CTA Project Assignment

Higher Diploma Software Development - GMIT

David Faulkner

[redacted]

Contents

Introduction

Sorting is the process of re-arranging an array of items into the correct order depending on the array type. In the case of an integer array, this is generally placing lowest valued items at the start progressively increasing in value.

There are many important characteristics to look at and understand before choosing the most appropriate sorting algorithm for a certain sorting task. Some of the questions that should be asked before selecting the appropriate sorting algorithm include:

- What is the size of the input array?
- Is it nearly or partially sorted already?
- Is there a need for a simple algorithm that is easy to read and write?
- What is the range of the items in the array?
- Should the algorithm be comparison or non-comparison based?
- Is the use of recursion a problem due to the amount of stack space it uses?
- Is a stable sorting algorithm required?
- Does the sorting need to be in-place?

An understanding of time and space complexities and performance is an important aspect of analysing sorting algorithms. The performance of a sorting algorithm is measured by many factors as stated above, but it is greatly impacted by time and space complexity. To have an efficient sorting algorithm, system resources must be used efficiently. A sorting algorithm that has good performance should use the least amount of additional memory as possible; this is measured in space complexity. The running time of an algorithm given the input size is measured in time complexity and this should also be as low as possible to achieve a good performance. However, it can be difficult to get both a good time and space complexity measurement of an algorithm at the same time where has to be “a trade-off between time and space. “If you want to reduce the space, then the time may increase”, (Time and Space Complexity Analysis of Algorithm, 2021), so there is a compromise required. Choosing the appropriate sorting algorithm needs to be thoughtful as some of these algorithms may perform well with input sizes that are lower even though they are very inefficient for large data sets.

In some circumstances it may be important that equally valued items in an array do not change order sequence during the sorting process. Sorting algorithms that do not change the order of identical values are stable. Some stable sorting algorithms that I will examine are Bubble sort, Merge sort, Counting sort and Insertion sort. An example of an un-stable sorting algorithm is Shell sort.

Some sorting algorithm programs require additional space to sort an array of items, using temporary storage to allow a switch to take place. If an algorithm that does not need extra storage space for the sorting process, then this is called in-place sorting. In-place sorting algorithms are useful in cases where data sets have huge volumes, and it is not efficient to use any additional space. Examples of in-place sorting algorithms are Bubble sort, Insertion sort and Shell sort. Sorting algorithms which do not work in-place are Counting sort and Merge sort.

Array inputs to be sorted may come in different types, some numerical (integers, doubles, real numbers, or numbers for time etc.) and others may need to be sorted alphabetically or lexicographically, in ascending or descending order. A comparison function would provide order for data set items that do not have natural order. Comparator functions are used to return a certain result if $x < y$, $x = y$ or $x > y$. These determine if a value is “less than”, “equal to”, or “greater than”.

The main difference between comparison and non-comparison-based sorting is the speed of the running time. Comparison based soring algorithms do not make any assumptions about the data set and use comparators to determine the relationship between data set items and define the ordering. Non-comparison algorithms are known as linear sorting algorithms as they have a time complexity of O(n) (Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) based Sorting Algorithms?, 2021). However, these algorithms can only be used to sort integer valued data sets. “A fundamental result in algorithm analysis is that no algorithm that sorts by comparing elements can do better than $n \log n$ performance in the average or worst cases” (Part 1 Sorting Algorithms, Patrick Mannion GMIT). This states that non-comparison algorithms are the most efficient when dealing with average and worst-case time complexities, which make them more reliable as a sorting solution. These sorting algorithms have the quickest running time, and I will look to prove this in my examination of comparison and non-comparison-based sorting algorithms.

Sorting Algorithms

Bubble Sort

The simple comparison-based sorting algorithm that I have chosen is Bubble Sort.

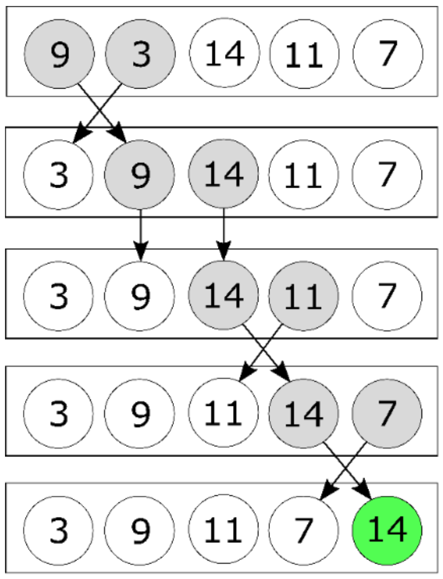
Although this sorting algorithm is okay to use when input array sizes are small or if the array is already sorted but still needs a check which would result in a best-case time complexity of O(n), it is evident that as the input size of an array to be sorted is increased, the efficiency of the sort is greatly reduced.

It is quite a simple algorithm to implement as it is a comparison of adjacent items. As a result, it is required to loop with a nested for-loop through the array multiple times and switches the larger items with the smaller items in-place until it is sorted, which proves to be extremely inefficient in larger arrays.

After every loop through the array, the largest of the items will be in its correct place and is ignored for the remainder of the program. Therefore, there are (n – 1) items to iterate through on each run, and n is reduced by 1 after each iteration. Therefore, the worst-case scenario where there is a reverse-sorted array or in an average case scenario, the time complexities are O(n^2) which would result in quadratic orders of growth.

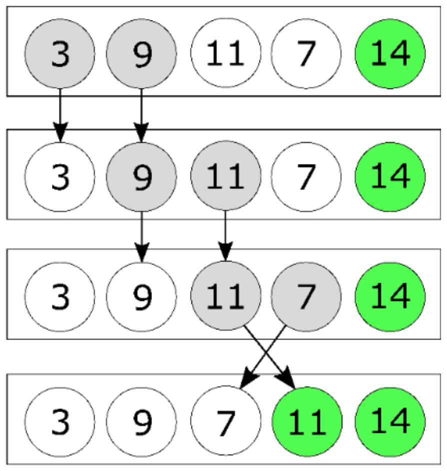
In Java, a temporary variable is created during each switch of items. This requires extra storage space. It uses a constant amount of extra space for variables and is in-place to adjust the order of the input array, so its space complexity results in a value of O(1).

Bubble sort is a stable sorting algorithm which preserves the order of a sorted input.

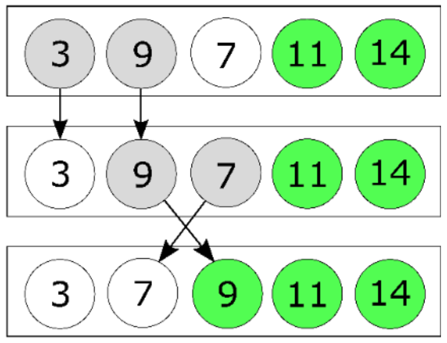


- Input array (n = 5). Start of first loop, 9 > 3 (switched).
- 9 < 14 (no exchange).
- 14 > 11 (switched).
- 14 > 7 (switched).
- 14 is the largest element and is in correct order.

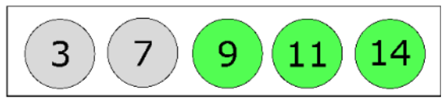
Figure 1: Bubble sort diagram



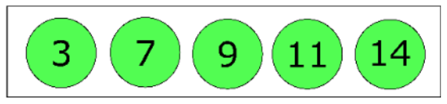
- Start of second loop, 3 < 9 (no exchange).
- 9 < 11 (no exchange).
- 11 > 7 (switched).
- 11 is the next largest element and is in correct order.



- 3 < 9 (no exchange).
- 9 > 7 (switched).
- 9 is the next largest element and is in correct order.



- 3 < 7 (no exchange).



- Array is in correct order.

Figure 2: Bubble sort diagram contd.

Merge Sort

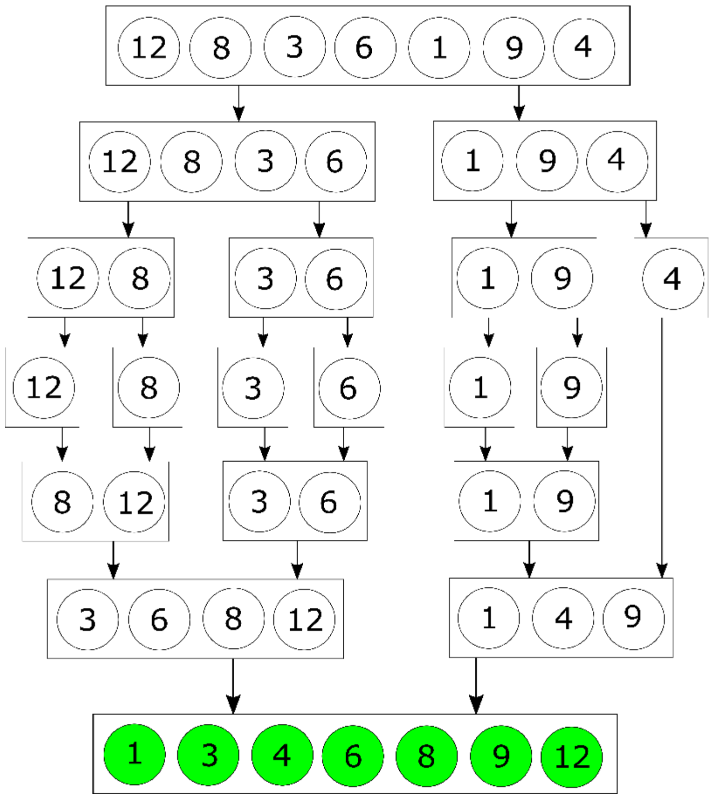
The efficient comparison-based sorting algorithm that I have chosen is Merge Sort.

Merge sort follows a recursive “divide and conquer approach” (java T point, 2021). The given array is broken down recursively into separate arrays until each item is contained into its own individual array. These sub-arrays are then merged back into the final array sorted.

The best, average and worst-case time complexities for Merge Sort is O(n log n). As a result, merge sort is very reliable and predictable given any input array type and is a very efficient comparison type sorting algorithm.

The only main downside of using Merge Sort is its space complexity is O(n) which can take up a lot of space and may slow down the program in some cases.

It is stable so items will maintain their position in relation to other identical items.



- Given array (n = 7).
- Split array into two pieces.
- Split arrays in half again.
- Split arrays in half again until all items are in individual arrays.
- Items are then brought back together into arrays of two, with correct order.
- Items are brought back into two arrays again with correct order.
- Array fully back together in correct order.

Figure 3: Merge sort diagram

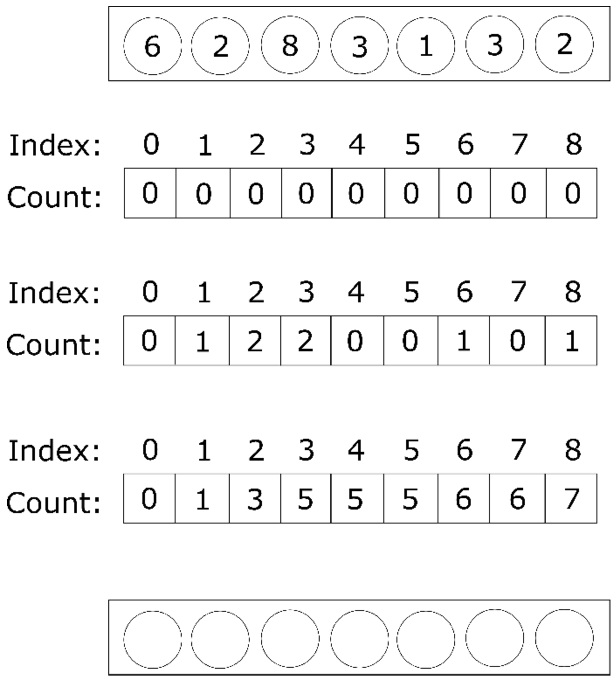
Counting Sort

The non-comparison sorting algorithm that I have chosen for this project is Counting Sort.

It is a linear time sorting algorithm that assumes the contents of the input array. It is most efficient when the array range is linear and consistent and, in the range, [0, k], k being the largest item in the array. The lower the value that k is in relation to n (n = input size), the more efficient the sorting algorithm will be. As a result, this cannot be used as a general-purpose sorting algorithm but when the input is aligned with this property than it is a very efficient and useful sorting algorithm (Baeldung, 2021).

That algorithm calculates the value of the number of occurrences of each item and stores this in a frequency array. The sum of items that are less than or equal to each item is then calculated in this array. This provides the correct ordered index for item that will be stored in the new sorted array.

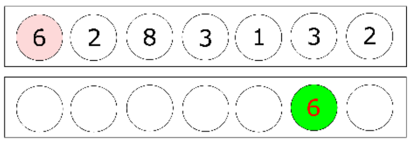
Counting sort is stable and achieves a best, average, and worst-case time complexities of O(n + k), so it is a very efficient sorting algorithm. Space complexity value for this sort is also O(n + k).



- Input array (n = 7).
- Create empty array to count the occurrence of each item in the array.
- Occurrences of each item.
- Calculate sum of items that are less than or equal to each item.
- Create new empty array to contain sorted elements.

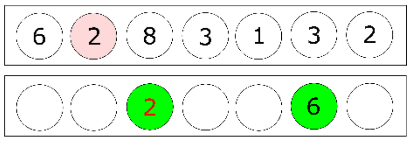
Figure 4: Counting sort diagram

Index:	0	1	2	3	4	5	6	7	8
Count:	0	1	3	5	5	5	6	6	7



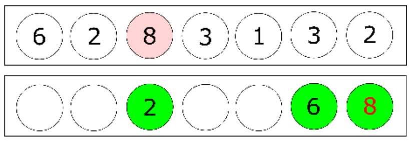
- Using the frequency array, the first element (6) in the input array is given the 6th position in the sorted array. The count of 6 in the frequency array is reduced by 1, and this continued below for each item.

Index:	0	1	2	3	4	5	6	7	8
Count:	0	1	3	5	5	5	5	6	7



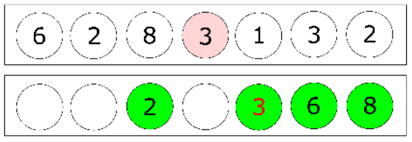
- 2 is given the 3rd position in the sorted array.

Index:	0	1	2	3	4	5	6	7	8
Count:	0	1	2	5	5	5	5	6	7



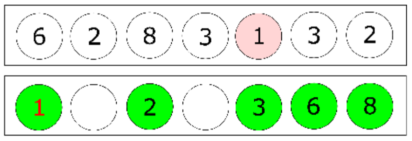
- 8 is given the 7th position in the sorted array.

Index:	0	1	2	3	4	5	6	7	8
Count:	0	1	2	5	5	5	5	6	6



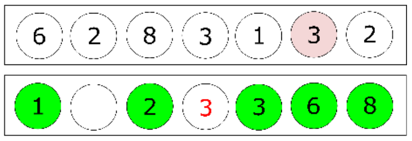
- 3 is given the 5th position in the sorted array.

Index:	0	1	2	3	4	5	6	7	8
Count:	0	1	2	4	5	5	5	6	6



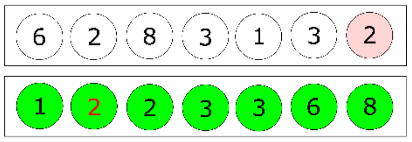
- 1 is given the 1st position in the sorted array.

Index:	0	1	2	3	4	5	6	7	8
Count:	0	0	2	4	5	5	5	6	6

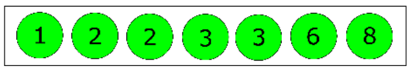


- 3 is given the 4th position in the sorted array.

Index:	0	1	2	3	4	5	6	7	8
Count:	0	0	2	3	5	5	5	6	6



- 2 is given the 2nd position in the sorted array.



- Sorting complete.

Figure 5: Counting sort diagram contd.

Insertion Sort

The fourth sorting algorithm that I will examine is another simple comparison-based sort, called Insertion Sort.

This algorithm is good to use in cases where an input array has a small n size, or if the array is nearly sorted already or if the user wants a simple and easy to write program. It has a best-case time complexity of O(n) and an average and worst-case time complexity of O(n^2). Therefore, as array input sizes become very large this program would become quite slow, unless the array is already nearly sorted. On the contrast, if the array input is small then the Insertion Sort algorithm can be very useful, sometimes overtaking the more complex sorting algorithms in speed and efficiency.

Insertion sort preserves the order of equal items in an array, so it is a stable sorting algorithm, and it is sorted in-place, it uses a constant amount of extra storage space and therefore, has a space complexity of O(1) (OpenGenus, 2021).

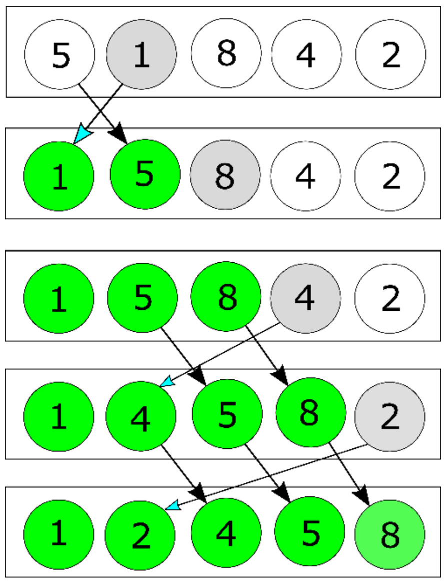


Figure 6: Insertion sort diagram

- The key is set at the second item in the array, assuming the first item is sorted.
- 5 > 1, so 1 moves to the 1st position and 5 is moved to the 2nd position. 8 is the new key.
- 8 > 5, so it stays in position and 4 is the new key.
- 4 < 8 and 4 < 5 but 4 > 1, so 4 moves to 2nd position, 5 and 8 shift up one position. 2 is the new key.
- 2 < 8, 2 < 5 and 2 < 4 but 2 > 1, so 2 moves to the 2nd position and the other items shift up by one position.
- The array is now sorted.

Shell Sort

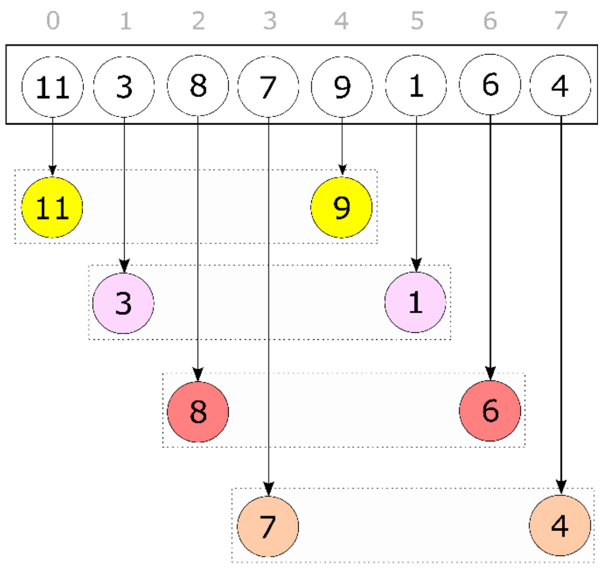
The final sorting algorithm that I will examine is another comparison-based sort called Shell Sort.

Shell sort is a hybrid sorting algorithm firstly comparing elements within an array that are initially far apart from each other and then repeats this but reduces the gap between items. The elements are switched if needed to correct order, and then insertion sort is used to complete the sort.

As comparisons are made with elements that are far apart reducing the length between the closely valued elements, the array is partially sorted, and the insertion sort process works well with an array in this state as there are less swaps required.

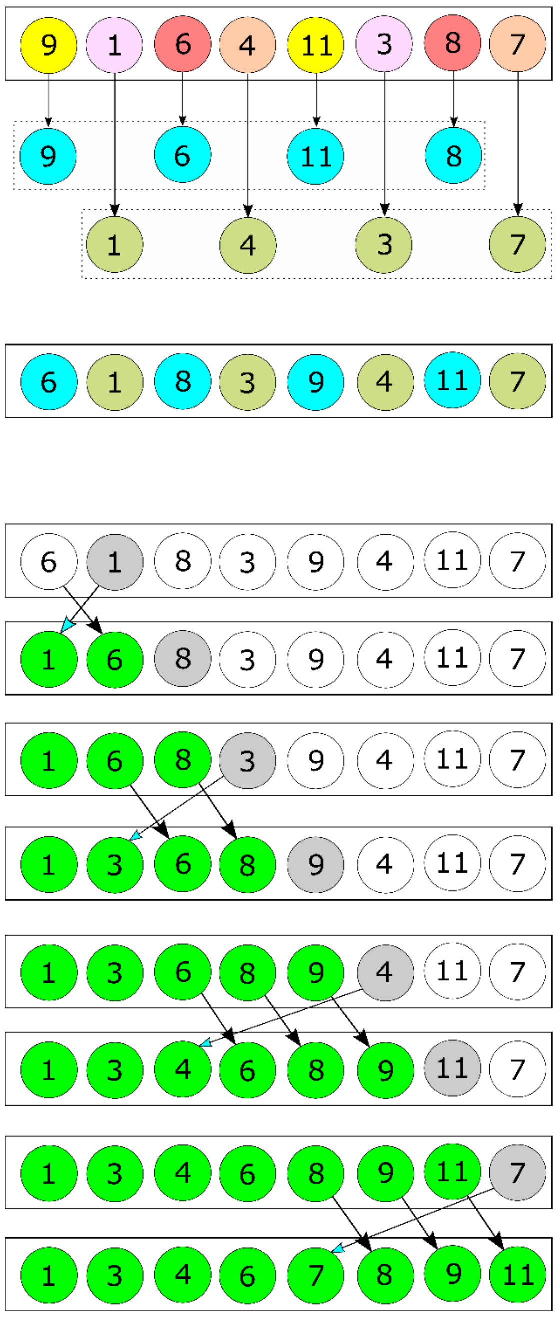
The resulting best case time complexity for Shell Sort is O(n log n), the average case and worst-case time complexities can be O(n log n) and O(n^2) respectively, but this can also depend on the array gap sequence.

The algorithm sorts in-place but is not stable as it "does not examine the elements lying in between the intervals" (Shell Sort (With Code), 2021) and it has in a space complexity of O(1). This sorting is useful for someone who requires a program which is quite simple and easy to write, the array input size is large, and it does not require much additional storage space.



- Given input array (n = 8).
- Comparisons made of intervals of 4 initially (n/2).
- Swaps of elements made if necessary (all items shown in diagram will be swapped as the values in the virtual sub-arrays to the right are smaller than the values in the left).

Figure 7: Shell sort diagram



- Updated array after first comparisons.
- Comparisons made of intervals of 2 (n/4).
- Order sub-arrays.
- Updated array after second comparisons. The array cannot be sorted any more in this fashion as the gap has been reduced as low as possible, so it is now ready for insertion sort.
- Begin insertion sort. Second element in the array is the key.
- 1 < 6, switched and 8 is new key.
- 8 > 6, no exchange and 3 is new key.
- 3 < 8, 3 < 6, 3 > 1, inserted as second element and 9 is new key.
- 9 > 8, no exchange and 4 is new key.
- 4 < 9, 4 < 8, 4 < 6, 4 > 3, inserted as third element and 11 is new key.
- 11 > 9, no exchange and 7 is new key.
- 7 < 11, 7 < 9, 7 < 8, 7 > 6, inserted as fifth element.
- Array is sorted.

Figure 8: Shell sort diagram contd.

Implementation and Benchmarking

For my sorting algorithm benchmark application, I used the Java language. Each of the sorting algorithms were given their own class to improve legibility and ease of use. Another class called `SortingAlgorithms` contained the methods to generate a random array, get the average of results at each array size, calculate the benchmark result for each sorting algorithm and the main method to initiate the application.

In the main method, I firstly created an array of integers called `sizes` to contain each array size that I would test the sorting algorithms with. These ranged from 100 to 25000. Then I printed the console output heading as shown in figure X containing the heading "`Size`" and column headings for each array size to be tested. I then created a `SortingAlgorithms` object called `sort`, and this was used to call the `benchmark()` method to initiate the testing of each sorting algorithm.

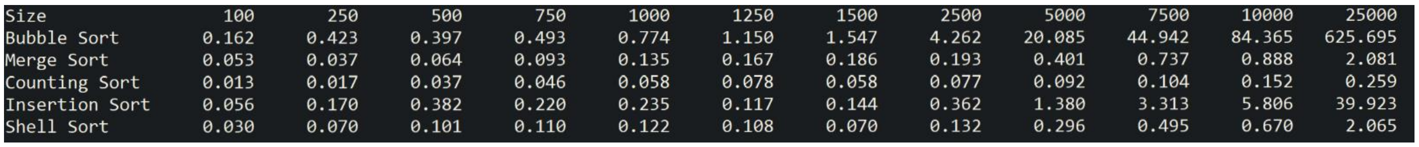
The `benchmark(int sort, int[] a)` method contains two arguments – `int sort` selects which sorting algorithms to choose (1: Bubble sort, 2: Merge sort, 3: Counting sort, 4: Insertion sort, 5: Shell sort) and `int[] a` takes in the `sizes` array which contains the various sizes of arrays to test. A new array is then created in this method called `results` which contains the results of each individual benchmark test run. A switch statement is used to select which sorting algorithm to be called and tested. This is used to reduce repetitive code and to contain the benchmark process in one method. After the runs are complete and the results array is full after each array size is tested, the `getAverage(double[] results)` method is called to calculate the average value of the ten runs in milliseconds and print this value to the console. This is repeated until all array sizes have been looped through the benchmarking test and printed to the console output.

In the `SortingAlgorithms` class there are instance variables set up for use in this class:

1. A random array variable which is used for each test run - `private int[] randArr;`
2. The number of runs is set at 10 with the variable `private int runs = 10;`
3. Bubble sort object – `private BubbleSort bs = new BubbleSort();`
4. Merge sort object – `private MergeSort ms = new MergeSort();`
5. Counting sort object – `private CountingSort cs = new CountingSort();`
6. Insertion sort object – `private InsertionSort is = new InsertionSort();`
7. Shell sort object – `private ShellSort ss = new ShellSort();`

These sort objects were created to call each class from the benchmark method in a non-static way.

Console output



```
Size                 100     250     500     750    1000    1250    1500    2500    5000    7500   10000   25000
Bubble Sort        0.162   0.423   0.397   0.493   0.774   1.150   1.547   4.262  20.085  44.942  84.365 625.695
Merge Sort         0.053   0.037   0.064   0.093   0.135   0.167   0.186   0.193   0.401   0.737   0.888   2.081
Counting Sort      0.013   0.017   0.037   0.046   0.058   0.078   0.058   0.077   0.092   0.104   0.152   0.259
Insertion Sort     0.056   0.170   0.382   0.220   0.235   0.117   0.144   0.362   1.380   3.313   5.806  39.923
Shell Sort         0.030   0.070   0.101   0.110   0.122   0.108   0.070   0.132   0.296   0.495   0.670   2.065
```

Figure 9: Console output of application shows results in table form.

Results table

	100	250	500	750	1000	1250	1500	2500	5000	7500	10000	25000
Bubble Sort	0.162	0.423	0.397	0.493	0.774	1.150	1.547	4.262	20.085	44.942	84.365	625.695
Merge Sort	0.053	0.037	0.064	0.093	0.135	0.167	0.186	0.193	0.401	0.737	0.888	2.081
Counting Sort	0.013	0.017	0.037	0.046	0.058	0.078	0.058	0.077	0.092	0.104	0.152	0.259
Insertion Sort	0.056	0.170	0.382	0.220	0.235	0.117	0.144	0.362	1.380	3.313	5.806	39.923
Shell Sort	0.030	0.070	0.101	0.110	0.122	0.108	0.070	0.132	0.296	0.495	0.670	2.065

Table 1: Results of application – all values are in milliseconds and are the average of 10 repeated runs.

Results Graphs

Running time shown max 60ms

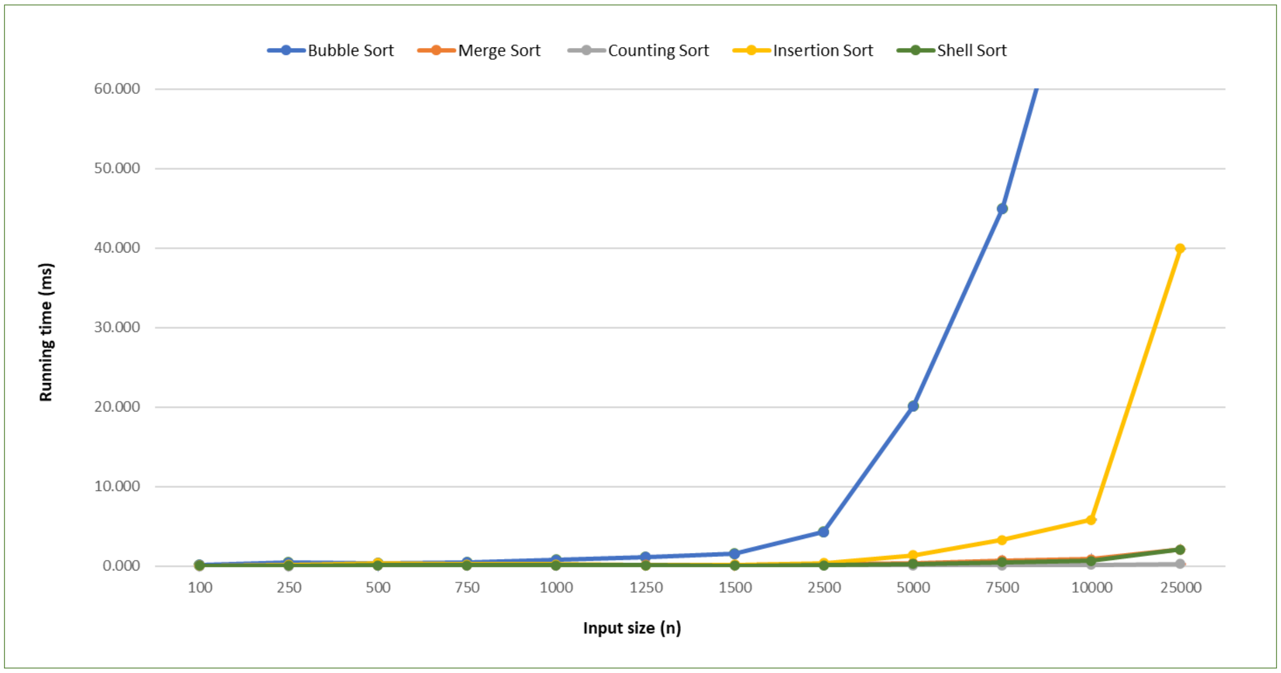


Figure 10: Results graph showing running time to a max of 60ms.

Running time shown max 2.5ms

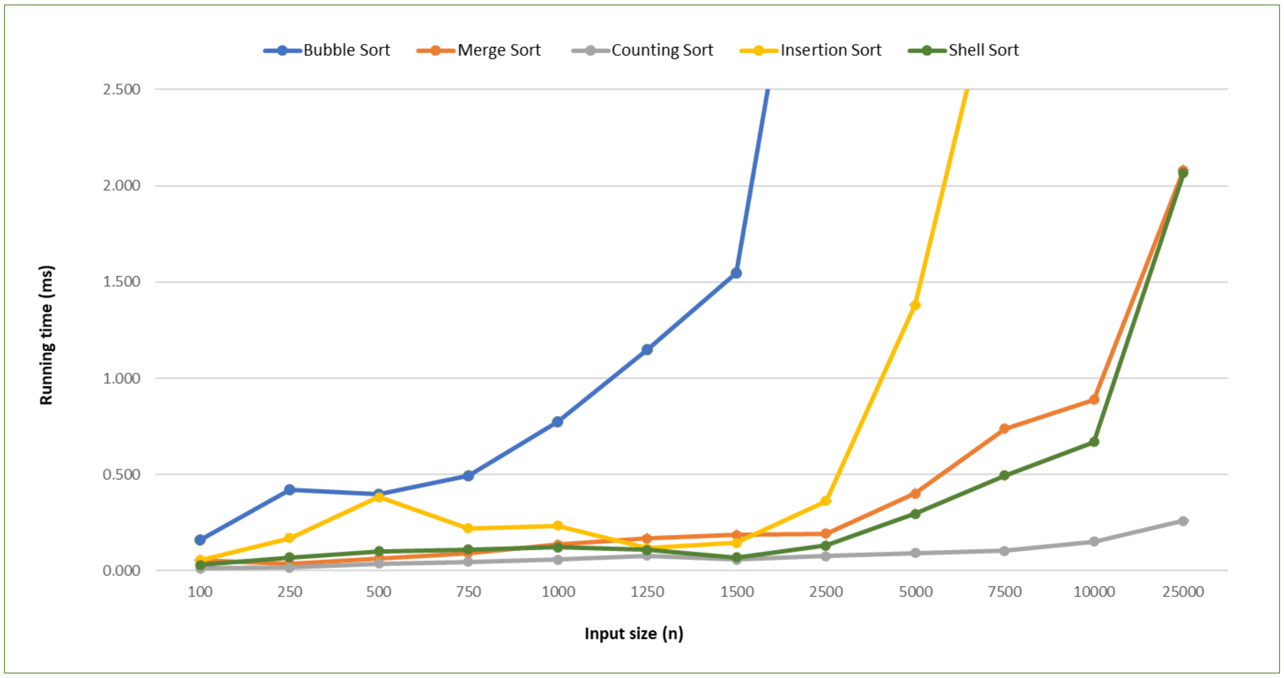


Figure 11: Results graph showing running time to a max of 2.5ms.

Analysis

As shown above in the results table, the Counting Sort algorithm produces the most efficient results in each array size test. The array input criteria suit the algorithm as the range of values in the array match the array size. It is important to note that if the range was much larger than the array size then it is possible that it may not perform as efficiently as it has above, however, it has proven to be an extremely effective and reliable sorting algorithm solution. As shown up close in figure 11, the counting sort algorithm never reaches above 2ms until an array of n = 25k is run through it. It remains consistently quick throughout the test proving a superb time complexity of O(n + k).

The Shell Sort algorithm comes in second place overall just ahead of Merge Sort. It proves to be slightly more efficient and stays true to its best-case time complexity of O(n log n). There shows to be a sharp increase in time taken between n = 10k and n = 25k which resembles an $O(n^2)$ case time complexity, however, this gap in array size is much larger than previous gaps so it is fair to say a time complexity of O(n log n) is achieved and it is a very efficient sorting algorithm.

Merge Sort also shows very efficient results on par with the results from Shell Sort. It proves that it has reliability, predictability, and a time complexity of O(n log n). This is a very useful and efficient sorting algorithm but one of the more complex algorithms.

Insertion Sort shows very promising and efficient results up until the array size reaches n = 5000. At n = 1500 it is one of the quickest sorts and then it reduces in efficiency greatly after n = 5000. Insertion Sort can be a very efficient algorithm to use for sorting but when array sizes get larger than there are other more reliable options. It has proven its best-case time complexity of O(n) up until n = 2500, and then there is evidence of a time complexity of $O(n^2)$.

The most unreliable and inefficient sorting algorithm in this group is Bubble Sort. Throughout the tests, it runs slowest and increases quadratically, proving a time complexity of $O(n^2)$. When the array size was very small (n = 100), it produces similar results to the other sorting algorithms, however, very quickly it appears to reduce in efficiency as the array size increases and it is therefore not a very useful sorting algorithm solution for any array types.

It was evident during the algorithm testing process, that out of the 10 individual runs for each array size, the initial run was quite slower than the remaining 9 runs in every sorting algorithm program. This is shown to be a result of Just-In-Time (JIT) compilation, when the source code is initially compiled and interpreted called the “JRockit approach” which “results in relatively longer startup times” (Understanding Just-In-Time Compilation and Optimization, Oracle, 2021).

References

Time and Space Complexity Analysis of Algorithm. 2021. Time and Space Complexity Analysis of Algorithm. [ONLINE] Available at: https://afteracademy.com/blog/time-and-space-complexity-analysis-of-algorithm. [Accessed 13 May 2021].

Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) based Sorting Algorithms? . 2021. Difference between Comparison (QuickSort) and Non-Comparison (Counting Sort) based Sorting Algorithms? . [ONLINE] Available at: https://javarevisited.blogspot.com/2017/02/difference-between-comparison-quicksort-and-non-comparison-counting-sort-algorithms.html#axzz6ukHzTRR5. [Accessed 13 May 2021].

www.javatpoint.com. 2021. Merge Sort - javatpoint. [ONLINE] Available at: https://www.javatpoint.com/merge-sort. [Accessed 11 May 2021].`

Baeldung. 2021. Counting Sort in Java | Baeldung. [ONLINE] Available at: https://www.baeldung.com/java-counting-sort. [Accessed 11 May 2021].

OpenGenus. 2021. Insertion Sort Complexity Analysis. [ONLINE] Available at: https://iq.opengenus.org/insertion-sort-analysis/. [Accessed 12 May 2021].

Shell Sort (With Code). 2021. Shell Sort (With Code). [ONLINE] Available at: https://www.programiz.com/dsa/shell-sort. [Accessed 12 May 2021].

Understanding Just-In-Time Compilation and Optimization. 2021. Understanding Just-In-Time Compilation and Optimization. [ONLINE] Available at: https://docs.oracle.com/cd/E15289_01/JRSDK/underst_jit.htm. [Accessed 12 May 2021].