Daniel Murray

G00387933

Contents

Introduction

Sorting is an essential concept in computer programming which is a method by which items/elements (usually in large numbers) are re-arranged in a specific manner based on a relationship shared between elements e.g. numerical or lexicographic. The applicability and usefulness of sorting lists cannot be understated - especially as the size of datasets increase and accessing data rapidly and efficiency is critical. However, not all sorting algorithms are equal. Each have both benefits, and drawbacks in terms of their efficiency. Some of the key concepts involved in sorting algorithms are:

Complexity

The computational complexity of an algorithm is a measure of the amount of computing resources (time and space) that a particular algorithm consumes when it runs. (Britannica, 2020, para. 5). Complexity is denoted by big O notation, with O referring to the order of magnitude – and is used when representing the time and space complexity of a program.
Time complexity refers to the number of operations performed in order to complete a task with respect to ‘n’, where ‘n’ is the size of the input. A measurement of how an algorithm performs as the input size increases gives a clear indication of the performance and efficiency of that program. Space complexity refers the amount of memory consumed by a program’s operation. The curve of a program’s complexity can be constant O(1), logarithmic O (log n), linear O(n), quadratic $O(n^2)$, or in the worst scenario, exponential - $O(c^n)$. It also worth noting that each algorithm will have a best, average, and worst case scenario in which any of the above performances may be demonstrated.

Performance

This relates to the allocation of time and memory on the computer’s hardware upon execution of a program. It is affected by the computers own Operating System and specifications. The complexity of the program itself is independent of a systems performance, but the converse is true in that the system performance can be affected by a programs complexity. Other factors like clock speed, RAM, internal memory, and CPU temperature all affect a systems performance. (Cosgrove 2020).

In-place sorting

In place sorting takes place when a sorting algorithm does not require additional memory/space to be created during the program’s execution, therefore space remains constant. Examples of in-place sorting algorithms include Bubble Sort, Insertion Sort, Heap Sort, and Insertion Sort.

Stability

The stability of a sorting algorithm is concerned with how the algorithm treats equal (or repeated) elements. Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don’t. (Srivastava 2020, para. 2). In other words, after sorting, a stable algorithm does not change the original ordering of two items with the same value.

Before	6	3	**4**	**4**	12	9

After	3	**4**	**4**	6	9	12

Figure 1. A visualization of stability being maintained after sorting

Examples of some stable sorting algorithms include: Bubble, Merge, Tim, Count, and Insertion sort.

One downside of stable sorting algorithms they can increased complexity and/or memory usage due to the increased resources required to preserve ordering.

In contrast, unstable sorting occurs when this ordering is not preserved upon sorting, with the resultant output not distinguishing between items of the same value in the sorted sequence. Unstable sorting algorithms are faster due to less resources used to preserve item ordering in stable sorting algorithms. Examples of some unstable sorting algorithms include: Selection sort, Shellsort, Quicksort, Heapsort

Comparator Functions

Not all items are primitive and can be compared implicitly to one another. Because of this, comparator functions act as a means to compare to items based on a particular property. In object oriented programming, one can order objects of a defined class using the comparator interface. Below is a coded example in java of how one might compare two 'Car' Objects, with the main logic implemented in the compareTo() method.

```
public class Car implements Comparator<Car> {
        //Various properties declared, only one can be chosen as the basis of comparison
        private String carModel;
        private int id; //This will be the unique identifier by which comparisons are made
        private String colour;
        private int YearMfg;

        //Constructor
        public  Car(String carModel, int id, String colour, int yearMfg) {
                Initialise…
        }

        //Comparing two Car Objects
        public int compareTo(Car c) {

                if (this.equals(c)) {
                        return 0;

                //if greater, returns +1
                } else if (this.getId() > c.getId()) {
                        return 1

                //If less, returns -1
                } else
                        return -1;
        }}

        //Override the equals method also to align with comparator method
        public boolean equals(CarCompare other) {
                return this.getId() == other.getId();
        }
```

Comparison-based sorts

In this type of sorting technique, elements are compared to each other to generate the sorted array. In this sense, they are compared with a single comparison operation (comparator). Because of the comparison operation carried out, these algorithms tend to be slower than their non-comparison-based counterparts. The best time complexity achieved by this class of algorithms is O(n*log(n)). (Javin Paul, 2017)

e.g. if $a \leq b$ and $b \leq c$ then $a \leq c$

Some comparison based algorithms include Quicksort, Heapsort, Bubble Sort, Insertion Sort, Timsort, Shell Sort.

Non-comparison-based sorts

These types of algorithms do not rely on a comparison operation, but can be based on assumptions about the input, and use keys to sort the input values – achieving O(n) in time complexity. (Javin Paul, 2017). Examples include Counting Sort, Radix Sort, and Bucket Sort.

Bubble Sort

This is a simple comparison-based sorting algorithm which compares adjacent elements in an input and swaps their positions if they are in the wrong order.

Take for an example a given input named *array*.

- First for loop begins with index $i = 0$.
- For each index i, an inner loop with index j completes an entire pass of $array.length - i$. Both adjacent elements are compared, and swapped if array[j]< array[j-1]

This causes the unsorted item to 'bubble' up to the end of the array, where it is then sorted. After each pass, the next inner pass iterates only to $array.length - i$. This is because the items in the positions after $array.length - i$. have already been sorted. Sorting passed this would be unnecessary.

An additional Boolean variable is also declared here. It is initially set to true on the each pass. If the assumption holds true – the method will return and the array is fully sorted. However, if a swap takes place, the Boolean is set to false and the looping can continue. Because you are only iterating to $array.length - i$. at each iteration of i, the Boolean check

Complexity

Bubble sort uses a nested array structure to sort its input, which results in an average time complexity of $\boldsymbol{n^2}$.

In the best case scenario, the array is already sorted. However, for each element n in the list, a traversal must occur to check the elements. Even with the improvement of traversing to $array.length - i$, and the additional Boolean check, the performance still gives an $\boldsymbol{n^2}$ time complexity.

In the worst case scenario, the list is in reverse order, and similarly for each element n, a second traversal must be completed resulting in an $\boldsymbol{n^2}$ time complexity.

Additional space is not required in this algorithm, hence a constant O(1) space complexity.

Bubble Sort Implementation

```java
public class BubbleSort {

    private void sort(int[] array) {
        boolean isSorted;

        for (int i = 0; i < array.length; i++) {
            isSorted = true;//assume true initially – if it holds true, return
            //second loop compares each element j with the previous item
            //This causes the element j to ‘bubble’ up to the end of the list
            for (int j = 1; j < array.length – i; j++)
                //if less, then swap with the previous item
                if (array[j] < array[j - 1]) {
                    swap(array, j, j - 1);
                    isSorted = false;
                }
            //if all items are sorted, return.
    }
    private void swap(int[] array, int index1, int index2) {
        var temp = array[index1]; // temporary variable
        array[index1] = array[index2]; // reassign
        array[index2] = temp; //setting array[index2] == array[index1]
    }
```

Visualization of Bubble Sort Algorithm

For the given input array = {6, 2, 5, 7, 10}

i = 0 -- {**6, 2**, 5, 7, 10} → {**2, 6**, 5, 1, 10}

Conditions	Before	After
First Pass		
i = 0, j = 1	{**6, 2**, 5, 1, 10}	{**2, 6**, 5, 1, 10}
i = 0, j = 2	{2, **6, 5**, 1, 10}	{2, **5, 6**, 1, 10}
i = 0, j = 3	{2, 5, **6, 1**, 10}	{2, 5, **1, 6**, 10}
i = 0, j = 4	{2, 5, 1, **6, 10**}	{2, 5, 1, **6, 10**}
Second Pass		
i = 1, j = 1	{**2, 5**, 1, 6, 10}	{**2, 5**, 1, 6, 10}
i = 1, j = 2	{2, **5, 1**, 6, 10}	{2, **1, 5**, 6, 10}
i = 1, j = 3	{2, 1, **5, 6**, 10}	{2, 1, **5, 6**, 10}
i = 1, j = 4	{2, 1, 5, **6, 10**}	{2, 1, 5, **6, 10**}
Third Pass		
i = 2, j = 1	{**2, 1**, 5, 6, 10}	{**1, 2**, 5, 6, 10}
i = 2, j = 2	{1, **2, 5**, 6, 10}	{1, **2, 5**, 6, 10}
i =2, j = 3	{1, 2, **5, 6**, 10}	**{1, 2, 5, 6, 10}**
	boolean isSorted = true	Method returns. Input sorted

Merge Sort

This algorithm works on the principle of recursively splitting the input (array) into smaller sub arrays. It continuously the array at the midpoint of each subsequent subarray, creating a binary tree structure, until each subarray only contains one element, and is therefore itself a sorted subarray. The single element subarrays are then merged back together in sorted order, and the combination of unsorted subarrays continues until the original array is fully sorted.

Complexity

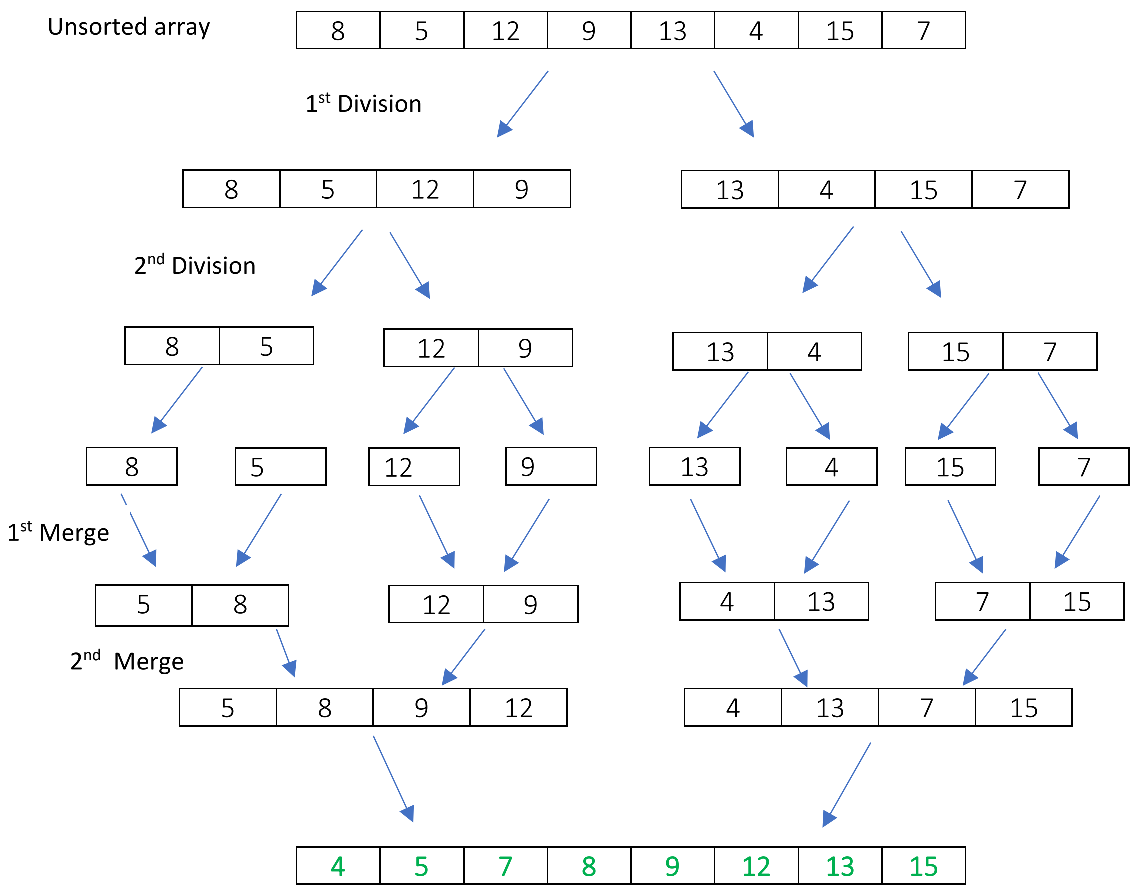
In terms of time complexity:

- Dividing the array is an $\boldsymbol{O(1)}$ operation.
- The recursive operation halves each subarray at every iteration, creating a binary tree structure, which takes $\boldsymbol{\log n}$ time.
- The merging step takes $\boldsymbol{O(n)}$ time as it must iterate through the subdivided 'n' elements.
- Combining all three steps – the constant time operation is ignored, and the overall complexity is therefore $\boldsymbol{O(n * logn)}$**.** This remains the case for both the best and worst case scenarios.
- Space complexity: The Merge sort approach allocates additional memory for each subarray, therefore requiring $\boldsymbol{O(n)}$ space.

Merge Sort Implementation

```
public void sort(int[] arr) {

        if (arr.length < 2) { // bass condition to stop recursion
            return;
        }
        // 1.Divide the array in half
        var middle = arr.length / 2; // Half

        int[] left = new int[middle];
        //Copy into left array
        for (int i = 0; i < middle; i++)
            left[i] = arr[i];
        //Copy into right array
        int[] right = new int[arr.length - middle];
        for (int i = middle; i < arr.length; i++)
            right[i - middle] = arr[i]; // i-middle ensures correct index

        // Recursive calls to keep dividing arrays until base case reached
        sort(left);
        sort(right);

        merge(left, right, arr);
    }

    // Merges our sorted left and right arrays back into our original array
    private void merge(int[] left, int[] right, int[] result) {
        int i = 0; // left counter
        int j = 0; // right counter
        int k = 0; // result arr counter
        while (i < left.length && j < right.length) {

            if (left[i] <= right[j]) // If left value is smaller
            //assign the left(smaller) value to array[k]
            result[k++] = left[i++]; // Increment both k & j

            else // if right value smaller
                result[k++] = right[j++];
        }

        while (i < left.length) // Copy remaining surplus values from left
            result[k++] = left[i++];

        while (j < right.length) // Copy remaining surplus values from right
            result[k++] = right[j++];

    }
```

Visualization of Merge Sort Algorithm



Counting Sort.

This is a non-comparison based algorithm that relies on the use of a second 'counts' array. The second array keeps a count of the frequencies for each element of the given input. Because of this additional array, you must also know the length of the original input. Once the counts array has been filled, it can then be iterated over, and the input array then refilled in sorted order. One drawback of this approach is that the input array's values need to be positive integers. Otherwise, you cannot use their values as indexes of the counts array. In addition, the range of values should ideally be close together as this will avoid unnecessarily large traversals through zero values.

Complexity

For time complexity, you first must:

- Iterate and populate the counts array. This requires iteration over the input array - $\boldsymbol{O(n)}$.
- Iterate the counts array, and refill the input array from this - $\boldsymbol{O(k)}$.
- This results in a time complexity of $\boldsymbol{O(n+k)}$, which can be simplified to $\boldsymbol{O(n)}$.

For space complexity, additional memory is needed for the extra counts array. Taking 'k' as the maximum value of the input array, then $\boldsymbol{O(k)}$ space is required for this algorithm.

Counting Sort Implementation

```
private void sort(int[] array) {
        int max = array[0];
        //Finds the largest element in the array
        for (int i = 0; i < array.length; i++) {
            if (array[i] > max)
                max = array[i];
        }
        //Count array declared and initialised to the same size as input
        int[] counts = new int[max + 1];

        for (int i = 0; i < array.length; i++) {
            counts[array[i]]++; //Increments counts at the same index
             position as the input elements value
        }

        var k = 0; // indexer for the input array
        for (int i = 0; i < counts.length; i++) {
            // second loop will iterate up to the frequency value
            for (int j = 0; j < counts[i]; j++) {
                array[k]++ = i;
           // inserts back into array at the correct sorted position
            }
        }
    }
```

Counting Sort Visualization

- Given an input array:

3	2	4	4	1	5

- Traverse this array to find largest element, so the counts array can be initialized to the correct length

  $int\ max\ =\ 0;$

1st Iteration	3	2	4	4	1	5

int max = 3;

2nd Iteration	3	2	4	4	1	5

int max = 3;

3rd Iteration	3	2	4	4	1	5

int max = 4;

4th Iteration	3	2	4	4	1	5

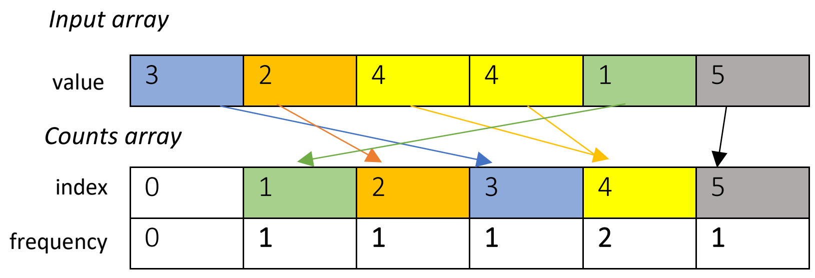
int max = 4;

5th Iteration	3	2	4	4	1	5

int max = 5;

6th Iteration	3	2	4	4	1	5

- Populate the counts array (full for loop excluded but intuition has been explained



Input array

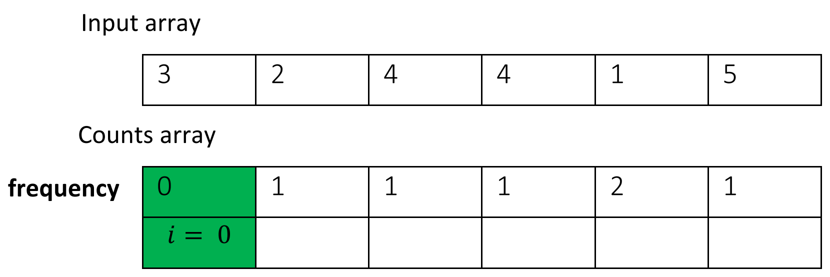
value	3	2	4	4	1	5

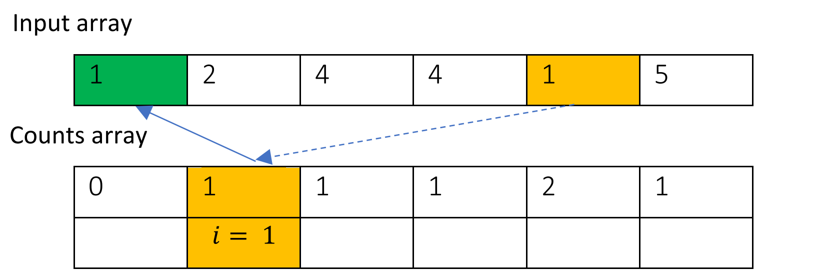
Counts array

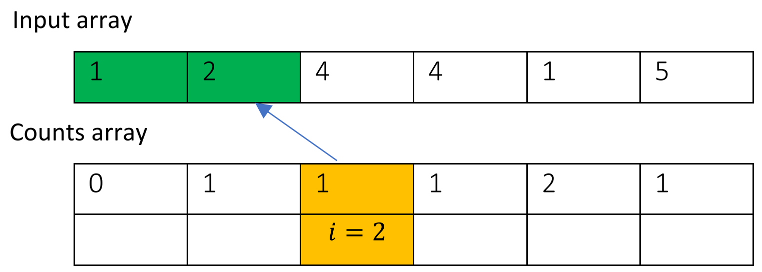
index	0	1	2	3	4	5
frequency	0	**1**	**1**	**1**	**2**	**1**

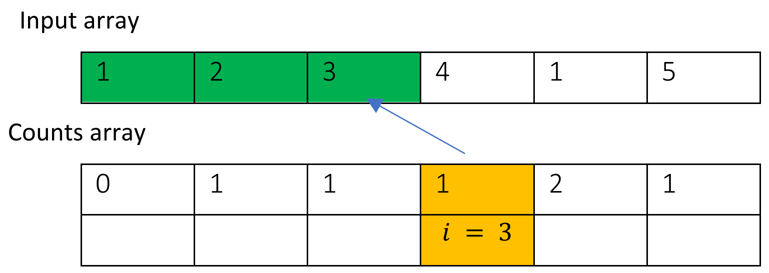
The above populated counts array therefore contains no zeros, a single one, a single two, a single 3, two fours, and a single 5.

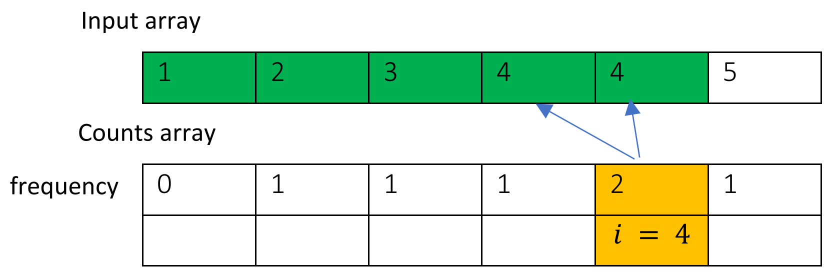
- Iterate over counts array and refill input array with corresponding numbers of frequencies occurring



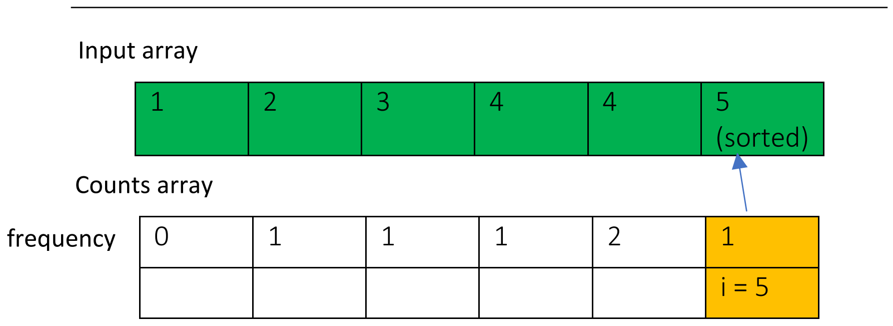








Note: for this iteration, an inner for loop as described in implementation will iterate up to count[i], which in this case is two. For each occurrence, it will then insert the value ‘4’ back into the original array.



Quick Sort

This is an efficient comparison based sorting algorithm, which also sorts the elements in place. It is based on the principle of dividing the input array into partitions that are separated by a 'pivot' point i.e. the target value to be sorted. Partitioning is done recursively in the case of this implementation. The pivot value is selected by the user and is generally chosen as either the last element, the median element, or the first element. It also relies on two index variables. One for iterating over the unsorted partition, and a second to mark the boundary between the sorted and unsorted partitions. i.e. edge of the left/unsorted partition.

Complexity

In terms of time complexity, you must:

- Partitioning the array. This involves iterating over the array and so is an $\boldsymbol{O(n)}$ operation for both best and worst case scenarios.
- However the number of time partitioning the array occurs varies. In the best case scenario, the pivots final position is in the middle of the array – dividing the array equally in half. This will result in a $\boldsymbol{\log n}$ time complexity (if this final pivot position is adhered to throughout the partitioning process).
- However, if the pivots final position is at a point where the left and right partitions are highly uneven i.e. left >> right or vice versa, the max efficiency that can be achieved moving the pivot is in $\boldsymbol{O(n)}$ time.
- For example, if a given array is already sorted, and the initial pivot position is chosen as the last element. For each partitioning, the pivot only moves one index to the left, and must traverse the entire array, giving an $\boldsymbol{O(n)}$ time complexity.

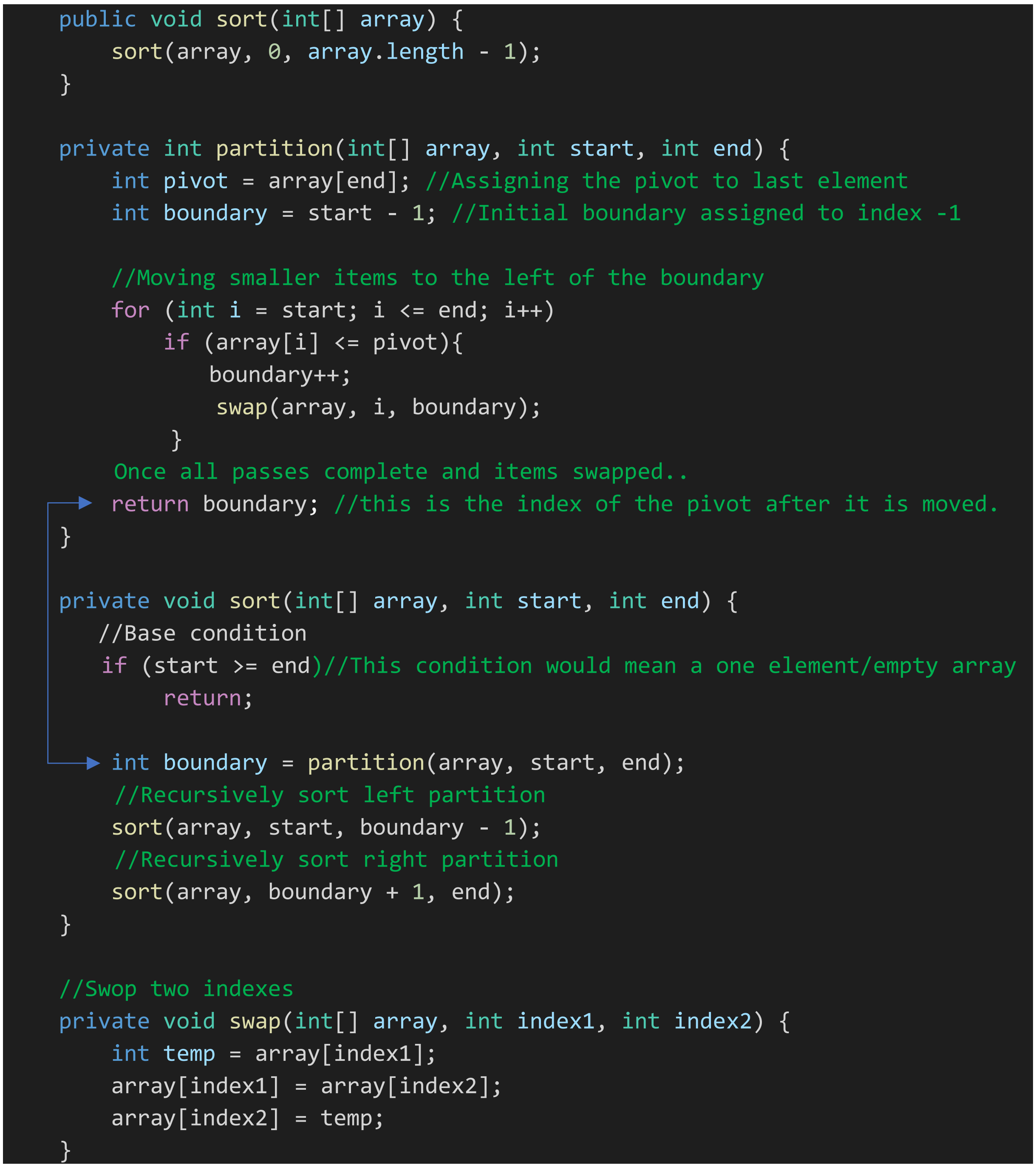
Combining the partitioning, and number of partitions, we get an O($\boldsymbol{n \log n}$) time complexity in the best case scenario, and a worst case scenario of $\boldsymbol{O(n\hat{}2)}$.

Quick Sort Implementation

Note: This implementation assigns the pivot as the last element in the array, and the boundary index is initially set to -1. See code comments for further explanation

The approach here is to:

1) Partition
2) Sort left recursively
3) Sort right recursively

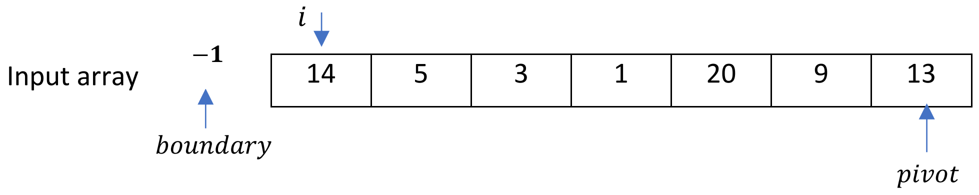


```java
public void sort(int[] array) {
    sort(array, 0, array.length - 1);
}

private int partition(int[] array, int start, int end) {
    int pivot = array[end]; //Assigning the pivot to last element
    int boundary = start - 1; //Initial boundary assigned to index -1

    //Moving smaller items to the left of the boundary
    for (int i = start; i <= end; i++)
        if (array[i] <= pivot){
           boundary++;
            swap(array, i, boundary);
         }
    Once all passes complete and items swapped..
    return boundary; //this is the index of the pivot after it is moved.
}

private void sort(int[] array, int start, int end) {
   //Base condition
   if (start >= end)//This condition would mean a one element/empty array
        return;

    int boundary = partition(array, start, end);
    //Recursively sort left partition
    sort(array, start, boundary - 1);
    //Recursively sort right partition
    sort(array, boundary + 1, end);
}

//Swop two indexes
private void swap(int[] array, int index1, int index2) {
    int temp = array[index1];
    array[index1] = array[index2];
    array[index2] = temp;
}
```

Count Sort Visualization

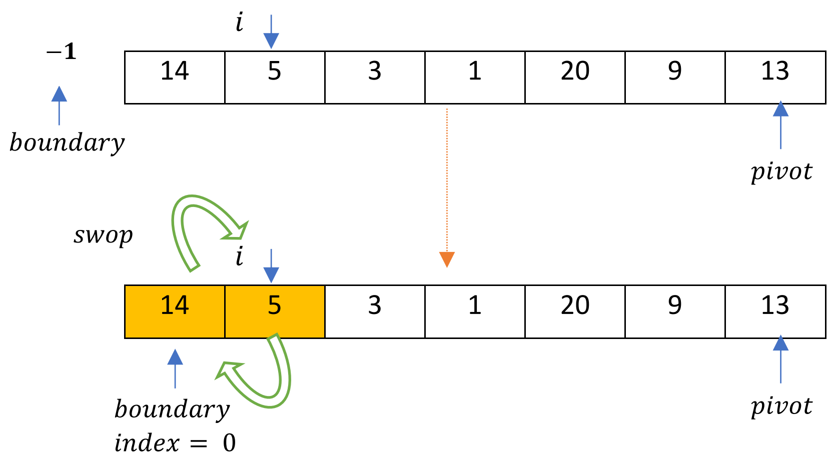
1st iteration

- Below is the initial set up of the algorithm, the boundary index is initially set to -1, the pivot is chosen as the last element, and our standard indexer i starts at the beginning of the array
- On the first iteration 14 > 13, so we continue.



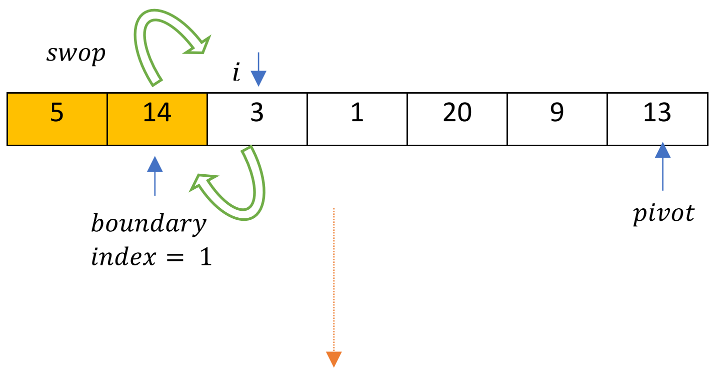
2nd iteration

- If the current item is smaller than the pivot, move it to the left partition.
- On the second iteration, 5 < 13, so we must move this to the left partition. First, increase the boundary, then swop the element at index $boundary$ with index i.

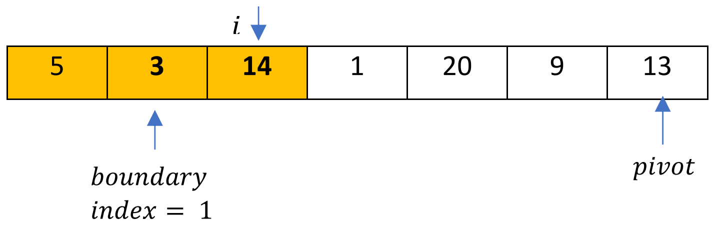


3rd iteration

- Increase $i. i < pivot$ again, so we increase the boundary, and swop it with the current value of i

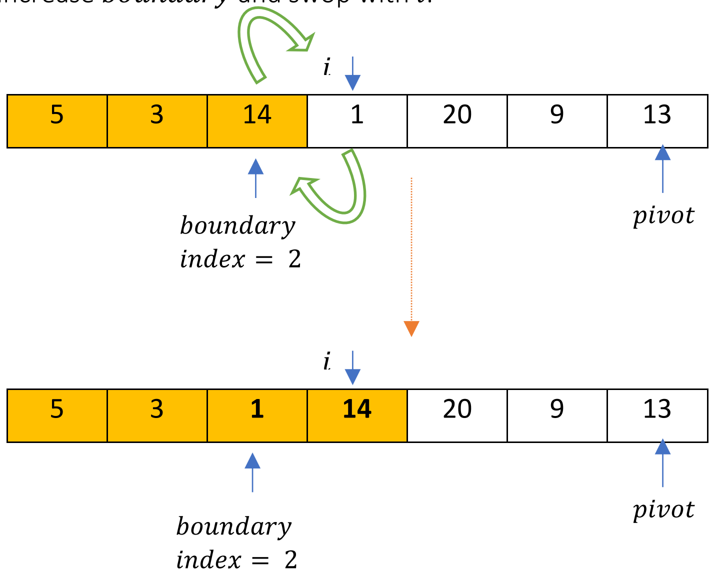


- Iteration 3 continued..



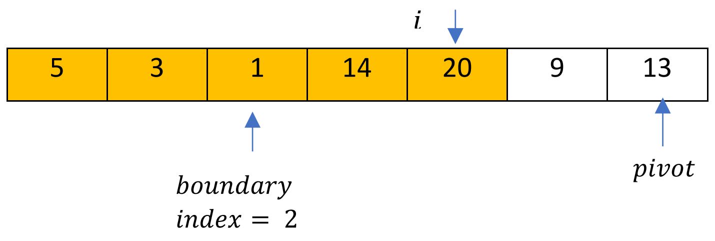
4th iteration

- $i < pivot$: increase $boundary$ and swop with i.



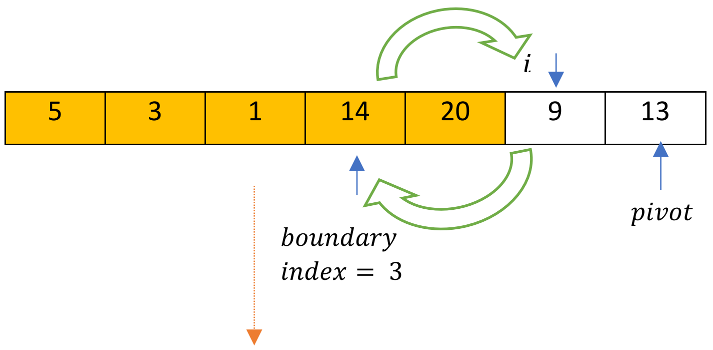
5th iteration

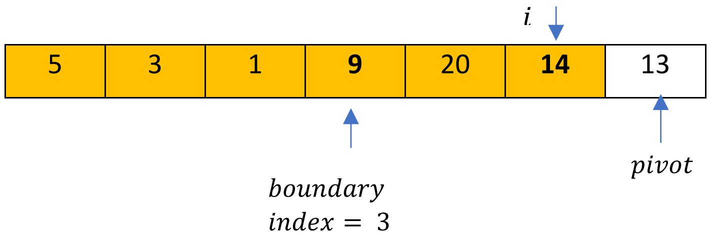
- i > pivot, so continue



6th iteration

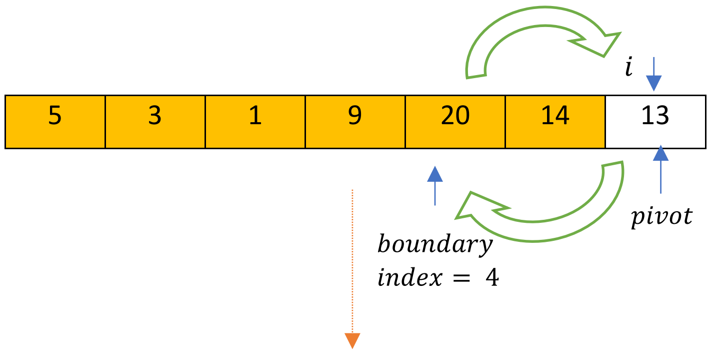
-

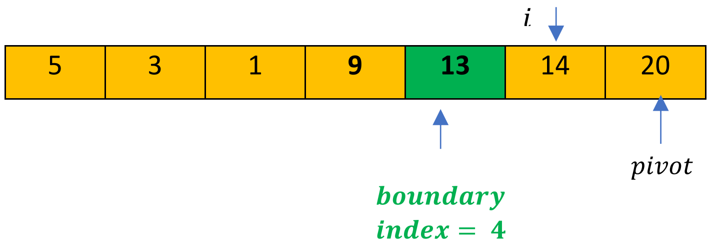




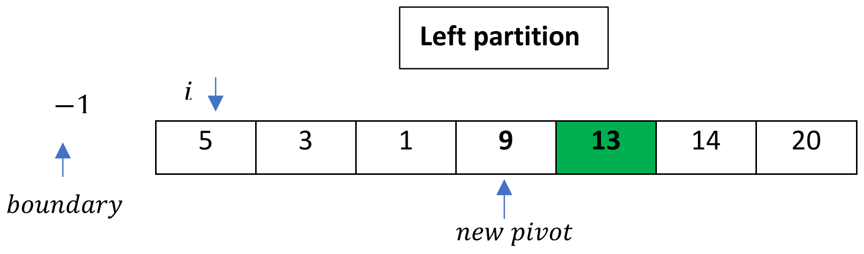
7th iteration

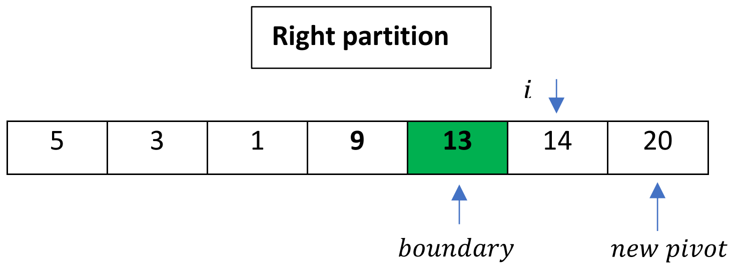
- Now index i has reached the pivot, the condition still holds true as $i <= pivot$. Increase $boundary$ and swop with i
- Once the pivot has been swopped, the full pass is complete and the pivot element has been swopped to its final sorted position.





With the first pass complete, the array has now been divided into both its left and right partitions. The recursive swopping can now take place on both partitions until the array is sorted. The last element of the left, and right partitions are chosen as the new pivots, shown below. The same logic as described in the previous diagrams is applied to both partitions recursively until the array is fully sorted.





Insertion Sort

Insertion sort is a comparison based sorting algorithm. Interestingly this does not technically 'swop' elements like many other algorithms, but a copying/shifting approach of elements is taken. Greater items are shifted to the right, and subsequently room is made for the current item to be inserted.

Complexity

- You must iterate over the input array, and read one item at a time. This is therefore an $\boldsymbol{O(n)}$ operation, for both best and worst case scenarios.
- A second iteration must occur to shift the items to the right if required. In the best case, the items are already sorted and this would therefore be an $\boldsymbol{O(1)}$ operation.
- In the worst case scenario, all the items will be in reverse order, requiring all elements to be shifted, resulting in $\boldsymbol{O(n)}$ time complexity.
- Therefore, the overall complexity for this algorithm is $\boldsymbol{O(n^2)}$.
- Space is conserved, with no additional space being required, making it $\boldsymbol{O(1)}$ in terms of memory.

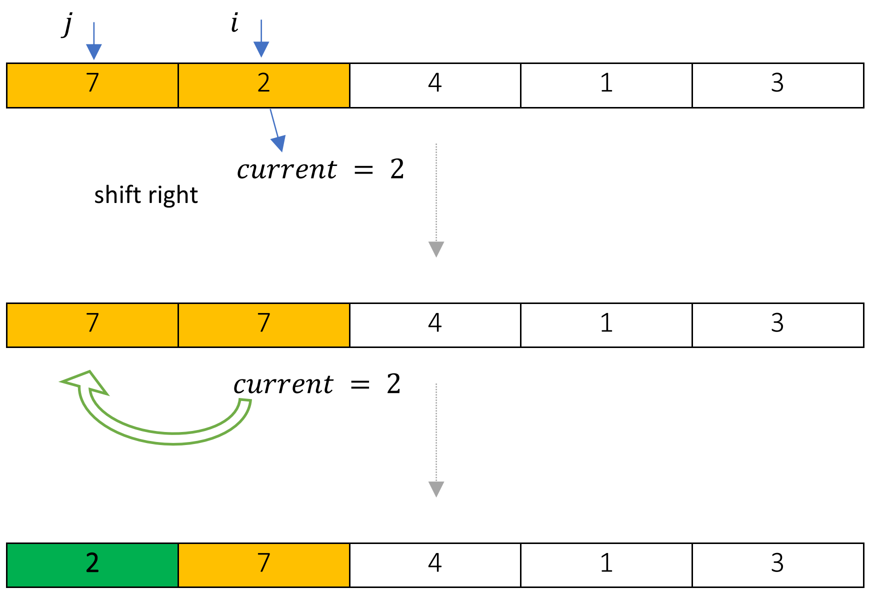
Implementation

```
    private int[] sort(int[] arr) {

        for (int i = 1; i < arr.length; i++) { //Start at 2nd index
            so comparison to previous can be made
            int current = arr[i]; //temporary storage variable

            int j = i - 1; //assign j to previous index
            while (j >= 0 && arr[j] > current) {
           //iterate backwards and shift larger elements right
            arr[j + 1] = arr[j]; //does not iterate passed the sorted elements
                j--; //decrements j to continue until loop conditions not met
            }
            arr[j + 1] = current;
           //insert the current item to correct position

        }
        return arr;
    }
```

The above implementation relies on the use of a temporary storage variable $current$. This is to prevent the element from being lost when shifting elements. The while loop condition does not allow for j to iterate passed the unsorted section of the array. The while loop also shifts all items greater than current to the right, and once complete, inserts current to the correct position. This cycle continues until the array is sorted.

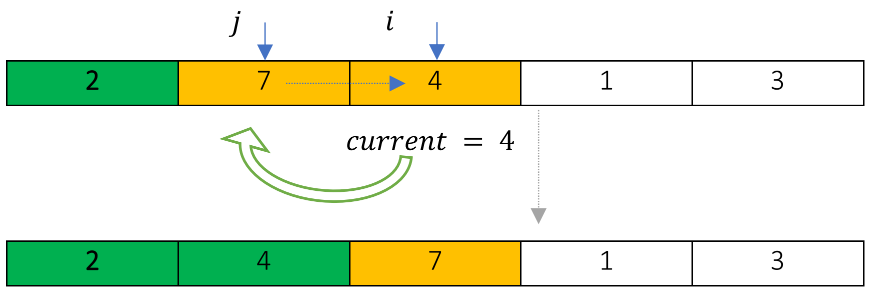
Insertion Sort Visualization

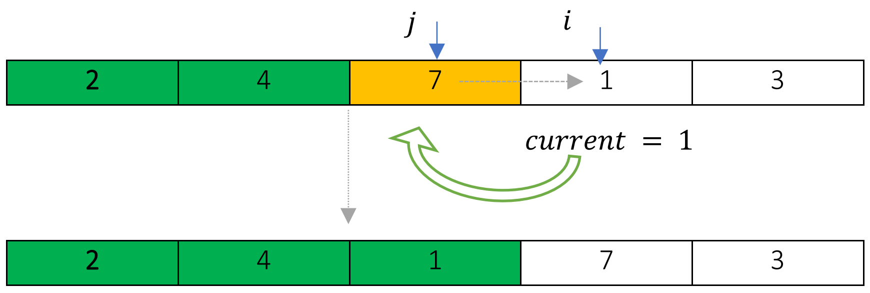
- In the first step, variables are assigned. Initially, since 7 is the first item in the array, it is assumed to be sorted. The while loop checks if $array[j] > array[i]$. In this case it is, and so the larger element is shifted to the right while the current variable is stored in a temporary variable. This temporary variable is then inserted to the previous position



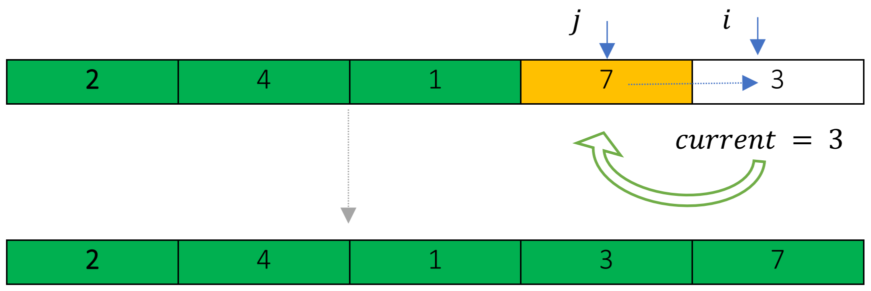
First element sorted

- The process is repeated again, 7 is shifted/copied to the next position while the current position 4 is stored in a temporary variable. It is then inserted to the previous/now sorted position.





- This process is now complete once all greater elements have been shifted right and smaller elements inserted.

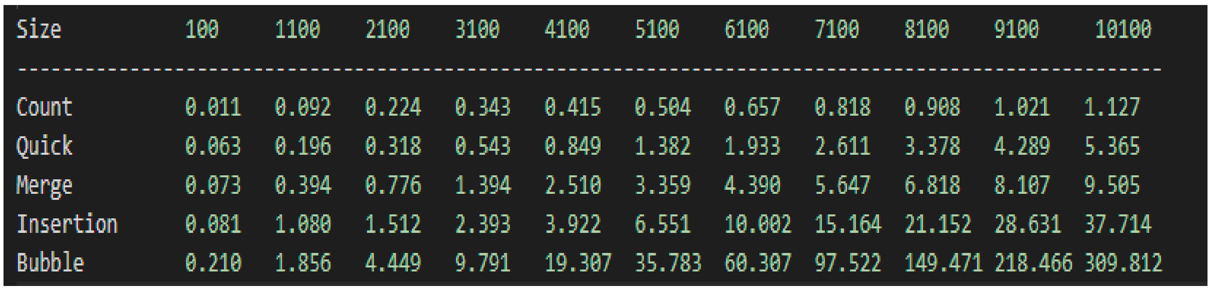


Implementation & Benchmarking

To implement the five chosen algorithms, I wanted to keep all functionality separate for each algorithm, therefore each algorithm is contained within its own class. The $Runner$ class then instantiates an object for each of the classes, and each then subsequently calls their respective $sortTester()$ method. With regards to each class, I separated any functionality I deemed necessary into separate private methods, to ensure a cleaner and easier read code.

The first issue in terms of implementing the program logic was ensuring that each algorithm generated 10 new random arrays and sorted each one, then obtaining the average of the ten runtimes. For this, I used a nested for loop approach. This ensured each iteration generated a new array of size $n + 1000$. The inner for loop sorts the random array 10 times and cumulatively stores the runtime, which is the simply divided by 10 to compute the average runtime.

In order to output the final table of results from the console to a file, I used the command terminal in the $src/ie.gmit.dip$ folder to first compile each class with the command $javac\ *$. Following this, I redirected the output of the executed program to a text file 'output.txt' with the command $java\ ie.gmit.dip.Runner\ >\ output.txt$. Below is an image of the average runtime of 10 separate code executions for each array size, for each of the five chosen algorithms



```
Size           100     1100    2100    3100    4100    5100    6100    7100    8100    9100     10100
----------------------------------------------------------------------------------------------------
Count          0.011   0.092   0.224   0.343   0.415   0.504   0.657   0.818   0.908   1.021   1.127
Quick          0.063   0.196   0.318   0.543   0.849   1.382   1.933   2.611   3.378   4.289   5.365
Merge          0.073   0.394   0.776   1.394   2.510   3.359   4.390   5.647   6.818   8.107   9.505
Insertion      0.081   1.080   1.512   2.393   3.922   6.551   10.002  15.164  21.152  28.631  37.714
Bubble         0.210   1.856   4.449   9.791   19.307  35.783  60.307  97.522  149.471 218.466 309.812
```

Table 1. Average runtimes for each algorithm

Count Sort Results

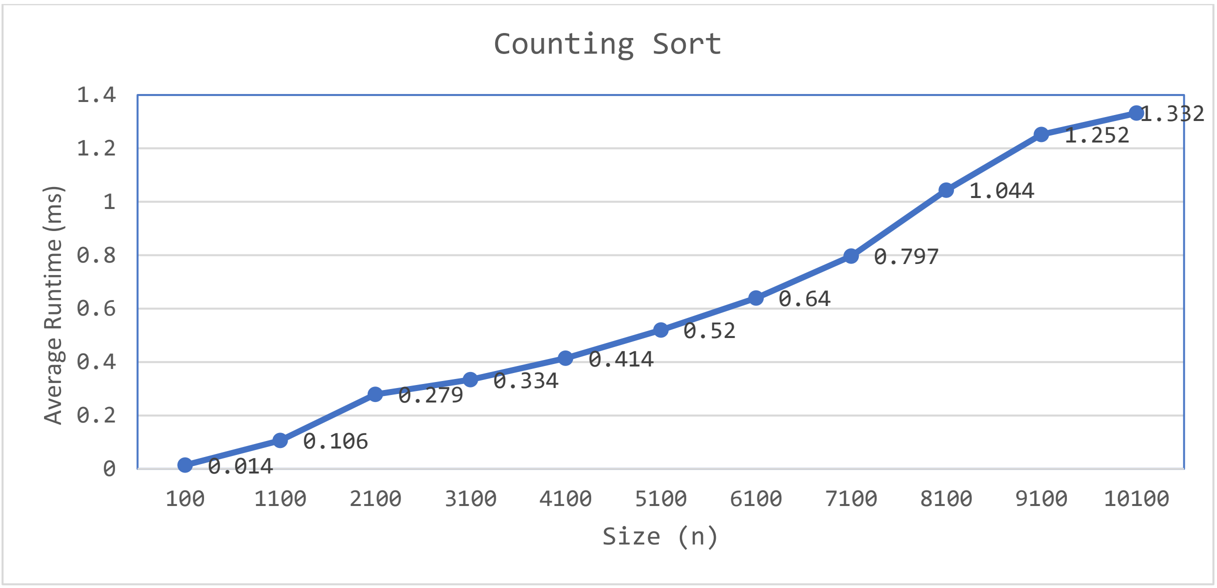


Figure 1. Average runtime in milliseconds vs input size (n) for Counting Sort

In terms of performance, counting sort was the fastest of all five sorting algorithms in overall average time scores. The graph generally adheres to the expected linear trend based on its $O(n)$ time complexity. This performance may not be fully representative of how this algorithm might perform in more realistic conditions, however. The random array generated only contains numbers in the range [0-100]. Because of this, the intermediate counting array to store frequencies is always a small array size, possibly making traversal a faster operation and less demanding on memory. This algorithm therefore seems to have performed under best case conditions.

Quick Sort Results

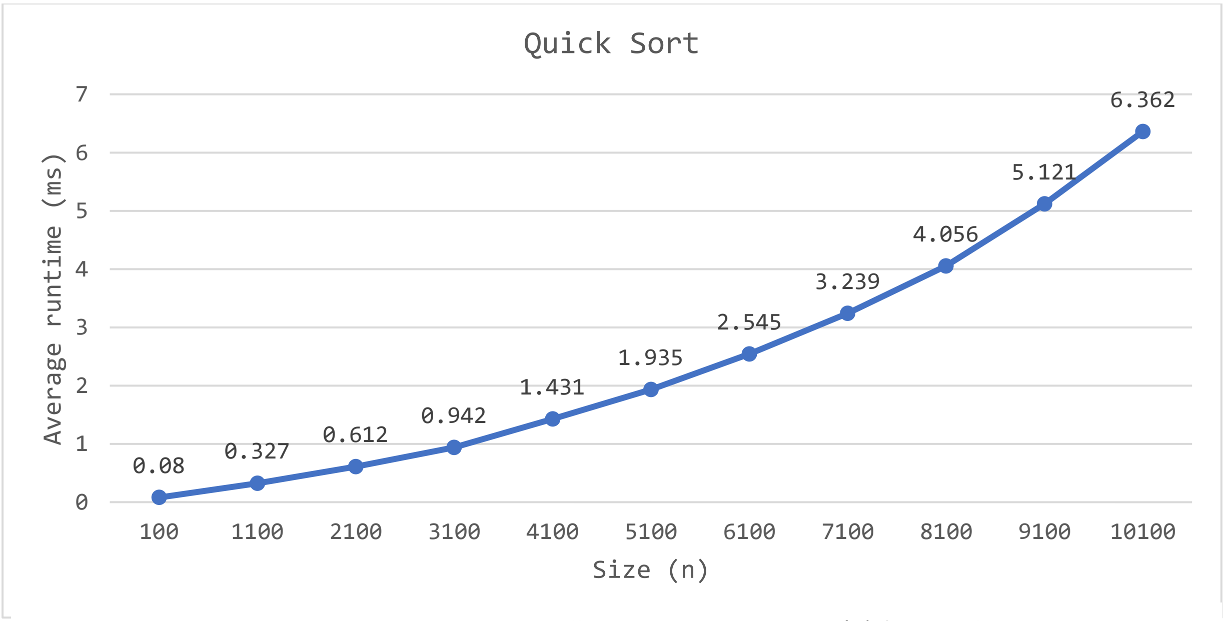


Figure 2. Average runtime in milliseconds vs input size (n) for Quick Sort

Here, the results appear to follow a 'low sloped' quadratic trend. Consequently, it can be assumed that the pivot position consistently did not result in well balanced partitions, and therefore its typical logarithmic efficiency was not achieved on average. However, it still significantly outperformed merge, bubble, and insertion as expected.

Merge Sort Results

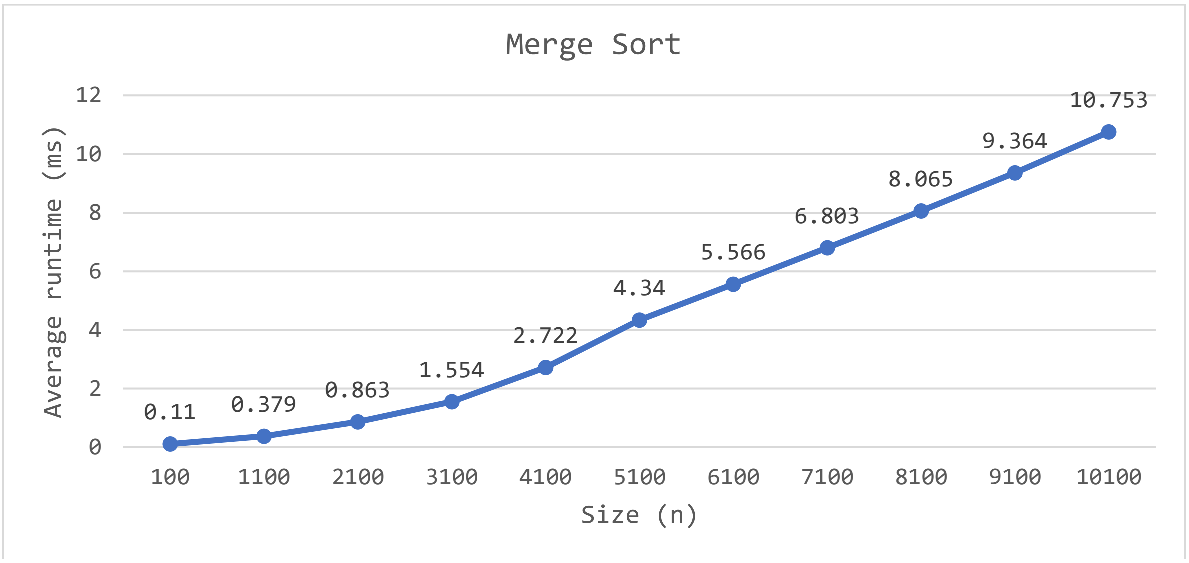


Figure 3. Average runtime in milliseconds vs input size (n) for Merge Sort

At lower array sizes, the results appear to be forming a quadratic trend, but after an array size of 5100, the results are almost perfectly linear – $O(n)$.

Insertion Sort Results

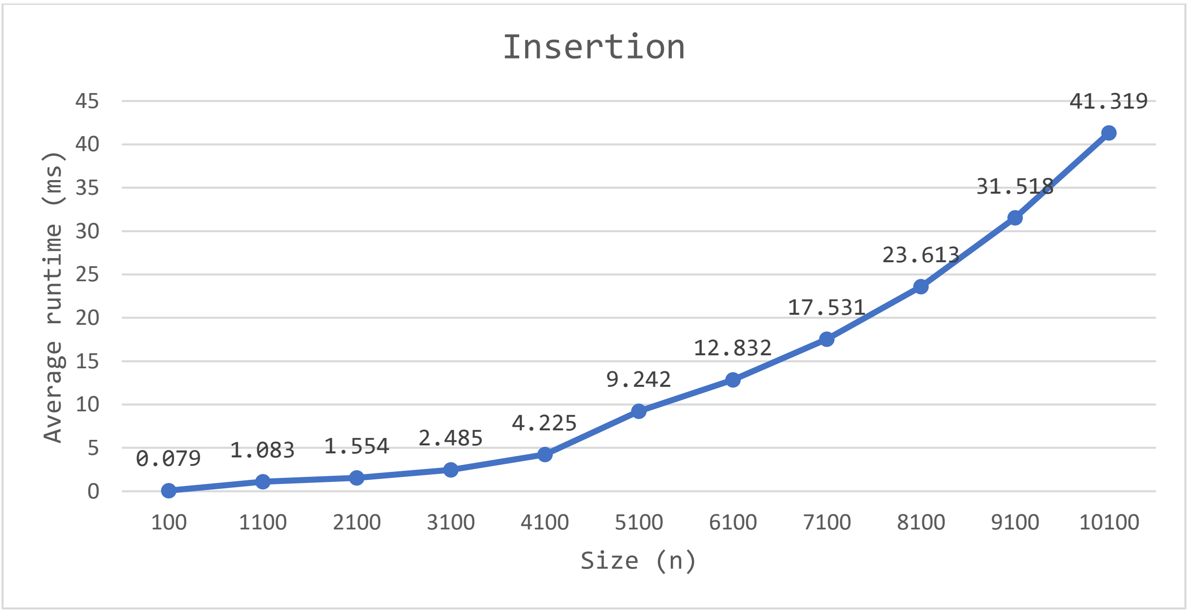


Figure 4. Average runtime in milliseconds vs input size (n) for Insertion Sort

At lower array sizes, a generally linear trend is observed but as the input size increases (>4100), the expected $O(n^2)$ complexity is clearly observed.

Bubble Sort Results

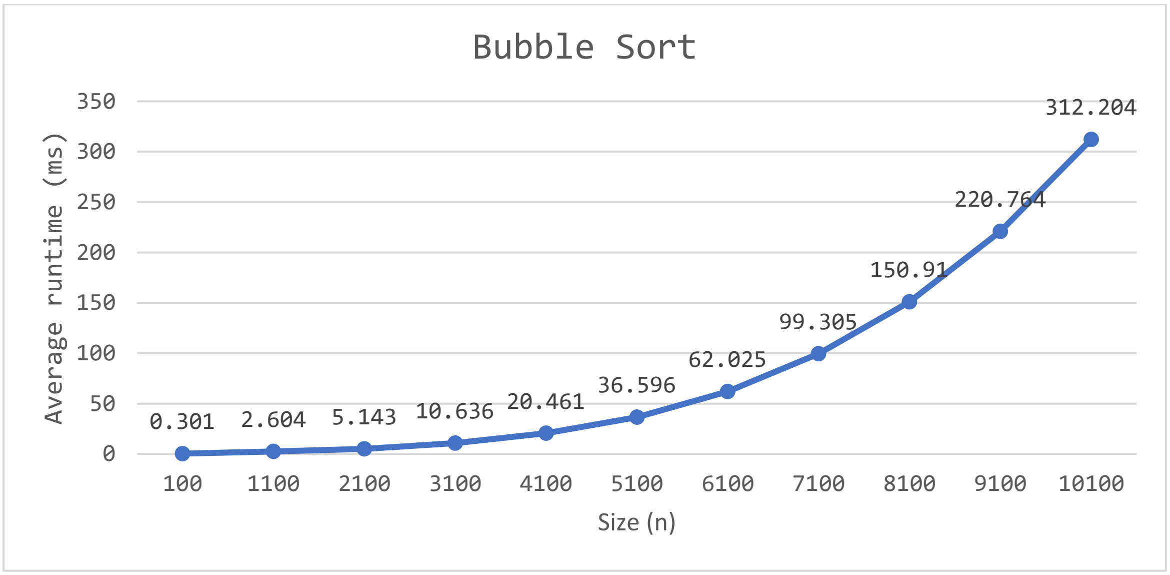


Figure 5. Average runtime in milliseconds vs input size (n) for Bubble Sort

Bubble sort expectedly had the longest runtimes, with a clear $O(n^2)$ trend being observed, and average runtimes much higher than all other algorithms, demonstrating its inefficiency particularly array sizes > 3100. In general, the average runtimes are very high and therefore demanding with time complexity.

Grouped Results

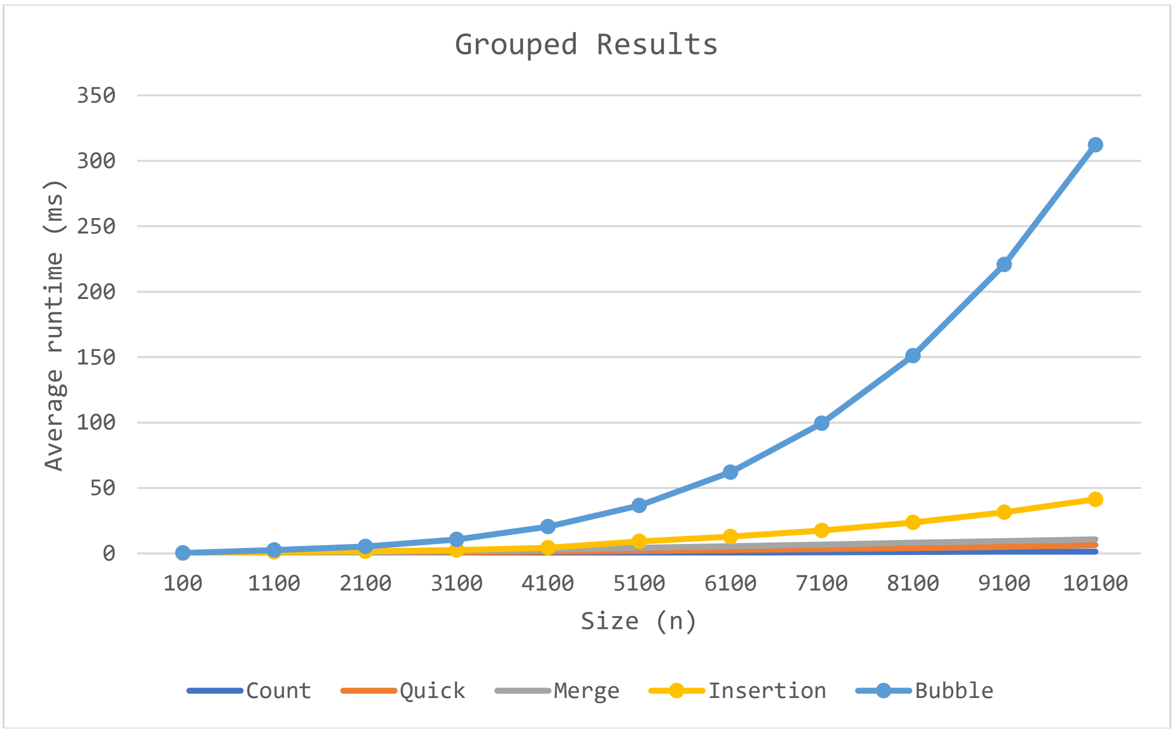


Figure 6. Average runtime in milliseconds vs input size (n) for all results.

Conclusion

This assignment has demonstrated the performance results of five well known sorting algorithms. A general discussion on sorting algorithms was carried out and complexity analysis was done on each chosen algorithm - along with implementation and explanation using visual examples with Microsoft Word's inbuilt tables and visual tools. Results were presented in both tabular and graphical format, and discussed in relation to their expected performance. In general, the algorithms adhered to the expected complexities, however seeming to tend toward worse case scenarios in each case.

References

1) Computer Science - Algorithms and Complexity | Britannica." 2020. In Encyclopædia Britannica. https://www.britannica.com/science/computer-science/Algorithms-and-complexity
2) Cosgrove, James. 2020. "10 Factors That Affect CPU Performance." GizmoFusion. June 23, 2020. https://www.gizmofusion.com/factors-affecting-cpu-performance.
3) Paul, Javin. n.d. "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.
4) opendatastructures.org. (n.d.). 11.1 Comparison-Based Sorting. [online] Available at: https://opendatastructures.org/ods-java/11_1_Comparison_Based_Sorti.html.

Note on system details

System type – 64-bit operating system.

RAM – 8GB

Processor – Intesl® Core™ i7-3520M, clock speed 2.90GHz.