Galway Mayo Institute of Technology

H. Dip. in Data Analytics

46887 Computational Thinking with Algorithms

Benchmarking Sorting Algorithms

Author:
Elizabeth Daly

Lecturer:
Dr. Dominic Carr

April 28, 2020


GMIT
INSTITIÚID TEICNEOLAÍOCHTA NA GAILLIMHE-MAIGH EO
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Abstract

This report describes an algorithm benchmarking project for module 46887 Computational Thinking with Algorithms. The project consists of a Python application (completed using Jupyter notebook) to benchmark five different sorting algorithms: insertion sort, merge sort, counting sort, quicksort, and heapsort. The Python application includes implementations of the sorting algorithms, and a single cell to benchmark them by measuring their running time using arrays of randomly generated integers. In this report we consider the concept of sorting and explain some of the relevant associated concepts. We introduce each of the algorithms, discuss their time and space complexity, and explain in detail how each works. The process followed for benchmarking is described, and the results presented. We examine if the results obtained matched theoretical expectations and we discuss observed differences between the algorithms in terms of their performances.

Contents

1 Introduction **4**
- 1.1 Space and Time Complexity of Algorithms . . . 5
  - 1.1.1 Big O notation . . . 7
- 1.2 Performance of Algorithms . . . 7
- 1.3 In-place Sorting . . . 8
- 1.4 Stable Sorting . . . 9
- 1.5 Comparator Functions . . . 9
- 1.6 Comparison-Based Sorts . . . 10
- 1.7 Non-Comparison-Based Sorts . . . 10

2 Sorting Algorithms **12**
- 2.1 Insertion Sort . . . 12
  - 2.1.1 How does it work? . . . 12
  - 2.1.2 Time and space complexity . . . 14
- 2.2 Merge Sort . . . 15
  - 2.2.1 How does it work? . . . 16
  - 2.2.2 Time and space complexity . . . 19
- 2.3 Counting Sort . . . 20
  - 2.3.1 How does it work? . . . 20
  - 2.3.2 Time and space complexity . . . 22
- 2.4 Quicksort . . . 22
  - 2.4.1 How does it work? . . . 23
  - 2.4.2 Time and space complexity . . . 24
- 2.5 Heapsort . . . 25
  - 2.5.1 How does it work? . . . 26
  - 2.5.2 Time and space complexity . . . 29
- 2.6 Summary of algorithms described . . . 29

3 Implementation & Benchmarking **32**
- 3.1 Instructions . . . 32
- 3.2 Coding the sorting algorithms in Python . . . 32
  - 3.2.1 insertion_sort . . . 32
  - 3.2.2 merge_sort . . . 33
  - 3.2.3 counting_sort . . . 33
  - 3.2.4 quick_sort . . . 33
  - 3.2.5 heap_sort . . . 33

3.3 Benchmarking the sorting algorithms 38
3.4 Benchmarking results 41
3.4.1 Expected behaviour 43
3.5 Performance on sorted and reversed arrays 43

4 Conclusion **51**

1 Introduction

What does sorting mean? It is the act of arranging an *unordered* list of elements into an *ordered* list [1]. The ordering rules to apply will depend on the type of the input data. For example, if the the unordered list consists of numbers, the aim of the sorting process may be to arrange them in ascending or descending order. If the unordered list consists of words, then the sorting process may consist of placing those words in their lexicographic (or alphabetical) order; think of the way books are arranged on a library shelf using alphabetical order of the author surname. Other examples of sorting include placing calendar events in date order, arranging certain products in ascending price order on an online purchasing website prior to making a decision, displaying transactions on a bank statement by transaction number, and displaying the results of a web search by relevance [2, 3]. A sorting algorithm is a step by step process which is to be followed in order to perform the particular sorting task. It will take the unsorted input, follow a series of pre-defined steps, and produce a sorted output. Efficient sorting algorithms are much sought-after, both as stand-alone processes and as the pre-processing steps for other tasks, for example in computer graphics [3]. Some of the general conditions for sorting are [3]:

- A list of elements is said to be sorted if each element in the list is less than or equal to its successor. The definition of less than depends on the particular problem.
- Duplicate items must be contiguous in the ordered list.
- The contents of the list must be the same before and after sorting.

Note that, for this project, we are concerned only with sorting random lists of integers. Therefore, we are dealing with a simple subset of sorting tasks. We will now discuss some of the concepts associated with sorting algorithms including: time and space complexity, algorithm performance, in-place sorting, stable sorting, comparator functions, comparison-based sorts, and non-comparison-based sorts. These are all important factors when comparing algorithms and deciding which one to use for a particular application or input instance size.

liz [2] General [1] Again [4]

1.1 Space and Time Complexity of Algorithms

One of the first things to consider when analysing or comparing algorithms is the idea of time and space complexity [5]. Complexity is different to performance, in that performance is a measure of how much time or memory a program requires; it is platform dependent, so the same algorithm may run more/less efficiently on different physical computers. Complexity, on the other hand, is a function of how the algorithm is designed, not a function of the hardware on which it is run. It can be difficult to seperate complexity from performance, and complexity affects performance but not the other way around [5]. The trick with complexity is to think of the problem in an abstract way, and therefore compare algorithms in a platform-independent way.

The **space complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to run and produce a result, usually quantified as a function of the size of the input [6, 7]. It includes memory required to store compiled algorithm instructions, memory required by method activation frames (of recursive calls for example), and memory required by data and variables that the algorithm uses [7]. In practice, it is the latter that is used to quantify space complexity. For example, a trivial algorithm that creates a list of length n has space complexity of order n; if n increases, the space requirement will increase linearly with n [8]. In this module we have been comparing the relative efficiency of various algorithms primarily by analysing their running time complexity [9]; that is the factor we are considering when benchmarking the algorithms in this report.

The **time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input [4]. Algorithms can be compared by evaluating their running time complexity on inputs of size n (the benchmarking process that we will be carrying out during the course of this project). This complexity usually falls into one of a number of families, so that execution time grows at a certain rate with respect to increasing input size n. Figure 1, taken directly from the lecture notes, illustrates different growth families. If the running time scales as n^2, this means that the running time depends on the size of the input in a quadratic fashion, for example. The most expensive (in terms of time) computation in an algorithm is the one which determines complexity. Therefore, if an algorithm has two steps, one linear and the other quadratic, it is the

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

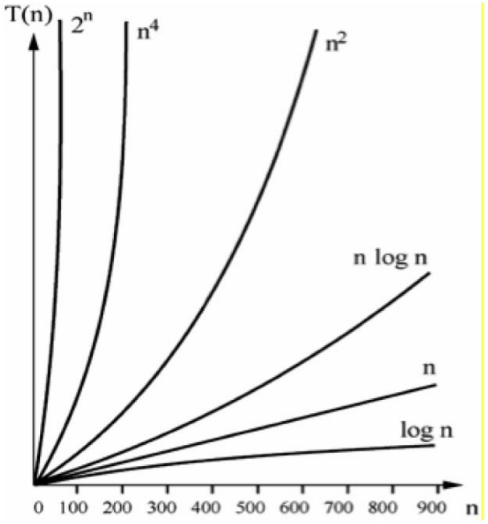

Figure 1: Running time complexity T(n) as a function of input size n.

quadratic term which dominates.

Complexity is often calculated for the best, worst, and average cases because, as well as the size of the input, the structure of the input data can affect the run time. For example, a search algorithm will run faster with input data that is already sorted compared to completely random input data.

- **Worst case:** the input instances which lead to the worst (maximum) runtime behaviour. It is often the easiest case to analyse and it provides a lower bound on performance. A simple example might be searching through an array of length n for a particular value only to find that the element you are looking for is the last element checked.

- **Average case:** describes the algorithm behaviour when the input is completely random. Sticking with the last example, the element you are searching for will be somewhere in the middle of the list most of the time.

- **Best case:** the particular set of inputs for which the algorithm does the least amount of work. In the search example, the element you are looking for happens to be the very first element checked. This rarely happens in practice.

1.1.1 Big O notation

Big O notation is used in computer science to classify algorithms according to how their time or space complexity scales as input size n grows, in the worst case [10, 11, 12]. It describes the behaviour of a function in the limit as the argument tends towards infinity (or some particular value). Big O is expressed as $O(n)$, for example, if the complexity of an algorithm grows linearly with input size n. The aim is always to identify the tightest upper bound, so while it is true to say that an algorithm that is $O(n^2)$ is also $O(n^3)$, $O(n^2)$ is the tighter bound. There are other notations used to describe the best (Ω) and average (Θ) case complexities of algorithms but we will not go into them in this project [13]. I found an excellent explanation of Big O notation, with lots of calculated examples, in lecture notes available at [10]. The first step in calculating Big O is to identify the individual steps/computations carried out and add them to calculate a total running time T(n). If T(n) is a sum of terms, only keep the largest growth rate term (most expensive computation). If any part of T(n) is a product, all constants can be omitted. We say that $T(n)$ is $O(g(n))$ if and only if

$$\lim_{n\to\infty} \frac{T(n)}{g(n)} < \infty$$

For example, if we analyse an algorithm and find that $T(n) = n^2+42n+7$, is $T(n)$ of $O(n)$? The answer is no because the limit tends towards ∞:

$$\lim_{n\to\infty} \frac{T(n)}{g(n)} = \lim_{n\to\infty} \left(\frac{n^2 + 42n + 7}{n} \right) = \lim_{n\to\infty} \left(n + 42 + \frac{7}{n} \right) = \infty.$$

Is $T(n)$ of $O(n^2)$? The answer is yes because

$$\lim_{n\to\infty} \frac{T(n)}{g(n)} = \lim_{n\to\infty} \left(\frac{n^2 + 42n + 7}{n^2} \right) = \lim_{n\to\infty} \left(1 + \frac{42}{n} + \frac{7}{n^2} \right) = 1 < \infty.$$

so $T(n)$ is $O(n^2)$. We will analyse the complexity of our chosen algorithms later in this report. Typically, an algorithm which iterates through an array of length n via a single for loop will be of $O(n)$, while two nested for loops will be of $O(n^2)$.

1.2 Performance of Algorithms

As we stated above, the performance of an algorithm is a measure of how much time or memory the code requires. It is platform dependent, so the

same algorithm may run more/less efficiently on different physical computers. A computer with many cores, more RAM, and a fast compiler will be able to run code faster than an older machine. Most discussions of algorithmic efficiency talk about the theoretical Big O approach we just described. On a practical level, if one wants to measure the platform specific performance of an algorithm, the easiest way to do that is to measure the time the algorithm takes to run for different sizes and instances of input [14]. In the Python programming language the time module allows one to record the start and end times, and therefore the running time, of any code snippet. We will make use of this module to benchmark sorting algorithms in this way later on in this report. While working on the project, I found that my algorithm performances were influenced by how many other processes (if any) I was running on my laptop at the same time as I was running the benchmarking code. An example of an empirical analysis for Java is given in the conference proceedings here [15].

1.3 In-place Sorting

A sorting algorithm is classified as **in-place** if it only uses a fixed additional amount of working space, regardless of the size of the input [3]. It is therefore associated with the space complexity of algorithms. If memory is limited, in-place sorting with efficient use of available memory is obviously desirable. All of the simple comparison-based sorting algorithms discussed in the lectures (bubble, selection, insertion sort) are in-place algorithms. Sorting algorithms that are not classified as in-place require additional space, and the amount required scales in some way with the size of the input. As [16] explains, the concept of an in-place algorithm is not unique to sorting, but it is one of the applications for which it is most important. The goal is to produce the sorted output using the same memory space that holds the unsorted input. This is done by successfully transforming the input until the output is produced; there is then no need to set aside one memory space for the input and another for the output. In sorting algorithms, this can be done by repeatedly exchanging elements in the unsorted list until a sorted list results. The alternative to in-place sorting would be to leave the original unsorted list unchanged and create a sorted copy of it. An example of a not-in-place sorting algorithm is merge sort, which is one of the algorithms chosen by me for benchmarking in this project [17].

1.4 Stable Sorting

A sorting algorithm is regarded as stable if it preserves the order of equal elements in the input when it produces the output [3, 17]. Stability is important if we wish to maintain the sequence of original elements in the input. Formally, if two elements a_i and a_j in the unordered list are equal, and if $i < j$, then the location of $A[i]$ must be to the left of $A[j]$ in the final sorted list (p. 55 [4]). The example given in [4] is of sorting information on an airport departures board; when sorting flight information by destination, flights to the same city still appear in order of departure time as they did on the original list. An unstable sort may or may not preserve the ordering of equal elements in the original list, and this may a quality that is desired.

1.5 Comparator Functions

According to p. 264 of [1], a comparator is a processor that takes as input two elements and outputs them in order - quite a vague statement. This general process is illustrated in Figure 2 taken from [1]. We discussed methods of ordering elements in the introduction, and each of these will require its own custom comparator function. On p. 55 of [4] a comparator function is described as one which compares two elements p and q and returns:

- a negative number if $p < q$,
- zero if $p = q$,
- a positive number if $p > q$.

When the elements to be sorted are numbers or strings the definitions of $<, =$, or $>$ are obvious (numerical and alphabetical ordering), but that may not always be the case. In general they will depend on the type of the input data. Luckily for this project we will be working with arrays of integers

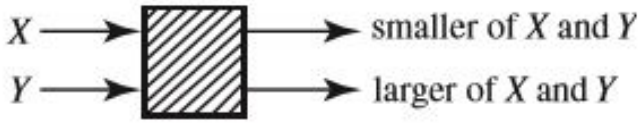

Figure 2: An abstract comparator.

where the meaning of $<, =$, or $>$ is clear. For complicated input data, a sort key may be required so that elements can be compared by proxy via their key values. The elements to be sorted may also have **satellite data**, which is any information that is associated with the sort key. This must travel with the key as the key is sorted. Satellite data might be the book itself as opposed to the author surname, or fields such as name, age, address associated with a customer number [18] in a retail database, for example. As this project is concerned with sorting simple arrays of integers, there is no associated satellite data that we have to worry about.

When sorting lists we often talk about the number of **inversions**. An inversion exists in a list if the elements at positions i and j are out of order: so even though $i < j$, element $A[i] > A[j]$. A list with a lot of inversions is further from being sorted than one that has few inversions. For example, the list [2, 1, 4] has only one inversion as element pair (2,1) is out of order; once they are swapped the list is sorted.

1.6 Comparison-Based Sorts

In comparison-based sorting, the elements of an array are compared with each other to determine the order in which they should be placed in the final sorted array [19]. These types of sorting algorithms *only* use comparison operations to decide on the order of elements in a sorted list. As we discussed above, the particular comparison operation to use will depend on the type of the input data. I found an excellent visualization of some common comparison-based sorting algorithms in action on the University of San Francisco Computer Science web pages, where the individual comparison steps (a pair of elements at a time) can be clearly seen [20]. Examples of well-known comparison-based sorts include bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort. We will be discussing some of these in detail below. For now, we just state that comparison-based sorting algorithms can do no better than $O(n \log n)$ performance in the average or worst cases, and they are very applicable to input data which is diverse [3].

1.7 Non-Comparison-Based Sorts

In contrast, non-comparison-based sorting algorithms use information about the input data to sort it in some other way, not by comparing elements. These algorithms can be used to sort input data that have a known distribution

or that fall into a certain range. Counting sort, bucket sort, and radix sort are non-comparison-based algorithms [21]. Counting sort sorts by key value, while bucket and radix sort examine parts of the key [22]. Their best worst-case time complexity is $O(n)$ as, at a minimum, each element must be checked one time. For this reason they are often referred to as *linear* sorts. We will benchmark a counting sort algorithm later on.

2 Sorting Algorithms

For this project we have been asked to benchmark five sorting algorithms, three according to specific criteria and then any other two. The five I have chosen are listed in Table 1. We must explain how each algorithm works (using diagrams and example input instances) and discuss their time and space complexity. We will be sorting arrays of randomly generated integers of different input sizes n.

2.1 Insertion Sort

To implement insertion sort in Python, I adapted code taken from two websites: Python Central [23] and Tutorial Edge [24], and used Geeks for Geeks [25] for some help with my understanding of the algorithm. This is clearly stated as comments to the code in my Jupyter notebook.

Insertion sort is a simple comparison-based sort. It is a good choice of algorithm if the size n of the list to be sorted is small, if the number of inversions is low (list already mostly sorted), and if we want to minimize the amount of code that needs to be written. It is often used in the latter stages of hybrid sorting algorithms where the early stages of the sorting task have already been achieved using other algorithms. It is also stable and in-place.

2.1.1 How does it work?

Insertion sort is an iterative algorithm which works by dividing a list into a sorted part and an unsorted part. On each iteration of the loop, the size of the sorted sublist grows by one element while the unsorted portion shrinks

1 Insertion sort	Simple comparison-based sort
2 Merge sort	Efficient comparison-based sort
3 Counting sort	Non-comparison-based sort
4 Quicksort	Efficient comparison
5 Heapsort	Efficient comparison

Table 1: Algorithms for benchmarking.

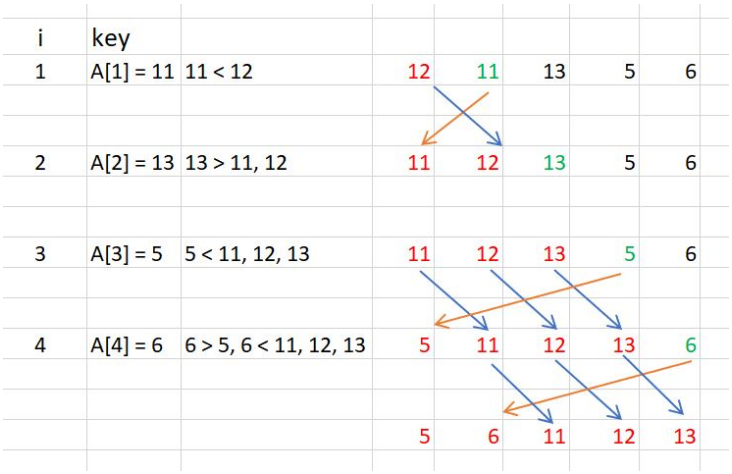

Figure 3: Insertion sort example 1.

by one element. I found a very good explanation of how the algorithm works on the Geeks for Geeks website [26]. Assume the input is a list A of length n with indices 0, 1, ..., n-1. The algorithm will iterate through the list, starting at index 1 and ending at index $n-1$. On the first iteration of the loop, the first element (at index 0) is assumed to be sorted and the second element (index 1) is the key. The key is compared to each of the elements in the sorted part of the list (at indices < 1 to its left) and inserted into the correct position there. Any elements $>$ the key are shifted one position to the right (a move called a neighbouring *transposition*) within the sorted sublist. On the second iteration of the loop the key is the element at index 2 and elements at indices 0 and 1 are assumed sorted. The key is compared to the sorted part of the list and inserted into the correct position, with any elements that are $>$ the key being shifted one position to the right. The process continues until the key is the last element in the list, at index n-1. Consider the example of an unsorted list A = [12, 11, 13, 5, 6]; it has n = 5 elements positioned at indices 0, 1, 2, 3, 4. The algorithm will iterate through indices 1, 2, 3 and 4 as illustrated in Figure 3 with the key coloured green at each iteration and the sorted part of the list red so that it can be seen as the the algorithm proceeds.

- Before sorting A = [12, 11, 13, 5, 6] so A[0] = 12, A[1] = 11, A[2] = 13, A[3] = 5, A[4] =6.

- Iteration one, i = 1. Key = A[1] = 11. Sorted sublist is A[0] = 12. 11 < 12 so 12 shifts right one position, 11 inserted at index 0. Now A = [11, 12, 13, 5, 6]
- Iteration two, i = 2. Key = A[2] = 13. Sorted sublist is A[0, 1] = [11, 12]. 13 > 11 and 13 > 12 so 13 stays where it is. Now A = [11, 12, 13, 5, 6]
- Iteration three, i = 3. Key = A[3] = 5. Sorted sublist is A[0, 1, 2] = [11, 12, 13]. 5 < 11 and 5 < 12 and 5 < 13 so 5 gets placed at index 0 and elements 11, 12, 13 each shift one place to the right. Now A = [5, 11, 12, 13, 6].
- Iteration four, i = 4. Key = A[4] =6. Sorted sublist is A[0, 1, 2, 3] = [5, 11, 12, 13]. 6 > 5 but 6 < 11, 6 < 12 and 6 < 13 so 6 gets placed at index 1 and elements 11, 12, 13 each shift one place to the right. Now A = [5, 11, 12, 13, 6].

Another simple example is illustrated in Figure 4 [23]. The list to be sorted is A = [5, 9, 1, 2, 0]. Note that, at each iteration of the loop the correct position of the key must be found so that it can be placed correctly into the sorted part of the list. The only way for this to happen is if the key is compared to each of the sorted elements at each iteration of the loop. This gives rise to a nested loop structure - an outer for loop to move through the keys and an inner for/while to find the correct insertion point in the sorted part of the list.

Insertion sort is a very intuitive algorithm as it is similar to the way in which a person might sort a deck of cards [27]. At all times the sorted cards might be held in the dealer's left hand, and each new card is placed into its correct position there.

2.1.2 Time and space complexity

The complexity of insertion sort is summarised on p. 57 and 59 of [4]. It exhibits $O(n^2)$ time complexity in the worst case (elements are completely in reverse order) because the algorithm must iterate over the entire sorted part of the list to find the correct position at which to insert the key. In contrast, the best case scenario occurs if the elements are already sorted (no inversions exist) and so the time complexity is linear. Insertion sorts works well with

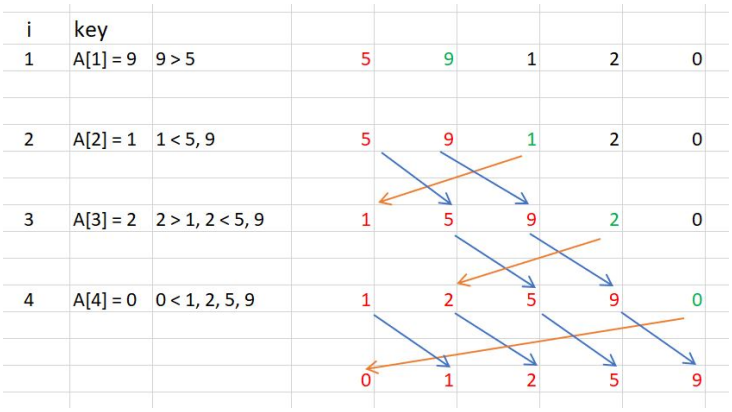

Figure 4: Insertion sort example 2.

a list that is already mostly sorted as the number of transpositions required is small. Reference [28] states that shift operations require only one third as much processing as exchange operations, which is part of the reason why insertion sort performs well in benchmarking studies.

There are n-1 passes of the for loop required to sort n elements in a list [28]. If the number of inversions is d, the total number of comparisons made is d + at most n-1. For a sorted list, this translates to the best case scenario and linear performance or $\Omega(n)$. Recall that, when considering complexity, constants are ignored so that n-1 approximates to n. A list with elements in full reverse order represents the worst case, with $(n-1)n/2$ inversions. The number of comparisons required is n-1 times this number of inversions, so complexity is of order $O(n^2)$ (keeping only the highest order terms). On average, with $(n-1)n/4$ inversions, complexity is also of order n^2, written as $\Theta(n^2)$.

The space complexity of insertion sort is $O(1)$ because, as an in-place sorting algorithm, it only requires a fixed amount of additional memory space (above that required for the program itself), that does not depend on the size of the input [29].

2.2 Merge Sort

I implemented merge sort in Python using code from Geeks for Geeks [30] and w3resource.com [31] as commented in the code.

Merge sort is an efficient comparison-based sort that was proposed by John van Neumann in 1945. It takes a recursive approach to divide the task of sorting into smaller problems. As explained on p. 81 of [4], it divides the list to be sorted into two equal (or close to equal) parts, each of which is separately sorted by recursively taking the same approach many times. The final step is to merge the two sorted parts of the list into one sorted list. It is a good algorithm to use if stable sorting is required or if one wishes to sort externally-stored data (p. 84 of [4]), but the standard implementation is not in-place with implications for the space complexity, although in-place implementations do exist.

2.2.1 How does it work?

An unsorted list containing n elements is recursively broken down into n lists each containing one element (each of these one-element lists is considered sorted). These one-element sorted sublists are considered the base case in the recursive process. The sublists are then repeatedly merged to create new (longer) sorted sublists until there is only one sorted list remaining [32, 33]. Each merge step involves comparisons so that every element is placed into the merged list at the correct position. This is often referred to as a top-down approach. I found a very good illustration of how the process works on Stack Abuse [34], and Figure 5 is reproduced directly from this website.

- Before sorting A = [4, 8, 7, 2, 11, 1, 3].
- First level of partitioning: A partitioned into lists L = [4, 8, 7, 2] and R = [11, 1, 3].
- Second level of partitioning: L becomes [4, 8] and [7, 2] R becomes [11, 1] and [3]. The list containing 3 is a base case - it's stored until final merge.
- Third level of partitioning: [4, 8] split into [4] and [8], [7, 2] split into [7] and [2], [11, 1] split into [11] and [1]. All base cases so merging can begin.
- First level of merging: $4 < 8$ so [4] and [8] merge to [4, 8]; $7 > 2$ so [7] and [2] merge to [2, 7], $11 > 1$ so [11] and [1] merge to [1, 11].

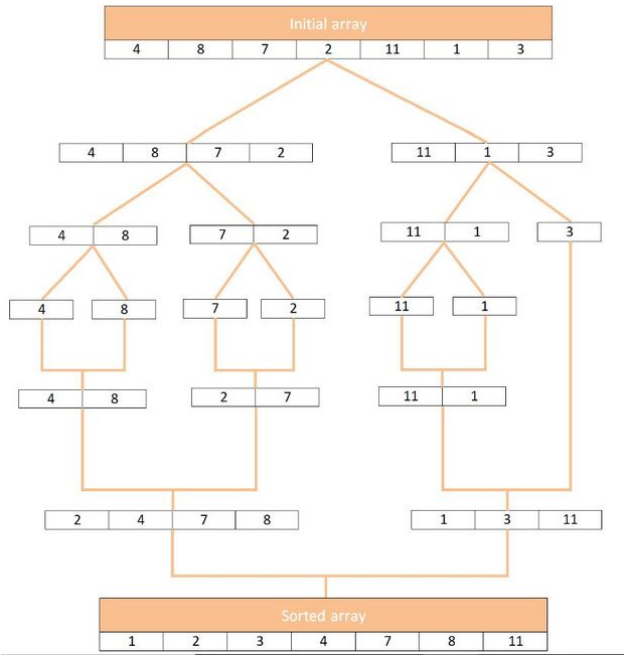

Figure 5: Merge sort example 1.

- Second level of merging: [4, 8] and [2, 7] merge to [2, 4, 7, 8]; [1, 11] and [3] merge to [1, 3, 11].
- Final merge: [2, 4, 7, 8] and [1, 3, 11] merge to [1, 2, 3, 4, 7, 8, 11] which is sorted.

Initially, I didn't really understand the order in which all of these processes happened. Figure 6 (from [30]) shows how the algorithm proceeds - the left half of the initial list is recursively sorted all the way down to a set of base cases which are merged to produce a sorted left half at every level of recursion. Then the right half of the list is recursively sorted down to a set of base cases which are merged to produce a sorted right half. I think of the algorithm as working from top down but also from left to right. The final step is to merge the sorted left and right halves of the list. Most of the work is done in this last step so it should be made as efficient as possible.

In the example shown in Figure 6, the list to be sorted is [38, 27, 43, 3, 9, 82, 10]. The sequence of steps is more like:

- Partition the list into [38, 27, 43, 3] and [9, 82, 10].

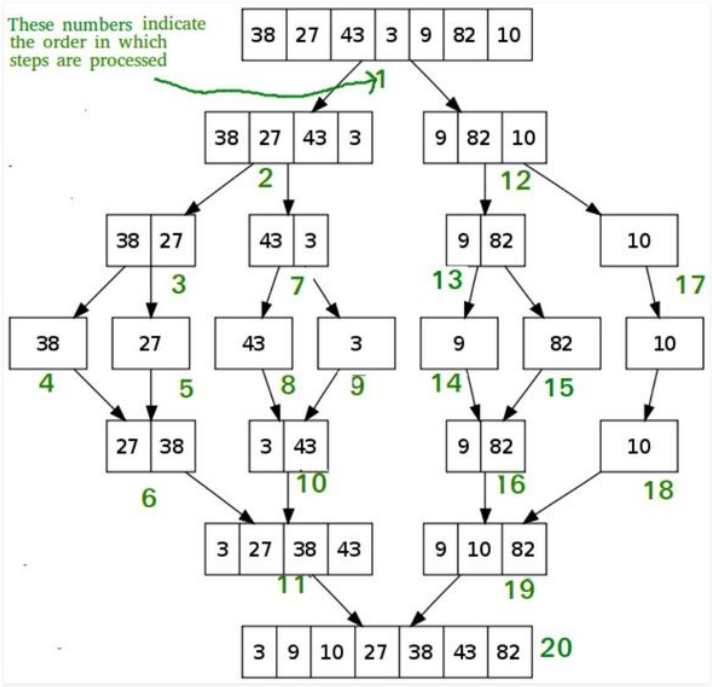

Figure 6: Sequence of steps in merge sort.

- List [38, 27, 43, 3] is partitioned into [38, 27] and [43, 3].
- List [38, 27] is partitioned into [38] [27], which are base cases of the recursive process.
- Lists [38] and [27] are merged (using comparison) to [27, 38].
- List [43, 3] is partitioned into [43] [3], which are base cases of the recursive process.
- Lists [43] and [3] are merged (using comparison) to [3, 43].
- Lists [27, 38] and [3, 43] are merged (using comparison) to [3, 27, 38, 43] (step 11 in the figure).
- The same processes occur with the right partition of the original list to produce [9, 10, 82] (step 19 in the figure).
- The final step is when [3, 27, 38, 43] and [9, 10, 82] are merged (using comparison) to produce [3, 9, 10, 27, 38, 43, 83].

2.2.2 Time and space complexity

Merge sort is one of the most time-efficient sorting algorithms, having a worst-case time complexity of $O(n \log n)$ as discussed on p. 141 of [1]. A proof of this is given in the solution to exercise 6.9 (c) on p. 412 of that book. The author states that a list of length n can be repeatedly split in half at most $1 + \log_2 n$ times; this is what happens during the partitioning part of the algorithm. So there are at most O(log (n)) stages to the algorithm. At each stage there are at most n comparisons carried out (merge steps), which gives a total worst-case time complexity of $O(n \log n)$. The time complexity is the same in the best and average cases because the algorithm always splits each list in half and takes linear time to merge (and sort) each half [30]. In the average case for example, there are still $\log n$ stages to the algorithm, with at least $n/2$ comparisons, leading to average case complexity which is the same as the worst case.

Merge sort is not an in-place sorting algorithm because it uses additional space to store temporary copies of arrays before merging them - p. 83 of [4]; it therefore has linear space complexity in the worst case, $O(n)$ [34]. In fact, in the worst case merge sort does about 39% fewer comparisons than quicksort does in the average case [32].

2.3 Counting Sort

I implemented counting sort using code from the Programiz website [35]; this is acknowledged as comments in my Jupyter notebook.

Counting sort is a non-comparison sort invented by Harold Seward in 1954. It can achieve linear running time in the best case, but only if certain assumptions are made about the data [36]. For this reason it is not very flexible. It exhibits best, average and worst case time complexities of n + k, where n is the size of the input and k is the range of possible key values (usually integers). The common implementation is stable and not in-place because it requires separate input and output arrays. An in-place implementation is possible but it is not stable [37]. What is its big advantage? Speed, but only if one is happy with the associated lack of flexibility. For example, the standard implementation cannot handle negative numbers although it can be extended to work for negative inputs; I realised this when I was testing my code with various arrays of integers. It is efficient if the range of input data is not significantly greater than the number of objects to be sorted [38].

2.3.1 How does it work?

This algorithm works by counting the number of elements that have distinct key values and then manipulating these counts to determine the position of each key in the output [39]. The procedure to sort an array of length n is:

1. Determine the key values k from the range of the input data (we assumed input ≥ 0 so in practice it's the maximum that is used).

2. Initialize an array *count* of length k+1. This will store the count of key values in the array and is initially all zeros.

3. Initialize an array *result* of length n to store the sorted output.

4. Iterate through the input array and store the number of times each key value occurs in the input.

5. Construct the sorted array *result* from the number of occurrences of each key value stored in *count*.

n = 7	k = 8								
A	4	2	2	8	3	3	1		
count	0	0	0	0	0	0	0	0	0
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
count	0	1	2	2	1	0	0	0	1

Figure 7: First part of counting sort.

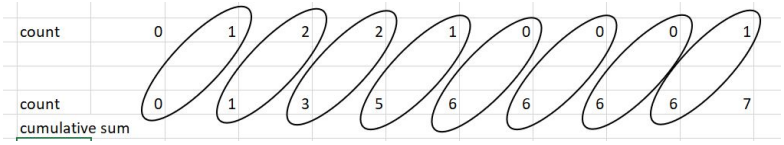

Figure 8: Calculating the cumulative sum of count array.

Consider the example of sorting an array A = [4, 2, 2, 8, 3, 3, 1]. n = 7 and k = max(A) = 8. Count array is initialized to zero and has length k + 1 = 9. Result array has size n = 7. First store the number of occurrences of each element in A at that index of count. So index 2 of count will hold 2 because there are two 2s in A; index 8 will hold 1 because there is only one 8 in A. Recall that indexing starts at zero. This first part is shown in Figure 7.

Next, store the cumulative sum of elements in the count array. This is illustrated in Figure 8. The elements inside the oval shapes are summed to find the cumulative sum directly below, for example, the second oval has 1 + 2 = 3.

The last step is to go through each element of the input A. At that index in the cumulative count find the element stored, subtract 1 from it and at that index in the result, place the element of A. That sounds complicated so I'll show the process in Figure 9. Take element 4 in A. count[4] = 6, place 4 at result[6-1=5] and decrement count[4] = 6-1=5. What about 8? count[8] = 7, place 8 at result[7-1=6] and decrement count[8] = 7-1=6. The decrementing of the count cumulative sum is to account for equal elements in A. Note that this example of the algorithm at work is not stable because the order of equal elements in A is reversed in result - see the placement of the two 2s for example. The solution is to iterate backwards through A and use the information that cumulative count gives us about the number of elements that are $\leq$ each element of A [40], thus preserving a mapping from

Figure 9: Calculating the result in counting sort.

each element from A to result.

2.3.2 Time and space complexity

Filling the count array takes $O(n)$ time and iterating through it takes $O(k)$ time. Iterating through the input to map each element to the sorted array also takes $O(n)$ time. So, the total time taken is $O(2n + k)$, which becomes $O(n+k)$ if coefficients are ignored as per our earlier discussion of complexity. The best, average and worst cases are identical. The space complexity of counting sort is also $O(n + k)$. For input instances in which the maximum key value is significantly smaller than the size of the input (n), counting sort can be highly space-efficient, as the only storage it uses other than its input and output arrays is the count array which uses space $O(k)$ [39].

2.4 Quicksort

I have chosen quicksort as my fourth algorithm for benchmarking. As acknowledged via comments in my Jupyter notebook, I adapted Python code from the Geeks for Geeks website [41].

Quicksort is an efficient comparison-based sort (like merge sort) and it is also a recursive divide and conquer algorithm (again like merge sort). It is not a stable algorithm as the order of equal elements in the input list is not necessarily preserved in the sorted output list. It is not an in-place algorithm either as it has a space complexity that is larger than $O(1)$ [42]. Quicksort

was developed in 1959 by Sir Tony Hoare when he was still in his 20s. It is one of the fastest sorting algorithms on average, and a good choice if one is interested in good average case behaviour [4]. A very short video explaining how it works can be found at [43].

2.4.1 How does it work?

On a high level, the steps of quicksort are:

1. Select one element in the array to be the *pivot*,
2. Partition the array so that elements $<$ the pivot are placed to its left, elements $>$ the pivot to its right. The pivot is now in its final position.
3. Recursively apply these two steps to each of the sub arrays.
4. The base case for the recursion is an empty array or one of length 1, which is assumed sorted.

Note that after step 2 above the full list has not been sorted, but it has been sorted in relation to the pivot element so that we won't have to compare elements in the left partition to those in the right partition, thus saving some time.

On investigating this algorithm one of the first questions I encountered was how to pick the pivot. Common approaches are to always pick the first or last element, use a random element, or pick the median of the elements in the array [44, 45]. Each option has implications for the running time of the algorithm. The best option is to pick a pivot that is the median of the list (in terms of size) so that the partition step will produce two sublists that are roughly equal in size [46]. For this project, for simplicity, I chose an implementation that always picks the last element in the array as the pivot.

As an example consider the list A = [9, 12, 10, 2, 17, 1, 6]. The algorithm steps are:

- Pivot = 6. Partition A into [1, 2] 6 [9, 12, 10, 17]. 6 is in its final position.
- Pivot = 2 (left sublist), partition into [1] 2. Pivot = 17 (right sublist), partition into [9, 12, 10] 17. 2, 6, 17 in their final positions.

- Left sublist is a base case so [1] is its own pivot. Keep going with the right sublist.
- Pivot = 10, partition into [9] 10 [12]. All base cases.
- Sorted list is all the pivots which are in their correct positions now [1, 2, 6, 9, 10, 12, 17].

The detail of how the algorithm arranges elements to the left and right of the pivot at the partition stage, and how it places the pivot in its final position, eluded me for a while, but I found an excellent explanation on [47] which I present in Figure 10. The example given is an array of integers [10, 25, 3, 50, 20]. The pivot is the last element in the list, 20, and it is red in the figure. The counter i progresses through the list selecting elements (in green) which are compared to the pivot. The counter p also progresses through the list, but how it does so depends on the results of comparisons between the pivot and each element. The steps are:

1. The pivot (20) is > test element (10), so p is incremented and the test element is swapped with the element at p, which in this case is itself.
2. The pivot (20) is < test element (25) so p is not incremented and no swaps are made.
3. The pivot (20) is > test element (3), so p is incremented and the test element is swapped with the element at p; 3 and 25 exchange places.
4. The counter i reaches its maximum value at the element before the pivot. The pivot (20) is < test element (50). Exchange the pivot and the element at position p so that 20 and 25 exchange places. The pivot is now it its final position. The process continues with recursive calls to partition the two sublists [10, 3] and [50, 25].

2.4.2 Time and space complexity

The time complexity of quicksort is $O(n^2)$ in the worst case and $n \log n$ in the average(Θ) and best (Ω) cases. The worst case can happen if the input is already almost sorted and the choice of pivot means that too many elements lie to the left of right of it [36, 45]; the extreme is for the list to be partitioned into an empty sublist and a sublist containing n-1 elements as on p. 69 of [4],

STEP												
	i											
1	10	25	3	50	20	20 > 10	10	25	3	50	20
	p					swop 10 with itself		*p*				
		i										
2	10	25	3	50	20	20 < 25	10	25	3	50	20
		p				do nothing		*p*				
			i									
3	10	25	3	50	20	20 > 3	10	3	25	50	20
		p				swop 3 with 25			*p*			
				i								
4	10	3	25	50	20	20 < 50	10	3	20	50	25
			p			swop 20 with 25						

Figure 10: Quicksort placement of elements in a sublist in relation to the pivot. The pivot is red and the test element is green. At the end of the process the pivot (and only the pivot) is positioned correctly in its final position.

leading to $O(n^2)$ behaviour because the algorithm must check all elements in the array to sort a single item, and do this n-1 times.

According to [42], although quicksort swaps elements within the input list, it requires $O(\log n)$ additional space to maintain the recursive call stack. This qualifies it as not in-place. As an aside, I've realised that this $\log n$ memory requirement seems to be a feature of recursive algorithms. However, [45] states that quicksort **is** an in-place algorithm under the broadest definition because it doesn't use any extra space for manipulating the input. On the other hand, in the lectures for this module we learnt that it was not in-place [3] so that is what I'll stick to. The worst case space complexity of quicksort is $O(\log n)$ [48].

2.5 Heapsort

The final algorithm I will benchmark is heapsort. The code I used was adapted from the Sanfoundry website [49] and this is acknowledged as comments in my Jupyter notebook.

Heapsort is another efficient comparison-based sort which is attractive if one is concerned about worst-case scenarios. It sorts without performing the

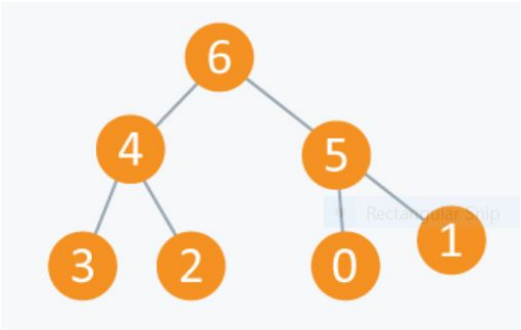

Figure 11: A binary tree.

n-1 comparisons one would normally require to find the largest element in a set of n elements [4], so it minimizes the number of direct comparisons which must be made. Heapsort was invented in 1964 by J. W. J. Williams, making use of a data structure called the heap. One type of heap is a binary tree, so called because the maximum number of children per node is two. The smallest element is always located at the root of the tree (p. 91 of [1]), making it particularly suitable for use in this sorting algorithm. Figure 11 shows an example of a binary tree taken directly from [50]. Notice that each node has greater value than any of its children, so $6 > 4$ or 5, $4 > 3$ or 2, and $5 > 0$ or 1. [50] also states that a heap containing n elements has the smallest possible height of $\log n$. Here, 6 is the *root* and 3, 2, 0, 1 are the *leaves*. Heapsort is an in-place algorithm but it is not stable.

2.5.1 How does it work?

The algorithm first places all the elements of the input array into a heap structure. It divides the input into a sorted part and an unsorted part and then repeatedly extracts the maximum element from the unsorted part to place it into the sorted list. It doesn't need to perform a linear search through the unsorted part of the list. Instead it takes advantage of the heap structure to easily access the maximum element in the heap. At a high level the steps to be followed are:

1. The input to be sorted is rearranged into a list representation of a binary heap by using the list indices - each index represents a node. The root is at index 0 with indices increasing towards the leaves. There are rules which determine the child indices of each parent.

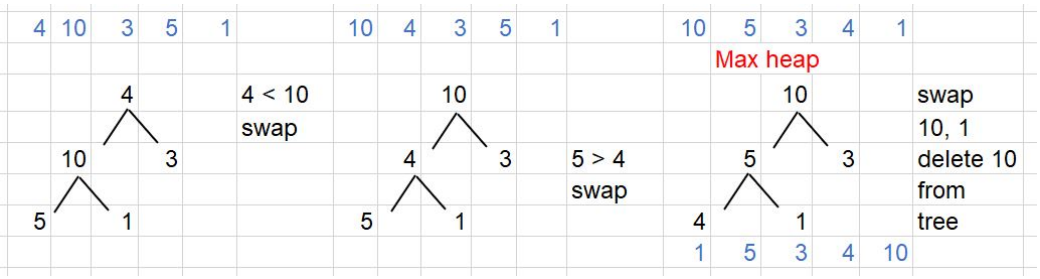

Figure 12: Building the heap and transforming to a max heap in heapsort.

2. The largest element in the heap (the root) is repeatedly removed and inserted into a sorted list. The heap is updated, following rules when removing elements from a heap in order to preserve its structure.

Consider the example of using heapsort on the list **A = [4, 10, 3, 5, 1]** as detailed in [51]. Start iterating through A to build the heap:

- A[0] = 4 is the parent at index 0; the left child of 4 is at index 2 x 0 + 1 = 1 A[1] = 10 and the right child of 4 is at index 2 x 0 + 2 = 2 A[2] = 3.
- Next calculate the left and right children of 10 and 4 starting on the left branch.
- 10 is at index 1; left child of 10 is at index 2 x 1 + 1 = 3 so A[3] = 5. The right child of 10 is at index 2 x 1 + 2 = 4 so A[4] = 1. There are no elements left in A so 3 has no children.
- A has been transformed into a binary heap as shown on the left of Figure 12. It is not yet a *max* heap.

Now transform this into a max heap where parent node $\geq$ child nodes.

- $10 \geq 5$ and 1 , ok.
- $4 < 10$ so swap nodes 4 and 10 in the tree and in the list: A = [10, 4, 3, 5, 1].
- $5 > 4$ so swap nodes 5 and 4 in the tree and in list: A = [10, 5, 3, 4, 1].

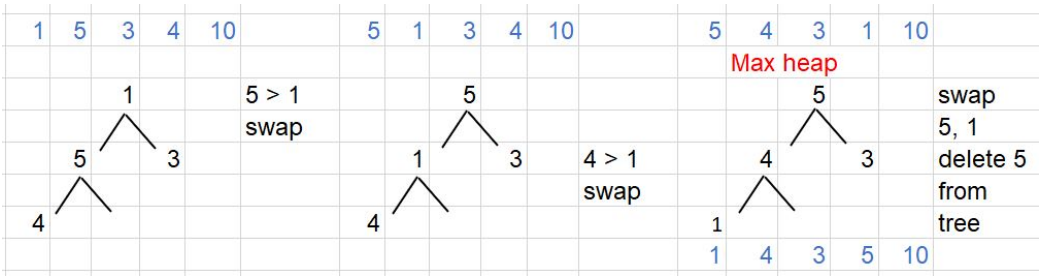

Figure 13: Transforming to a max heap to find max node in heapsort.

Figure 14: Transforming to a max heap to find max node in heapsort.

- Structure is now a max heap. Swap max (10) and min (1) nodes in the tree and delete the max node. Make the swap in A also: A = [1, 5, 3, 4, 10].

We now must transform our new heap into a max heap again where each parent node $\geq$ child nodes. The steps are shown in Figure 13.

- $5 \geq 4$ ok.
- $1 < 5$ so swap nodes 1 and 5 in the tree and in list: A = [5, 1, 3, 4, 10].
- $1 < 4$ so swap nodes 1 and 4 in the tree and in list: A = [5, 4, 3, 1, 10].
- Structure is now a max heap. Remove max node by swapping max (5) and min (1) nodes in tree and list, and delete max node from the tree: A = [1, 4, 3, 5, 10].

Transform into a max heap again where parent node $\geq$ child nodes. Steps shown in Figure 14.

Figure 15: Last step in heapsort.

- $4 > 1$ so swap nodes 1 and 4 in the tree and in list: A = [4, 1, 3, 5, 10].
- Structure is a max heap. Remove max node by swapping max (4) and min (3) nodes in tree and list and remove max node: A = [3, 1, 4, 5, 10].

The structure is already a max heap, so swap max (3) and min (1) nodes in tree and list and delete the max node: A = [1, 3, 4, 5, 10]. The steps are shown in Figure 15 and the list is now sorted.

2.5.2 Time and space complexity

On p. 143 of [1] it is stated that the total running time of heapsort is $O(n \log n)$ because there are n elements in the input list/tree and each operation to extract the max element from the tree takes logarithmic time. This is related to the max length/depth of a binary tree being $\log n$ (p. 66 [4]. This reference also points out that the running time advantage gained by implementing heapsort non-recursively versus recursively is lost as the size of the input increases (see Table 4-4 on p. 67). Heapsort is an in-place algorithm and, as such, requires no additional space - elements are sorted inside the original list so the space complexity is $O(1)$ and does not depend on the input in any way [52]. Because of this constant space complexity, and worst-case time complexity of $O(n \log n)$, embedded systems with real-time constraints or systems concerned with security often use heapsort, such as the Linux kernel [53].

2.6 Summary of algorithms described

In our lectures for this module we often compared the various sorting algorithms by looking at their time and space complexities and whether or not

Algorithm	**Type**	**Stable?**	**In-place?**	**Why me?**
Insertion sort	Simple comparison	Y	Y	Small n, nearly sorted, little code required
Merge sort	Efficient comparison	Y	N	Stable sorting
Counting sort	Non-comparison	Y	N	Speed for limited input instances
Quicksort	Efficient comparison	N	N	Good average-case behaviour
Heapsort	Efficient comparison	N	Y	Good worst-case behaviour

Table 2: Summary of algorithms for benchmarking.

they were stable and in-place [3, 54, 36]. This type of comparison is also frequently found in various websites and books, for example, [55, 56, 2]. In that spirit, Table 2 contains a high-level summary of some of the properties of the algorithms I studied for this project and Table 3 summarizes their time and space complexities. Of these five algorithms, note that insertion sort and heapsort are the only ones which are in-place, and this is reflected in their worst-case space complexity of $O(1)$. As to the question of what is the *best* sorting algorithm? The answer depends very much on the task at hand. Some of the criteria for choosing a sorting algorithm are summarized on p. 56 of [4]. Insertion sort is a good option if there are only a small number of elements to be sorted and if the list is already mostly sorted. For this reason, is it often used as the last stage in hybrid sorting algorithms where most of the hard work has already been done by a partner sorting algorithm. Introsort uses a combination of quicksort, insertion sort and heapsort [57] while Timsort combines the best of merge and insertion sort [58]. Merge sort is a good choice if stable sorting is required, quicksort if average-case behaviour is the primary concern, and heapsort if one is interested in the worst-case scenario. Counting sort is very fast, but it is not suitable for sorting large arrays (because of how the space complexity grows with n and k) and it can only be used to sort discrete values (either the list itself or the helper array of key values). So, it is not suitable for sorting strings for example. Hopefully, the properties of the various algorithms will be reflected in the results of my benchmarking, described in the remainder of this report.

Algorithm	**Time complexity**			**Space complexity**
	best	*average*	*worst*	*worst*
Insertion sort	n	n^2	$O(n^2)$	$O(1)$
Merge sort	$n \log n$	$n \log n$	$O(n \log n)$	$O(n)$
Counting sort	$n + k$	$n + k$	$O(n + k)$	$O(n + k)$
Quicksort	$n \log n$	$n \log n$	$O(n^2)$	$O(n)$
Heapsort	$n \log n$	$n \log n$	$O(n \log n)$	$O(1)$

Table 3: Time and space complexities of sorting algorithms chosen as a function of input size n. For counting sort, k is the range of possible key values.

insertion_sort

In [2]:
```
# Define an insertion sort function which operates on a list a
# Code adapted from: https://www.pythoncentral.io/insertion-sort-implementation-guide/
# and: https://tutorialedge.net/compsci/sorting/insertion-sort-in-python/

def insertion_sort(a):
    # Iterate through each element of a, starting at the second element (index = 1).
    for i in range(1,len(a)):
        # Key element is element at position i (first iteration key is at index=1).
        key = a[i]
        # Define a counter to check elements to the left of the key.
        position = i
        #print("i= ", i)
        # Set up a loop to compare the current key with the elements to its left i.e. the sorted portion of the list.
        while position > 0 and a[position - 1] > key:
            # Move elements one position to the right.
            a[position] = a[position - 1]
            # Decrement position counter.
            position -= 1
            # Insert the key value in its new position.
            a[position] = key
        #print("position= ", position, "key= ", key, ": ", a)
    # Return the sorted list in-place of unsorted input list.
    return a
```

Figure 16: Algorithm to perform insertion sort.

3 Implementation & Benchmarking

3.1 Instructions

The Python code for this project is contained in a single Jupyter notebook of filename **EDalySortingAlg.ipynb** [59]. Each sorting algorithm occupies its own cell, as does each stage of the benchmarking and results process. The code for the algorithms was modified to ensure that the sorted array was returned from each function. To view this file, download it and start Jupyter notebook in the folder containing the file. Use the command **Jupyter notebook** on the command line. I have placed the notebook, this report, and some results files in a GitHub repository called **comp-thinking-project** at https://github.com/elizabethdaly/comp-thinking-project

3.2 Coding the sorting algorithms in Python

3.2.1 insertion_sort

Python code was adapted from two websites: Python Central [23] and Tutorial Edge [24], as acknowledged as comments in the notebook. I added my own comments in order to explain what was happening at each step. This is the case for all five algorithms so I won't state that again. A screen shot of the code is shown in Figure 16 and a test of the algorithm at work in Figure 17.

```
In [3]: # Perform a simple test of insertion_sort on a small list.
        a = [7, 5, 2, 3, -2, 1]
        print("Unsorted list:", a)
        print("Sorted list:", insertion_sort(a))

        Unsorted list: [7, 5, 2, 3, -2, 1]
        Sorted list: [-2, 1, 2, 3, 5, 7]
```

Figure 17: Test of insertion sort.

3.2.2 merge_sort

I implemented merge sort in Python using code from Geeks for Geeks [30] and w3resource.com [31]. A screen shot of the code is shown in Figure 18 and a test of the algorithm at work in Figure 19.

3.2.3 counting_sort

I implemented counting sort using code from the Programiz website [35]. A screen shot of the code is shown in Figure 20 and a test of the algorithm at work in Figure 21.

3.2.4 quick_sort

I adapted Python code from the Geeks for Geeks website [41]. A screen shot of the code is shown in Figure 22 and a test of the algorithm at work in Figure 23.

3.2.5 heap_sort

The code I used was adapted from the Sanfoundry website [49]. A screen shot of the code is shown in Figure 24 and a test of the algorithm at work in Figure 25.

merge_sort

In [4]:
```
# Define a merge sort function which operates on a list a
# Code adapted from https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercis

def merge_sort(nlist):
    # Divide the list to be sorted into two smaller ones of roughly equal sizes.
    # Do this if the list length is > 1.
    if len(nlist) > 1:
        # Determine the midpoint of the list for splitting.
        mid = len(nlist) // 2
        # Left list is up to (but not including) midpoint.
        lefthalf = nlist[:mid]
        # Right half is from midpoint to end.
        righthalf = nlist[mid:]

        # Recursively apply the function on each half of the list.
        merge_sort(lefthalf)
        merge_sort(righthalf)

        # Counter to move through left, right and merged sublists.
        i = j = k = 0

        # Comparison steps during merge to ensure correct order of elements after merging.
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                nlist[k] = lefthalf[i]
                i += 1
            else:
                nlist[k] = righthalf[j]
                j += 1
            k += 1

        while i < len(lefthalf):
            nlist[k] = lefthalf[i]
            i = i + 1
            k = k + 1

        while j < len(righthalf):
            nlist[k] = righthalf[j]
            j = j + 1
            k = k + 1
    #print("Merging ",nlist)

    # Return the sorted list in-place of unsorted input list.
    return nlist
```

Figure 18: Algorithm to perform merge sort.

In [5]:
```
# Perform a simple test of merge_sort on a small list.
a = [14, 46, 43, 27, -1, 57, 41, 45, 21, 70]
print("Unsorted list:", a)
print("Sorted list:", merge_sort(a))
```

```
Unsorted list: [14, 46, 43, 27, -1, 57, 41, 45, 21, 70]
Sorted list: [-1, 14, 21, 27, 41, 43, 45, 46, 57, 70]
```

Figure 19: Test of merge sort.

counting_sort

In [6]:
```
# Code adapted from: https://www.programiz.com/dsa/counting-sort
# Commented print statements are for checking code.
# Note that this implementation (a standard one for counting sort) will not sort negative numbers, demonstrating
# one of the limitations of counting sort.

def counting_sort(array):
    size = len(array) # Length of array to be sorted.
    # print(size)
    # Empty array of same size to hold sorted values.
    output = [0] * size

    # Max value in the array so that key range can ve determined.
    upper = max(array)
    # print(lower, upper)

    # Create an empty count array of length (max value + 1).
    count = [0] * (upper + 1)
    # print(count)

    # Store count of each element at it's index in count array.
    for i in range(0, size):
        count[array[i]] += 1
    # print(count) # Check count array.

    # Store cumulative sum of elements of the count array.
    for i in range(1, upper + 1):
        count[i] += count[i - 1]
    # print(count) # Check cumulative counts.

    # For each element in array, use that as an index into cumulative count array.
    # Place that element at index = (cum count value - 1) in output array.
    i = size - 1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1

    for i in range(0, size):
        array[i] = output[i]

    # Return the new array containing the sorted elements (so not in-place).
    return output
```

Figure 20: Algorithm to perform counting sort.

In [7]:
```
## Perform a simple test of counting_sort on a small list.
## No negative numbers please counting_sort cannot handel them!
b = [6, 5, 3, 2, 1]
print("Unsorted list:", b)
print("Sorted list:", counting_sort(b))
```

```
Unsorted list: [6, 5, 3, 2, 1]
Sorted list: [1, 2, 3, 5, 6]
```

Figure 21: Test of counting sort.

quick_sort

In [8]:
```
# Code adapted from https://www.geeksforgeeks.org/python-program-for-quicksort/
# This code is contributed by Mohit Kumra
# It takes the last element as the pivot.
# It places the pivot element at correct position in the sorted array.
# It places all smaller elements (smaller than pivot) to left of pivot and all greater elements to right of pivot.

def partition(arr, low, high):
    i = low - 1         # Index of smaller element
    pivot = arr[high]     # pivot = last element in this implementation.

    for j in range(low , high):

        # If current element is smaller than or equal to pivot
        if   arr[j] <= pivot:

            # increment index of smaller element.
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i + 1)

# The main function that implements quick_sort
# arr = Array to be sorted,
# low  = Starting index,
# high = Ending index

# Function to do quick_sort
def quick_sort(arr, low, high):
    if low < high:

        # pi is partition index, arr[p] is now at right place
        pi = partition(arr, low, high)

        # Separately sort elements before partition and after partition
        # Done by repeated recursive calls to quick_sort
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)

    # Return the sorted list in-place of unsorted input list.
    return arr
```

Figure 22: Algorithm to perform quicksort.

In [9]:
```
## Perform a simple test of quick_sort on a small list.
q = [6, 5, 30, 0, 2, -1]
print(len(q))
print("Unsorted list:", q)
print("Sorted list:", quick_sort(q, 0, len(q)-1))
```

```
6
Unsorted list: [6, 5, 30, 0, 2, -1]
Sorted list: [-1, 0, 2, 5, 6, 30]
```

Figure 23: Test of quicksort.

heap_sort

In [10]:
```
# Code from: https://www.sanfoundry.com/python-program-implement-heapsort/

def heap_sort(alist):
    # Transform the input into a heap structure.
    build_max_heap(alist)

    for i in range(len(alist)-1, 0, -1):
        alist[0], alist[i] = alist[i], alist[0]
        # Turn this into a max heap, where each node has greater value than any of its children.
        max_heapify(alist, index = 0, size = i)
    # Return the sorted list in-place of unsorted input list.
    return alist

# Code to determine index of a parent node.
def parent(i):
    return (i - 1)//2
# Code to determine index of a left child node.
def left(i):
    return 2*i + 1
# Code to determine index of a right child node.
def right(i):
    return 2*i + 2

def build_max_heap(alist):
    length = len(alist)
    start = parent(length - 1)

    while start >= 0:
        max_heapify(alist, index = start, size = length)
        start = start - 1

def max_heapify(alist, index, size):

    l = left(index)
    r = right(index)

    # Finding the max element in the heap.
    if (l < size and alist[l] > alist[index]):
        largest = l
    else:
        largest = index

    if (r < size and alist[r] > alist[largest]):
        largest = r

    if (largest != index):
        # Swapping elements in the tree/list.
        alist[largest], alist[index] = alist[index], alist[largest]
        # Recursively apply the function to pairs of nodes to generate a max heap structure.
        max_heapify(alist, largest, size)
    #print(alist) # Check.
    return alist
```

Figure 24: Algorithm to perform heapsort.

In [11]:
```
## Perform a simple test of heap_sort on a small list.
h = [10, 17, 8, 9, 1, 5, -3]
print("Unsorted list:", h)
print("Sorted list:", heap_sort(h))
```

```
Unsorted list: [10, 17, 8, 9, 1, 5, -3]
Sorted list: [-3, 1, 5, 8, 9, 10, 17]
```

Figure 25: Test of heapsort.

3.3 Benchmarking the sorting algorithms

I followed the process outlined in the project instructions when benchmarking the sorting algorithms. I gradually built up to a single Jupyter notebook code cell that could test all algorithms over all input sizes, while performing 10 runs at each input size, in the following way:

- I first checked that each algorithm worked as expected on a small input array by using lots of print statements; these were commented out in the code when I was certain that the algorithms were working.
- I used randint from Python's random module to generate random arrays of integers of size n over a certain range. For example, for given input size n, the array of random integers covered the range 0 to n, inclusive.
- The Python time module was used to record start and end times for code snippets. All times were converted to ms for this report.
- I measured the running time (in ms) of each algorithm 10 times and calculated the average, for a single input array of size n. All running times reported here are the average of 10 runs. Each run of a sorting algorithm took place on a new random array of correct size.
- I measured the average running time of each algorithm for a set of input instances n.
- Finally, in a single cell I iterated over all five algorithms, over all input instances, and performed 10 runs per (algorithm, n) combination. The results are ouput to the console, stored in a Pandas dataframe, and written to a csv file called **benchmark_results.csv**. A typical output is shown in Figure 26, and the code to produce it in Figures 27, 28, 29, and 30. The notebook cell was just too long to capture in a single screenshot without zooming out a lot, so I did it in sections with line numbers.

The input sizes measured were n = [100, 250, 500, 750, 1000, 2000, 4000, 6000, 8000, 10000]. Note that insertion sort turned out to be very slow when sorting large arrays with n $>$ 4000. The list of n values can be edited to exclude large n if you want to run the notebook quickly.

	Size	Insertion	Merge sort	Counting	Quicksort	Heapsort
0	100	0.598	0.100	0.100	0.097	0.499
1	250	3.391	0.798	0.302	0.304	1.299
2	500	11.880	1.703	0.399	0.695	2.386
3	750	27.581	2.602	0.297	1.401	4.302
4	1000	49.231	3.738	0.495	1.789	5.582
5	2000	203.920	7.877	0.992	3.977	12.570
6	4000	822.857	17.560	2.090	8.770	27.243
7	6000	2016.373	25.732	3.394	12.779	42.077
8	8000	3546.052	35.604	4.294	17.949	57.423
9	10000	5546.753	45.669	5.387	23.442	74.551

Figure 26: Typical output of the benchmark code. All times in ms

```
# Intialise a Pandas dataframe to hold results.
n_values = [100, 250, 500, 750, 1000, 2000, 4000, 6000, 8000, 10000]
data = {'Size':n_values}

# Create DataFrame
df = pd.DataFrame(data)
# df # Check ok
```

Figure 27: Code cell to benchmark all algorithms over all input sizes n.

```
runs = 10
print("Size", "\t\t", end='')
for j in n_values:
    print(j, "\t\t", end='')
print('\n')

# Array to hold algorithm name.
alg_name = ["Insertion", "Merge sort", "Counting", "Quicksort", "Heapsort"]
# Array to hold function names.
algs = [insertion_sort, merge_sort, counting_sort, quick_sort, heap_sort]

# Iterate through sort algorithms and their names at same time with zip.
for c, alg in zip(alg_name, algs):
    # print(c, alg) # Check names match algorithm being called.
    print(c, "\t", end='')
    each_alg = []

    # Iterate through the input arrays of differenet size.
    for i in n_values:
        # print("Size", i) # Check

        # Initialize runtimes to zero
        runtimes = []

        # Run each sort algorithm 10 times.
        for r in range(runs):
```

Figure 28: Code cell to benchmark all algorithms over all input sizes n.

```
        # Run each sort algorithm 10 times.
        for r in range(runs):
            # Re-set the input so that its unsorted before each run.
            ip = random_array(i)
            # print("Size=", i, "run=", r, "IP=", ip) # Check ok

            # Call each sorting algorithm in turn.
            ########################################################################
            # Need to pass 3 args to quick_sort, only 1 to the others.
            if alg == quick_sort:
                # print("3 pars required")
                # Start time
                ts = time.time()
                alg(ip, 0, len(ip)-1)
                # Finish time
                tf = time.time()
                # print("Size=", i,"run=", r, "OP=", d(ip, 0, len(ip)-1)) # Check ok
            else:
                # print("1 par required")
                # Start time
                ts = time.time()
                alg(ip)
                # Finish time
                tf = time.time()
                # print("Size=", i,"run=", r, "OP=", d(ip)) # Check ok
            ########################################################################
```

Figure 29: Code cell to benchmark all algorithms over all input sizes n.

```
            ########################################################################

            # Calculate difference between finish and start times
            time_taken = tf - ts

            # Place time taken on each run into runtimes array
            runtimes.append(time_taken)

        # Average of 10 runs.
        avg_runtime = (np.average(runtimes)) * 1000
        print("%.3f" % avg_runtime, "\t\t", end='')

        # List of avg_runtimes for each size for a single algorithm.
        each_alg.append(avg_runtime)

    #print(each_alg) # Check ok

    # Store algorithm name and avg runtimes for each size in a list.
    times_all_n = {c: each_alg}
    # print(times_all_n) # Check it matches what's printed to console ok

    # Make a new dataframe out of the list.
    dfnew = pd.DataFrame(times_all_n)
    # Append it to existing results.
    df = pd.concat([df, dfnew], axis=1)

    print('\n')
```

Figure 30: Code cell to benchmark all algorithms over all input sizes n.

Size n	100	250	500	750	1000	2000	4000	6000	8000	10000
Insertion sort	0.598	3.391	11.880	27.581	49.231	203.920	822.857	2016.373	3546.052	5546.753
Merge sort	0.100	0.798	1.703	2.602	3.738	7.877	17.560	25.732	35.604	45.669
Counting sort	0.100	0.302	0.399	0.297	0.495	0.992	2.090	3.394	4.294	5.387
Quicksort	0.097	0.304	0.695	1.401	1.789	3.977	8.770	12.779	17.949	23.442
Heapsort	0.499	1.299	2.386	4.302	5.582	12.570	27.243	42.077	57.423	74.551

Table 4: Average running times (ms) of sorting algorithms as a function of input instance size n, laid out as specified in project description document.

3.4 Benchmarking results

Benchmarking code was run on a Lenovo ideapad 330S laptop. The processor is an Intel(R) Core (TM) i5-8250u CPU 1.6 GHz with 4 cores. The machine has a 64 bit operating system, is running Windows 10 Home, and has 8GB of RAM. I found that the benchmarking code runs much faster when I close down all other applications on my laptop. I also noticed that, although the results are similar from run to run, the actual runtime numbers are a little different. However, trends of fastest, slowest, and the ranking of algorithms don't change.

Initial results of the benchmarking process are shown in Figure 31 and in Table 4. What is immediately obvious is that insertion sort is *much* slower that all other algorithms studied when the size of the input is greater than a few hundred. Figure 32 shows the same results zoomed in on the y axis so that differences between the other four algorithms are clear. Without any analysis, it also looks as if insertion sort is increasing quadratically with input size n. This is the expected average case behaviour for this algorithm - randomly generated arrays of integers correspond to the average case. Best case would be if the arrays were already sorted, worst case if they were in complete reverse order. From these results we can conclude that:

1. Insertion sort is the slowest of the algorithms we tested.
2. For all input sizes n (bar perhaps n = 100) counting sort is the fastest algorithm.
3. If we rank the algorithms from fastest to slowest the order is: counting sort, quicksort, merge sort, heapsort and insertion sort.

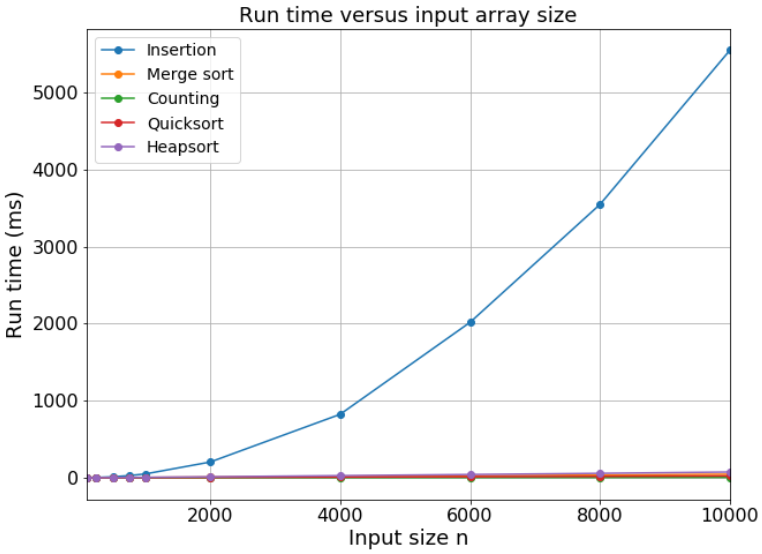

Figure 31: Run time performance of the algorithms as a function of input size n.

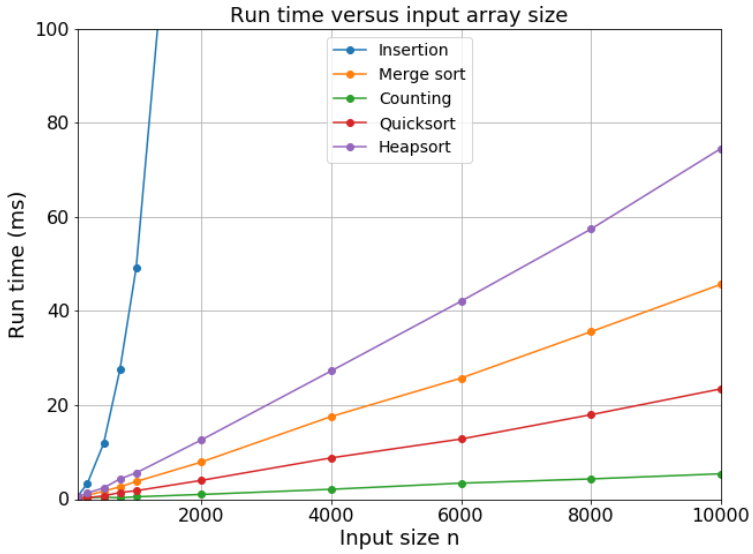

Figure 32: Performance of the algorithms with a zoom on y.

Algorithm	**p0**	**p1**	**p2**	R^2
Insertion sort	0.000057	-9.557e-3	-2.5621e-1	0.934
Merge sort	0.000342	0.236186		1.0
Counting sort	0.000537	0.024753		0.997
Quicksort	0.000175	0.034930		0.999
Heapsort	0.000557	0.161490		1.0

Table 5: Fitting of benchmarking results using NumPy polyfit.

3.4.1 Expected behaviour

Are these results in line with what we expect from out theoretical analysis of the five algorithms? Let's start with the run time complexity, which is summarized in Table 3. Here, we see that the average-case runtime for insertion sort should scale quadratically with input size n, for counting sort it should scale linearly with n, and for the other three algorithms the runtime is proportional to $n \log n$. I used the NumPy [60] function polyfit to fit the measured times to these theoretical curves. The results are summarized in Table 5. The fit was to a second-order polynomial for insertion sort $y = p0x^2 + p1x + p2$, and a first order polynomial for the other algorithms $y = p0x + p1$. Note that the x axis for fitting was transformed to $n \log n$ for merge sort, quicksort, and heapsort so that a polynomial of order one could be fit to the data. The x axis is therefore different for these three algorithms in the plots that follow. It's clear that the results are as expected for each of my chosen sorting algorithms [4, 1].

3.5 Performance on sorted and reversed arrays

Out of interest I also performed some benchmarking on best and worst case scenarios; recall that, in general, the best case occurs if an input array is already sorted and the worst case is if the input array is in reverse order. I performed this test for only a few input array sizes, mainly because I ran into a problem running quicksort on already-sorted arrays, where the maximum recursion depth was reached for n $>$ 2000 approximately.

To test with an already-sorted input, I generated a random array of integers as before and used my fastest algorithm (counting sort) to sort it before passing it to each of the five algorithms for benchmarking. I also generated

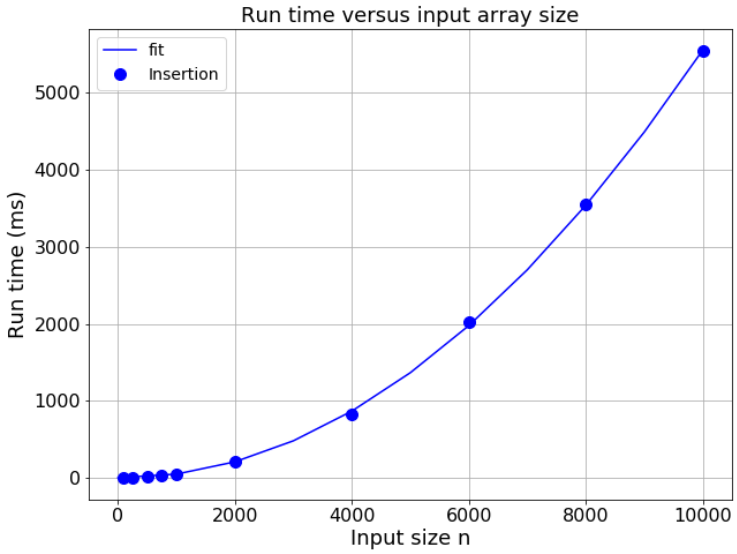

Figure 33: Fitting insertion sort run times to a second-order polynomial.

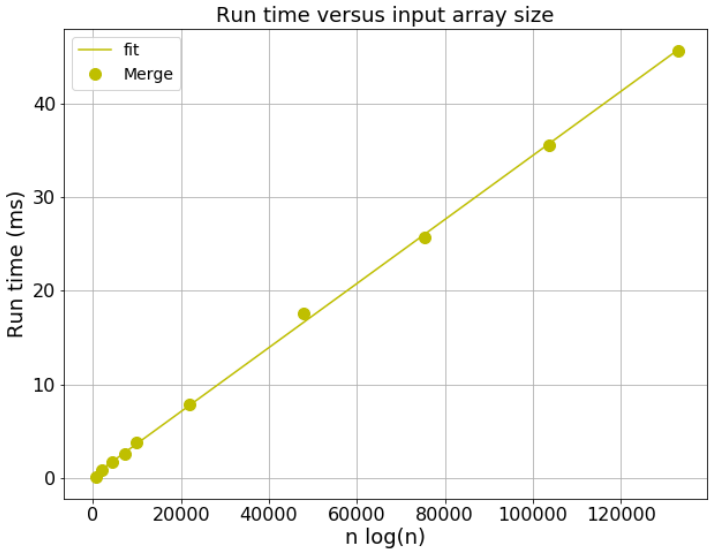

Figure 34: Fitting merge sort run times to a first-order polynomial. Note the x axis.

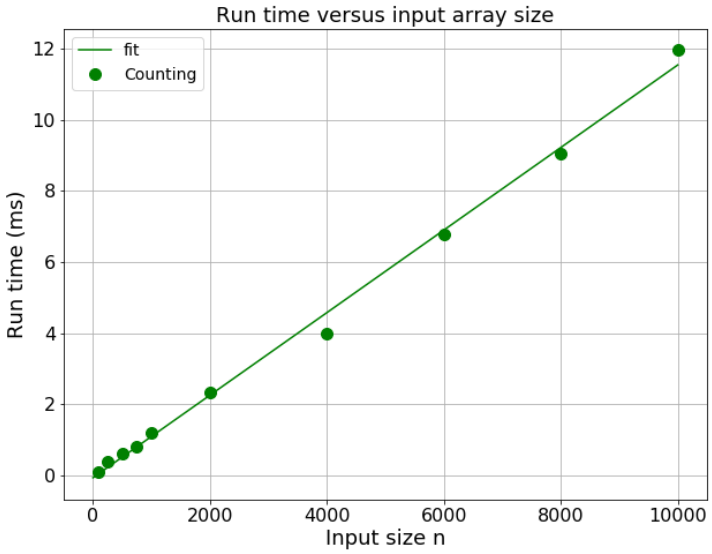

Figure 35: Fitting counting sort run times to a first-order polynomial.

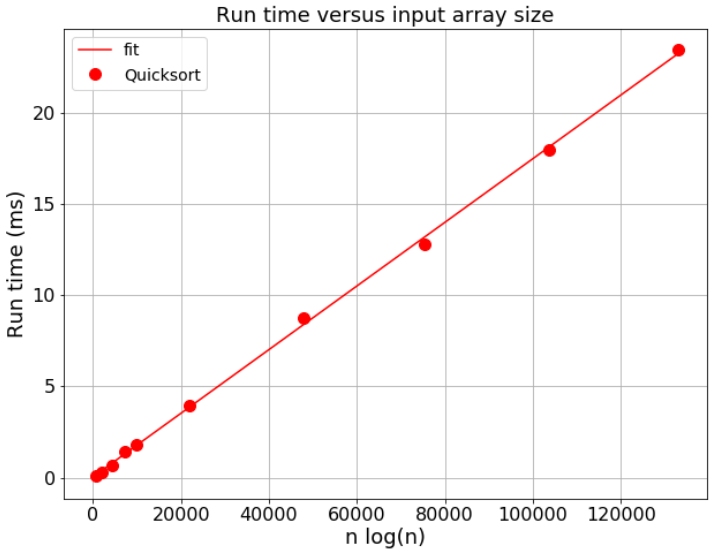

Figure 36: Fitting quicksort run times to a first-order polynomial. Note the x axis.

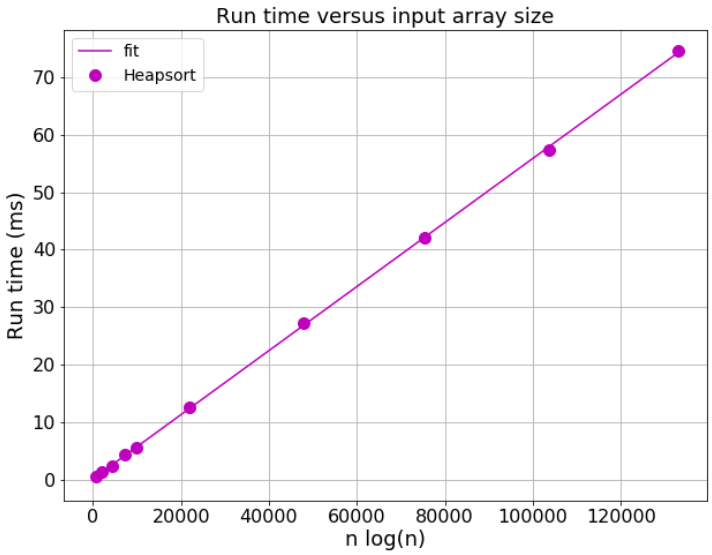

Figure 37: Fitting heapsort run times to a first-order polynomial. Note the x axis.

a random array of integers as before, used a fast algorithm (counting sort) to sort it, reversed it in place with Python's *reverse()* command, and then passed it to each of the five algorithms for benchmarking. Each run of a sorting algorithm started with a new random array of integers and described in section 3.3. Microsoft Excel was then used to to compare numbers and produce the comparison plots in the section.

I found that there was very little difference between the performance of some of the algorithms when applied to sorted, random, and reversed arrays: merge sort, counting sort, and heapsort. The results for this part of the benchmarking are shown in Figures 38, 39, and 40. This is exactly the behaviour we expect for these three algorithms. Table 3 shows that merge and heapsort have $n \log n$ complexity in the best, average and worst cases while counting sort has $n+k$ complexity in all three scenarios. So, the runtime performance doesn't really change with different input array characteristics. Counting sort is the fastest algorithm over this range of n, with merge next, and heapsort coming third.

The situation was very different for insertion sort (Figure 41) and quick-

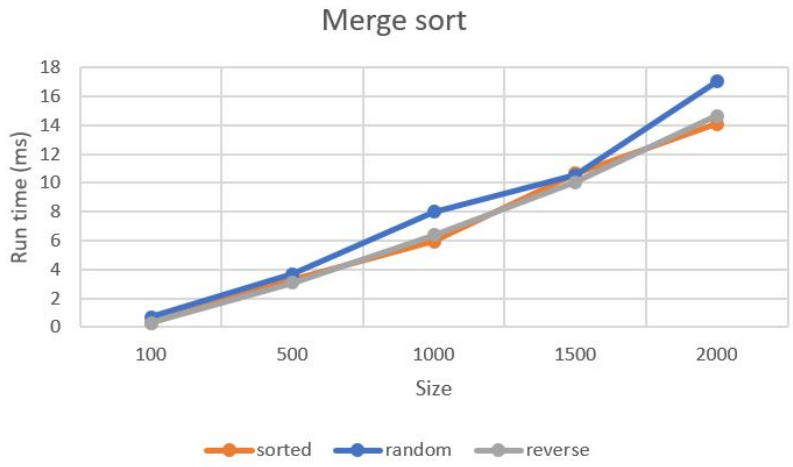

Figure 38: Merge sort run times for different input instance types.

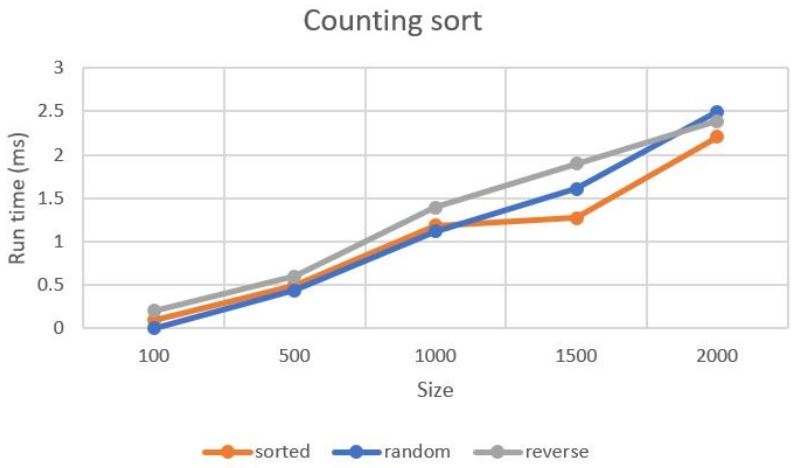

Figure 39: Counting sort run times for different input instance types.

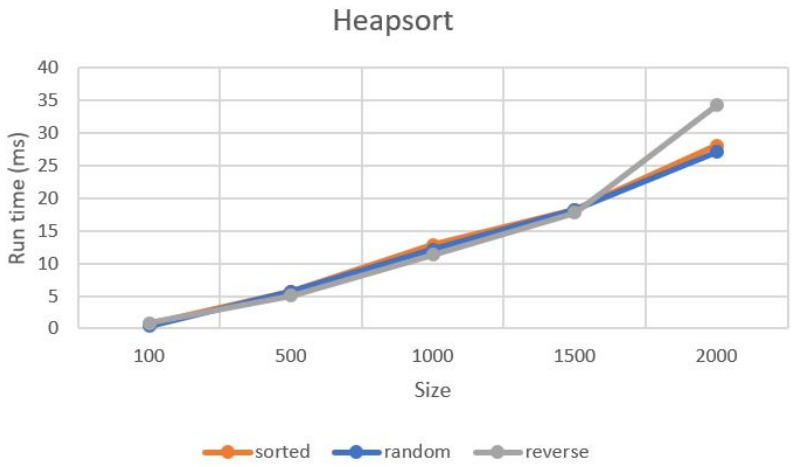

Figure 40: Heapsort run times for different input instance types.

sort (Figure 42) with theoretical time complexities given in Table 3. Insertion sort has worst and average time complexity of n^2 but linear n dependence in the best case. A simple best case situation is if the input is already sorted, so that only one comparison operation happens at each iteration, while a worst case occurs with a reverse-sorted array [29]. It's clear from Figure 41 that insertion sort is really fast with a sorted input, faster even than counting sort. It's slowest with a reverse-sorted input. The results we see here seem to make sense.

For quicksort the worst case time complexity is n^2 and that is what we appear to be seeing in the figure for arrays that are either already sorted or in complete reverse order. It's actually the worst performing algorithm with an already-sorted input. Of course, the choice of pivot plays a large part in the performance of quicksort not just the characteristics of the input array. The worst possible choice is to repeatedly have partitions that are one element long. This happens if the pivot is always the first or last (our case) element with a list that is either sorted or in reverse order [61]. Again, the results seen here make sense. Best case behaviour could be achieved by re-coding the algorithm to make a better choice of the pivot.

Finally, I include the run times for each algorithm with sorted, random, and reversed arrays in Table 6.

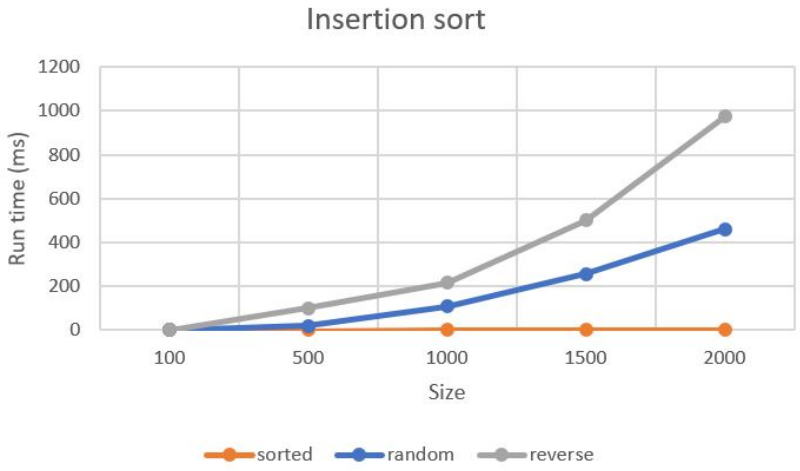

Figure 41: Insertion sort run times for different input instance types.

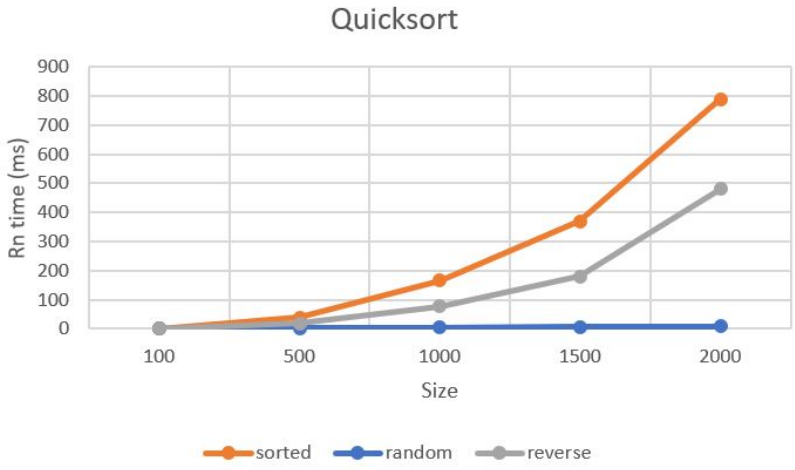

Figure 42: Quicksort run times for different input instance types.

Size	Sorted Insertion	Merge sort	Counting sort	Quicksort	Heapsort
100	0.000	0.299	0.097	1.694	0.697
500	0.000	3.306	0.499	39.691	5.685
1000	0.276	5.958	1.186	165.350	12.866
1500	0.413	10.676	1.272	369.960	18.251
2000	0.499	14.071	2.204	788.804	28.042
Size	**Random Insertion**	**Merge sort**	**Counting sort**	**Quicksort**	**Heapsort**
100	1.138	0.698	0.003	0.100	0.399
500	20.553	3.687	0.439	1.995	5.694
1000	107.740	7.978	1.115	4.005	12.167
1500	255.185	10.548	1.610	5.944	18.237
2000	461.521	17.023	2.496	8.871	27.059
Size	**Reversed Insertion**	**Merge sort**	**Counting sort**	**Quicksort**	**Heapsort**
100	1.743	0.298	0.202	1.295	0.799
500	99.202	3.094	0.600	18.962	5.087
1000	217.042	6.385	1.392	77.594	11.279
1500	500.315	10.053	1.895	179.321	17.736
2000	975.219	14.660	2.389	479.952	34.266

Table 6: Benchmarking with sorted, random, and reversed inputs. Time in ms.

4 Conclusion

For this project we were asked to benchmark the running time of five different sorting algorithms as a function of input array size. The input arrays consisted of random integers, generated using the Python **randint** function from the **random** module. The array sizes I tested were n = [100, 250, 500, 750, 1000, 2000, 4000, 6000, 8000, 10000]. The Python **time** module was used to calculate the running time of algorithms (in ms) by measuring the time immediately before and after the call to the particular sorting algorithm. Each algorithm was measured 10 times for each value of n, and the average of those 10 runs was calculated to be the running time. I chose to benchmark insertion sort, merge sort, counting sort, quicksort, and heapsort.

For these random arrays of integers, I found that counting sort was the fastest algorithm for all n values, apart from n = 100 where we can't really distinguish between it and merge sort. For all values of n the sorting algorithms could be ranked in order, from fastest to slowest, as: counting sort, quicksort, merge sort, heapsort, and insertion sort. With n = 10000 quicksort took 4.6 times longer to run than counting sort, merge sort took 9.2 times longer, heapsort took 15 times longer, and insertion sort took 1109 times longer to run than counting sort. The caveat here is that counting sort cannot handle negative integers. The results were presented in section 3.4.

In section 3.4.1, the measured running time as a function of n was compared to the theoretical behaviour for each algorithm by fitting the measured values using NumPy **polyfit**. I found that insertion sort run time increased quadratically with n, and for counting sort it increased linearly with n, as expected. The average case behaviour predicts $n \log n$ runtime as a function of n for merge sort, quicksort, and heapsort. This was also confirmed using **polyfit** to the transformed x axis data.

I also performed some benchmarking on best, average, and worst case scenarios as described in section 3.5. Here, I assumed that the best case for sorting was represented by asking each algorithm to sort an already-sorted array; the worst case was represented by asking each to sort a reverse-sorted array. The average case, as before, was represented by a random array of integers. I performed these tests for a reduced set of input sizes: n = [100, 500, 1000, 1500, 2000]. The maximum input size was limited here because the maximum recursion depth was reached with quicksort for n *gte* 2000 and an already sorted array. I found that there was no significant differences between best, average, and worst cases for merge sort, counting sort, and heapsort.

This is to be expected because, according to Table 3, the time complexity of these algorithms is the same for all three scenarios. Insertion sort was fastest for an already-sorted array (expected linear dependence on n) and slowest if the input was reverse-sorted (expected quadratic dependence on n). In fact, for a sorted input, insertion sort was even faster than counting sort over this range of n; recall that counting sort has been my fastest sorting algorithm so far. Quicksort was possibly the most interesting algorithm tested under best, average, and worst cases. Of course, the choice of pivot is an important factor in determining exactly what is the best and worst case for this algorithm, not just the structure of the input array. The pivot was always the last element in the input array in my implementation of quicksort. This means that sorted or reversed arrays both represent a worst case scenario and quadratic dependence on n is expected. That is what I saw, with the sorted input actually taking longest to run, reverse-sorted a bit less, and random input was the fastest.

In conclusion, counting sort was the fastest algorithm I tested when the input consisted of a random array of positive integers. Insertion sort was the slowest for this task, but the fastest if the input array was already sorted. Therefore, it is important to consider carefully the task at hand when choosing the *best* algorithm for it.

References

[1] David Harel with Yishai Feldman. *Algorithmics The Spirit of Computing*. Pearson Education Limited, Harlow, England, 3rd edition, 2004.

[2] Essential programming: Sorting algorithms. `https://towardsdatascience.com/essential-programming-sorting-algorithms-76099c6c4fb5`. Accessed: 2020-03-17.

[3] Sorting algorithms part 1. 46887 Computational Thinking with Algorithms lecture notes. Semester 3: H. Dip. Data Analytics.

[4] George T Heineman, Gary Pollice, and Stanley Selkow. *Algorithms in a Nutshell A PRACTICAL GUIDE*. O'Reilly Media Inc., CA, United States, 2nd edition, 2016.

[5] Hackerearth: Time and space complexity. `https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/`. Accessed: 2020-03-26.

[6] Geeks for geeks: What does 'space complexity' mean? `https://www.geeksforgeeks.org/g-fact-86/`. Accessed: 2020-03-26.

[7] studytonight: Space complexity of algorithms. `https://www.studytonight.com/data-structures/space-complexity-of-algorithms`. Accessed: 2020-03-26.

[8] Afteracademy: Time and space complexity analysis of algorithm. `https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm`. Accessed: 2020-03-27.

[9] Anaysing algorithms part 2. 46887 Computational Thinking with Algorithms lecture notes. Semester 3: H. Dip. Data Analytics.

[10] Wikipedia: Big o notation. `https://en.wikipedia.org/wiki/Big_O_notation`. Accessed: 2020-03-28.

[11] Khan academy: Big o notation. https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation. Accessed: 2020-03-28.

[12] Medium.com: A simplified explanation of the big o notation. https://medium.com/karuna-sehgal/a-simplified-explanation-of-the-big-o-notation-82523585e835. Accessed: 2020-03-28.

[13] Towards data science: Big o. . Accessed: 2020-03-28.

[14] Medium.com: https://medium.com/@swainson/the-3-steps-in-an-empirical-approach-for-analyzing-the-run-time-of-algorithms-f48d90e18908. https://medium.com/@swainson/the-3-steps-in-an-empirical-approach-for-analyzing-the-run-time-of-algorithms-f48d90e18908. Accessed: 2020-03-28.

[15] An empirical study of algorithms performance in implementations of set in java. https://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2012/0565_Kucakatal.pdf. Accessed: 2020-03-28.

[16] stackoverflow: Sorting in place. https://stackoverflow.com/questions/16585507/sorting-in-place. Accessed: 2020-03-28.

[17] Tutorials point: Data structure - sorting techniques. https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm. Accessed: 2020-03-29.

[18] Stack exchange: Explanation of satellite data from a programmers perspective. https://softwareengineering.stackexchange.com/questions/160197/explanation-of-satellite-data-from-a-programmers-perspective. Accessed: 2020-03-30.

[19] Wikipedia: Comparison sort. https://en.wikipedia.org/wiki/Comparison_sort. Accessed: 2020-03-29.

[20] University of san francisco cs: Visualization of different comparison sorts. https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html. Accessed: 2020-03-29.

[21] University of wisconsin–madison cs: Non-comparison sorting algorithms. http://pages.cs.wisc.edu/~paton/readings/Old/fall08/LINEAR-SORTS.html. Accessed: 2020-03-30.

[22] Javarevisited: Difference between comparison (quicksort) and non-comparison (counting sort) based sorting algorithms? https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html#ixzz6IBaW89ro. Accessed: 2020-03-30.

[23] Python central: Insertion sort: A quick tutorial and implementation guide. https://www.pythoncentral.io/insertion-sort-implementation-guide/. Accessed: 2020-03-17.

[24] Tutorialedge.net: Implementing the insertion sort algorithm in python. https://tutorialedge.net/compsci/sorting/insertion-sort-in-python/. Accessed: 2020-03-17.

[25] Geeks for geeks: Python program for insertion sort. https://www.geeksforgeeks.org/python-program-for-insertion-sort/. Accessed: 2020-03-17.

[26] Geeks for geeks: Insertion sort. https://www.geeksforgeeks.org/insertion-sort/. Accessed: 2020-03-31.

[27] Stack abuse: Insertion sort in python. https://stackabuse.com/insertion-sort-in-python/. Accessed: 2020-03-17.

[28] Brad Miller and David Ranum. *Problem Solving with Algorithms and Data Structures using Python*. Franklin, Beedle & Associates Inc, Wilsonville, United States, 2nd edition, 2013. Available online at runestone.academy/runestone/books/published/pythonds/index.html.

[29] Wikipedia: Insertion sort. https://en.wikipedia.org/wiki/Insertion_sort. Accessed: 2020-04-01.

[30] Geeks for geeks: Merge sort. https://www.geeksforgeeks.org/merge-sort/. Accessed: 2020-03-20.

[31] Python data structures and algorithms: Merge sort. https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-8.php. Accessed: 2020-03-20.

[32] Wikipedia: Merge sort. https://en.wikipedia.org/wiki/Merge_sort. Accessed: 2020-03-20.

[33] Merge sort: A quick tutorial and implementation guide. https://www.pythoncentral.io/merge-sort-implementation-guide/. Accessed: 2020-03-20.

[34] Stack abuse: Merge sort in python. https://stackabuse.com/merge-sort-in-python/. Accessed: 2020-03-20.

[35] Programiz: Counting sort algorithm. https://www.programiz.com/dsa/counting-sort. Accessed@ 2020-03-21.

[36] Sorting algorithms part 3. 46887 Computational Thinking with Algorithms lecture notes. Semester 3: H. Dip. Data Analytics.

[37] stackoverflow: Is counting sort in place & stable or not? https://stackoverflow.com/questions/30222523/is-counting-sort-in-place-stable-or-not. Accessed@ 2020-04-02.

[38] Geeks for geeks: Counting sort. https://www.geeksforgeeks.org/counting-sort/. Accessed@ 2020-03-21.

[39] Wikipedia: Counting sort. https://en.wikipedia.org/wiki/Counting_sort. Accessed@ 2020-04-03.

[40] Mit computer science & artificial intelligence laboratory: Counting sort. https://courses.csail.mit.edu/6.006/spring11/rec/rec11.pdf. Accessed@ 2020-04-03.

[41] Geeks for geeks: Python program for quicksort. https://www.geeksforgeeks.org/python-program-for-quicksort/. Accessed: 2020-03-22.

[42] stackoverflow: Is quicksort in-place or not? https://stackoverflow.com/questions/22028117/is-quicksort-in-place-or-not. Accessed: 2020-04-02.

[43] Youtube quicksort. https://youtu.be/HDQd6_0TJIE. Accessed: 2020-03-22.

[44] Hackernoon: Algorithms explained quicksort. https://hackernoon.com/algorithms-explained-quicksort-324305b8757b. Accessed: 2020-03-22.

[45] Geeks for geeks: Quicksort. https://www.geeksforgeeks.org/quick-sort/. Accessed: 2020-04-02.

[46] Medium.com: Pivoting to understand quicksort [part 1]. https://medium.com/basecs/pivoting-to-understand-quicksort-part-1-75178dfb9313/. Accessed: 2020-04-02.

[47] www.log2base2.com: Quicksort. https://www.log2base2.com/algorithms/sorting/quick-sort.html. Accessed: 2020-04-02.

[48] Eric rowell: Big o cheatsheet. https://www.bigocheatsheet.com/. Accessed: 2020-04-02.

[49] Sanfoundry: Python program to implement heapsort. https://www.sanfoundry.com/python-program-implement-heapsort/. Accessed: 2020-03-25.

[50] Hackerearth: Heaps/priority queues. https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/tutorial/. Accessed: 2020-04-03.

[51] Geeks for geeks: Heapsort. https://www.geeksforgeeks.org/heap-sort/. Accessed: 2020-04-03.

[52] stackoverflow: Why does heap sort have a space complexity of o(1)? https://stackoverflow.com/questions/22233532/why-does-heap-sort-have-a-space-complexity-of-o1. Accessed: 2020-04-08.

[53] Wikipedia: Heapsort. https://en.wikipedia.org/wiki/Heapsort. Accessed: 2020-03-25.

[54] Sorting algorithms part 2. 46887 Computational Thinking with Algorithms lecture notes. Semester 3: H. Dip. Data Analytics.

[55] Scanftree.com: Complexity comparison of sorting algorithms. https://scanftree.com/Data_Structure/time-complexity-and-space-complexity-comparison-of-sorting-algorithms. Accessed: 2020-04-01.

[56] Geeks for geeks: Analysis of different sorting techniques. https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/. Accessed: 2020-03-29.

[57] Geeks for geeks: Introsort or introspective sort. https://www.geeksforgeeks.org/introsort-or-introspective-sort/. Accessed: 2020-04-08.

[58] Wikipedia: Timsort. https://en.wikipedia.org/wiki/Timsort. Accessed: 2020-04-08.

[59] Project jupyter. https://jupyter.org/.

[60] Numpy scientific computing package. https://numpy.org/.

[61] Stack exchange: What makes for a bad case for quick sort? https://softwareengineering.stackexchange.com/questions/

257002/what-makes-for-a-bad-case-for-quick-sort. Accessed: 2020-04-13.