Benchmarking sorting algorithms

Fiachra O' Donoghue

G00398776

1 Introduction

1.1 Sorting

Sorting is the process of rearranging the members of a collection such that each member has a value less than or equal to that of the member to its right, and that equal values are contiguous. More formally, if A is a collection, and A_i and A_j are the i_{th} and j_{th} members of that collection, then in order for that collection to be sorted the following condition must be satisfied for all values of i and j; if $\text{A}_\text{i} < \text{A}_\text{j}$ then $\text{i} < \text{j}$; and, if $A_i = A_j$ then there must be no k such that $i < k < j$ and $A_i \neq A_k$ (Heineman et al., 2016, p. 53). A sorted collection A' must also be a permutation of the original collection A, that is, all of the members in A must appear in A' (Cormen et al., 2001, p. 15).

1.1.1 Comparators

If the members of a collection to be sorted are, for instance, integers, then it is clearly and intuitively understood what is meant by the relations '$<$', '$>$', '$=$', '$\neq$', etc. For other kinds of object however, it may not be obvious how they are to be compared. The definition of the relation is not the job of the sorting algorithm, which should be able to operate independently of the details of the relationship between the objects it is sorting. The job of defining the relationship should fall to a comparator function. The comparator, or "ordering relation" defined on a collection, must satisfy the conditions that, given any three items, a, b, and c from the collection, only one of $(a < b)$, $(a > b)$, $(a = b)$ may be true; and if $(a < b)$ and $(b < c)$, then $(a < c)$ (Knuth et al., 1968, p. 5).

In general, when a sorting algorithm is implemented, it is assumed that the user has access to, implicitly or explicitly, a comparator function, which, when passed two values a and b, will return 0 if $a = b$, -1 if $a < b$, and 1 if $a > b$ (Heineman et al., 2016, p. 55). A pseudocode example of a comparator function for comparing numbers is defined below.

Algorithm 1 A function for comparing numerical values

```
1: procedure COMPARATOR(a, b)
2:     if a < b then return -1
3:     if a = b then return 0
4:     if a > b then return 1
```

1.1.2 Inversions

In a collection of items, A, where A_i and A_j are the i_{th} and j_{th} members of that collection, if $i < j$ but $A_i > A_j$, then the pair (A_i, A_j) is an inversion (Knuth et al., 1968, p. 11). The degree of "sortedness" can be quantified by counting the number of inversions in a collection. For instance, the collection [1,3,4,2,5] is a permutation of the sorted collection [1,2,3,4,5], with two inversions; (3,2) and (4,2). The degree of sortedness, and therefore the number of inversions, of a collection can be an important consideration when choosing an appropriate sorting algorithm for a problem, as some algorithms perform particularly well, or poorly, on nearly-sorted or maximally unsorted inputs. A reverse-sorted collection will have the maximum possible number of inversions for that collection, and for some algorithms, such as insertion sort (see section 2.1) this represents the worst possible case in terms of efficiency.

1.2 Properties of sorting algorithms

In choosing a sorting algorithm, and deciding if it is a good fit for a particular problem, its is useful to consider the following properties.

1.2.1 Stability

Stability is the property whereby equally valued elements in the input collection maintain their positions with respect to one another in the output (sorted) collection. Stability can be important where satellite data must be moved around with the items being sorted, or where the output of the algorithm is to be used as the input to some other algorithm. Radix sort, for instance is dependent on the stability of counting sort (Cormen et al., 2001, p. 170).

1.2.2 Efficiency

Efficiency refers to the algorithm's use of resources, in particular, how it scales to increasing input size. Usually it is measured by runtime or memory use as a function of input size. A more detailed discussion of the topic will be found in the section on analysing algorithm complexity (section 1.4). However, an important property of algorithms as they relate to memory usage which it seems appropriate to include here, is whether or not their sorting is conducted *in-place*, meaning that no extra copies of the input data need to be made in the process of sorting.

In-place sorting algorithms might be particularly desirable in low memory environments such as embedded systems, while non in-place algorithms such as merge sort might be desirable for sorting data stored in a file external to the system (Heineman et al., 2016, p. 81)

1.2.3 Suitability for a particular input

In addition to size of input, specific characteristics of the input can affect the performance of an algorithm. For instance, given two sorting algorithms, one may outperform the other on an input with very few inversions, while the other may be more efficient given an input which is less sorted initially. Similarly, one algorithm may perform particularly well on smaller inputs while

performing poorly on larger ones. Insertion sort, for example, examined in more detail in section 2.1, performs extremely well on input sizes up to $\sim n{=}20$ (see figure 12 for a demonstration), but performance falls off as n increases giving it by far the worst overall performance of the five algorithms examined here. The differential efficiency depending on input characteristics can be exploited by algorithm designers to make highly efficient adaptive hybrid algorithms which switch strategy to best tackle the particular nature of the part of the input data on which they are currently working. Timsort, for instance, identifies and merges pre-existing runs in its input, and will add to those runs, where they are below a minimum threshold, using insertion sort which is very efficient on small inputs (Peters, 2002). The implementation of introsort used here, described in section 2.5, uses a recursive partitioning algorithm to divide the sorting task into sublists and switches to heapsort when a maximum recursion depth is reached, as well as to insertion sort when a sublist contains fewer than 20 elements.

It follows from the fact that an algorithm will perform better or worse depending on the nature, as well as on the size of its input, that choice of algorithm does not rest solely on its complexity relative to other competing algorithms. It also depends on the nature of the input in terms of its size, degree of sortedness, and the underlying probability distribution of the specific inputs it is likely to encounter.

1.3 Classes of sorting algorithms

Sorting algorithms may be classified in any number of ways, but only one will be examined here; whether the algorithm sorts by comparing items or by some other method.

1.3.1 Comparison

Comparison-based sorting algorithms, such as insertion sort (section 2.1) and quicksort (section 2.2), order a collection by comparing pairs of elements until the collection is sorted. No comparison sorting algorithm can perform better than $O(n \log n)$ in the average or worst cases. Quicksort and indeed, heapsort; see section 2.3, then, at $O(n \log n)$, are "asymptotically optimal" (Cormen et al., 2001, pp. 165–168), while insertion sort at $O(n^2)$ is not.

1.3.2 Non-comparison

Non-comparison-based sorting algorithms, such as bucket sort, radix sort, and counting sort, bypass the $O(n \log n)$ limit of comparison-based sorting algorithms by leveraging a priori knowledge about the input data to classify and count values, allowing them to sort their inputs in linear time. Counting sort (section 2.4), for instance, must construct an array equal in length to the range of the data. To do this it must have access to that information (Cormen et al., 2001, p. 168). Bucket sort needs a hash function that will uniformly distribute the input space into "buckets" (Heineman et al., 2016, p. 74). Both counting sort and bucket sort can complete their task in linear time but only under certain conditions. Bucket sort needs an input with a uniform distribution, and an ordered hash function (Heineman et al., 2016, p. 75), while counting sort degrades as n grows larger. See the plot in figure 11 for an example of this.

1.3.3 Hybrid

Hybrid sorting algorithms, such as timsort (Peters, 2002), and introsort (section 2.5), invoke different sorting algorithms depending on the characteristics of the data they are sorting. Introsort, for instance, defaults to quicksort, which recursively partitions the input, but switches to insertion sort for short subarrays, and heapsort when the recursion depth is likely to cause quicksort's performance to degrade (Musser, 1997).

1.4 Analysing algorithm complexity

Efficiency of algorithms can be assessed empirically or theoretically. To assess an algorithm empirically it should be, as described in section 3, executed and the resources of interest monitored. Usually those resources will be time and space, that is the time taken for the algorithm to complete its task, and the amount of memory it requires to do it. The results obtained through empirical analysis are useful, but they are tied to the implementation, the platform on which they are run, the size of their input, and a number of other variables unrelated to the algorithm itself. It is really a measure of performance of that implementation of the algorithm under very specific conditions.

To describe the efficiency of an algorithm in a way which is useful for comparison between algorithms independent of specific circumstances, it is necessary to isolate its description from its specific implementation. It is therefore analysed in terms of the number of operations it takes to complete its task for a given input size. More specifically it is analysed in terms of how the number of elementary operations scales with increasing values of n. For the purposes of these analyses all elementary operations are assumed to take the same amount of time.

If the complexity of an algorithm is described by a function connecting the size of its input to the number of operations it must perform, then we can classify those functions into performance families based on the order of their growth functions (Heineman et al., 2016, p. 18). As the size of n increases, the highest order terms of which it is a part come to dominate the value of the growth function to the degree that the constants and lower order terms become insignificant. This term allows the function to be assigned to a "performance family". Common performance families in algorithm complexity analysis are, in order of decreasing efficiency (Heineman et al., 2016, p. 19);

- Constant ($f(n) = 1$)
- Logarithmic ($f(n) = \log n$)
- Sublinear ($f(n) = n^d$ for $d < 1$)
- Linear ($f(n) = n$)
- Linearithmic ($f(n) = n \log n$)
- Quadratic ($f(n) = n^2$)
- Exponential ($f(n) = 2^n$)

Because algorithms may exhibit very different efficiency when applied to inputs with different characteristics, it is not possible to unambiguously define $f(n)$ without knowing what n is, and it is not possible to express a useful $f(n)$ that covers every possible input set the algorithm could possibly encounter. It is, however, possible to predict how the algorithm would perform in best case possible, i.e. when passed an input that allows it to perform the least possible number of operations. Similarly, it is possible to predict its performance in the worst and average cases. These functions are called asymptotic, and they express the upper bound of $f(n)$, in the worst case, the lower bound, in the best case, and the lower and upper bounds of the average case. The worst case performance is expressed as $O(f(n))$, the best case is expressed as $\Omega(f(n))$, and the average case is expressed as $\Theta(f(n))$ (Heineman et al., 2016, p. 18).

The best, worse and average case complexities for the five algorithms examined below are listed in table 1 in section 3.2.

2 Sorting Algorithms

2.1 Insertion sort

Insertion sort is a comparison-based sorting algorithm which works in linear time in the best case (Ω(n)), i.e. given an already sorted input, and quadratic time in the average and worst cases ($\theta(n^2)$ and $O(n^2)$). The algorithm sorts in-place so space complexity is constant. It is efficient when sorting small or almost sorted inputs, and where there are many duplicate elements (Heineman et al., 2016, p. 60).

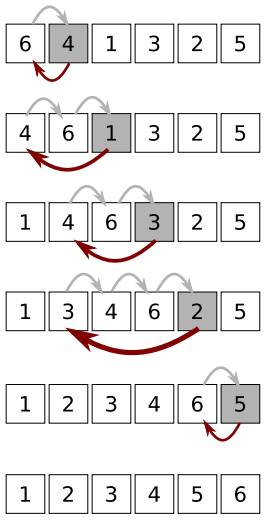

Figure 1: Insertion sort.

The algorithm consists of two loops, one nested inside the other. The outer loop iterates from the second element in the input list to the end. If the index of the current outer loop value is i, the inner loop compares the value at i, known as the *key* with the value to its left (i - *1*). If the element at i - *1* is less than, or equal to, the key, no action is necessary as the two elements are sorted with respect to one another. However, if the value at i - *1* is greater than the key then the value at i - *2* is compared with the key, and the inner loop continues leftward until either the beginning of the list is reached or a value smaller than the key is found. When this occurs, the key is inserted into the appropriate slot in the list and all of the elements that were compared with it and found to be larger are shifted one position to the right.

Figure 1 demonstrates the insertion sort operating on an unsorted list of six digits. Each row represents an iteration of the outer loop. The key in each iteration is represented by the grey box. The red arrow denotes the insertion of the key into its new position, and the grey arrows represent the shifting of all of the intervening, larger, values one position to the right. Note that the elements to the left of the key are always sorted with respect to one another, while the elements to the right of the key have yet to be inserted into this sorted sub-array. Note also that each grey arrow represents one iteration of the inner loop. The inner loop needs to run once only for each inversion of which the current key is a part. This is why the algorithm has a running time of Ω(n) for an already sorted list — because a single run of the outer loop — n - 1

iterations — with one comparison per iteration, and without any iterations at all of the inner loop, will confirm that the input is sorted.

The worst case running time is equal to $n \times (n-1) \times \frac{1}{2} = \frac{n^2-n}{2}$ and the average case time is $n \times (n-1) \times \frac{1}{4} = \frac{n^2-n}{4}$ (Woltmann, 2020b). Taking only the highest order term into account as this will be the dominant term as n grows, this gives us average and worst case performances of $\theta(\mathrm{n}^2)$ and $O(n^2)$. Finally, the algorithm is stable, as, if the algorithm encounters a value equal to the key to the key's left, it will simply move back to the outer loop and onto the next element, leaving the two equal elements where they are.

2.2 Quicksort

Quicksort is a comparison-based sorting algorithm which recursively partitions its input data based on whether each element is higher or lower than a sometimes arbitrarily chosen *pivot* value. The algorithm runs in $O(n \log n)$ time in the best and average cases, and $O(n^2)$ in the worst case.

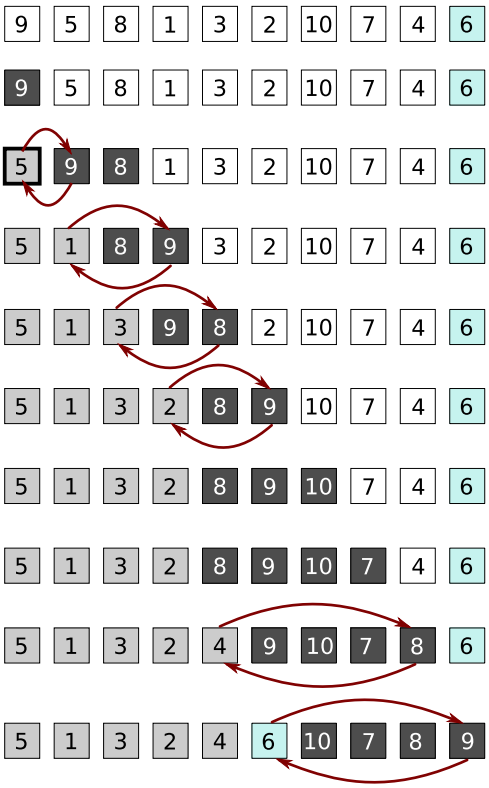

Figure 2: Quicksort partitioning.

Figure 2 illustrates the partitioning mechanism of the algorithm. First a pivot is chosen. Much can be said on the subject of pivot selection, and the matter is discussed below, but for the purposes of this demonstration, the pivot is taken to be the last element of the input array. The pivot value is highlighted in blue. As the algorithm progresses, each value in the array is compared to the pivot value. If the current value is greater than the pivot value it is assigned to the right-hand sub-array — represented by the dark boxes in figure 2. If the current value is less than the pivot value then it is swapped with the first value of the right-hand sub-array. Finally, when the pivot value, which is located in the last array position, is reached, it is swapped with the first value of the right-hand sub-array. This places it between a sub-array containing only values smaller than it, and a sub-array containing only values that are equal to or exceed it. It also places it in the position it will occupy in the fully sorted array. The partitioning portion of the quicksort algorithm will then return the location of the pivot value so that two recursive calls to quicksort can partition one of the two sub-arrays each.

Figure 3 illustrates the overall operation of quicksort. The first two rows show the original array to be sorted and the result of the call to the partition algorithm shown in figure 2. After partitioning, the original pivot value, 6, is in its final sorted position. The two sub-arrays on either side of 6, one containing only values less than and one containing only values greater than, 6. The last value in each array is again chosen as the pivot (row 3) and both subarrays are again partitioned (row 4). The two pivot values, 4 and 9, for the sub-arrays partitioned in row 4 are now in their final sorted positions. Of the four sub-arrays produced by the partitioning in row 4, two contain just a single value, meaning that they are now in their final sorted positions; numbers

5 and 10 are now sorted. The remaining two sub-arrays are partitioned, producing two more sorted pivot values, 2 and 8 (row 6), and three single-element arrays (1, 3, and 7), completing the array sort.

The array has been sorted in-place, without any overhead, giving a space complexity of n. As for time complexity, it has already been mentioned that quicksort finishes in $O(n \log n)$ time for best and average cases. Bentley (1999, p. 119) makes the point that as comparison algorithms, such as quicksort, are provably limited to $O(n \log n)$, quicksort's performance is "close to optimal." He goes on to say, however that the algorithm's performance degrades to $O(n^2)$ given some common inputs such as an array with long runs of identical elements.

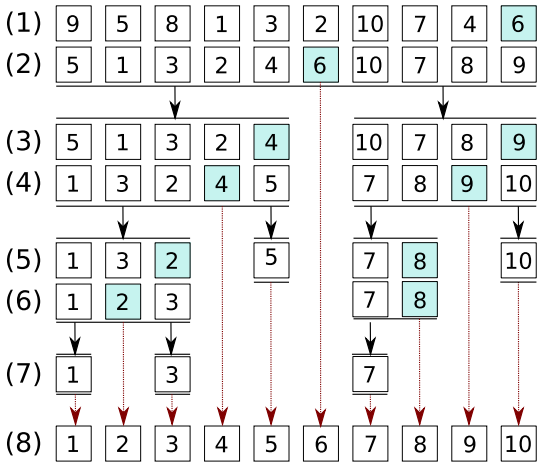

Figure 3: Quicksort sorting.

Poorly chosen pivot values can cause poor performance. Performance is likely to be best — all else being equal — when the pivot, for the most part, is located near the middle of the data, so that the two partitioned sub-arrays are of roughly size. This being the case, suggests Sedgewick (1978, p. 851), the median may be an optimal value for the pivot, and he goes on to outline a method for sampling three values — the first, the last, and the middle, from each array or sub-array to be partitioned, and setting the pivot value to the median of those three. Other frequently used values are the first value in the array, the last value, or a random value (Heineman et al., 2016, p. 73).

One final interesting optimisation, attributed without citation by Bentley (1999, p. 121) to Sedgewick, is the use of insertion sort to sort small sub-arrays for which quicksort would be inefficient. The suggested implementation would entail halting partitioning when array length fell below a certain threshold, and using insertion sort to complete sorting of the almost-sorted final array. This solution starts to resemble introsort (Musser, 1997), which is discussed below in section 2.5.

2.3 Heapsort

Heapsort is a sorting algorithm that sorts data by first rearranging it so that it forms a *max-heap*, then taking the top value, and recursively rearranging the remaining elements into a max-heap, then taking the new top value, and so on, until the heap is exhausted and the array is sorted. The algorithm runs in $O(n \log n)$ time in the best, average, *and* worst cases. It sorts in-place so its space complexity is $O(1)$ (Cormen et al., 2001, p. 129).

A heap is a binary tree which is completely filled, with the possible exception of the lowest level, which, to the extent it *is* filled, is populated from left to right, with the rightmost nodes filled last. A max-heap is a heap in which the value of every node, with the exception of the root node, is less than or equal to that of its parent (Cormen et al., 2001, pp. 127–129). A max-heap can be stored as 1-based array with the root node at index, $i = 1$ and such that given the index i of any node, the parent of that node will be at $i/2$, and its left and right children will be at $2i$ and $2i + 1$, respectively (Cormen et al., 2001, p. 128; Bentley, 1999, p. 148). In the implementation used here, however, the root node is stored at $i = 0$, putting a node's parent at $(i\text{-}1)/2$, and its left and right children at $2i + 1$ and $2i + 2$, respectively. An example of a max-heap can be seen in

figure 4.

Any array can be seen as a heap. Note the array indices attached to the heap nodes in figure 4; the nodes are simply filled from left to right, and top to bottom. Rearrangement of the heap, (i.e. rearrangement of the underlying array) so that it complies with the requirement that each parent node be greater than or equal to each of its child nodes, will produce a max-heap. The heapsort algorithm sorts an array, A, by repeatedly forming a max heap and removing the top node — which, by definition, is the greatest value in the array, and which will be located at position A[0]. The removed maximum value is swapped with the value at the end of the array. From this point the array is split into a heap sub-array and a sorted sub array. At each iteration of the algorithm, the greatest remaining value in the heap sub-array is identified by rearrangement of that sub-array into a max-heap, and is swapped with the last value in the heap sub-array. Thus, the sorted sub-array grows larger and the heap sub-array grows smaller, until the array is fully sorted. Figure 5 illustrates the process.

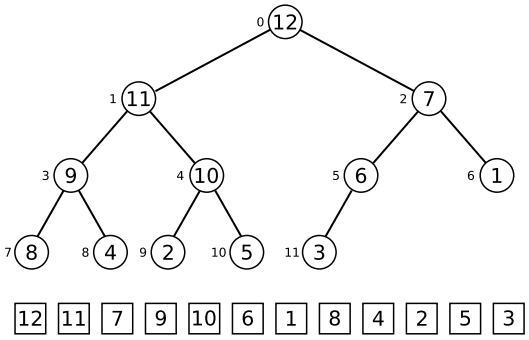

Figure 4: A max heap generated from an array holding the numbers 1 to 12. The small number to the left of each node is the index of that node's value in the array below.

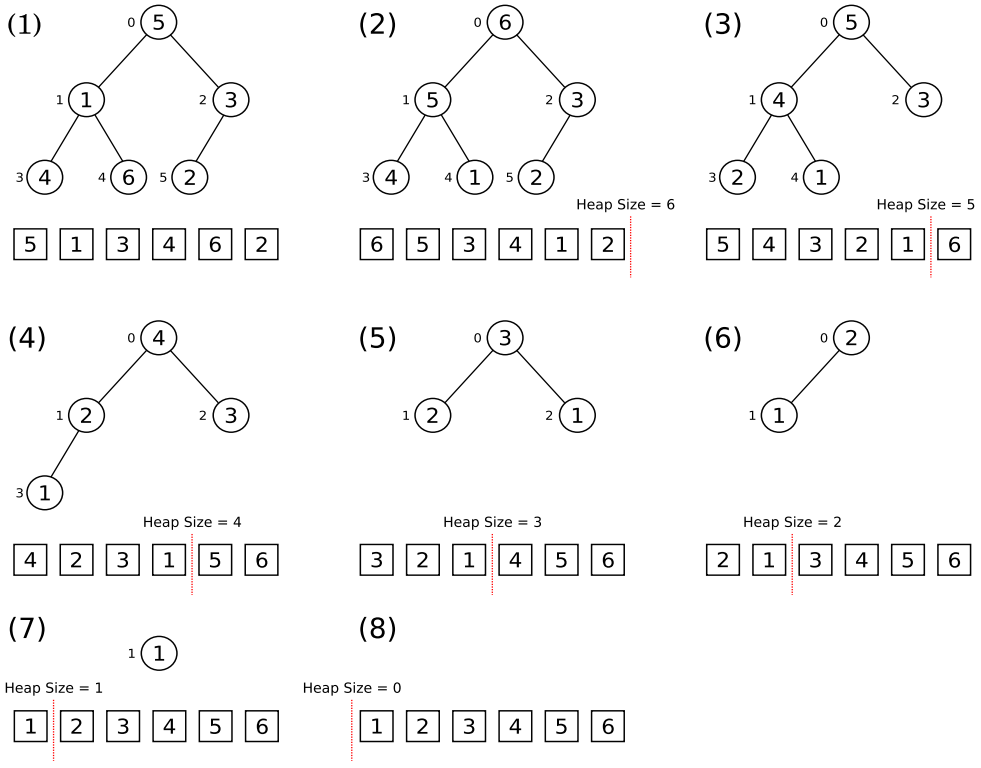

Figure 5: The process of sorting a 6 element array using heapsort: (1) the original array viewed as an array and as a heap; (2) the array rearranged as a max-heap; (3) the top value in the max-heap is swapped with the last value in the array. A new max heap is formed; (4-6) max heaps are formed, top value is swapped into last position of heap sub-array, heap-size is decremented; (7) max-heap of 1 node is "swapped" into the sorted subarray; (8) the sorted array.

2.4 Counting sort

Counting sort is the first non-comparison sort reviewed here. The algorithm sorts data by counting the number of occurrences of each unique value in the input and then calculating the index in the sorted output of the last incidence of each value. With this information it is able to populate a sorted output list. Counting sort runs in $O(n)$ time when the range of possible values, k is $O(n)$ (Cormen et al., 2001, p. 168). It is stable, and it has a space complexity of $O(n + k)$ (Woltmann, 2020a).

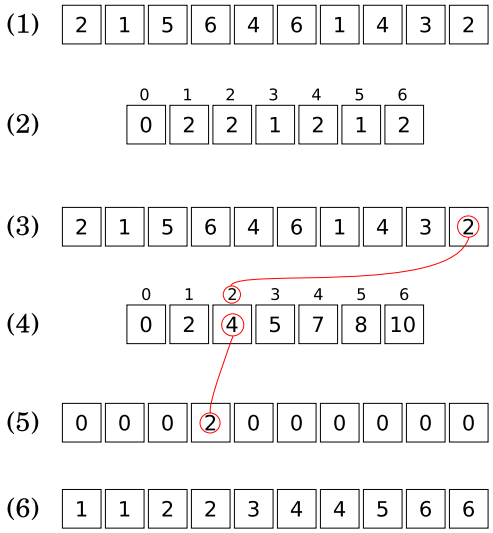

Figure 6: The process of a counting sort algorithm: (1) The array to be sorted; (2) The counter array — the value at index i is incremented when the value i is encountered in the input array; (3) the input array is iterated through in reverse; (4) the count array now holds a cumulative sum of counts; (5) the output array is populated using the cumulative sum of counts — effectively showing positions of last occurrences of values; (6) cumulative counts are decremented as they are used, and the sorted output array is populated.

Counting sort must know the range of values, k in its input data. When it is passed an array for sorting, the algorithm creates two further arrays: a count array, of length k and an output array equal in length to the input array. The count array should be initialised with zeroes. The algorithm begins by iterating through the input. For each element it encounters it increments the value in the count array the index of which is equal to the value encountered. For example, if the current element in the input array has a value of 5, then the value in the count array at index 5 is incremented. When the entire input array has been traversed, the values in the count array are converted to a cumulative sum of value counts, by iteratively adding each value to the value that precedes it. This effectively provides a list of positions of the last occurrence of each value in the sorted output array. To make use of this list, the input array is iterated through in reverse and, for each value encountered, the value at the index in the count array that equals that value provides the position for that value in the output array (see the red lines connecting steps 3-5 in figure 6 for a visual example of this). The value in the count array is then decremented so that when that value is encountered in the input array again it will be placed in the next position to the left.

2.5 Introsort

Introsort is a hybrid sorting algorithm developed to address some shortcomings with quicksort. Musser (1997, p. 983) points out that while quicksort runs in $O(n \log n)$ time in best and average cases, it degrades to $O(n^2)$ in the worst case, when recursion depth passes some threshold. Furthermore, while heapsort runs in $O(n \log n)$ in the worst case, it remains between 2 and 5 times slower than quicksort. Sedgewick (1978) addresses the issue with quicksort's poor worst case performance, suggesting alternative pivot selection methods, but also suggesting the use of insertion sort on small partitions. Musser (1997, p. 986) adopts Sedgewick's suggestions, using insertion sort for partitions below a threshold and also using "'median-of-three"' pivot selection, but he also attempts to stave off recursion problems by switching to heap sort when a recursion depth of $\log n$ levels is reached (Heineman et al., 2016).

The resulting algorithm runs in $O(n \log n)$ in best, worst and average cases. Its space complexity is $O(\log n)$ as its constituent algorithms all sort in-place and the only extra requirement is quicksort's stack space.

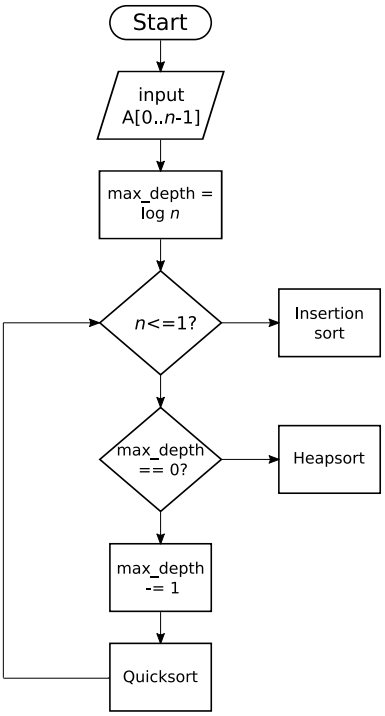

Figure 7: Introsort flow chart

3 Implementation & Benchmarking

3.1 Implementation

The five algorithms described above are implemented in `benchmark.py` as `insertion_sort()`, `quicksort()`, `heapsort()`, `counting_sort()`, and `introsort()`. All take as input a Python list of comparables, i.e. anything that implements the `__lt__`, `__gt__`, and `__eq__` methods. Some of the sorting functions require additional parameters. These functions have been so arranged, using wrapper functions or defaults, as to be callable using only an input list, simplifying automated benchmarking. Where an algorithm uses a helper function, the second function is defined as in inner function of the first. An exception to this is the `partition()` function which comprises the bulk of the quicksort algorithm, and which is defined as a separate top-level function, because it must also be available to `introsort()`.

Benchmarking is handled by the `benchmark()` function, which expects, at a minimum, a list of functions implementing the algorithms to be benchmarked, and a list of input array sizes to be generated and used as test data.

When `benchmark()` is executed, a list of lists of random integers is generated according to user specification. The number and length of the lists is provided to `benchmark` by a parameter called `arr_sizes`, and the range of possible values in the lists can be specified in a parameter called

`intrange` which defaults to (0, 100). Following input generation, in a series of nested loops, each function is repeatedly run on each input array. The number of repetitions can be specified in a parameter called `reps`, but it defaults to 10. As most of the algorithms tested sort in-place, a copy is made before each pass, and it is the copy which is sorted and then discarded. In any one run of the benchmark function, each of the algorithms benchmarked is timed on an identical set of inputs.

Python's `time.time()` is called immediately before and immediately after each run of each algorithm, and the duration of the run is calculated. Each algorithm's mean runtime on each input list size is recorded. After all benchmarks complete, the list of lists holding mean runtimes per algorithm and input size, is returned to the caller. Optionally, all of the individual times, as well as all the input lists, can be dumped to disk in a timestamped json file for further analysis.

The `main()` function in `benchmark.py` initialises the list of algorithms and a list of input array sizes and passes them to the `benchmark()` function. The resulting table of mean runtime by algorithm and input size is stored in a variable, `result`. This variable is used to generate a LaTeX-formatted table of results. This table is written to the working directory.

Finally, some plots are produced using the mean time data and saved to the working directory with timestamped filenames. The plots are generated, using `matplotlib`, by the `write_plot()` function, also defined in `benchmark.py`.

3.2 Benchmarking

Algorithm	Best case	Worst case	Average case	Space complexity	Stable
Insertion sort	n^2	n^2	n^2	1	Yes
Quicksort	$n \log n$	n^2	$n \log n$	1	No
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Counting sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No

Table 1: Complexity of each of the five algorithms.

Table 1 shows the time and space complexity for each of the five algorithms. Assuming the random integer list generated by `benchmark` represented somewhat average cases for these algorithms to sort, and based on the orders of time complexity in the average case associated with each of the algorithms, the algorithms might be expected to be ordered as follows in terms of runtime (from slowest to fastest):

1. Insertion sort (n^2)
2. Quicksort, Heapsort, and Introsort (all $n \log n$)
3. Counting sort (n)

Figure 8 shows a chart of mean times per algorithm for different values of n. Insertion sort is clearly the slowest sorter, by far, at least on the data it was given. Looking more closely at the performance of the remaining algorithms (by removing insertion sort), in figure 9, quicksort, heapsort, and introsort are clumped together, as the complexity table predicts, and counting sort

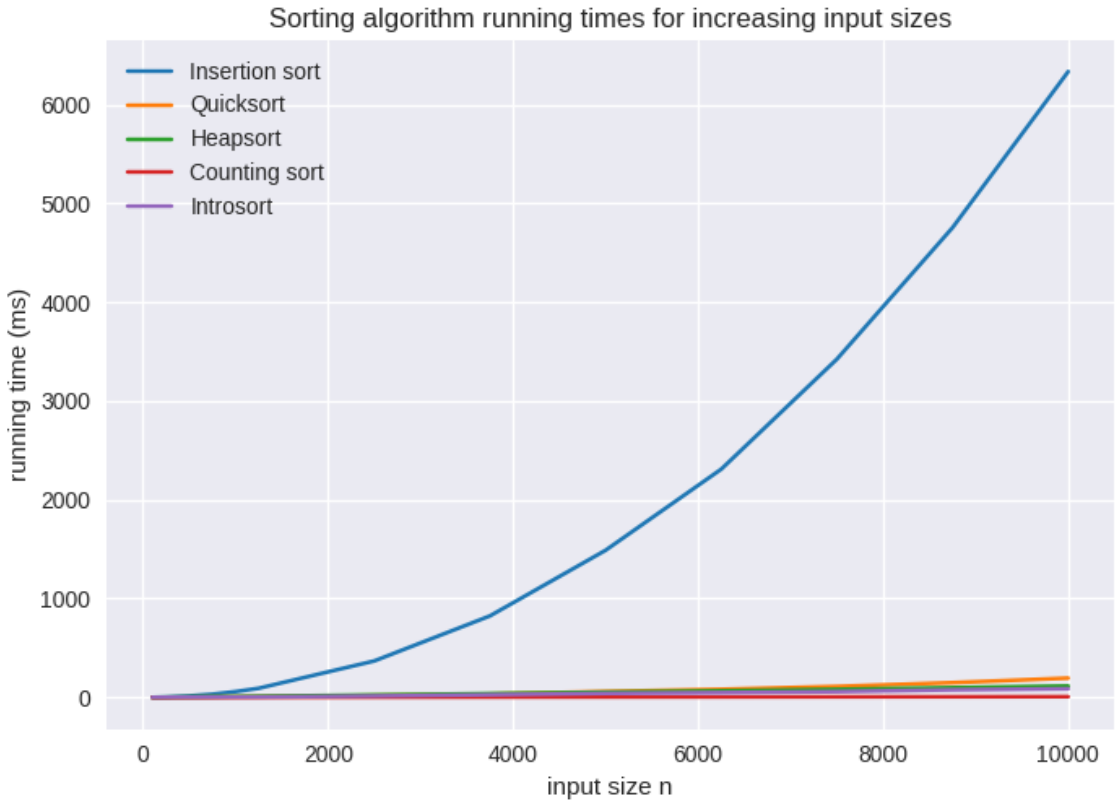

Figure 8: Relative performance of five algorithms

performs best of all, as expected. The relative performances of all five algorithms is probably best illustrated in figure 10 which shows all five on a log y-scale. Table 2 holds the data from which these charts were generated.

Of course, different algorithms perform differently on different *types* of input too. If that wasn't the case, based on the benchmarks here, no-one would use anything but counting sort. To illustrate this we can observe what happen if the benchmark is run with exactly the same parameters as before but with a range of (0, 1000000) rather than (0, 100) (figure 11). Counting sort fails to perform efficiently because its sensitive to k, the maximum value of the input array.

Similarly, insertion sort excels at short and nearly-sorted arrays, as can be seen in figure 12 where, up until about 25 elements, it outperforms all the other algorithms. It is because of these different suitabilities that so many sorting algorithms exist and that hybrid algorithms, such as introsort and timsort are so successful.

$n=$	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**3750**	**5000**	**6250**	**7500**	**8750**	**10000**
Insertion sort	0.657	3.658	14.715	31.833	58.102	91.984	368.778	824.728	1489.387	2310.453	3424.353	4755.968	6335.822
Quicksort	0.422	0.861	1.982	3.547	5.251	6.596	19.474	34.223	59.113	82.051	110.927	148.855	195.104
Heapsort	0.609	1.671	3.787	5.971	8.495	10.940	24.428	38.608	53.251	66.133	81.637	98.850	115.702
Counting sort	0.086	0.173	0.370	0.557	0.771	0.915	1.802	2.784	3.757	4.422	5.378	6.176	7.040
Introsort	0.263	0.598	1.400	2.380	3.391	4.025	13.526	26.568	40.257	48.054	58.275	76.574	87.299

Table 2: Times in milliseconds to sort arrays of size n for each of the algorithms.

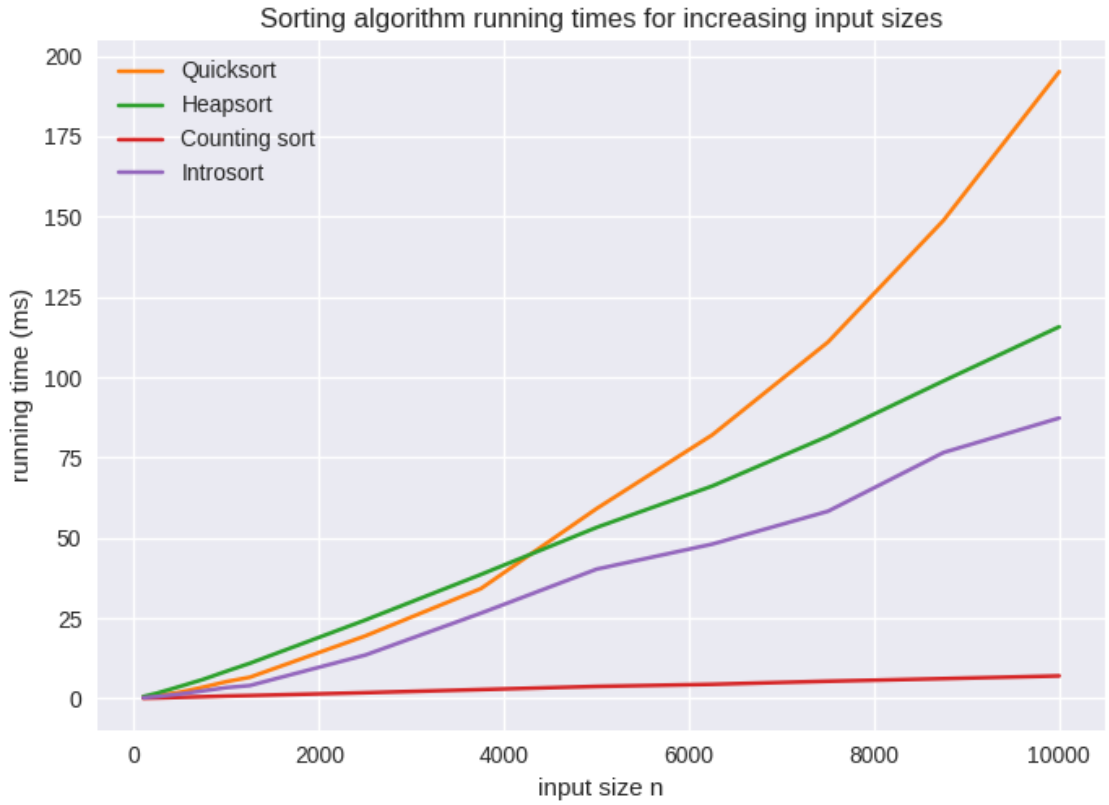

Figure 9: Relative performance of four algorithms — insertion sort has been excluded

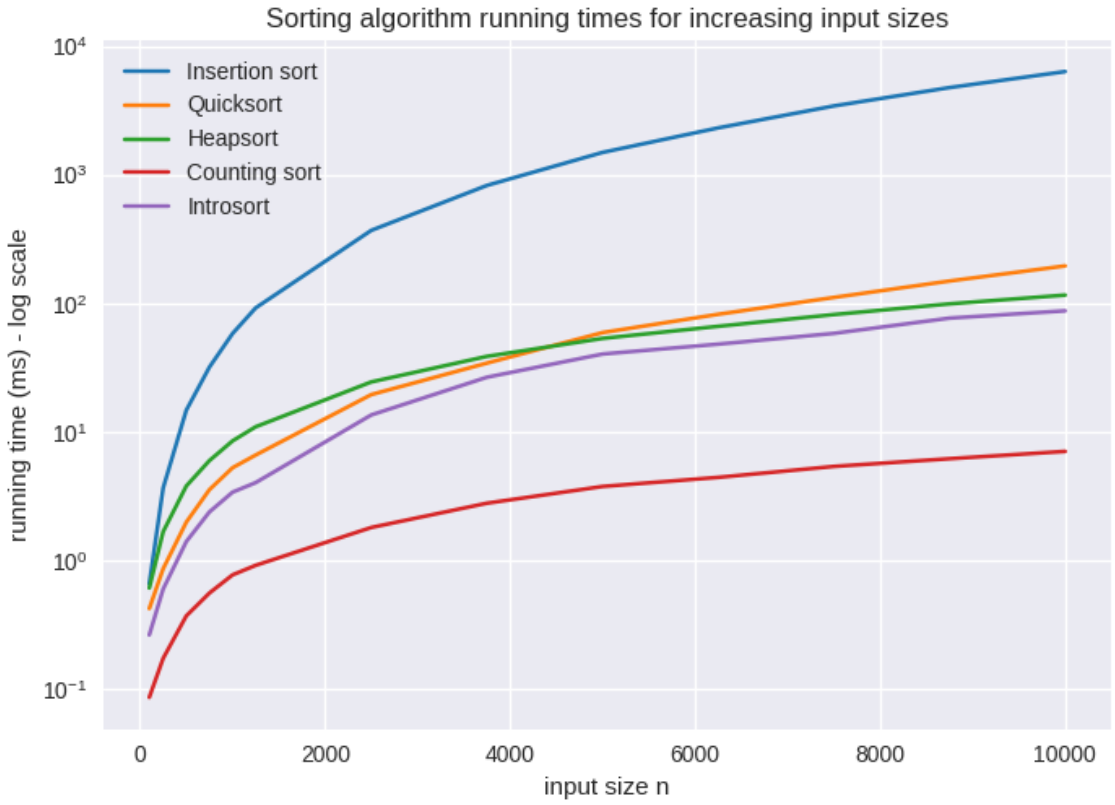

Figure 10: Relative performance of five algorithms using a log y-scale

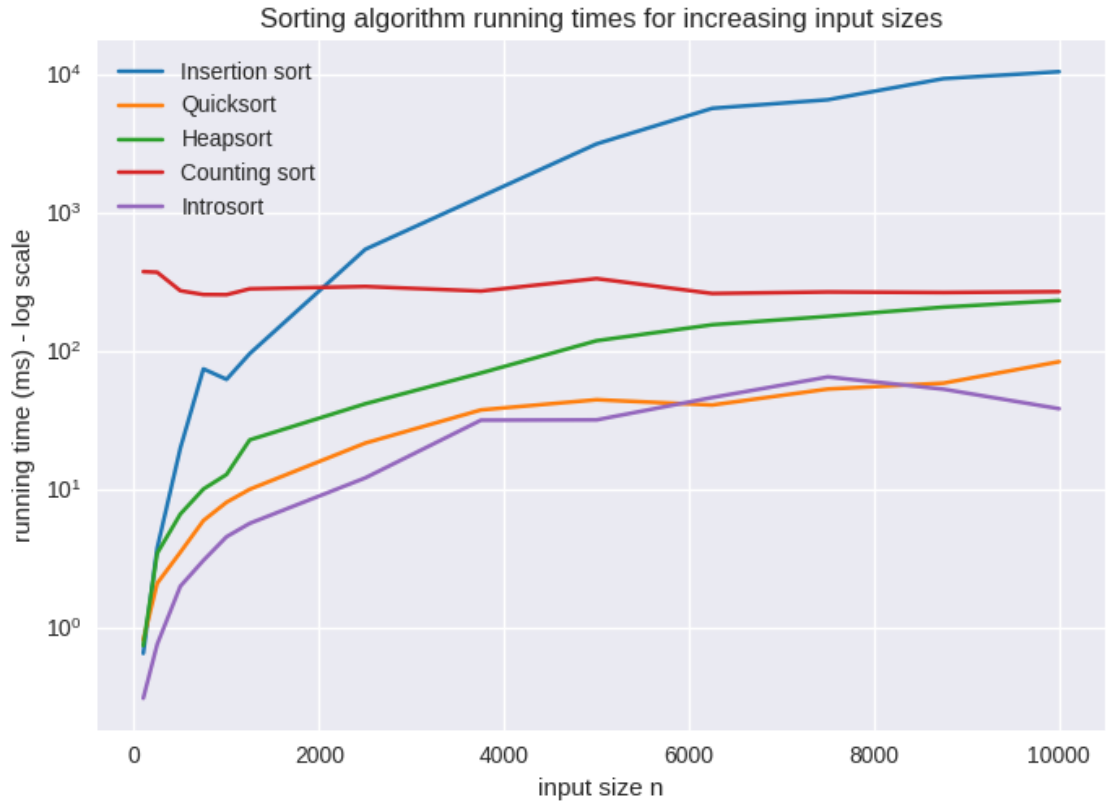

Figure 11: Relative performance of five algorithms using a log y-scale, and input range (0, 1000000)

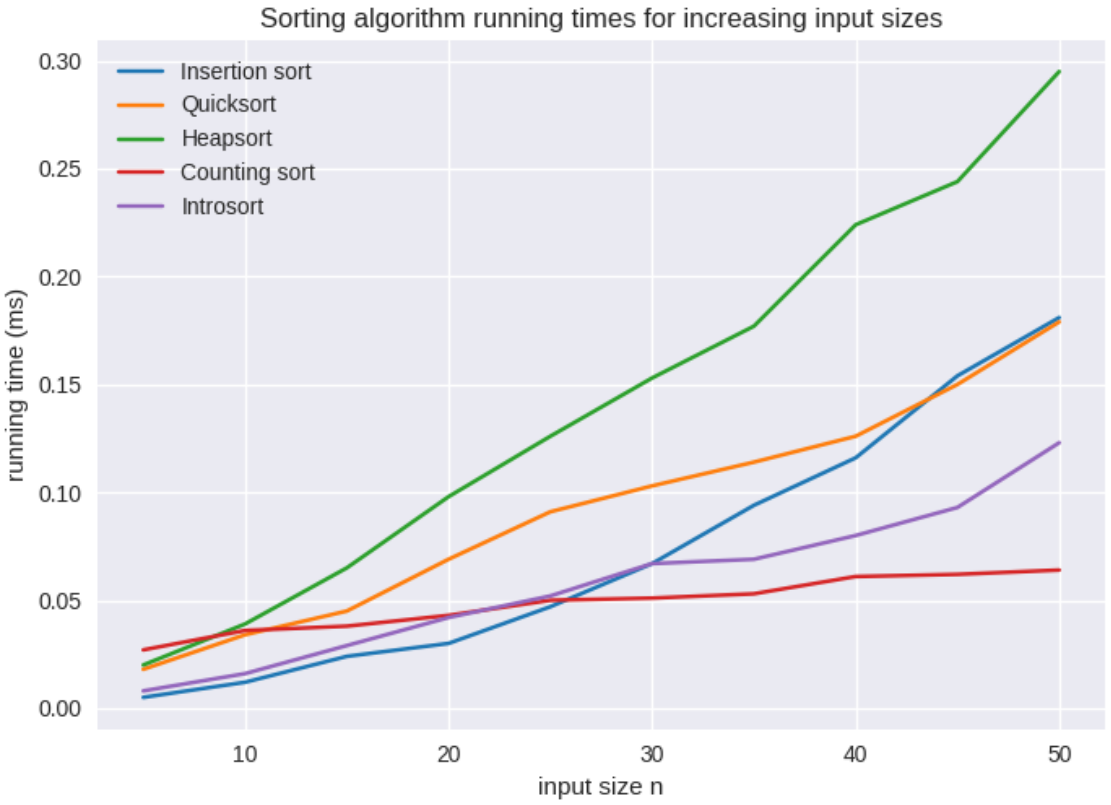

Figure 12: Relative performance of five algorithms with maximum 50-element input

References

Bentley, J. (1999). *Programming pearls (2nd ed.)* ACM Press/Addison-Wesley Publishing Co.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (Second). MIT Press.

Heineman, G., Pollice, G., & Selkow, S. (2016). *Algorithms in a nutshell*. O'Reilly. https://books.google.ie/books?id=95zVsgEACAAJ

Knuth, E., Varga, R., Knuth, D., & Harrison, M. (1968). *The art of computer programming: Sorting and searching*. Addison-Wesley Publishing Company. https://books.google.ie/books?id=ZQu9mgEACAAJ

Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, *27*(8), 983–993. https://doi.org/https://doi.org/10.1002/(SICI)1097-024X(199708)27:8

Peters, T. (2002). [python-dev] sorting. Retrieved May 10, 2021, from https://mail.python.org/pipermail/python-dev/2002-July/026837.html

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, *21*, 847–857. https://doi.org/10.1145/359619.359631

Woltmann, S. (2020a). Counting sort – algorithm, source code, time complexity. Retrieved May 10, 2021, from https://www.happycoders.eu/algorithms/counting-sort/

Woltmann, S. (2020b). Insertion sort - algorithm, source code, time complexity. Retrieved May 10, 2021, from https://www.happycoders.eu/algorithms/insertion-sort/