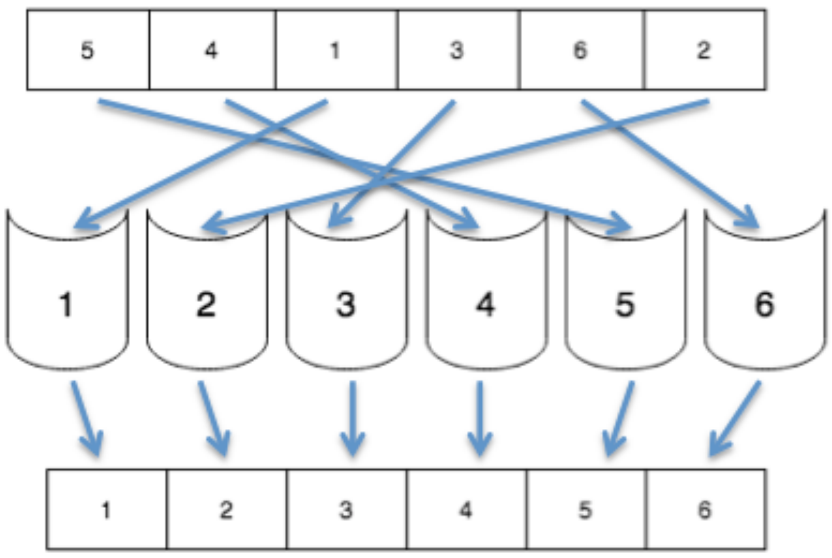
Why use this algorithm?

Quick sort, unlike merge sort, is in-place which means no auxiliary space is needed in its implementation. It is considered to be one of the most efficient sorting algorithms due to its performance in the average.

3. Bucket Sort

Bucket sort is generally a non-comparison based sorting algorithm, however, insertion sort is often used in its implementation. In such case, it could technically be considered a mix of comparison and non-comparison based methods. This algorithm creates buckets that correspond to the size of the input array and assigns each item to a bucket depending on its size. Bucket sort is known to exhibit the scatter/gather approach which is observed in the explanation below.

To illustrate the process of bucket sort without using comparison sort, a simplified example containing integers ranging from 1-6 are utilised in this case.



1. Empty buckets are created matching the number of items contained in the array. The size of each bucket is established by dividing the largest value in the array by the length of the array (6 % 6 = 1)
2. The items are placed into the buckets which corresponds to their size.
3. Finally, the buckets are merged into a sorted list.

Complexity of Bucket Sort

Generating the empty buckets, placing each item in their corresponding bucket and merging the buckets to create an ordered list occurs in *0(n)* time. Sorting the individual buckets (which was not necessary in the example above) varies depending on the method used. If using insertion sort, we know that the best-case is *o(n)* and worst-case is $O(n^2)$. Thus, bucket sort's time complexity would coincide with the aforementioned. In terms of worst-case space complexity, it measures at *O(nk),* faring significantly worse than merge sort as we will see later.[14]

Is it stable and in-place?

Its stability **depends** on the underlying algorithm used for sorting i.e., if insertion sort were used in the implementation, it would be stable. As bucket sort creates buckets for storing array items, this algorithm is considered **out-of-place**.

Why use this algorithm?

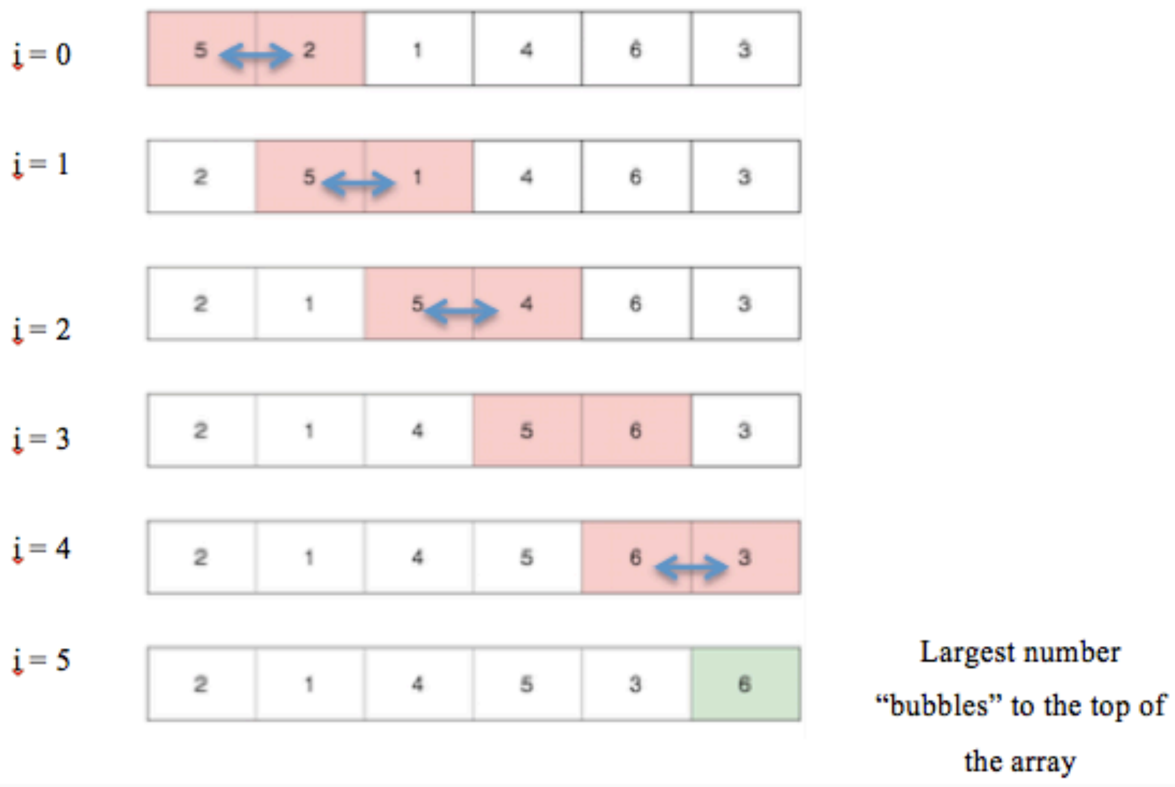
It is quicker than bubble sort, as we will see, because the individual buckets can be sorted individually. This reduces the amount of comparisons taking place.

[14] What is bucket sort – Ladinirenbeliz (2021)

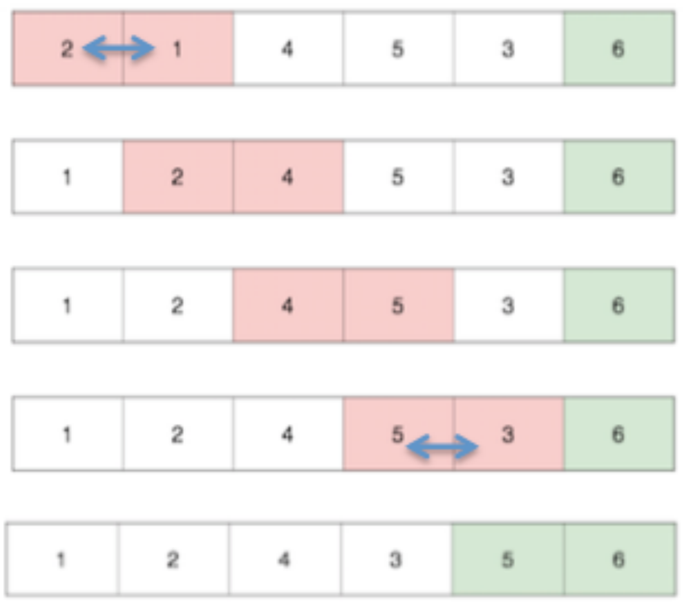
4. Bubble Sort

Bubble sort is considered to be the simplest sorting algorithm. It compares two adjacent items in an array [a, b] and swaps them if they are not in the intentional order, for example, if a > b. As we will see in the example, with each iteration the largest value “bubbles” to the top, forming an ordered array on the right side of the array.

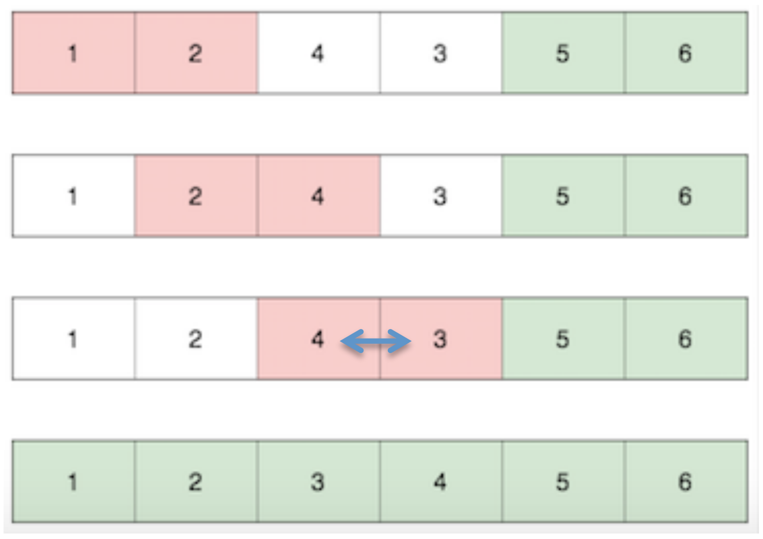
Iteration 1:



Iteration 2:



Iteration 3:



Complexity of Bubble Sort

Bubble sort has a best-case time of *O(n)*, occurring when the array is already sorted. The average and worst-case is $O(n^2)$. Average-case occurs when the array items are shuffled, while worst-case presents when the array is organised in descending order.

Is the algorithm stable and in-place?

Bubble sort is **stable** and **in-place**.

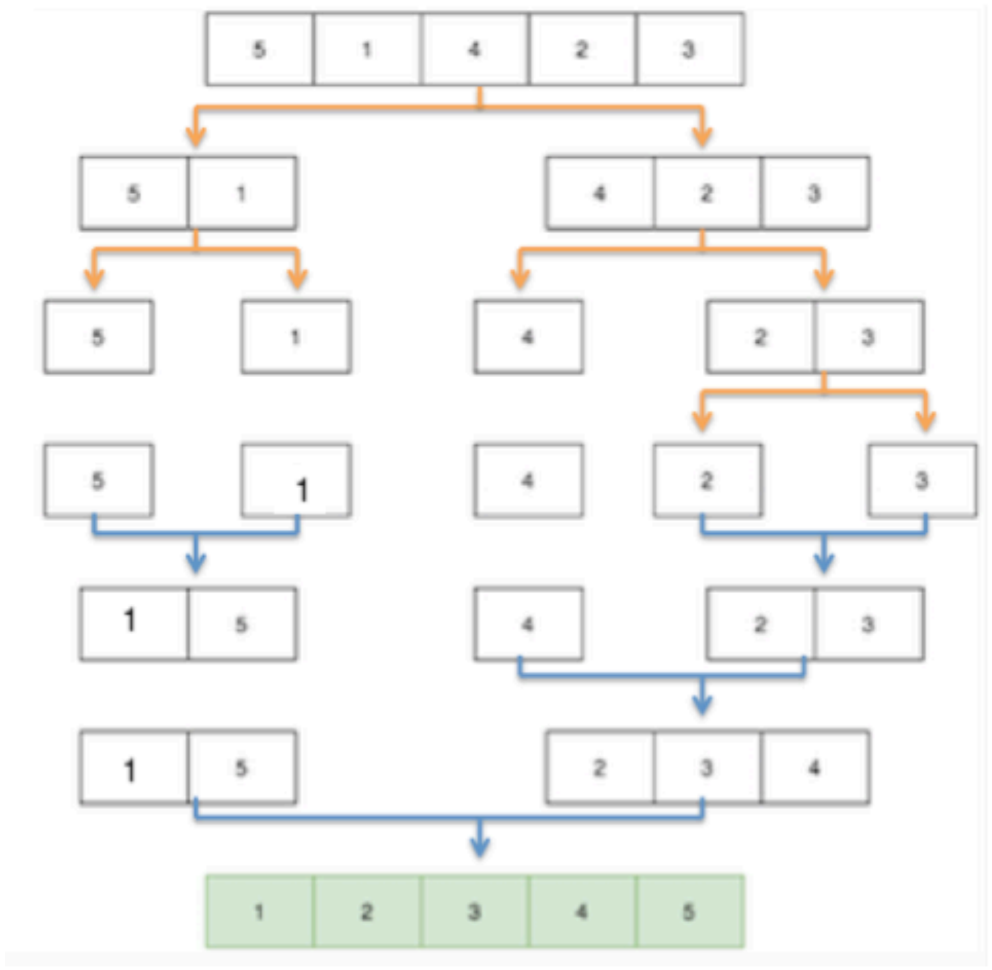
Why use this algorithm?

Bubble sort's application is advantageous when complexity is not an issue and concise, simple code is favoured.

5. Merge Sort

Merge sort, like quick sort, is also a divide and conquer algorithm but that uses recursion. There are two functions in the implementation of merge sort. The recursive function recursively halves the input array with each iteration while the second function merges the arrays to form a final sorted one.

The first call is to the function that splits the array. The midpoint is at index 2. Thus, the array is split in two. The first is [:2] and second is [2:] which is [5, 1] and [4, 2, 3]. The call is made recursively until there are two or fewer items in the array. It is the call to the merge function that merges sub-arrays in the correct order, until all arrays have been merged to a single sorted array. See example below.



The **<u>yellow</u>** arrows represent splitting the array in half.

The **<u>blue</u>** arrows signify the merging off arrays.

Complexity of Merge Sort

The merge function in bubble sort takes in two inputs with a combined length of *n* at most, *n* being the size of the original array. This function looks at each item once which means the runtime complexity for this function is *O(n).* With regards to the second function, splitting the input in half until one items remains has a runtime complexity of $log_2 n$. Further, since the merge function is called from within this function, the total runtime complexity of the algorithm is $o(n\ log_2 n)$. In fact, $o(n\ log_2 n)$ is the best worst-case runtime that can be achieved by a sorting algorithm".[15]

Is the algorithm stable and in-place?

Merge sort is **stable** and with the particular implementation discussed above, it is **not in-place**.

Why use this algorithm?

Merge sort is very efficient and scales well with an increasing input size. The arrays can be dealt with in parallel when necessary. However, when processing smaller array sizes, algorithms such as bubble and insertion sort are faster. As mentioned, merge sort is out-of-place due to the extra memory it uses creating copies of the arrays by recursively calling itself. As a result of this limitation, merge sort is not ideal in situations where memory constraints are a concern.

[15] Sorting Algorithms – Real Python (2021)

3. Implementation & Benchmarking

The algorithms above were implemented on python 3 and executed on a macOS Sierra Intel Core 2 Duo, version 10.12.6 (mid 2010) @ 2.4 GHz and 8 GB of RAM. Each algorithm processed input sizes from 100 to 10000. Using the inbuilt *time* method function, the average of 10 executions was recorded in milliseconds.

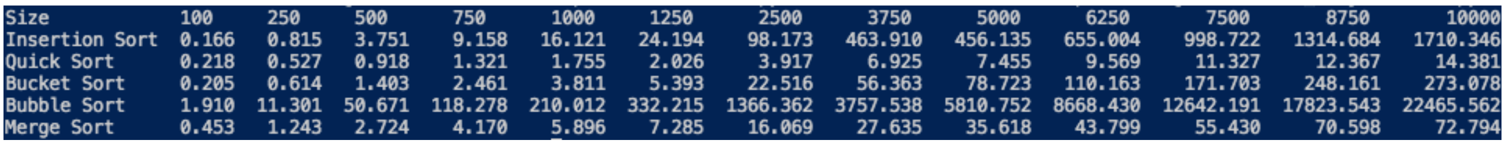
Performance expectations

Before looking at the empirical evidence, here are the expectations of how the algorithms should perform based on the knowledge gained from this report so far from fastest to slowest.

1. **Merge Sort**
   Merge sort's worse-case complexity is $O(n \log n)$ which is the same as quick sort's best case and should perform well on larger arrays. However, it does use auxiliary space unlike quick sort which will be a disadvantage.
2. **Quick Sort**
   Its worst-case $O(n^2)$ should be avoided because the input array is randomly generated and in all probability a decent pivot will be selected which improves performance.
3. **Bucket Sort**
   Bucket sort should outperform insertion sort because the input is random which will be favourable insofar as there will likely be an even spread of values (un-clustered) which can then be evenly distributed into the buckets.
4. **Insertion Sort**
   While insertion sort has a fast best case running time ($O(n)$), it will likely perform closer to its upper bound complexity $O(n^2)$ with the randomised array.
5. **Bubble Sort**
   Bubble sort will perform the worst as its worst-case runtime of $O(n^2)$ occurs when the array is shuffled (which is the case with randomly generated arrays) and because it must iterate through the array n times.

Benchmarking Results

The Pandas library was used to output the benchmark results of all five algorithms for various input sizes to a DataFrame (a 2-dimensional data object). See figure 1.



```
Size            100     250     500     750    1000    1250     2500     3750     5000     6250      7500      8750     10000
Insertion Sort  0.166   0.815   3.751   9.158  16.121  24.194   98.173  463.910  456.135  655.004   998.722  1314.684  1710.346
Quick Sort      0.218   0.527   0.918   1.321   1.755   2.026    3.917    6.925    7.455    9.569    11.327    12.367    14.381
Bucket Sort     0.205   0.614   1.403   2.461   3.811   5.393   22.516   56.363   78.723  110.163   171.703   248.161   273.078
Bubble Sort     1.910  11.301  50.671 118.278 210.012 332.215 1366.362 3757.538 5810.752 8668.430 12642.191 17823.543 22465.562
Merge Sort      0.453   1.243   2.724   4.170   5.896   7.285   16.069   27.635   35.618   43.799    55.430    70.598    72.794
```

Figure 1: Benchmark Averages in milliseconds

The Matplotlib library was used to create three plots (figure 2 & 3) depicting the results of the benchmark. The first plot shows all five algorithms. The second displays four algorithms, while leaving out bubble sort.

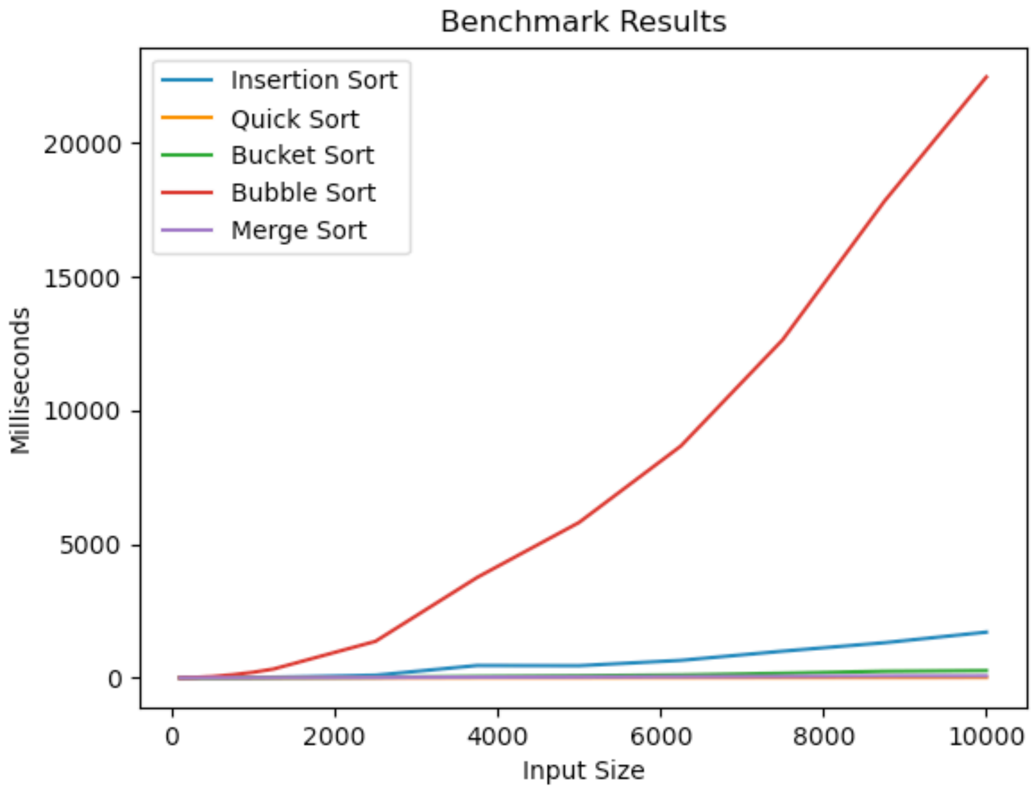


Figure 2: Plot of benchmark results of all 5 algorithms

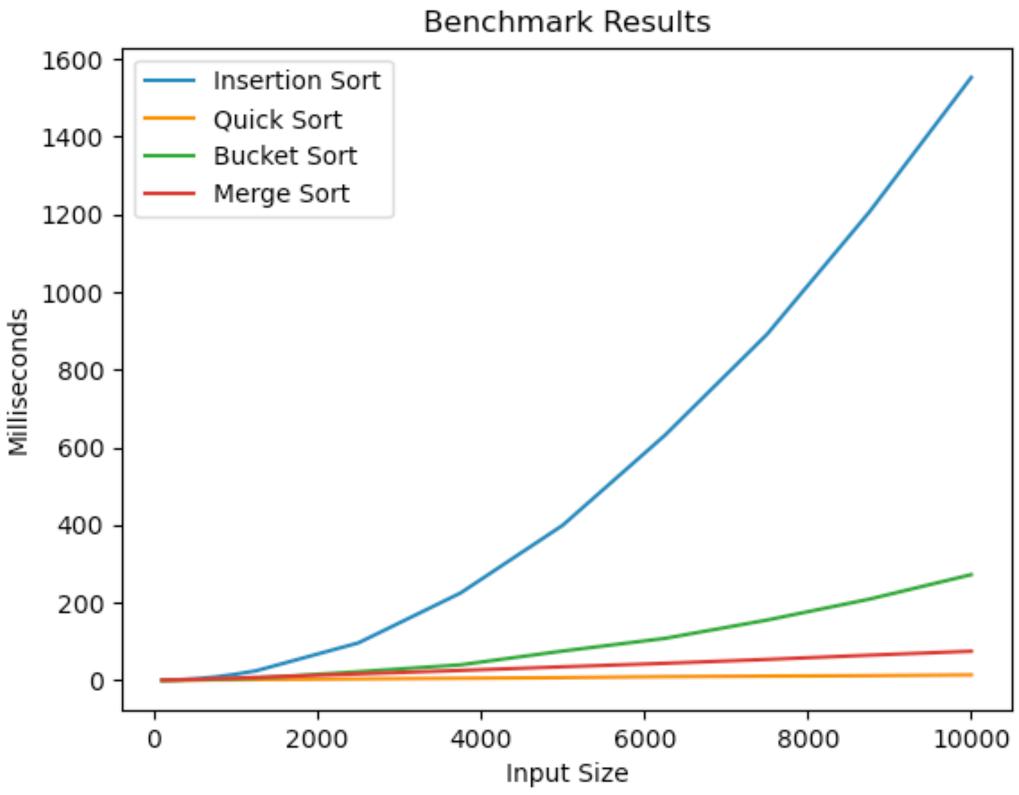


Figure 2: Benchmark results excluding bubble sort

The performance of algorithms is most similar for smaller input sizes. However, as the inputs increase their performance starts to become distinct. This is especially the case with bubble sort, whose performance eventually leads to an order of magnitude in difference. It is the least efficient sorting algorithm of the five, as expected. Unexpected, however, was the result of quick sort and merge sort with quick sort proving to be the most efficient sorting algorithm. It appears that the requirement of extra memory in the implementation of merge sort affects its efficiency; this seems especially noticeable when processing large unsorted data inputs.

4. Conclusion

The objective of this report was to comprehensively research sorting algorithms including their characteristics and conditions, time and space complexity, stability and memory use as well as to choose five sorting algorithms, discuss their implementation, benchmark their run-times and discuss the results. The analysis included the processing of randomly generated arrays of various sizes which resulted in large distinctions in running time between the algorithms. Notably, bubble sort proved thoroughly inefficient when dealing with large input sizes. However, when processing small input sizes, both bubble sort and insertion sort prove advantageous due to concise and simple code and the fact they do not use auxiliary space. On the contrary, quick sort and merge sort demonstrated good running time efficiency with larger input sizes. While their running times were quite similar, quick sort took precedence regarding efficiency. Various factors must have influenced this result such as the randomly selected pivot and the fact that it is in-place sorting. However, while quick sort was faster, it is important to remember it does not preserve the relative ordering of the original data as opposed to merge sort, which is a stable sorting algorithm.

Bibliography

Baeldung. “Guide to in-Place Sorting Algorithm Works with a Java Implementation.” *Baeldung*, 3 Oct. 2020, www.baeldung.com/java-in-place-sorting.

Brilliant. “Insertion Sort.” *Brilliant.org*, 15 Aug 2021, brilliant.org/wiki/insertion/

Collins English Dictionary. Glasgow: HarperCollins Publishers. Print. 1994.

Geeks for Geeks. Counting Inversions. Retrieved August 1, 2021 from https://www.geeksforgeeks.org/counting-inversions/

Dijkstra, Edsger Wybe, et al. *A discipline of programming*. Vol. 613924118. Englewood Cliffs: prentice-hall, 1976.

Heineman, George T., Gary Pollice, and Stanley Selkow. *Algorithms in a nutshell: A practical guide*. " O'Reilly Media, Inc.", 2016.

Ladinirenbeliz. “What Is Bucket Sort?” *Educative: Interactive Courses for Software Developers*, www.educative.io/edpresso/what-is-bucket-sort.

Knuth, D. E. "Sorting and Searching. Third edn. Volume 3 of The art of computer programming." 1997.

Mu, Qi, Liqing Cui, and Yufei Song. "The implementation and optimization of Bitonic sort algorithm based on CUDA." *arXiv preprint arXiv:1506.01446*, 2015.

Patrick, Mannion, “Sorting Algorithms Part 1”. COMP08033 Computational Thinking with Algorithms, GMIT, 2018.

Real Python. “Sorting Algorithms in Python.” *Real Python*, 15 May 2021, realpython.com/sorting-algorithms-python/#the-merge-sort-algorithm-in-python.