Galway Mayo Institute of Technology

H. Dip in Science
(Software and Computing)

Computational Thinking and Algorithms

Benchmarking Sorting Algorithms in Java

Author
Kieran P. Somers

Lecturer
Dr. Dominic Carr

May 3, 2020

1 Introduction

"Data is the new oil" – in 2017 *The Economist* reported that the World's most valuable resource was no longer petroleum [1, 2]. We are entering the age of the 4^{th} Industrial Revolution [3], an era characterised by the convergence of technologies, biologies, and societies and one which is forecast to change the way in which individuals live and work, and the way in which entire societies and economies operate and interact. Indeed one can hardly go a day without encountering a popular science or news article reporting on the latest breakthroughs facilitated by "Big Data", "AI", "Smart Devices", or the "Internet of Things".

At both the forefront and foundations of this new era is information technology, its central role cemented by advances in materials and hardware – from small scale wearable devices, to massively parallel Exascale computing infrastructures [4], to the development of the first generation of commercial quantum computers [5]. In parallel with these physical advances, are advances in the software we use to exploit available computing power – be it to improve our social experiences *via* the latest applications and games, or the development of advanced numerical scientific libraries optimised for novel tensor-processing-units (TPUs) [6] to be used in machine-learning and AI applications. There is no doubt that computational technology is developing at ever increasing rates as a result of the synergistic development of novel computational hardware and software.

Whilst developing novel hardware is largely an experimental endeavour, albeit one which is almost certainly accompanied by theory (i.e. the design of novel superconducting materials), the analysis and development of novel software and algorithms certainly requires a greater emphasis on theoretical insight, particularly when performance is important. The ability to analyse a problem in a computational manner, and transform that into an efficient and robust algorithm is a critical skill for a modern software developer. The theoretical under-pinnings of algorithm analysis and design are much too broad to consider in any detail here, rather, we will focus on gaining some entry-level insight by way of one of the most frequently occuring computational operations carried out – the sorting of a list.

A "greenhorn" approach is as follows: "Hey Alexa. What's the fastest way to sort a list?". Whilst Alexa may offer some naive insight, if the same question were asked of two seasoned computational thinkers (i.e. programmers), they may well ponder for some time, and return some time later-again, having devised two different solutions to the same problem, each equally valid within its own set of constraints (input size, available hardware, software restrictions, etc.). What is likely, or at least what one hopes would have

occured, is that both programmers have justified their approaches based on sound scientific principles – their design has been accompanied by some formal or informal analysis, be it theoretical or empirical. In this report we will take such an approach, and analyse the performance of various sorting algorithms using both theoretical and empirical methods.

The aim of all sorting algorithms is to take some list or collection of objects, be they primitive or complex types, and to place them in some predefined order of relevance to the type. In some cases the "natural order" is obvious, for instance an algorithm which sorts the natural numbers would place the number 1 before the number 2, 2 before 3, etc. based on increasing numerical value. Similarly, the natural ordering of strings of text would place *a* before *b*, *b* before *c* and so on, in an increasing lexicographical order. In cases where no obvious natural order emerges, the precise definition of whether a list of objects is sorted is dependent on some definition of equivalence as defined by a *comparator* function, which will vary depending on the objects being compared, and the situation. For instance, the natural ordering for a collection of dogs is ambiguous, and one could sort them by age (a natural numerical ordering), name (a natural lexicographical order), color, breed, sex, or some combination of all of the above. What is important is that the definitions of equality are put in place prior to any sorting happening by the definition of *some* comparator function.

Irrespective of the details of equivalence, for a sorting algorithm to be deemed functional it must typically satisfy two criteria. The first is that it places the elements in increasing order where equivalent elements are contiguous. For an array A with at least two integer indices i and j, if $i < j$ the corresponding array elements $A[i]$ and $A[j]$ must also satisfy the criteria that $A[i] \leq A[j]$ or else the array is unsorted. By the same logic an array A is deemed unsorted if it contains an element $A[i] > A[j]$ where the index $i < j$. In terms of contiguity, if for an array A there is an element $A[i]$ equal to an element $A[j]$, there can be no element $A[k]$ such that $i < k < j$ and $A[i] \neq A[k] \neq A[j]$. The second condition which must be satisfied when evaluating the behavour of a sorting algorithm is that given an unsorted array A_u, the sorted array A_s must be some permutation of A_u – that is, the contents of both arrays must be the same, only arranged in a different order, and elements cannot be removed, or changed to satisfy sort criteria.

Note the frequent use of the logical $>$, $<$ and $=$ operators which define whether two elements are less than, greater than, or equal to each other. Again the precise definition of less than, greater than, or equal to may vary depending on the task at hand. For primitive numeric types, such as int, $float$, and $long$ types the familiar mathematical definitions apply, and in the current work we aim to place a series of positive integers in order of increasing

value, from smallest to largest.

All definitions of equality described above effectively rely on comparisons of values using the logical $>$, $<$ and $=$ operators, and these operators are employed by so-called comparison-based methods. Certain algorithms, entitled non-comparison-based methods, forego these operations entirely, and these types of algorithms are particularly useful when comparing non-numeric data, as in the dog sort example given previously. Both comparison- and non-comparison based algorithms will be considered herein.

Sorting algorithms are further differentiable by other desirable features – their efficiency (both in terms of speed and memory, often termed their time- and space-complexity), their stability (whether logically equivalent elements remain in the same relative order pre- and post-sorting), and whether the sorting occurs "*in-place*" or "*out-of-place*" (whether auxiliary memory is required to perform the sort).

Generally speaking, the perfect sorting algorithm would satisfy all of the criteria required of a functional sorting algorithm described previously but with low time- and space-complexity (indicating efficiency), in a stable manner (if required), and the sorting would take in-place such that memory-usage is minimized. As we will see, trade-offs must be struck between these features – certain algorithms outperform others on some of these properties, but typically not all of them, and algorithms tend to be chosen based on their overall performance which is likely based on a combination of the above factors, and the hardware which is available.

One issue which arises with respect to benchmarking sorting algorithms is experimental design. Assuming that the algorithm, input size, and programming languages are constants, differences in hardware (chip speed, memory size and bandwidth) can easily lead to factors of two or greater differences in the time required to sort a list of items. Similarly, for a given input size and set of hardware, an algorithm written in low-level or compiled languages such as FORTRAN, C++ or JAVA is quite likely to outperform higher-level or interpreted languages such as PERL, JAVASCRIPT, and PYTHON. Of course, not only do hardware and software influence how a piece of software performs, but if these are both assumed to be constant one must then consider the size of the input provided to the algorithm – an algorithm which provides pleasing performance when operating on 10 inputs, may not provide such pleasant results when tasked with millions, or even hundreds, of inputs.

With so many variables at play, the question arises as to how one can objectively measure which algorithms are better than others? In order to do so, idealised approaches which are hardware and software agnostic, are required to analyse different algorithms. The most common of these is the $\mathcal{O}(n)$ or "Big-O" analysis, sometimes referred to as Bachmann-Landau notation [7, 8]

after its inventors, where $\mathcal{O}$ implies the order-of-magnitude performance of an algorithm for an input size, n. This report will carry out such an analysis, with the aims being effectively three-fold:

1. to analyse a number of different sorting algorithms from a theoretical perspective in order to understand and model their limiting behaviours, and to thus classify in terms of time- and space-complexity ($\mathcal{O}$-notation), stability, and whether they are in-place

2. to implement these algorithms in a modern object-orientated software development language (Java) such that one can practically employ them to sort arbitrary-sized arrays of integers

3. to empirically benchmark the performance of each algorithm by measuring their time-efficiencies for arrays of varying size, and to analyse and discuss these empirical results in light of item 1 above

Each algorithm will now be conceptually and procedurally described, specific examples will be provided to illustrate their features, time- and space-complexities will be analysed, along with descriptions of other desirable featuresa alluded to above. Subsequent sections will then discuss how these algorithms were implemented in a Java application such that meaningful benchmark results could be obtained. In the final section the benchmark results obtained will be presented and discussed for each algorithm, and different algorithms will be compared and contrasted in light of the above. In terms of analysis of algorithms, their complexities, and numerous derivations and approximations presented in the text, no single authoritative source could be found, and a number of sources proved useful throughout the project including the module course notes [9] and recommended text books [10, 11], StackOverflow [12] and indeed Wikipedia [13]. Where algorithms have been adopted from the literature, their sources are specifically acknowledged in the text.

2 Sorting Algorithms

2.1 Simple comparison-based sort: BubbleSort

BubbleSort is amongst the simplest sorting algorithms that exist, and although too slow for most practical applications, its simplicity in terms of its conceptual approach, and its programmatic implementation, make it a useful entry point for understanding sorting algorithm performance. Whilst likely

in use as early as the 1950's, the first explicit usage of the term "BubbleSort" can be attributed to Iverson in 1962 [14], the algorithm getting its name from the fact the largest value "bubbles" towards the top, or end, of the array as the array is traversed and individual elements are sorted. It is an "in-place" algorithm – it requires a constant amount of auxiliary memory to perform the sort, irrespective of array size, and it is also stable in that the order of equivalent elements are preserved. As a comparison-based sorting method, it uses only logical comparison based operations ($<$, $>$) to determine whether elements are in the desired order.

The BubbleSort algorithm implemented in Java in the current project has been adapted from the course notes of Mannion [9]. The adapted source code is given in Listing 1, and a diagramatic representation is provided in Table 1.

The BubbleSort procedure relies on two loops, an outer loop and a nested inner loop, with the inner loop responsible for testing the array elements such that for indices j and $j+1$, $A[j] < A[j+1]$. The outer loop iterates through the array in descending order of elements, reducing its starting value defined by the index i by 1 unit per iteration. The outer loop therefore defines an upper bound for testing during the execution of the inner loop, which sequentially examines all pairs of elements at indices j and $j+1$, for values of j from $j = 0$ to $j < i$. Procedurally, the approach can be described as follows:

1. initialize the outer loop with index i equal to the last index in the array (the array length minus 1)
2. initialize the inner loop, with index j equal to the first index in the array (0)
3. if $A[j] > A[j+1]$, swap the two values such that $A[j] < A[j+1]$
4. increment the counter j and repeat step 3 for all indices $j < i$
5. when j is equal to $i - 1$ terminate the inner loop. At this point:
    - the largest value between index 0 and i has now been put in place at index i
6. decrement the counter i, and repeat steps $2-5$ until the exit condition for the outer loop is met ($i = 1$).

Note that with respect to Step 3 above, and line 16 in Listing 1 below BubbleSort can be deemed "stable" – that is, if two sequential elements, $A[j]$ and $A[j+1]$, have the same value they will not be swapped, and so BubbleSort preserves the original ordering of same-valued elements.

```java
public static void iterativeBubbleSort(int[] array) {

  // outer array starts at last index of array
  // and decrements while it is greater than 0
  for (int i = array.length - 1; i > 0; i--) {

    // inner loop starts at first index of array
    // and increases while it is less than i
    // it will terminate therefore when i = 1, and j = 0
    for (int j = 0; j < i; j++) {

      // if the value of array[j] > than next element in array
      // push the j value into a temp array
      // then swap element j+1 which is smaller than element j
      // with element j, which is greater
      if (array[j] > array[j + 1]) {
        int temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
    }
  }
}
```

Listing 1: Implementation of a basic Bubblesort algorithm in Java [9]. Note that there is no logic to terminate the sort procedure for an array that is already sorted.

Table 1 shows the sort procedure for a simple array of $n = 5$ elements, [5,3,6,4,1], which has a total of 7 inversions: [5,3], [5,4], [5,1], [6,4], [6,1], [3,1], and [4,1].

In terms of time complexity the outer loop indexed by i runs a total of $n - 1$ times. The inner loop indexed by j runs $n - 2$ times on the first iteration, $n-3$ times on the second iteration, etc, until it runs a total of once on the final iteration.

The total number of iterations can be computed as the sum of the iterations of the inner loop, a sequence which corresponds to a sum of natural numbers (4+3+2+1), leading to a triangular number (10), whose value can be computed analytically from the number of elements in the array as $n(n-1)/2$, or $(n^2 - n)/2$.

Within the inner loop there are a number of constant time operations – the greater than operation within the if statment, and three three subsequent value assignments for $temp$, $array[j]$ and $array[j + 1]$. Within $\mathcal{O}$ notation, these constant time operations and terms are ignored, thus leading to the well-known average-case, Θ, and worst-case, $\mathcal{O}$, time-complexities of Bubble

Sort of $\approx n^2$.

In the best-case scenario, Bubble Sort is known to have a time complexity, $\Omega = n$ for an array that is already sorted, i.e. $A = [1, 2, 3, 4, 5]$, and therefore one which has no inversions. Note that the algorithm presented in Listing 1 does not account for this and will perform $(n^2 - n)/2$ operations irrespective of whether the array is already sorted.

This can be remedied *via* the alterations presented in Listing 2, which includes specific logic to count the number of swap operations performed (these operations are constant time and so can be neglected in "Big-O" determinations). In the event that a sorted array is provided as input and no inversions are present in the array over a complete a series of iterations of the inner loop, the implementation in Listing 2 will terminate after $\approx n$ iterations over the inner loop, and one iteration over the outer loop, thus leading to the best-case performance of BubbleSort of $\Theta(n)$.

With respect to the space-complexity of BubbleSort, and ignoring the memory required to store the *IntegerSorter* instance class and its methods – memory is required to store n array elements, 2 *int* types in the for loops (i and j), and 1 temporary variables (*int temp*) to perform the swap. The memory required to store the n array elements can effectively be ignore, and only the extra, or auxiliary, memory should be considered. In such a case the extra memory required is effectively constant, and is independent of n, leading to a space-complexity of $\mathcal{O}(1)$ for the algorithm shown in Listing 1. For the algorithm show in Listing 2 more memory is required to store a single extra integer, $numberInversion$, although this is a minor burden in terms of memory requirements and $\mathcal{O}(1)$ space complexity still applies. Overall the space-complexity of BubbleSort, along with its simplicity, are amongst its only positive features – as we will see its performance is less than desirable for arrays of any meaningful size given its n^2 average and worst case time-complexity.

```java
public static void iterativeBubbleSort(int[] array) {

  // add a variable to count the number
  // of inversions carried out
  int numberInversion = 0;

  // outer array starts at last index of array
  // and decrements while it is greater than 0

  for (int i = array.length - 1; i > 0; i--) {

    // inner loop starts at first index of array
    // and increases while it is less than i
    // it will terminate therefore when i = 1, and j = 0
    // if the value of array[j] > than next element in array
    // push the j value into a temp array, then swap element j+1
    // which is smaller than element j, with element j, which is
        greater
    for (int j = 0; j < i; j++) {
      if (array[j] > array[j + 1]) {
        numberInversion++;
        int temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
    }

    // if the number of inversions
    // is zero the array is already sorted
    // and so the outer loop should break
    // this will give best case O(n)
    // running time for a pre-sorted array
    if (numberInversion == 0) {
      break;
    }
  }
}
```

Listing 2: Modification of the iterative bubble sort algorithm [9] to account for a best case scenario where no inversions are present.

iteration	***i***	***j***	**arr[0]**	**arr[1]**	**arr[2]**	**arr[3]**	**arr[4]**	***action/comment***
1	*4*	*0*	5	3	6	4	1	*compare, swap*
2	*4*	*1*	3	5	6	4	1	*compare, don't swap*
3	*4*	*2*	3	5	6	4	1	*compare, swap*
4	*4*	*3*	3	5	4	6	1	*compare, swap*
N/A			3	5	4	1	6	*one element sorted; i = i - 1; j = 0;*
5	*3*	*0*	3	5	4	1	6	*compare, don't swap*
6	*3*	*1*	3	5	4	1	6	*compare, swap*
7	*3*	*2*	3	4	5	1	6	*compare, swap*
N/A			3	4	1	5	6	*two elements sorted; i = i - 1; j = 0;*
8	*2*	*0*	3	4	1	5	6	*compare, don't swap*
9	*2*	*1*	3	4	1	5	6	*compare, swap*
N/A			3	1	4	5	6	*three elements sorted; i = i - 1; j = 0;*
10	*1*	*0*	3	1	4	5	6	*compare, swap*
exit			1	3	4	5	6	*all elements sorted; exit*

Table 1: Table illustrating the BubbleSort procedure for an array of 5 elements: [5, 3, 6, 4, 1]. Iterations, and values for the indices i and j are enumerated in the first three columns. Values for each array element at each iteration are in columns $4-8$. Cells in light grey correspond to two elements current being compared. Cells in dark grey correspond to the current index of i (the last element which will be compared), unless the element at index i is being compared, in which case it is highlighed in light grey. Elements which have been sorted are highlighted in green cells. Comments are outlined in the right most column. This diagram was generated by the author *via* an online LaTeXtable generator [15].

2.2 Efficient comparison-based sort: QuickSort

QuickSort was developed by Hoare in 1959 and published in 1961 [16]. Like BubbleSort it is a comparison-based sort which relies only on logical comparison operators ($<$, $\leq$, $>$, $\geq$, $=$) to determine whether elements are sorted, and this sorting effectively be carried out in-place as the auxiliary memory requirements are low. However it is not guaranteed to be stable and the order of equivalent elements may not be preserved pre- and post-sorting. It is a recursive divide-and-conquer algorithm which works by splitting an array into two sub-arrays based on the value chosen for a single element which defines the boundary for the partition known as the pivot. The two sub-arrays are then recursively sorted *via* the same procedure. The QuickSort algorithm implemented in this work is derived from multiple self-consistent sources [17, 18, 19], and will now be described in detail.

The QuickSort algorithm can be loosely split into three steps:

1. selection of the pivot element
2. partitioning of values about the pivot
3. recursive sorting of the two partitions *via* steps 1 and 2.

The pivot element is an element about which other elements will be sorted – all elements whose values are less than the pivot should be placed into the "left" sub-array, and all element whose values are greater than the pivot should into a seperate "right" sub-array. The pivot itself may exist in either. In terms of choosing an appropriate pivot, selection of the last, first, median or random elements are popular strategies. Whilst superficially the selection of the pivot appears trivial, many strategies exist to select a pivot element which leads to greatest efficiency – for sorted, or nearly sorted arrays, selection of the last element can lead to lower efficiencies. As the arrays being sorted in the current work are all composed of randomly generated integers, the pivot element is always chosen to be the median element, whose value should also be effectively random.

The crucial step in QuickSort is the subsequent partitioning of elements. This is shown algorithmically in lines 16–45 of Listing 3 below, and schematically in Table 2. The basic strategy is as follows. Given a pivot value, P, find an element $A[i]$ such that $A[i] \geq P$, and find an element $A[j]$ such that $A[j] \leq P$. Once the search criteria are satisfied the elements $A[i]$ are swapped if and only if $i < j$, and the search criteria is terminated if during the search i becomes greater than j. Note that there are no restrictions on the pivot element, $A[p]$, satisfying the above criteria and being used in swaps. Algorithmically the partitioning procedure can be summarised as follows:

1. provide an array to be sorted, a lower sort index (*lower_index*) and an upper sort index (*upper_index*) with the latter two defining the limits of the search operations (the first and last elements of the array on the first call)

2. if the $lower\_index \geq upper\_index$ terminate the recursion as a base recursion condition has been met

3. define two index-counting integers i and j to be equal to the *lower_index* and *upper_index* respectively

4. while the index i is less than or equal to the index j:

   (a) search for an element $A[i] \geq P$

   (b) subsequently search for an element $A[j] \leq P$

   (c) if $i \leq j$, swap the values for $A[i]$ and $A[j]$ and increment i and j

5. go to step 1 with the *upper_index* = the current value for j

6. go to step 1 with the *lower_index* = the current value for i

Note that in Listing 3 and Table 2 the partioning procedure leads to two distinct sub-arrays defined by the bounds $A_1[lower\_index...j]$ and $A_2[i...high\_index]$. However, new arrays are not created post-partioning or during recursive sort calls, rather, one is merely changing the lower and upper bounds of the indices used in the search- and sort-space. The algorithm can therefore be deemed in-place given that all swap operations take place on the original unsorted array and only a small amount ($\approx \log n$) of fixed memory is required to perform the sort.

In order for recursive calls to terminate such that infinite recursion is not encountered, the base condition (Listing 3 lines 5–7.) requests termination when the search-space, defined by the difference between the *lower_index* and *higher_index*, is one element or less.

```java
public static void quickSort(int[] array, int lower_index, int
    upper_index) {

  // base case: lower index >= higher index
  // i.e. search range < 2
  if (lower_index >= upper_index) {
    return;
  }

  // get the start and end indices and set the pivot value
  // based on the median element in the array
  int i = lower_index;
  int j = upper_index;
  int pivot = array[(lower_index + upper_index) / 2];

  // iterate while the index i <= index j
  while (i <= j) {

    // find an element in left side
    // whose value is greater than or equal to the pivot
    // if no larger values will stop when
    // pivot reached
    while (array[i] < pivot) {
      i++;
    }

    // find an element in right hand side
    // whose value is less than or equal to the pivot
    // if no smaller values will stop when
    // pivot reached
    while (array[j] > pivot) {
      j--;
    }

    // if the index i
    // is less than or equal to
    // index j swap the two elements
    // and increment i and decrement j
    if (i <= j) {
      int temp = array[i];
      array[i] = array[j];
      array[j] = temp;
      i++;
      j--;
    }
  }

  // recursively call quick sort on
```

```
48      // array[lower_index...j]
49      // and array[i...upper_index]
50      IntegerSorter.quickSort(array, lower_index, j);
51      IntegerSorter.quickSort(array, i, upper_index);
52
53  }
```

Listing 3: QuickSort algorithm [17, 18, 19] implemented in the current project.

comment	*arr[0]*	*arr[1]*	*arr[2]*	*arr[3]*	*arr[4]*	*arr[5]*	*arr[6]*	*arr[7]*	*arr[8]*
elements	2	1	9	5	**7**	8	4	6	10
index labels	***i***				***p***				***j***
find arr[i] >= pivot	2	1	9	5	**7**	8	4	6	10
			i						***j***
find arr[j] <= pivot	2	1	9	5	**7**	8	4	6	10
			i					***j***	
i <= j therefore swap and increment i and j	2	1	6	5	**7**	8	4	9	10
				i			***j***		
find arr[i] >= pivot	2	1	6	5	**7**	8	4	9	10
					i		***j***		
find arr[j] <= pivot	2	1	6	5	**7**	8	4	9	10
					i		***j***		
i <= j therefore swap and increment i and j	2	1	6	5	4	8	**7**	9	10
						i,j			
find arr[i] >= pivot	2	1	6	5	4	8	**7**	9	10
						i,j			
find arr[j] <= pivot	2	1	6	5	4	8	**7**	9	10
					j	***i***			
i >j don't swap, partitioning complete, recurse	2	1	**6**	5	4	8	**7**	9	10
index labels	***low***		***p***		***j***	***i***	***p***		***high***

Table 2: Tabular illustration of the QuickSort partition procedure. Dark grey cells with bold text correspond to the pivot. Light grey cells track the indices i and j. Green cells denote swapped elements. Cyan and purple cells correspond to the partioned sub-arrays with new pivots highlighted in bold. This diagram was generated by the author *via* an online LaTeXtable generator [15].

In terms of the time-complexity for a simple two-way-partition QuickSort algorithm, one can expect $\Theta(n \log n)$ and $\Omega(n \log n)$ performance in the average- and best-cases respectively, and $\mathcal{O}(n^2)$ performance in the worst case. Before analysing these scenarios we should first consider the time complexity of the partitioning step.

In order to partition each element on either side of the pivot, all n elements have to be inspected either in the first or second while loop, or *via* a combination of both. The partitioning procedure is therefore an $\Theta(n)$, $\Omega(n)$, and $\mathcal{O}(n)$ operation in the average-, best- and worst-cases. Note that the time required for constant time swap operations is ignored in all cases.

The differentiation between the $n \log n$ average and best case performances, and the n^2 performance in the worst case stems is related to the "balancing" of the sub-lists formed by the initial partioning step, and the subsequent number of recursive calls required to sort these.

Worst-case performance occurs in the case of a completely unbalanced pair of sub-arrays, where one of the sub-lists is effectively empty and the other contains $\approx n$ elements, as might happen when the pivot is chosen as the smallest or largest element in the array. If all subsequent recursive calls partition the sub-arrays in the same way, $n - 1$ nested recursive calls are required to sort the array, effectively one element at a time. Thus the recursion executes effectively n times with the partition step carrying out n operations each time, and the n^2 worst-case time-complexity reminiscent of BubbleSort is obtained.

A specific example of a worst-case scenario is the array $A = [14, 12, 45, 3, 20, 52]$, where if on the first iteration 52 is chosen as the pivot, on the second iteration 45 is chosen, etc. This situation is unlikely to happen for arrays of random elements where the median is chosen as the pivot, as is the case here. Another case where worst-case performance can be observed is when the first or last element is always taken as the pivot for pre-sorted ($A = [2, 3, 4, 5, 6]$) or reverse sorted ($A = [6, 5, 4, 3, 2]$) arrays, or if one is sorting arrays where all elements are equal ($A = [5, 5, 5, 5, 5]$).

The best case performance occurs in the case of a perfectly balanced series of sub-arrays containing $n/2$ elements after each recursion. In such a case one can only make $\log_2(n)$ recursive calls before the original array is split into individual lists of size 1. Hence $\log_2(n)$ recursive calls to QuickSort would be made, with n operations carried out during each partition step, and the $\Omega(n \log n)$ best-case performance of QuickSort is thus rationalised.

The formal analysis of average case running times is perhaps more complex, and although general methods do exist [20], it is beyond the scope of the current work. What is critical to note is the deleterious worst-case performance of QuickSort under the conditions described above, and that

for arrays where elements are randomly or near randomly distributed, the $\Theta(n \log n)$ and $\Omega(n \log n)$ time-complexity of QuickSort make it a much more powerful tool than simple comparison-based sorts such as BubbleSort and InsertionSort, albeit with a slight cost in terms of space-complexity – $\log n$ space-complexity is observed as a result of the constant amount of fixed extra memory required to store auxiliary variables for each recursive call.

2.3 Non-comparison sort: CountingSort

Unlike the previously described BubbleSort and QuickSort algorithms which are comparison-based sorting methods, CountingSort is a non-comparison-based approach which uses *keys* to group and then sort elements. Its invention is attributed to Seward in 1954 [21].

The basic principle of CountingSort, and other non-comparison based methods such as BucketSort, is to first group elements together based on their unique values, termed the *key*, and to then iteratively count the number of elements with that *key* value. The sort can then be performed assuming that the sort order of the *keys* is known *a priori*, and knowing the number of times, or the frequency with which, each *key* is present in the original array.

The basic procedure for implementing CountingSort for an array of integers is as follows, with the specific algorithm [22, 23] implemented in the current project given in Listing 4:

1. find the minimum and maximum values in the array to determine the *range* of values
2. create a new array with $n = range$ elements (one for each *potential key*)
3. iterate over the array and count the number of occurrences (n_{key}) of each *key*, incrementing the appropriate *key* counter accordingly
4. for each *key* count the cumulative sum of all preceding *keys* – this determines the starting index of each *key* in the sorted array
5. insert each ordered *key* back into the original array n_{key} times

Steps $1 - 4$ above are schematically represented in Table 3, with step 5 represented in Table 4. Note that neither Table 3 nor Table 4 illustrate a subtlety in the algorithm whereby each *key* is offset by the minimum *key* value, line 24 of Listing 4, to allow the *count* array, which effectively acts as a bucket array, to be indexed from 0 to n where n is the difference between the maximum and minimum values of the array elements.

```java
public static void countingSort(int[] array) {

  // find the min and max values in the array
  int min = array[0], max = array[0];
  for (int i = 0; i < array.length; i++) {
    if (array[i] < min) {
      min = array[i];
    }
    if (array[i] > max) {
      max = array[i];
    }
  }

  // set the range and create a
  // new integer array with n = range elements
  int range = max - min + 1;
  int[] count = new int[range];

  // count the number of occurences of each
  // key, correcting the index for the min
  // key so indices start at 0
  for (int i = 0; i < array.length; i++) {
    int key = array[i];
    count[key - min]++;
  }

  // count the cumulative sum of elements
  // preceding this element/key - defines
  // the starting index for each value
  for (int i = 1; i < range; i++) {
    count[i] += count[i - 1];
  }

  // populate the original array with each number accounting for
      its frequency
  int j = 0;
  for (int i = 0; i < range; i++) {
    while (j < count[i]) {
      array[j++] = i + min;
    }
  }
}
```

Listing 4: CountingSort algorithm implemented in the current project [22, 23].

	arr[0]	*arr[1]*	*arr[2]*	*arr[3]*	*arr[4]*	*arr[5]*	*arr[6]*	*arr[7]*	*arr[8]*	*arr[9]*	
elements	3	2	4	1	2	4	5	3	1	0	
find the min and max values											
iteration	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	
elements	3	2	4	1	2	4	5	3	1	0	
min or max	***min, max***	***min***	***max***	***min***			***max***			***min***	
count key occurrences											
iteration	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	
elements	***3***	***2***	***4***	***1***	***2***	***4***	***5***	***3***	***1***	***0***	***Sum***
count[0]	0	0	0	0	0	0	0	0	0	***1***	***0***
count[1]	0	0	0	***1***	***1***	***1***	***1***	***1***	**2**	***2***	***1***
count[2]	0	***1***	***1***	***1***	***2***	***2***	***2***	***2***	***2***	***2***	***2***
count[3]	***1***	***1***	***1***	***1***	***1***	***1***	***1***	***2***	***2***	***2***	***5***
count[4]	0	0	***1***	***1***	***1***	***2***	***2***	***2***	***2***	***2***	***7***
count[5]	0	0	0	0	0	0	***1***	***1***	***1***	***1***	***9***

Table 3: Tabular illustration of the minimum and maximum key finding and key counting approaches in QuickSort. Light grey cells correspond to the total counts, or frequencies, for each key upon summation, and dark grey cells are the cumulative sum of the frequencies of all preceding keys. This diagram was generated by the author *via* an online LaTeXtable generator [15].

In terms of time-complexity, CountingSort is relatively straight forward to understand given that it contains only simple loop structures and there are no recursive calls, unlike QuickSort. The initial determination of the minimum and maximum array values (Listing 4, lines $4-12$) requires inspection of every element in the array and is an $\mathcal{O}(n)$, $\Theta(n)$, $\Omega(n)$ operation, as is the determination of the frequency of each key value (Listing 4, lines $22-25$).

However, unlike previous sorting algorithms whose time-complexity was solely dependent on the number of array elements, n, CountingSort is also dependent on the range of the elements' values in the array being sorted.

For example, an array containing 100 elements in the range $1-5$ requires only 5 unique buckets to index and tally frequencies for the set of elements {1,2,3,4,5}. Conversely an array of only 5 elements whose values are in the range $1-100$, e.g. $A = [100, 50, 75, 1, 25]$, will require an array of 100 buckets to store all possible values in this range. The two for loops (lines $30-32$, $36-40$ of (Listing 4) which iterate over this range of values must therefore do so k times. This added complication gives CountingSort its characteristic worst-, average- and best-case performances of $\mathcal{O}(n+k)$, $\Theta(n+k)$, $\Omega(n+k)$ where n is the number of elements and k is the number of natural numbers between the minimum and maximum element values.

Similarly, $\mathcal{O}(n+k)$, $\Theta(n+k)$, $\Omega(n+k)$ space-complexity are observed if one includes the requirements to store the n array elements, and the k values present in the range. In this case, no extra space requirements are required to store the n array elements to be sorted, and the space-complexity can be simplified to $\mathcal{O}(k)$, $\Theta(k)$, $\Omega(k)$ if one only counts the auxiliary space required to store the frequency of each integer in the range (and ignoring the trivial amounts of space required to store various int types used to define the min, max etc.)

Finally, both in-place and out-of-place versions of the algorithm exist. The version implemented herein can be considered in-place given that the original array is used to store the final sorted array elements. However, a constant amount of memory is required to store the k elements defined by the range, and one should not neglect this added storage requirements for large values of k.

Similarly with respect to stability, both stable and unstable versions of CountingSort exist, and depending on the version implemented or the types being sorted, stability is not guaranteed. For the integer based sorts presented herein, stability is not necessarily as important a property as overall sorting efficiency as the order of equivalent integers is of little concern. For sorting of more complex objects the use of satellite data may be required to enforce stability.

		loop over $i \rightarrow$						
	key	***0***	***1***	***2***	***3***	***4***	***5***	
	count[key]	***1***	***2***	***2***	***2***	***2***	***1***	***arr[j]***
	0	0	$\rightarrow$					***0***
	1		1	$\rightarrow$				***1***
	2		1	$\rightarrow$				***1***
	3			2	$\rightarrow$			***2***
	4			2	$\rightarrow$			***2***
loop over j $\downarrow$	***5***				3	$\rightarrow$		***3***
	6				3	$\rightarrow$		***3***
	7					4	$\rightarrow$	***4***
	8					4	$\rightarrow$	***4***
	9						5 $\rightarrow$	***5***

Table 4: Tabular illustration of the nested *for* (over i) and *while* (over j) loops in lines $36-40$ of the CountingSort algorithm shown in Listing 4, for the array shown in Figure 3. The order of insertion runs over i then j as sorted elements are inserted into the final array (represented in grey cells). This diagram was generated by the author *via* an online LaTeXtable generator [15].

2.4 Chosen sort 1: InsertionSort

InsertionSort, like BubbleSort, is a simple comparison-based sorting algorithm, which is both stable and in-place. Attributing a source to its development is tricky – the algorithm itself is intuitive, likely pre-dating computers, and it closely resembles the manner in which one would sort a hand of playing cards – a single card (the *key*) is chosen, the array of cards is then scanned from left to right, and the *key* card is *inserted* after *all* cards which are smaller or equal to it, and before the $first$ card that is greater than it, and this procedure is repeated for all cards. Algorithmically this process can be implemented *via* two nested loops as shown in Listing 5 (adapted from [11]) – an outer for loop which iterates over elements from left to right so that their overall insertion position can be determined, and an inner *while* loop which iterates from right to left, pushing elements to the right if they are greater than the *key* so that the *key* can be inserted in the correct place. The basic procedure can be summmarised as follows, where the index i is initialized as $i = 1$:

1. set the value of the *key* to the element $A[i]$ with index i
2. set the value for j equal to $i - 1$
3. while $j > 0$ and $A[j] > key$
   (a) swap element $A[j + 1]$ with element $A[j]$ and decrement j
4. set the element $A[j]$ equal to the *key* value, and increment i

This sorting procedure is represented schematically in Table 5 for the array $A = [3, 2, 9, 5, 7, 6, 4, 8]$ which has 8 elements and 10 inversions: [3,2], [9,5], [9,7], [9,6], [9,4], [9,8], [5,4], [7,6], [7,4] and [6,4]. Note that the outer loop over index i (Listing 5, lines $5 - 22$) execues $n - 1$ times. Disregarding the cost of the inner loop momentarily by assuming it is never executed, InsertionSort will have $\Omega(n)$ time-complexity in the best-case. A specific example is the pre-sorted array $A = [1, 2, 3, 4, 5]$ which will never satisfy the condition that $A[j] > key$.

In all other cases the cost of the inner *while* loop over index j must be considered. For any given iteration over i, the inner loop will execute when $A[j] > key$. This condition will be satisfied for every inversion present in the list, which can be denoted k. For the specific array of 8 elements presented in Table 5, a total of 10 inversions are present and $k = 10$. One could then approximate the time-complexity of InsertionSort as the number of iterations over the outer loop, n, plus the number of iterations over the inner loop, k and the total number of operations is therefore some function $\propto (n + k)$.

```java
public static void insertionSort(int[] array) {

  // start looping at second element of the array
  // and increment index until we get to the last element
  for (int i = 1; i < array.length; i++) {

    // assign the key as the current element
    // and the j counter as the previous element
    int key = array[i];
    int j = i - 1;

    // while the counter j is greater than or equal to 0,
    // and element[j] is greater than the key
    // push element[j] to the right and decrement j
    while (j >= 0 && array[j] > key) {
      array[j + 1] = array[j];
      j = j - 1;
    }

    // insert the key at its new index
    array[j + 1] = key;
  }
}
```

Listing 5: InsertionSort algorithm adadapted from [11] and implemented in the current project.

The formula for the maximum number of possible inversions in an array of arbitrary dimension will not be derived here, but is given by $k = n(n+1)/2$, or $k = (n^2 + n)/2$. From this relation the worst case time-complexity, $\mathcal{O}$, of InsertionSort can be estimated as $\mathcal{O}(n + k)$ which upon substituting for k leads to $\mathcal{O}(n+(n^2+n)/2)$. The constant terms (n and 2) can be omitted when approximating the asymoptotic behaviour and the well-known $\mathcal{O}(n^2)$ time-complexity for InsertionSort results. In the average-case the time-complexity stems from an estimate of the average number of inversions in an array of randomly distributed elements, and is frequently given as half the number of inversions in the worst-case scenario: $k = (n^2 + n)/4$. Ignoring constant terms leads to the $\Theta(n^2)$ average-case time-complexity of InsertionSort.

In terms of space-complexity, InsertionSort has performance equivalent to BubbleSort. Discounting memory costs associated with storing the array being sorted (which is carried out in-place), a constant amount of memory is required to store the *key* value and the counting index j, and $\Omega(1)$ space-complexity results, thus implying low memory overheads are required for this algorithm.

Table 5: Tabular illustration of the InsertionSort algorithm shown in Listing 5. Grey cells indicate the current *key* value ($arr[i]$), light cyan cells track the element with index j, dark cyan cells indicate the result of pushing an element to the right, and green cells indicate elements which have been sorted as part of a previous iteration over index i. This diagram was generated by the author *via* an online LaTeXtable generator [15].

comment	*i*	*j*	*arr[0]*	*arr[1]*	*arr[2]*	*arr[3]*	*arr[4]*	*arr[5]*	*arr[6]*	*arr[7]*
key=arr[i]; j=i-1	*1*	*0*	3	2	9	5	7	6	4	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*1*	*0*	3→	3	9	5	7	6	4	8
j<0; arr[j+1]=key; i++	*1*	*-1*	2	3	9	5	7	6	4	8
key=arr[i]; j=i-1	*2*	*1*	2	3	9	5	7	6	4	8
j>=0; arr[j]<key; i++	*2*	*1*	2	3	9	5	7	6	4	8
key=arr[i]; j=i-1	*3*	*2*	2	3	9	5	7	6	4	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*3*	*2*	2	3	9→	9	7	6	4	8
j >= 0; arr[j] <key; arr[j+1] = key; i++	*3*	*1*	2	3	5	9	7	6	4	8
key=arr[i]; j=i-1	*4*	*3*	2	3	5	9	7	6	4	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*4*	*3*	2	3	5	9→	9	6	4	8
j >= 0; arr[j] <key; arr[j+1] = key; i++	*4*	*2*	2	3	5	7	9	6	4	8
key=arr[i]; j=i-1	*5*	*4*	2	3	5	7	9	6	4	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*5*	*4*	2	3	5	7	9→	9	4	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*5*	*3*	2	3	5	7→	7	9	4	8

Table 5: Tabular illustration of the InsertionSort algorithm shown in Listing 5. Grey cells indicate the current *key* value ($arr[i]$), light cyan cells track the element with index j, dark cyan cells indicate the result of pushing an element to the right, and green cells indicate elements which have been sorted as part of a previous iteration over index i. This diagram was generated by the author *via* an online LaTeXtable generator [15].

comment	*i*	*j*	*arr[0]*	*arr[1]*	*arr[2]*	*arr[3]*	*arr[4]*	*arr[5]*	*arr[6]*	*arr[7]*
j >= 0; arr[j] <key; arr[j+1] = key; i++	*5*	*2*	2	3	5	6	7	9	4	8
key=arr[i]; j=i-1	*6*	*5*	2	3	5	6	7	9	4	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*6*	*5*	2	3	5	6	7	9→	9	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*6*	*4*	2	3	5	6	7→	7	9	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*6*	*3*	2	3	5	6→	6	7	9	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1	*6*	*2*	2	3	5→	5	6	7	9	8
j >= 0; arr[j] <key; arr[j+1] = key; i++	*6*	*1*	2	3	4	5	6	7	9	8
key=arr[i]; j=i-1	*7*	*6*	2	3	4	5	6	7	9	8
j>=0; arr[j]>key; arr[j+1] = arr[j]; j=j-1;	*7*	*6*	2	3	4	5	6	7	9→	9
j >= 0; arr[j] <key; arr[j+1] = key; i++	*7*	*6*	2	3	4	5	6	7	8	9
i = array.length → exit; sorted			2	3	4	5	6	7	8	9

2.5 Chosen sort 2: MergeSort

The final algorithm to be discussed is the comparison-based MergeSort algorithm. Like QuickSort it employs a divide-and-conquer approach which relies on recursion to efficiently sort large arrays at the expense of an increase in space-complexity. Whilst still in popular use, detailed descriptions from Goldstine and von Neumann were published not long after World War II [24], and it may well pre-date less efficient algorithms such as BubbleSort in this regard. In terms of stability it may or not preserve the order of equivalent elements in the final sorted array depending on the specific implementation. It it is inherently an out-of-place approach, requiring potentially large amounts of auxiliary memory depending on the size of the list being sorted, and thus the recursion depth. Whilst multiple flavours of MergeSort exist, including top-down and bottom-up approaches, only the top-down approach is considered here [25], with the source code detailed in Listing 6.

Procedurally the recursive procedure for a top-down MergeSort is:

1. provide an array to the MergeSort method
2. if the array contains less than two elements a base condition has been met and a return statement is executed, otherwise
3. create two sub-arrays to store the $left$ and $right$ hand sides of the input array which is to be split at the median element
4. iterate over the input array assigning all elements with index less than the median index to the $left$ array, and all other elements to the $right$ array
5. recursively call MergeSort on the $left$ array
6. recursively call MergeSort on the $right$ array
7. merge the $left$ and $right$ sub-arrays, adding elements in sorted order

A couple of points are noteworthy with respect to the above procedure. Steps 3 and 4 which outline the creation of two sub-arrays to store both halves of the input array indicate that MergeSort is out-of-place, with auxiliary memory required to store these new arrays during each recursive call. The merge operation outlined in lines $34 - 58$ of Listing 6 is frequently written as a seperate method, but has been included directly in the MergeSort method in this case. We will now illustrate the procedure more clearly with specific examples, as shown in Figures 1 and 2.

```
1  public static void mergeSort(int[] array) {
2
3    // base case: arrays of length 0 or 1 are sorted
4    if (array.length < 2) {
5      return;
6    }
7
8    // calculate the median index
9    // and populate two new sub arrays
10   int mid = array.length / 2;
11   int[] left = new int[mid];
12   int[] right = new int[array.length - mid];
13   for (int i = 0; i < array.length; i++) {
14     if (i < mid) {
15       left[i] = array[i];
16     } else {
17       right[i - mid] = array[i];
18     }
19   }
20
21   // recursively sort the sub arrays
22   IntegerSorter.mergeSort(left);
23   IntegerSorter.mergeSort(right);
24
25   // indexes for iterated over left, right and merged array
26   int i = 0;
27   int l = 0;
28   int r = 0;
29
30   // while the l and r indices are less
31   // than their respective array lengths
32   // find the smaller element of the two and insert it into
33   // the final array in the correct position
34   while (l < left.length && r < right.length) {
35     if (left[l] < right[r]) {
36       array[i] = left[l];
37       i = i + 1;
38       l = l + 1;
39     } else {
40       array[i] = right[r];
41       i = i + 1;
42       r = r + 1;
43     }
44   }
45
46   // now add the remaining ``left'' elements
47   while (l < left.length) {
48     array[i] = left[l];
```

```
49      i = i + 1;
50      l = l + 1;
51    }
52
53    // now add the remaining ``right'' elements
54    while (r < right.length) {
55      array[i] = right[r];
56      i = i + 1;
57      r = r + 1;
58    }
59  }
```

Listing 6: MergeSort algorithm adadapted from [25] and implemented in the current project.

Figure 1 shows the recursive sequences of calls made by the MergeSort algorithm when provided with the input array $A = [2, 3, 9, 5, 7, 6, 4, 8]$. A number of important features emerge.

The first is that unlike QuickSort, Table 2, MergeSort does not rely on the selection of a pivot element and subsequent array iteration to partition the data into two distinct sub-arrays. Rather, arrays are split on the median element into evenly-balanced binary (left/right, red/black) trees. MergeSort will therefore not suffer from the problems which arise with QuickSort related to optimal pivot selection – recall for the latter that in the worst-case scenario n^2 performance is observed when the largest array element is consistently chosen as the pivot. MergeSort therefore has the benefit of requiring a fixed (deterministic) number of recursive calls to split an array of n elements into n arrays of 1 elements, which will be reflected in its worst-, average-, and best-case running times, *vide infra*.

Secondly one should note the order, or sequence, of recursive calls. Whilst sub-arrays 1 and 8 appear to be called sequentially in Listing 6, the left-most tree is explored entirely until the base recursive condition is met, that is, when an array containing only one element is obtained.

Figure 2 in turns shows the subsequent merging of arrays into a final sorted array. The merge operation occurs after the second recursive call to the MergeSort algorithm, lines $34 - 58$ of Listing 6. When merging arrays the left and right arrays are traversed at the same time and the element at index 1 in the first array is compared with the element at the same index in the second array. The lesser of the two elements is put into the sorted array first, followed by the other, and this process is repeated for all shared indices.

In cases where there are two sub-arrays of unequal size, the elements in the larger array are added in sequence once the two sub-arrays have been compared, as shown in lines $47 - 58$ of Listing 6. Note therefore that one

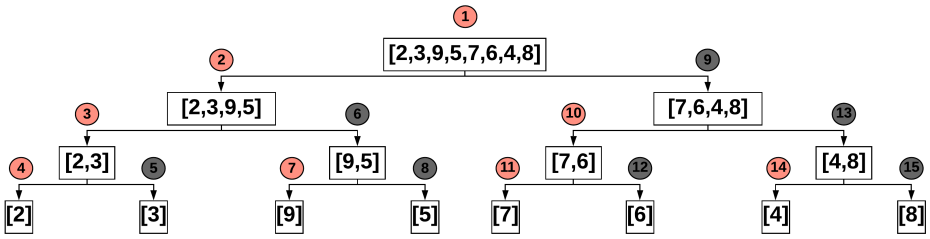

Figure 1: Schematic of the recursive sequence of calls made by the MergeSort algorithm to create sub-arrays when provided with the input array $A = [2, 3, 9, 5, 7, 6, 4, 8]$. Each arrow denotes a recursive call with the exact sequence of calls denoted by the number in a each circle, and the elements of each recursive sub-array shown between square brackets. Red- and grey-shaded circles correspond to first and second recursive calls in the MergeSort algorithm shown in Listing 6. This diagram was generated by the author using Lucidchart [26].

would expect a subtely different set of diagrams depending on the number of elements in the original array, and thus each sub-array, but the algorithmic and logical approaches are entirely self-similar and the algorithm shown in Listing 6 is applicable to arrays of any number of elements – heap space allowing.

In terms of time-complexity MergeSort is equivalent to QuickSort in the average- and best-case scenarios, with both displaying $\Theta(n \log n)$ and $\Omega(n \log n)$ performance. The initialiazation of the two sub-arrays from the n elements in the base array is an $\Theta(n)$, $\Omega(n)$ and $\mathcal{O}(n)$ operation as all elements must be individually accessed during the loop. Similarly, the merge operations, shown as three individual *while* loops in lines $34 - 58$ of Listing 6 all have $\Theta(n)$, $\Omega(n)$ and $\mathcal{O}(n)$ time-complexity.

The $\log n$ component of MergeSort's time-complexity stems from the recursive calls required to split the array of n elements into n arrays of 1 element, which requires $\approx \log(n)$ recursive calls. For each of the $\log(n)$ recursive calls, the $\mathcal{O}(n)$ operations described previously must be carried out and ignoring constant terms the asymptotic $\Theta(n \log n)$, $\Omega(n \log n)$ and $\mathcal{O}(n)$ behaviour is thus rationalised as the product of these two terms.

As noted previously, unlike QuickSort, MergeSort does not rely on the

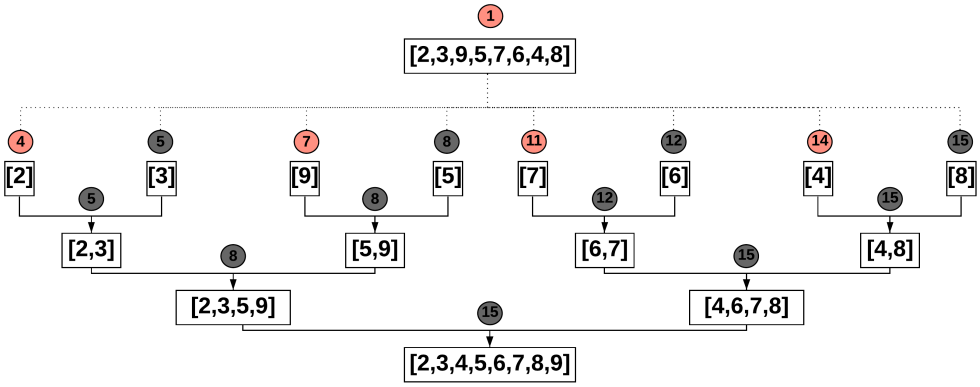

Figure 2: Schematic of the sequence of merge operations made by the MergeSort algorithm when provided with the input array $A = [2, 3, 9, 5, 7, 6, 4, 8]$. The corresponding recursive creation of sub-arrays is shown in Figure 1 along with a description of each symbol. Each arrow indicates the merging of two-subarrays into a sorted array, as shown algorithmically lines $34-58$ of Listing 6. This diagram was generated by the author using Lucidchart [26].

selection of a pivot element about which the partitioning of sub-arrays is dependent and the poor worst-case $\mathcal{O}(n^2)$ performance of QuickSort is avoided entirely with MergeSort as a result of the consistently well-balanced left and right trees. However, this may not be reflected in overall performance, as will be shown and discussed in subsequent benchmarking studies. One reason for this is MergeSort's added requirements in terms of space. During each recursive call the n array elements are first partitioned into and stored in two newly created arrays, requiring auxiliary storage which leads to $\mathcal{O}(n)$ space-complexity, versus $\approx \log n$ space-complexity in the case of QuickSort which can be implemented entirely in-place without large auxiliary memory.

3 Implementation and Benchmarking

3.1 Java Application Overview

The specific implementation of each algorithm being benchmarked has been described in the previous section and will not be elaborated upon here. This section will give a broader perspective on the main features of the Java application used to benchmark these algorithms. The Java application was developed with object-orientated programming principles in mind, that is, abstraction, encapsulation, cohesion, coupling, and composition. In this regard

there are several packages in the final JAVA project – *runner*, *sorters*, *random*, and *printers* which, in tandem, produce meaningful algorithm benchmarks and output which is useful for the user. Note that unlike the previous code Listings which provided complete details of each algorithm, the code Listings below are condensed in places, in some instances code has been removed for brevity, and the main aim is to give an indicative overview the various classes, methods and procedures implemented in the current project. Comprehensive detail of all packages, classes and methods are provided in the Java source code, and the accompanying JavaDocs, which have been compiled and included in the *doc* directory of the project source code.

The package central to *this* project is the *sorters* package, which contains two classes. The first is *sorters.AbstractSorter*, a minimalist abstract base class defining the methods expected of a concrete *sorter* class – a *setSortMethod()*, a *getSortMethod()*, and crucially, a *sort()* method. The first and last methods give a good indication of the procedures used herein – to sort an array a "sorter" type must be used, the "setSortMethod" must be called to define a specific algorithm to be used, and finally, the "sort" function should be called. The concrete *sorters.IntegerSorter* class extends the *sorters.AbstractSorter* abstract base class by implementing the aforementioned methods, and each sorting method implemented in the current project is present as an individual method in the *sorters.IntegerSorter* class.

Code Listing 7 below provides a condensed overview of the *IntegerSorter* class and its methods. The sort methods to be used can be set via the *IntegerSorter.setSortMethod(String sortMethod)* method, where a single lowercase *String* denotes the sort method to be used – the options are:

- *bubble* for BubbleSort
- *quick* for QuickSort
- *counting* for CountingSort
- *insertion* for InsertionSort
- *merge* for MergeSort
- *selection* for SelectionSort

Note that although SelectionSort was not described previously, it has been implemented [27] in Java and benchmarked as a result of the Author's curiosity with little extra burden in terms of application design or configuration.

```java
public class IntegerSorter extends AbstractSorter<int[]> {

  private String sortMethod;

  public IntegerSorter() {}

  public String getSortMethod() {}

  public void setSortMethod(String sortMethod) {}

  @Override
  public long sort(int[] input) {

    sortMethod = sortMethod.toLowerCase();
    long runTime = -1;

    if (sortMethod.equals("bubble")) {
      // start the timer
      long startTime = System.nanoTime();

      // run the sort
      IntegerSorter.iterativeBubbleSort(input);

      // compute the CPU time in nanoseconds
      runTime = System.nanoTime() - startTime;
    }
    else if (sortMethod.equals("quick")) {}
    else if (sortMethod.equals("counting")) {}
    else if (sortMethod.equals("merge")) {}
    else if (sortMethod.equals("insertion")) {}
    else if (sortMethod.equals("selection")) {}
    else {}

    // return the run time in nanoseconds
    return runTime;
  }

  public static void iterativeBubbleSort(int[] array) {}
  public static void insertionSort(int[] array) {}
  public static void selectionSort(int[] array) {}
  public static void countingSort(int[] array) {}
  public static void mergeSort(int[] array) {}
  public static void quickSort(int[] array, int low, int high)
      {}
}
```

Listing 7: An overview of the *IntegerSorter* class and its methods.

Once a sorting method has been chosen, arrays of integers can be sorted

via the *IntegerSorter.sort(int[] input)* method, which will delegate the responsibility for sorting to a specific sort method. The *IntegerSorter.sort(int[] input)* ultimately returns a *long* type containing the benchmark CPU time in nanoseconds.

Importantly with respect to all benchmark CPU times computed, the benchmark time includes only the call to a specific sort method, and a subsequent call to *System.nanoTime()* – no other methods (i.e. array generation), which might conflate or artificially inflate the CPU times, are called between the call to the sort method and the timer call. The computed benchmark CPU time are therefore wholly representative of the effiency of a given algorithm (as implemented), and these efficiencies are comparable between iterations of a single algorithm, and between different algorithms.

In order to run benchmarks and store benchmark times for a given algorithm, array size, and iteration, an *AlgorithmBenchmarker* class has been developed and included in the *runner* package. An *AlgorithmBenchmarker* instance has responsibility for running benchmarks, and storing relevant information for a single algorithm and array size (i.e. a single "cell" in the output tables). However, it also stores information for *multiple* statistical runs, thus allowing for summary statistics to be readily computed, stored and retrieved. The *AlgorithmBenchmarker* class contains ≈ 20 methods in total, including *Bean*-methods (getter/setters), not all of which will be described here. The most useful and pertinent classes relate to:

- benchmarking an algorithm n times *via*:
  - *runIntegerBenchmark()*
- computing statistical properties *via*:
  - *calculateMeanRunTime()*
  - *calculateStandardDeviation()*
  - *sampleStandardDeviation()*
  - *populationStandardDeviation()*)
- retrieving best, average and worse case run times *via*:
  - *getMeanRunTime()*
  - *getBestRunTime()*
  - *getWorstRunTime()*
  - *getProperty()* (which retrieves info. from a data dictionary)

Perhaps the most important method is the *AlgorithmBenchmarker* class is the *runIntegerBenchmark()* method, as outlined in Listing 8 below which provides an overview of some, but not all, of the source code. This method takes as input a) any type which extends an *AbstractSorter* type, i.e. an *IntegerSorter* b) a single integer defining the size of the array to be benchmarked, and c) a single integer defining the number of iterations to run. A loop is then initialized to run n times, once for each iteration requested, and during each loop a new array of random integers of a specific size is generated via a call to the *random.ArrayFactory.generateIntegers()* method, which will generate an array of random positive integers. An important point is that for every algorithm, array size, and iteration that is benchmarked, a new, unsorted, array of random integers is generated.

```
1  public void runIntegerBenchmark(
2                AbstractSorter<int[]> sorter,
3                int arraySize, int numberOfIterations
4                ) {
5
6    // set the array size and samples per array size
7    this.arraySize = arraySize;
8    this.numberOfIterations = numberOfIterations;
9
10   // benchmark the sorting algorithm numberOfIterations times
11   for (int i = 0; i < this.numberOfIterations; i++) {
12
13     // generate an array of random numbers of
14     // a given size before each sorting operation
15     int[] array = ArrayFactory.generateIntegers(arraySize);
16
17     // run the sort method and store the runTime
18     // in the AlgorithmBenchmarker.iterationData ArrayList
19     long runTime = sorter.sort(array);
20     this.iterationData.add(runTime);
21   }
22 }
```

Listing 8: An overview of the *runIntegerBenchmark()* method in the *AlgorithmBenchmarker* class.

The *IntegerSorter* class coupled with the *AlgorithmBenchmarker* class therefore allow benchmarks to be carried out for multiple sort algorithms, multiple array sizes, and for multiple iterations in a reasonably straight forward manner *via* the *Runner.runIntegerSortBenchmarks()* method. Listing 9 below provides an overview of the general approach.

```
1  private static ArrayList<AlgorithmBenchmarker>
       runIntegerSortBenchmarks(
2      int[] arraySizes,
3      int samplesPerArraySize,
4      String[] algorithms) {
5
6    // create a single integer sorter instance
7    // to be used to call various sort methods
8    IntegerSorter integerSorter = new IntegerSorter();
9
10   // a list to store all the AlgorithmBenchmarker instances
11   // which will run benchmarks and store results
12   ArrayList<AlgorithmBenchmarker> algorithmBenchmarkers = new
         ArrayList<AlgorithmBenchmarker>();
13
14   // for each algorithm ...
15   for (String algorithm : algorithms) {
16
17     // set the sort method
18     integerSorter.setSortMethod(algorithm);
19
20     // for each array size ... i.e. 100 elements, 200 elements
21     for (int arraySize : arraySizes) {
22
23       // create a new AlgorithmBenchmarker instance
24       // to run the benchmarks
25       AlgorithmBenchmarker benchmarker = new
             AlgorithmBenchmarker(algorithm);
26       algorithmBenchmarkers.add(benchmarker);
27
28       // run the benchmark by providing the
29       // integerSorter instance, the array size
30       // to be benchmarked and the number of
31       // times to run the benchmark
32       benchmarker.runIntegerBenchmark(integerSorter, arraySize,
             samplesPerArraySize);
33     }
34   }
35
36   // return the list of algorithm benchmarkers
37   return algorithmBenchmarkers;
38 }
```

Listing 9: An overview of the *runIntegerSortBenchmarks()* method in the *Runner* class.

The *runIntegerSortBenchmarks()* takes as input an array of integers (*arraySizes*) specifying the array sizes to be benchmarked (i.e. 100 elements, 200 elements, etc), an integer defining the number of samples to be run for each array size (*samplesPerArraySize*), and a *String* array containing the names of the various sorting methods to benchmark (*algorithms*).

For each algorithm, and for each array size the benchmark is then run n times with the results stored in a specific *AlgorithmBenchmarker* instance. Each instance of the latter is added to an *ArrayList* of *AlgorithmBenchmarker* instances, each of which can be later retrieved and used for data analysis.

The above approach outlines how sorting benchmarks can be carried out in batches of arbitrary size and complexity – that is, multiple algorithms, multiple array sizes, and multiple iterations per array size. These procedures are initiated from the *Runner* class *via* its *main* method as shown in Listing 10 below.

```
1  public static void main(String[] args) {
2
3    // get the list of algorithms to run
4    String[] algorithms = Runner.getAlgorithmsToRun();
5
6    // define the number of samples to be run for
7    // each algorithm/array size
8    int samplesPerArraySize = 10;
9
10   // generate a list of sample sizes, n,
11   // which determine the number of elements
12   // in an array to be sorted
13   int[] arraySizes = Runner.generateArraySizes(100, 20);
14
15   // run a batch of integer sort benchmarks
16   ArrayList<AlgorithmBenchmarker> algorithmBenchmarkers;
17   algorithmBenchmarkers = Runner.runIntegerSortBenchmarks(
         arraySizes, samplesPerArraySize, algorithms);
18
19   // print the output data by delegation to the
20   // printers.BenchmarkDataPrinter class
21   Runner.printOutputData(algorithms, arraySizes,
         samplesPerArraySize, algorithmBenchmarkers);
22
23 }
```

Listing 10: An overview of the *main()* method in the *Runner* class.

Firstly a list of algorithms are generated *via* a call to the *getAlgorithmsToRun()* method in the *Runner* class. The number of iterations to run for each sorting benchmark is subsequently defined.

Next an array of integers corresponding to the array sizes which will

be sorted is generated *via* a call to the *generateArraySizes()* method in the *Runner* class. The array sizes are generated by provided the lower bound to the number of array elements (100 elements in the Listing 10 above). This lower bound is increased in multiples of two n times, where n is defined as 20 on line 13 of Listing 10 above. Note that this exponential increase in array sizes will quickly lead to some inefficient algorithms taking undesirable lengths of time to complete the sorting procedure. For this reason, specific upper bounds to array sizes for all algorithms have been specified in the *getAlgorithmSpecificMaxArraySizes()* method.

For inefficient comparison based sort algorithms, this upper bound is set to $\approx$ 60,000 elements. For the most efficient algorithms arrays containing over 1×10^8 elements could be sorted on reasonable timescales. Although not shown in Listing 9, which gives an overview of how batches of benchmarks are run, these restrictions on whether a specific array size will be benchmarked for a specific algorithm are enforced therein, in the *runIntegerSortBenchmarks()* method in the *Runner* class.

Once the algorithms to be benchmarked are defined, the number of iterations for each array size are set, and the list of array sizes to run have been initialized, the benmchmarking procedure is initiated *via* a call to the *runIntegerSortBenchmarks()* method in the *Runner* class, line 17 of Listing 10.

Finally, the list of *AlgorithmBenchmarker* instances returned from the benchmarking runs are passed to the *printOutputData()* method of the *Runner* class, along with the list of algorithms, array sizes etc. The *printOutputData()* method delegates responsibility for producing neatly-formatted output to the *BenchmarkDataPrinter* class in the *printers* package.

The *BenchmarkDataPrinter* class contains a number of useful print methods which will be briefly described but source code will not be discussed in detail. The *printBenchmarkPropertyToScreen()* method prints various benchmark results to the terminal or console that one runs the code in - it is useful for quick inspection of results. The *printBenchmarkPropertyToFile* and *printIterationDateToFiles()* write similar outputs to files. The *printBenchmarkPropertyToFile()* method produces tabular output in multiple files in the *./summary_data* directory, which will be created in whatever directory the source code is called from. Similarly the *printIterationDateToFiles()* prints iteration-specific benchmark CPU times to files in the *./iteration_data* directory, which will also be created if it does not exist.

For each algorithm and array size, the benchmark properties are computed over n iterations, and the outputs printed to the screen and to disk include:

- the mean run time (μ)
- the best-case run time
- the worst-case run time
- the variation in run times (σ)
- the coefficient of variation in run times as a % ($100 \times \sigma/\mu$)

The mean simulation time, μ is given by

$$\mu = \frac{1}{n}\sum_{i=1}^{n} t_i \tag{1}$$

where t_i is the time taken to complete a single array sort, and n is the total number of sorts performed. The variation, σ is given by:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(t_i - \mu)^2} \tag{2}$$

when $n \leq 100$, and:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(t_i - \mu)^2} \tag{3}$$

when $n > 100$, corresponding to sample and population standard deviations respectively. Irrespective of how σ is computed, the final variations in benchmark times for a given algorithm and array size are reported as 2σ uncertainties, thus representing a $\approx 95\%$ confidence interval.

In terms of visualisation of results, the tabular data written to file were designed to be compatible with the PANDAS library of PYTHON, with subsequent data visualisation and analysis carried out using the MATPLOTLIB library. All benchmark data presented in the next section, and the corresponding python files, are provided in the *./results_plotters* directory.

Whilst specific CPU times are presented with units of time (nanoseconds internally, milliseconds when written to file and plotted below), one should note that CPU times vary depending on the specific hardware employed and other system usage settings (e.g. availability of memory for recursive algorithms, garbage collection). All computations were carried out on a quad-core 8^{th} generation INTEL® CORE ™ i5-8250U with a clock frequency of 1.60 GHz – a modest performance chip given current standards. The L1, L2 and L3

caches are 32 kb, 256 kb, and 6144 kb. Whilst noting the CPU specifications is done for thoroughness, algorithms will ultimately be compared in terms of their $\mathcal{O}(n)$ notation and relative CPU times, and absolute CPU times are therefore of lesser importance.

Finally, the ECLIPSE integrated development environment (IDE) was used for code development and testing with the following Java environments:

- openjdk 11.0.6 2020-01-14
- OpenJDK Runtime Environment
  (build 11.0.6+10-post-Ubuntu-1ubuntu118.04.1)
- OpenJDK 64-Bit Server VM
  (build 11.0.6+10-post-Ubuntu-1ubuntu118.04.1, mixed mode, sharing)

The application as provided is configured to regenerate the results reported in Section 3.2. The total time to run those benchmarks is approximately 600 seconds. For faster tests one should probably modify the largest array size to be sorted by changing line 13 of the *Runner.main()* method as shown in Listing 10, and re-compile the source code.

To run the Java code provided the following command suffices in a linux shell:

```
1  $ cd code/sort_benchmarks/bin/
2  $ java runner.Runner
```

Listing 11: Calling the default JRE to run the algorithm benchmarks.

```
1  $ cd code/sort_benchmarks/bin/
2  $ /usr/lib/jvm/java-11-openjdk-amd64/bin/java runner.Runner
```

Listing 12: Calling a specific JRE to run the algorithm benchmarks class.

If errors are encountered they are likely related to trying to run the code using a different version of Java than that used to compile the source. The quickest solution is to recompile the source code using a local Java compiler. Example terminal output is given in Listings 13 and Figure 3 below. Note that the timings therein are illustrative only and were produced during developmental testing.

```
1  ------------------------------------------------------------
2  -Runner.main()
3  ------------------------------------------------------------
4  ------------------------------------------------------------
5  -Runner.getAlgorithmsToRun()
6  ------------------------------------------------------------
7  -Will run the following algorithms:
8    -bubble
9    -insertion
10   -selection
11   -quick
12   -counting
13   -merge
14 ------------------------------------------------------------
15 ------------------------------------------------------------
16 -Runner.generateArraySizes()
17 ------------------------------------------------------------
18 -Run each algorithm for 20 different array sizes
19   -Smallest array size = 100 elements
20   -Largest array size = 52428800 elements
21 ------------------------------------------------------------
22 ------------------------------------------------------------------
23 -Runner.runIntegerSortBenchmarks()
24 ------------------------------------------------------------------
25   -For each array size, and each algorithm run 10
26   -samples to generate statistics on performance
27 ------------------------------------------------------------------
28 ------------------------------------------------------------
29 -Runner.getAlgorithmsMaxArraySizes()
30 ------------------------------------------------------------
31   ------------------------------------------------------------
32   -Running [bubble] sort for array size of [100] integers
33   ------------------------------------------------------------
34   -mean  run time (nanoseconds)                          = 319351
35   -best  run time (nanoseconds)                          = 279584
36   -worst run time (nanoseconds)                          = 371549
37   -variation in run time [+- 2.sigma] (nanoseconds) = 63720
38   -variation in run time [+- 2.sigma] (% of mean)   = 19%
```

Listing 13: Example terminal output from running the runner.Runner() method.

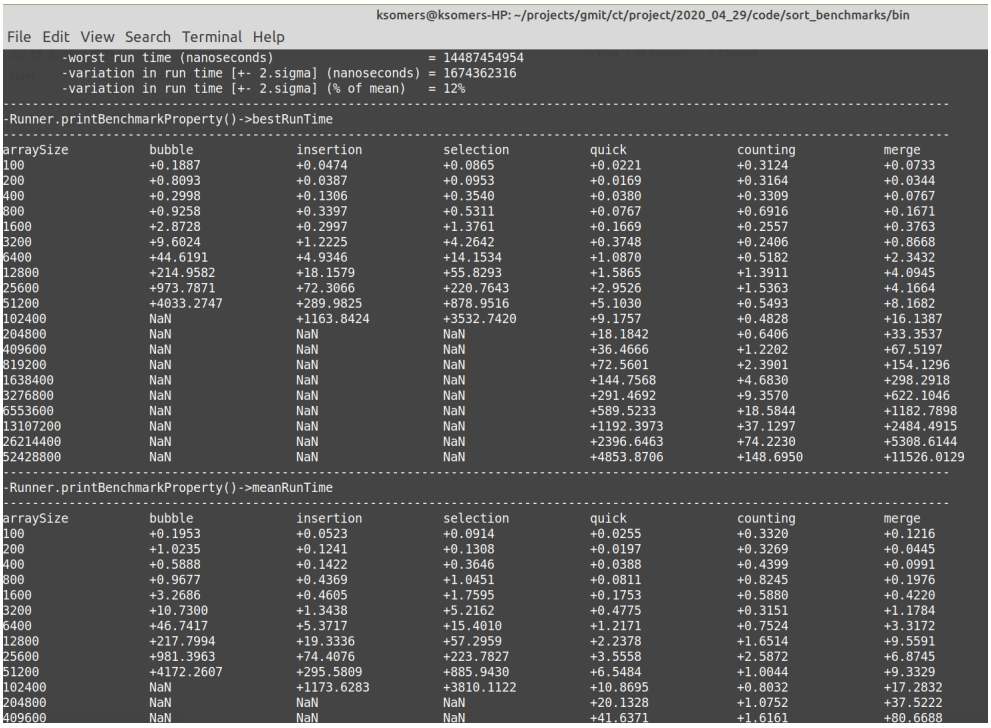

Figure 3: An illustrative screenshot of CPU times in milliseconds (best run time, mean run time) being written to a terminal for different algorithms and array sizes. NaN values correspond to algorithms which were not benchmarked for a particular array size owing to their inefficiency.

3.2 Benchmarking Results

This section will describe and analyse the benchmarking results obtained from the algorithms described and implemented in previous sections. The specifics of each algorithm will not be re-iterated in detail as they have been described comprehensively in previous sections, and important conclusions from previous sections will merely be referred to when appropriate. The focus of this section will be to interpret the benchmarking results in terms of the theoretical characteristics of the algorithms previously presented. Figure 4 and Table 3.2 present the key results – mean CPU times as a function of array size for each algorithm, each averaged over 10 samples. Given the integer arrays being sorted are randomly distributed, it is safe to assume average-case Θ time-complexities when comparing algorithms.

In Figure 4 each symbol corresponds to the mean CPU time, with corresponding shaded regions indicating the bounds for the largest (worst-case) and smallest (best-case) sorting times for a given array size n. One should

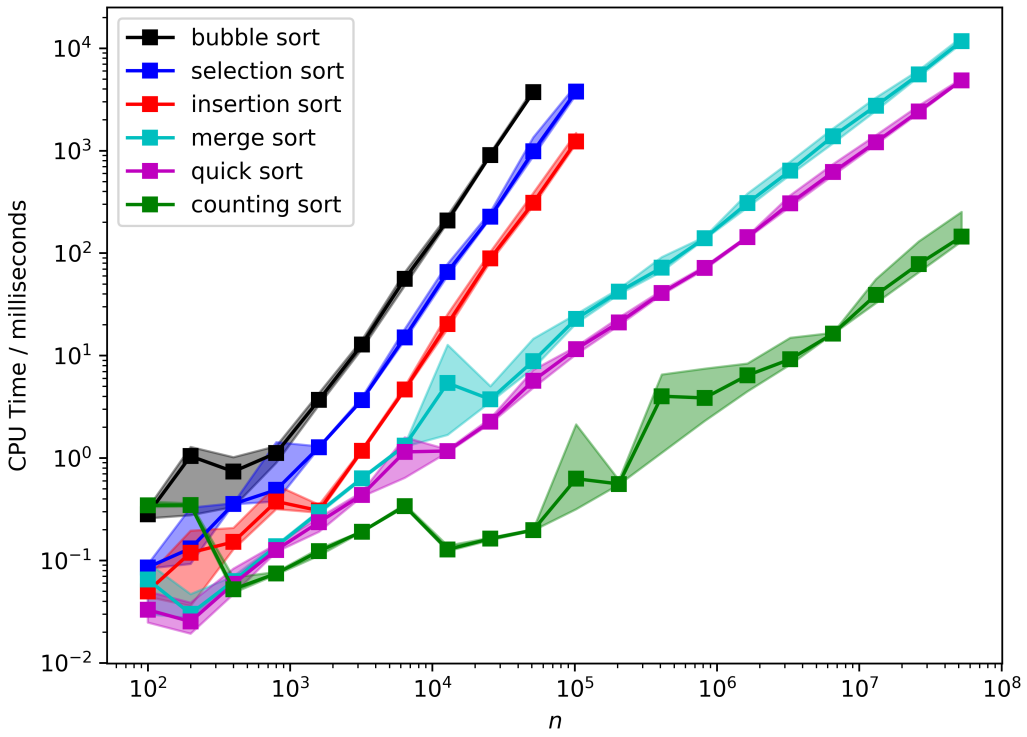

Figure 4: Benchmarking results obtained from the algorithms implemented in the current work where the y-axis is the CPU time in milliseconds, and the x-axis, n, is the number of elements in a given array. Lines and symbols represent the mean CPU time computed over 10 iterations for each algorithm and array size. The shaded regions represent the upper (worst-case) and lower (best-case) limits of the CPU times determined over 10 iterations. Both the x and y axes are represent on logarithmic scales.

note therefore that there are fluctations between CPU times computed for a given algorithm/array size but the general trends are still sensible, different algorithms can be compared and differentiated with certainty, and the fluctuations are generally modest for large values of n which are of most interest. It is also worth stating that Figure 4 presents the benchmark results on a log-log scale in order to emphasise similarities and differences more clearly, and so that different asymptotic behaviours, as implied by the slope of each line, can be clearly observed for algorithms with different time-complexity.

The three simple comparison-based sorts, BubbleSort, InsertionSort, and SelectionSort (which is included in the benchmarks for comparative and illustrative purposes [27], but not described in detail previously) all have theoretical average-case time-complexity of $\Theta(n^2)$. Figure 4 clearly shows they

have same asymptotic behaviour with all methods having the same slope.

BubbleSort consistently shows the worst overall performance for any appreciable value of n. Note that the BubbleSort algorithm implemented herein was modified, as shown in Listing 2, to account for scenarios where no inversions are required, a modification which was not introduced to other algorithms. Developmental testing showed that this modification had only a modest influence on the absolute CPU times, and no effect on the apparent asymptotic behaviour for large values of n. Despite the fact that InsertionSort and SelectionSort have better absolute performance than BubbleSort, which makes them objectively better sorting algorithms from an efficiency perspective, they all follow the same behaviour as n tends to infinity, which effectively rules all three out as effective sorting algorithms for all but the smallest arrays/lists.

QuickSort and MergeSort should also be analysed in tandem, given that they are both recursive comparison-based methods with $\Theta(n \log n)$ time-complexity, with both methods clearly showing the same behaviour in terms of the dependency of CPU times on array size. Notably, their asymptotic behaviour is markedely different from the previous algorithms which have n^2 time-complexity, and both algorithms can be employed to sort arrays orders-of-magnitude greater in size than the n^2 algorithms.

QuickSort tends to outperform MergeSort by $\approx$ a factor of 2 in terms of absolute CPU times, although the differences tend to become greater as n increases – for arrays of $\approx 1 \times 10^5$ elements MergeSort is a factor of 1.9 times slower than QuickSort, this ratio increasing to a factor of 2.4 for the largest array of $\approx 5 \times 10^7$ elements. The likely reason is the greater space-complexity of MergeSort relative to QuickSort. QuickSort carries out sorting in-place, requiring a small amount of fixed auxiliary memory to store variables during each recursive call, and $\log n$ space-complexity is the norm. Conversely, MergeSort has $\mathcal{O}(n)$ space-complexity, requiring extra memory to store array elements in two newly created sub-arrays during each recursive call. Added to this is the time required to build these arrays, and the subsequent merging operations, both of which are $\mathcal{O}(n)$ operations, although the constant multipliers representing the number of $\mathcal{O}(n)$ operations is omitted in asymptotic analysis.

Whilst QuickSort outperforms MergeSort in the present case, which is effectively representative of average-case performance given that the arrays are randomly distributed, one should note the caveats discussed in Sections 2.3 and 2.6 related to pivot selection. MergeSort is guaranteed to have $n \log n$ time-complexity in the worst-, average- and best-cases owing to the well balanced binary trees which result upon recursion. Whilst QuickSort has the same performance in the average and best cases, its worst-case $\mathcal{O}(n^2)$

behaviour when pivot selection is poor should be considered, and MergeSort may outperform QuickSort in certain circumstances and may well be chosen in practice.

What is clear from the present analysis is that both QuickSort and MergeSort are superior to the simple comparison-based sorts, in terms of time-efficiency, for arrays greater than 10^3 elements. For small arrays containing $n < 10^3$ elements, or situations where memory is an issue, InsertionSort may still be of use, as it has comparable efficiency to the recursive methods for small arrays, but it has the added benefits of being simple to implement, and has $\mathcal{O}(1)$ space-complexity.

Finally we consider the performance of the only non-comparison-based method implemented in the current project – CountingSort. Figure 4 shows that it outperforms all other methods for arrays of size $n \geq 1 \times 10^3$ elements. Its theoretical time-complexity is given by $\mathcal{O}(n+k)$, $\Theta(n+k)$ and $\Omega(n+k)$ in the worst-, average-, and best-case scenarios, where n is the array size and k is the range of discrete values between the minimum and maximum values in the array.

Given that the time-complexity of CountingSort is dependent on both the number of elements n, and the range of these elements, k, the results presented in Figure 4 neither account for nor give insight into the influence of k on its performance. Therefore, an important detail which must be considered when considering its performance as presented in Figure 4 is the upper limiting value which could be assigned to any single integer during random array generation. For all algorithms, array sizes and iterations the randomly generate integer arrays were allowed to contain any integer in the range $0 - 100000$. As the arrays used for benchmarking were generated randomly from values in this range, the absolute value for k is not known precisely for each array size n, and each iteration, when CountingSort was implemented.

In order to get a better sense of the influence of k on the performance of CountingSort, further simulations were run using CountingSort for various array sizes n, and fixed values of k. For these simulations k was fixed by first generating an array of n random integers, and subsequently defining two elements to be equal to 0 and $k - 1$.

Figure 5 subsequently shows the combined influence of both n and k, with k clearly influencing the absolute CPU times for a given n. As expected, when $n << k$ the efficiency of the algorithm is dominated by the k term, as reflected by the near constant CPU times for a given k when $k \geq 1 \times 10^4$ and $100 \geq n \leq 1 \times 10^3$. Conversely, where $n >> k$ the n term is dominant, as illustrated for simulations where $n \geq 1 \times 10^6$ and $100 \geq k \leq 1 \times 10^5$. The intermediate case where $n \approx k$ results in the effiency being governed by some

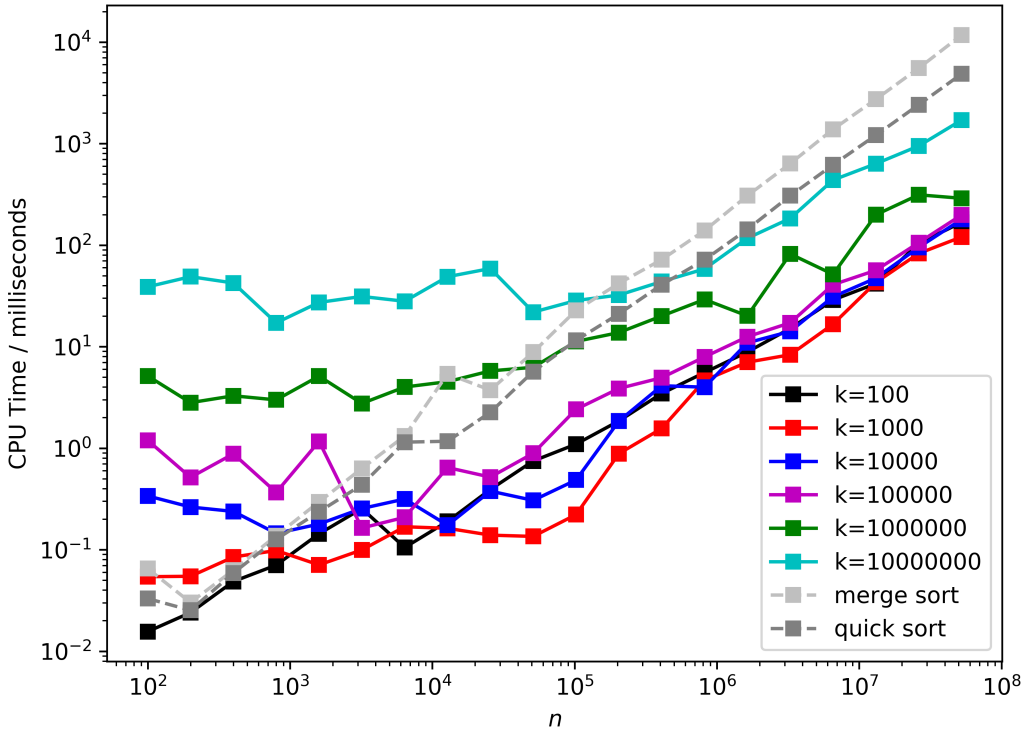

Figure 5: Benchmarking results obtained from the CountingSort algorithm implemented in the current work where the y-axis is the CPU time in milliseconds, and the x-axis, n, is the number of elements in a given array. Lines and symbols represent the mean CPU time computed over 10 iterations for each algorithm and array size. Each series corresponds to simulation results where the range of integers in each array has been specficially set to the value indicated by k. Results from MergeSort and QuickSort from Figure 4 are included for comparitive purposes. Both the x and y axes are represent on logarithmic scales.

function $\propto n + k$ and both terms contribute to performance.

When considering the superior performance of CountingSort for large values of n in Figure 4, one should therefore be mindful of the caveat that k is restricted to values no greater than 1×10^5. Indeed both MergeSort and QuickSort are shown to outperform CountingSort under certain conditions in Figure 5, and although not shown, BubbleSort, InsertionSort and SelectionSort may well outperform CountingSort where $n < 1 \times 10^3$ and $k > 1 \times 10^4 - 1 \times 10^5$.

To summarise, this report aimed to benchmark a number of comparison and non-comparison based sorting algorithms largely in terms of their time-

and space-complexities. A number of popular sorting algorithms have been implemented in Java, and analysed from a theoretical perspective in order to understand their limiting behaviours, and to thus classify them *via* $\mathcal{O}$-notation. An object-orientated suite of tools was developed in Java in order to carry out benchmark calculations of their CPU times for arrays of randomly generated integers of varying size. The benchmark results provided realistic empirical data on their performance, which was subsequently analysed, and interpreted in terms of their theoretical characteristics.

Table 6: Mean CPU benchmark times (milliseconds, averaged over 10 iterations) as a function of array size (n) for various sorting algorithms benchmarked in the current work. NaN results indicate that no sorting benchmarks were performed for a particular algorithm/array size as a result of their poor sorting efficiency.

n	bubble	insertion	selection	quick	counting	merge
100	2.825×10^{-1}	4.990×10^{-2}	8.550×10^{-2}	3.300×10^{-2}	3.449×10^{-1}	6.520×10^{-2}
200	$1.042\times10^{+0}$	1.189×10^{-1}	1.310×10^{-1}	2.530×10^{-2}	3.469×10^{-1}	3.020×10^{-2}
400	7.365×10^{-1}	1.514×10^{-1}	3.552×10^{-1}	5.890×10^{-2}	5.230×10^{-2}	6.290×10^{-2}
800	$1.116\times10^{+0}$	3.764×10^{-1}	4.886×10^{-1}	1.265×10^{-1}	7.450×10^{-2}	1.374×10^{-1}
1,600	$3.685\times10^{+0}$	3.061×10^{-1}	$1.275\times10^{+0}$	2.365×10^{-1}	1.230×10^{-1}	2.956×10^{-1}
3,200	$1.279\times10^{+1}$	$1.177\times10^{+0}$	$3.677\times10^{+0}$	4.352×10^{-1}	1.908×10^{-1}	6.322×10^{-1}
6,400	$5.588\times10^{+1}$	$4.655\times10^{+0}$	$1.503\times10^{+1}$	$1.146\times10^{+0}$	3.394×10^{-1}	$1.317\times10^{+0}$
12,800	$2.080\times10^{+2}$	$2.032\times10^{+1}$	$6.533\times10^{+1}$	$1.170\times10^{+0}$	1.278×10^{-1}	$5.387\times10^{+0}$
25,600	$9.053\times10^{+2}$	$8.860\times10^{+1}$	$2.261\times10^{+2}$	$2.258\times10^{+0}$	1.629×10^{-1}	$3.731\times10^{+0}$
51,200	$3.713\times10^{+3}$	$3.099\times10^{+2}$	$9.842\times10^{+2}$	$5.683\times10^{+0}$	1.974×10^{-1}	$8.821\times10^{+0}$
102,400	NaN	$1.231\times10^{+3}$	$3.779\times10^{+3}$	$1.154\times10^{+1}$	6.274×10^{-1}	$2.276\times10^{+1}$
204,800	NaN	NaN	NaN	$2.103\times10^{+1}$	5.578×10^{-1}	$4.199\times10^{+1}$
409,600	NaN	NaN	NaN	$4.075\times10^{+1}$	$4.009\times10^{+0}$	$7.207\times10^{+1}$
819,200	NaN	NaN	NaN	$7.171\times10^{+1}$	$3.851\times10^{+0}$	$1.392\times10^{+2}$
1,638,400	NaN	NaN	NaN	$1.430\times10^{+2}$	$6.400\times10^{+0}$	$3.071\times10^{+2}$

Table 6: Mean CPU benchmark times (milliseconds, averaged over 10 iterations) as a function of array size (n) for various sorting algorithms benchmarked in the current work. NaN results indicate that no sorting benchmarks were performed for a particular algorithm/array size as a result of their poor sorting efficiency.

n	bubble	insertion	selection	quick	counting	merge
3,276,800	NaN	NaN	NaN	$3.065\times10^{+2}$	$9.202\times10^{+0}$	$6.361\times10^{+2}$
6,553,600	NaN	NaN	NaN	$6.174\times10^{+2}$	$1.636\times10^{+1}$	$1.384\times10^{+3}$
13,107,200	NaN	NaN	NaN	$1.210\times10^{+3}$	$3.908\times10^{+1}$	$2.735\times10^{+3}$
26,214,400	NaN	NaN	NaN	$2.415\times10^{+3}$	$7.815\times10^{+1}$	$5.537\times10^{+3}$
52,428,800	NaN	NaN	NaN	$4.859\times10^{+3}$	$1.446\times10^{+2}$	$1.172\times10^{+4}$

References

[1] "The world's most valuable resource is no longer oil, but data." https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data.

[2] K. Bhageshpur, "Council Post: Data Is The New Oil – And That's A Good Thing." https://www.forbes.com/sites/forbestechcouncil/2019/11/15/data-is-the-new-oil-and-thats-a-good-thing/.

[3] K. Schwab, *The fourth industrial revolution*. New York: Crown Business, first u.s. edition ed., 2016.

[4] "Aurora Supercomputer: Accelerating the convergence of high performance computing (HPC) and AI at Exascale.." https://www.intel.com/content/www/us/en/high-performance-computing/supercomputing/exascale-computing.html.

[5] J. Aron, "IBM unveils its first commercial quantum computer." https://www.newscientist.com/article/2189909-ibm-unveils-its-first-commercial-quantum-computer/.

[6] "Google supercharges machine learning tasks with TPU custom chip." https://cloud.google.com/blog/products/gcp/google-supercharges-machine-learning-tasks-with-custom-chip/.

[7] P. G. H. Bachmann, *Die analytische Zahlentheorie. Dargestellt von Paul Bachmann*. Leipzig B.G. Teubner, 1894.

[8] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig B.G. Teubner, 1909.

[9] D. P. Mannion, "Lecture notes in computational thinking and algorithms," April 2020. Galway-Mayo Institute of Technology.

[10] D. Harel and Y. A. Feldman, *Algorithmics: the spirit of computing*. Harlow, Essex, England ; New York: Addison Wesley : Pearson Education, 3rd ed ed., 2004.

[11] G. T. Heineman, S. Selkow, and G. Pollice, *Algorithms in a nutshell*. Beijing ; Sebastopol [Calif.]: O'Reilly, 2009. OCLC: ocn291100353.

[12] "Stack Overflow - Where Developers Learn, Share, & Build Careers." https://stackoverflow.com/.

[13] "Wikipedia Main Page," Apr. 2020. https://en.wikipedia.org.

[14] K. E. Iverson, *A programming language*. 1962. OCLC: 523128.

[15] TablesGenerator.com, "Create LaTeX tables online – TablesGenerator.com." https://www.tablesgenerator.com/.

[16] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, p. 321, July 1961.

[17] "Quicksort in Java - Tutorial." https://www.vogella.com/tutorials/JavaAlgorithmsQuicksort/

[18] "Implement quick sort in java. - Java sorting algorithm programs." https://www.java2novice.com/java-sorting-algorithms/quick-sort/.

[19] "QUICKSORT (Java, C++) | Algorithms and Data Structures." http://www.algolist.net/Algorithms/Sorting/Quicksort.

[20] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *ACM SIGACT News*, vol. 12, pp. 36–44, Sept. 1980.

[21] H. Seward, *Information Sorting in the Application of Electronic Digital Computers to Business Operations*. M.I.T. Digital Computer Laboratory, M.I.T. Digital Computer Laboratory, 1954.

[22] "Counting sort," May 2014. www.growingwiththeweb.com/2014/05/counting-sort.html.

[23] "Implement Counting Sort using Java + Performance Analysis | JavaInUse." https://www.javainuse.com/java/countingsort.

[24] H. H. Goldstine and J. Von Neumann, "Planning and coding of problems for an electronic computing instrument, part ii, volume ii," 1948.

[25] I. Educative, "How to implement a merge sort in Java." Library Catalog: www.educative.io.

[26] L. S. Inc., "Lucidchart.com."

[27] "Selection Sort in Java - Javatpoint." https://www.javatpoint.com/selection-sort-in-java.