

# Sorting Algorithms Benchmarking Project

## Computational Thinking with Algorithms

Derek Higgins  
g00398357@gmit.ie

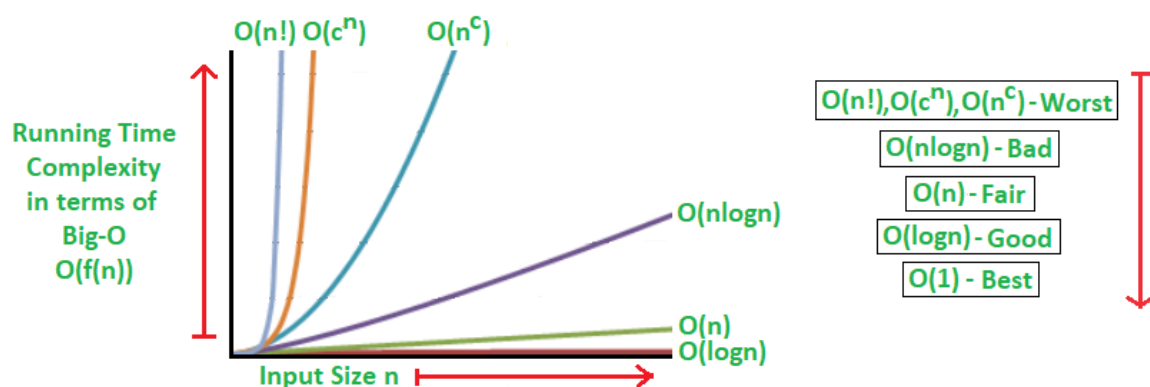
### Introduction

Sorting is the process of arranging items in sequence or grouping them based on the comparison of one versus another. With numeric objects the basis of sorting is on numeric value typically from least to greatest value. For other objects some form of comparison method must be implemented to allow objects to be sorted relative to each other.

### Complexity

Complexity of an algorithm can be stated in terms of both time and space. Time complexity refers to the execution time required by an algorithm to complete its operation on the data set and is commonly notated as Big-O Complexity.

Space complexity refers to the memory space utilised by an algorithm to solve its problem. In relation to sorting algorithms as discussed here this would be influenced by whether the problem is solved using an in-place method or whether the partially sorted data is stored elsewhere in memory during execution such as with Merge Sort type algorithms. Big-O notation is also used to express space complexity.



Analysis of Algorithms | Big-O analysis - GeeksforGeeks. (2018). Retrieved from  
<https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/>

## Performance

As the size of the data set increases, the time and space complexity of an algorithm can have a greater and greater impact on execution time. Ranging from linear to exponential growth rates in execution time, the best case sorting algorithms provide for  $n \log n$  complexity with the most basic providing exponential growth rates.

For applications which can be anticipated to only have the requirement to sort small data sets the implementation of basic sorting algorithms can be adequate. However where data sets can be expected to grow to larger sizes the time complexity of the sorting algorithm being employed must be considered and more advanced methods implemented.

## In-Place Sorting

Algorithms considered to sort in-place typically have good space complexity and do not require additional memory to hold data during execution of the algorithm.

In the case of merge sort for example, additional arrays are created as the problem is divided and this extra memory requirement increases its space complexity and would be an example of an algorithm which does not employ in-place sorting.

## Stable Sorting

When objects are being sorted and after comparison are found to be equal, an algorithm which is considered to be stable sorting will always arrange these instances of equal objects in the same relative order and that given the same original input array this would be maintained on successive runs of the sorting algorithm.

In a non-stable sorting algorithm this arrangement of equal objects would not occur.

## Comparator Functions

Comparator functions on objects allow them to be compared on the basis of a property of that object. With integers for example this is the numeric value, for horses, it could be based on their height. For an object compared by its colour, a comparator function would need to be created to determine which object is greater than ( $>$ ), equal to ( $=$ ) or less than ( $<$ ) when compared against another of the same type. In the case of colour this could be achieved through manipulation of its RGB values.

## Comparison Based Sorting

Comparison based sorting employs the comparator functions of objects to achieve sorting. All elements are inspected and the best time complexity that can be achieved is  $O(n \log n)$  performance.

## Non-Comparison Based Sorting

By making assumptions about the data set that is to be sorted, such as the distribution of values, it is possible to avoid comparison of objects in some instances leading to more efficient sorting. Bucket sort is an example of this and in the best case scenarios can achieve linear execution  $O(n)$  however depending on the data set the worst case scenario for this algorithm is  $O(n^2)$ .

## Algorithms

### Bubble

This comparison based sorting algorithm is named for the way in which elements bubble to the top when visualised. This is a simple algorithm that compares adjacent elements and swaps them when they are in the wrong order. This process is repeated until the list has been sorted.

### Space and Time Complexity

Best case scenario is when the array is already in a sorted condition and no swaps are required. In this instance time complexity will be  $O(n)$ .

In the worst case, when the data set must be fully compared for all objects, the time complexity is  $O(n^2)$ .

This algorithm has a space complexity of  $O(1)$  as it is an in-place algorithm.

### Diagrams

Initial condition as follows:

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7

Initial Array Condition


Elements 0 and 1 are compared, found out of order and swapped:

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7

First Swap

Now the next two elements are compared, found out of order and swapped.

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7


 Swap

This continues the inspection of elements traversing to the right. Here, elements are found in order and no swap required.

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7


Again, traversing to the right, elements are swapped

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7

 Swap

This continued to the end of the array.

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
	7	3	1	5	8	2	4	7	9

 Swap

Now the last item in the array is known to be sorted, having 'bubbled' to the top. The algorithm must now only consider n-1 elements. This repeats until the array is fully sorted as so.

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						
		1	2	3	4	5	7	8	9

## Different Inputs

There is a high level of variability between the best and worst case scenario with this algorithm. It benefits from the data set already being sorted. On unsorted data, after each swap it must carry out a comparison again on all data so performs poorly on badly sorted data sets.

## Merge Sort

Another comparison-based sorting algorithm, this is considered to be an efficient algorithm which divides the problem recursively until the base case, when an individual array of two items is being compared and sorted.

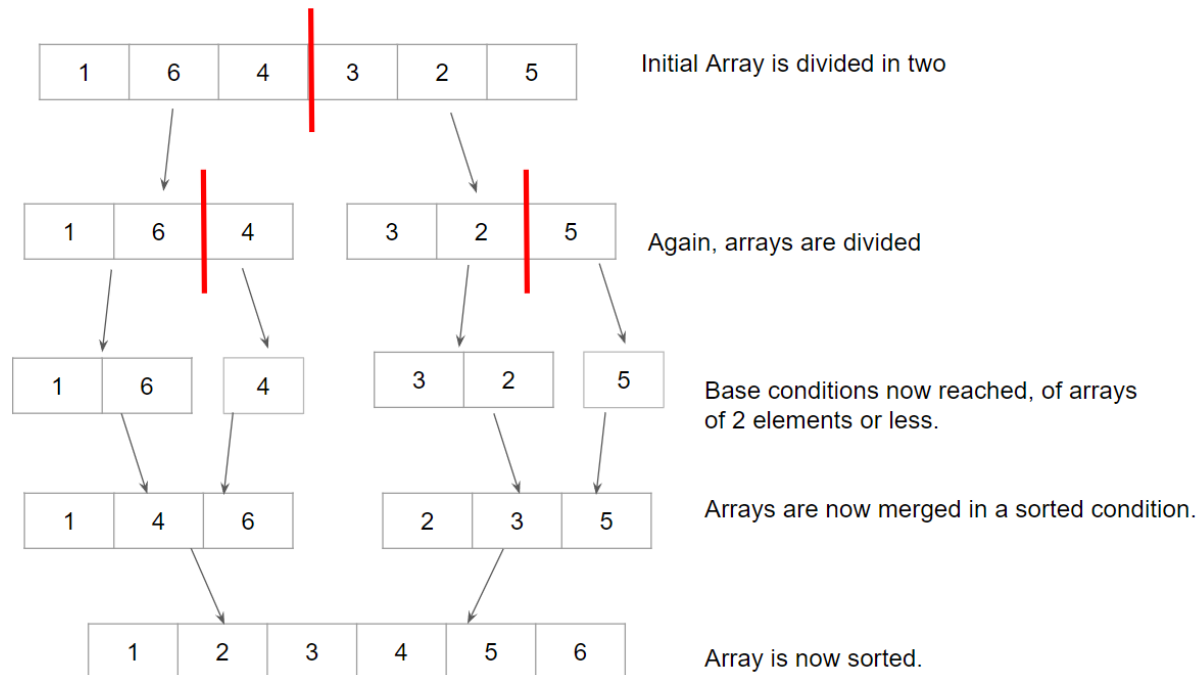
A stable sorting algorithm it will reproduce the same arrangement of duplicate objects on successive runs.

## Space and Time Complexity

This algorithm provides both Best and Worst case time complexity of  $O(n \log n)$  and is therefore considered to be a very good option.

The drawback for this algorithm is its space complexity as due to the recursive divide and conquer approach it has a space complexity of  $O(n)$ .

## Diagrams



## Different Inputs

Merge sort is quite stable for data sets with varying degrees of initial sorting. This stability of achieving  $O(n \log n)$  for the best and worst case of time complexity is at the expense of space complexity as  $O(n)$  becomes expensive and memory intensive on large data sets.

## Radix

This is a non-comparison based sorting algorithm whereby individual digits in values are examined as opposed to the whole value. The radix of a value is essentially the power of its numeric value to the base, ie. base 2, or base 10.

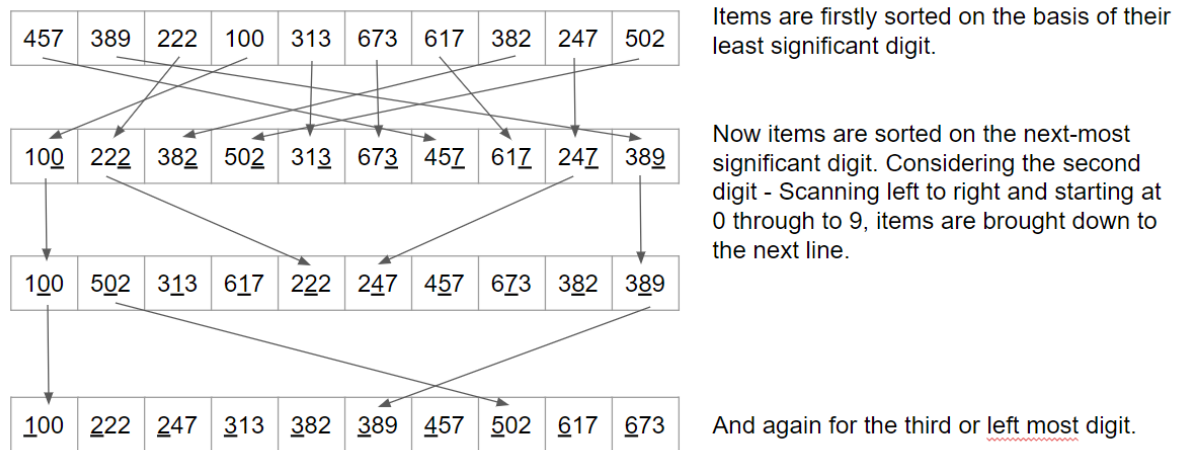
There are two approaches to radix sort, least significant digit (LSD) and most significant digit (MSD). LSD is favoured for sorting of integers and MSD is the preferred options for sorting objects such as strings or other objects with fixed length.

## Space and Time Complexity

The time complexity of Radix sort is given as  $O(n \times d)$  where  $d$  is the maximum number of digits in a value to be sorted.

The space complexity is essentially  $O(n)$  or  $O(b(n + d))$  where  $d$  is the number of digits and  $b$  is the base.

## Diagrams



## Different Inputs

The performance of the radix sort algorithm degrades with longer values i.e. where the base of the value is small and more digits are required to represent the value.

## Selection

A simple comparison-based sorting algorithm where the array is divided between sorted and unsorted. The minimum value in the array is selected and swapped with the first item, the first item is now sorted and the process continues.

This is a non-stable style algorithm and the arrangement of equal objects depends on their position in the array.

## Space and Time Complexity

The time complexity for this algorithm is  $O(n^2)$  and is therefore a poor choice for large sets of data.

As an in-place method the space complexity of  $O(1)$  is good.

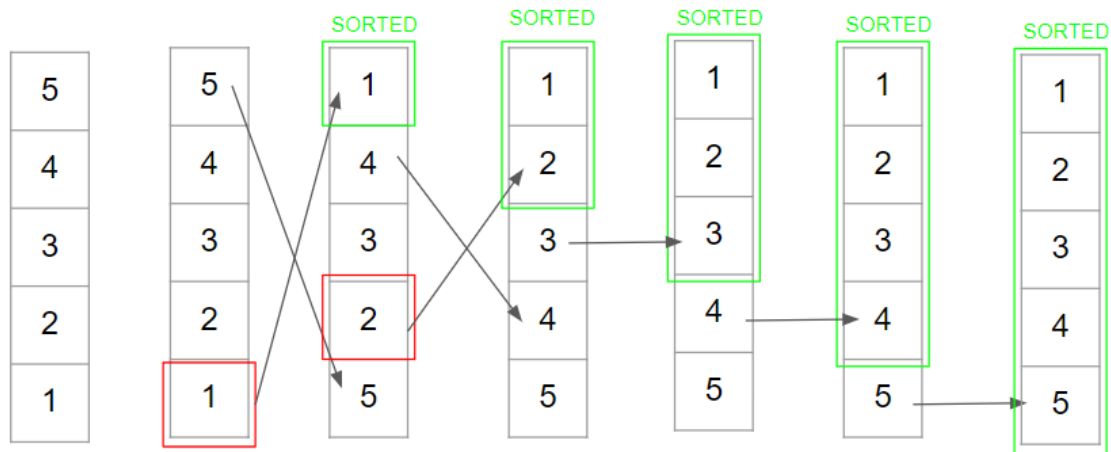
## Diagrams

Step 1: The least value is selected in the array and swapped with the first entry. Below, value 1 at position 4 in the array is swapped with 5 at position 0.

Step 2: The first entry is now considered sorted. Of the remainder, the unsorted section of the array, the least value is again selected and swapped now with the second element of the array. Below, value 2 is swapped with 4.

Step 3: The first and second entries of the array are now considered sorted. Again, in the unsorted section the least value is selected, in this case 3. It is already placed at the correct position in the array.

Step 4, 5 and 6: Again the process is repeated, in these instances the least values are found to be in the correct position and finally the array is found to be sorted.



## Different Inputs

Selection sort does not provide much improvement in performance between best and worst case scenarios in relation to time complexity.

## Insertion

Insertion is another comparison based sorting algorithm whereby the array is divided between sorted and unsorted sections. The array is divided in two sections, the division starting at the second element of the array. The second section of the array is essentially sorted into the first.

## Space and Time Complexity

The time complexity of this algorithm is  $O(n^2)$  and is therefore a poor choice for large data sets.

As an in-place method with space complexity is good at  $O(1)$ .

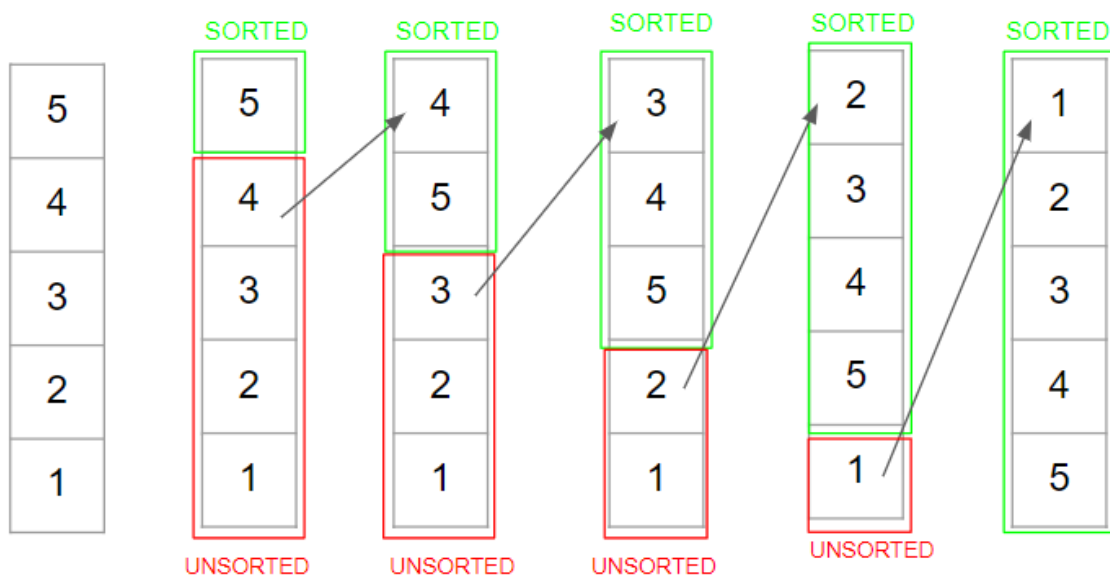


## Diagrams

Step 1: The array is split, the first element is considered sorted, the remainder of the array is considered unsorted.

Step 2: The unsorted portion of the array is considered, the first element is examined and placed into the sorted portion in the appropriate position. This process may require an iterative or recursive approach to perform as the item to be inserted into the 'sorted' section of the array must itself be compared and sorted.

Step 3: The process in Step 2 is repeated for the unsorted portion of the array until the array has been fully sorted.



## Different Inputs

The insertion algorithm is considered to be a stable algorithm that maintains the arrangement of duplicate objects.

The performance of the algorithm ranges between  $O(n)$  and  $O(n^2)$  depending on the degree to which data in the array is already sorted.

# Implementation and Benchmarking

## Process

The benchmarking program developed averages the results of 10 iterations of each sorting algorithm on a random array of size ranging from  $n = 100$  to 10,000 as per the project specification. The random arrays are populated with values between 0 and 99 and each run of each algorithm is applied to a clone of the same unsorted array for consistency.

The program is to be compiled and executed using the Runner class which contains the Main method. The algorithms applied are detailed in the next sections.

## Bubble Sort

```
/*
 * BUBBLE SORT
 * Reference: https://www.geeksforgeeks.org/bubble-sort/
 */

public void bubble(int[] sortThis) {

    int elementsSorted = 1; // Use one to start and last index of array

    boolean swaps = false;

    do {

        swaps = false; // Reset swaps flag

        for(int i = 0; i < sortThis.length - elementsSorted; i++) {
            //Iterate over array for comparisons

            if(sortThis[i] > sortThis[i+1]) { // Compare in pairs

                int temp = sortThis[i];
                sortThis[i] = sortThis[i+1];
                sortThis[i+1] = temp;

                swaps = true; // Set flag that swap occurred
            }

            elementsSorted++; // Increase last element sorted
        }

    }while(swaps); // Finish when sorted
}
```

## Merge Sort

```
/*
 * MERGE SORT
 * Reference: https://www.geeksforgeeks.org/merge-sort/
 */

private void merge(int[] toSort, int left, int mid, int right) { //
Method to sort sub-arrays and merge them

    int splitA = mid - left + 1; // Get the length of the two sub-arrays
to sort/merge
    int splitB = right - mid;

    int firstPart[] = new int [splitA]; // Temporary arrays to store the
sub-arrays
    int secondPart[] = new int [splitB];

    for(int i = 0; i < splitA; i++) { // Populate the temporary arrays

        firstPart[i] = toSort[left + i]; // Based on index of left
portion start incrementing with i
    }

    for(int j = 0; j < splitB; j++) {

        secondPart[j] = toSort[mid + j + 1]; // Based on index of
midpoint where split + 1 incrementing with j
    }

    int i = 0, j = 0;

    int k = left; // Variable to locate the start of the
index of the array to sort and then traverse

    while(i < splitA && j < splitB) { //Do-While there both
sub-arrays

        if(firstPart[i] <= secondPart[j]) {
//Compare, assign and move index on array to sort
            toSort[k] = firstPart[i];
            i++; // Increment i
        } else {
            toSort[k] = secondPart[j];
            j++; // Increment j
        }
        k++; // Increment k
    }

    while (i < splitA) { //Check for any element left in the sub-arrays
and assign it to the array
        toSort[k] = firstPart[i];
        i++; // Increment i
        k++; // Increment k
    }
    while (j < splitB) {
        toSort[k] = secondPart[j];
        j++; // Increment j
        k++; // Increment k
    }
}
```

```

    }

    //Done, recursion in "sort()" will continue and sort/merge the next
sub-array until all done
    }

    //Method to recursively divide the array until we reach sub-arrays of 1
element (base case)
    //Merge is always a sub-array of at least two elements
    public void sort(int toSort[], int partA, int partB) {

        if(partA < partB) { // Act on smaller portion

            int midPoint = (partA + partB)/2; //Get the index in the
middle

            sort(toSort, partA, midPoint); //Recursively get half of
both sub-arrays and sort
            sort(toSort, midPoint + 1, partB);

            merge(toSort, partA, midPoint, partB); // Now merge

        }

    }
}

```

## Radix Sort

```

/*
 * RADIX SORT
 * Reference: https://www.geeksforgeeks.org/radix-sort/
 *
 */

//Method to sort the array by Least Significant Digit
//Parameters: Input Array, Array Length

public void radix(int[] toSort, int n) {

    int max = getMax(toSort, n); //Find max value of the array using
method

    for(int exponent = 1; max/exponent > 0; exponent *= 10) { // Loop
over array examining the significant digit beginning with the rightmost, i.e.
remainder of m/exp
        countSort(toSort, n, exponent); //Send to counting sort
method with the radix to examine
    }

}

private int getMax(int[] toSort, int n) { //Find the maximum value of
the array

    int max = toSort[0]; //Set the first element as the max

    for(int i = 1; i < n; i++) { //Compare and swap if larger get
the max

```

```

        if (toSort[i] > max) {
            max = toSort[i];
        }
    }

    return max;
}

private void countSort(int[] toSort, int n, int exp) { // Method
implementing Counting Sort Algorithm

    int[] output = new int[n]; // Array for storing result

    int i;

    int[] count = new int[10]; //Array for storing counts of each
value (0-9 as decimal)
    Arrays.fill(count, 0); // Set counts/instances to zero for all
digits

    for (i = 0; i < n; i++) { //Loop over array 'counting' the
number of instances of each digit
        count[(toSort[i]/exp)%10]++;
    }

    for(i = 1; i < 10; i++) { // Loop over setting the index of where
each value in sorted array should rank
        count[i] += count[i - 1];
    }

    for(i = n - 1; i >= 0; i--) { // Loop from right (for stability of
result) and assign to output array the value of toSort, i.e. the amount of times
the value appears
        output[count[(toSort[i]/exp)%10] - 1] = toSort[i];
        count[(toSort[i]/exp)%10]--;
    }

    for(i = 0; i < n; i++) { // Change output array toSort with the
sorted result
        toSort[i] = output[i];
    }
}

```

## Selection Sort

```

/*
 * SELECTION SORT
 * Reference: https://www.geeksforgeeks.org/selection-sort/
 *
 */

public void selection(int[] sortThis) {
    //Variables for the Loops
    int leftSection = 0;
    int rightSection = 0;
    int sorted = 0;
}

```

```

        for(leftSection = 0; leftSection < sortThis.length -1;
leftSection++) { //Loop over the array

            sorted = leftSection; // Set the sorted portion

            for(rightSection = leftSection + 1; rightSection <
sortThis.length; rightSection++) { // Loop over array from next element (left+1)

                if(sortThis[rightSection] < sortThis[sorted]) { //
Compare each element with the minimum
                    sorted = rightSection;
                }
            }

            int temp = sortThis[leftSection]; // Swap minimum value with
the left position
            sortThis[leftSection] = sortThis[sorted];
            sortThis[sorted] = temp;
        }
    }
}

```

## Insertion Sort

```

/*
 * INSERTION SORT
 * Reference: https://www.geeksforgeeks.org/insertion-sort/
 */

public void insertion(int[] toSort) {

    for(int i = 1; i < toSort.length; i++) { //Loop over the array
starting at the second element

        int key = toSort[i]; //Select the element to
compare
        int j = i-1; //Select index where to start to
compare (to the left)

        while(j >= 0 && toSort[j] > key) { // While loop moves along
elements and compares to key

            toSort[j+1] = toSort[j];
            j = j-1; //Move the index of comparison one position
to the left

        }

        toSort[j+1] = key; // Insert key value

    }
}

```

## Results

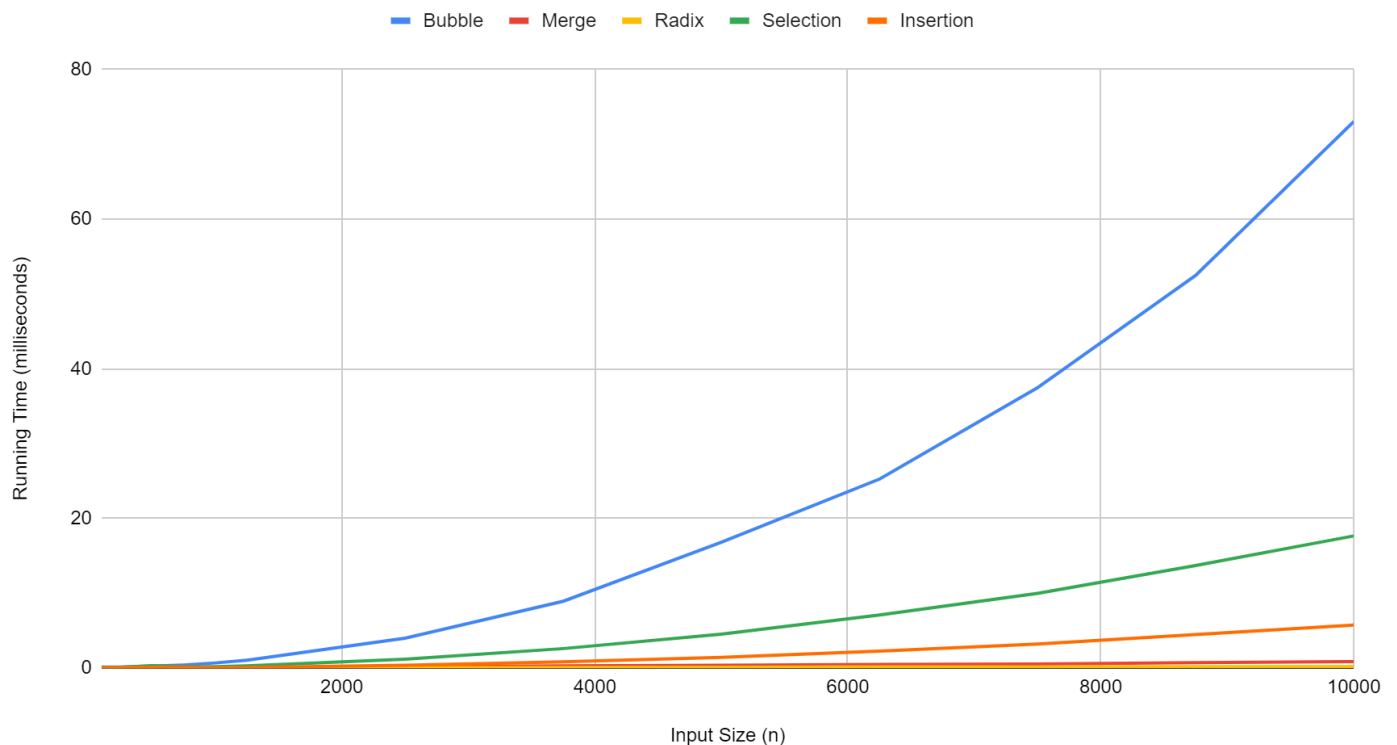
From the results in the table and chart below the behaviour of the various algorithms and their time complexity is readily demonstrated.

The three  $O(n^2)$  algorithms - Bubble Sort, Selection Sort and Insertion Sort can be seen to rise considerably in running time as the size of input increases and as expected.

In comparison, the  $O(n \log n)$  and  $O(n \times d)$  performance by Merge Sort and Radix Sort can be seen to behave similarly and are suited to larger data set sizes. The  $O(n^2)$  performance of Bubble Sort, Selection Sort and Insertion Sort increases at a higher rate as the input size increases.

Size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble	0.158	0.162	0.279	0.418	0.71	1.076	4.02	8.956	16.848	25.275	37.499	52.511	73.048
Merge	0.046	0.027	0.054	0.055	0.078	0.095	0.195	0.311	0.403	0.498	0.569	0.738	0.893
Radix	0.026	0.035	0.07	0.099	0.061	0.083	0.089	0.109	0.135	0.179	0.185	0.216	0.222
Selection	0.083	0.119	0.342	0.187	0.213	0.327	1.206	2.625	4.544	7.118	10.012	13.73	17.68
Insertion	0.048	0.109	0.14	0.15	0.067	0.101	0.387	0.846	1.461	2.277	3.236	4.504	5.763

Benchmark Times



## References

### **Space complexity - Wikipedia**

Space complexity - Wikipedia. (2022). Retrieved 15 May 2022, from [https://en.wikipedia.org/wiki/Space\\_complexity](https://en.wikipedia.org/wiki/Space_complexity)

### **Analysis of Algorithms | Big-O analysis - GeeksforGeeks**

Analysis of Algorithms | Big-O analysis - GeeksforGeeks. (2018). Retrieved 15 May 2022, from <https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/>

### **Bubble Sort - GeeksforGeeks**

Bubble Sort - GeeksforGeeks. (2014). Retrieved 15 May 2022, from <https://www.geeksforgeeks.org/bubble-sort/>

### **Selection Sort - GeeksforGeeks**

Selection Sort - GeeksforGeeks. (2014). Retrieved 15 May 2022, from <https://www.geeksforgeeks.org/selection-sort/>

### **Insertion Sort - GeeksforGeeks**

Insertion Sort - GeeksforGeeks. (2013). Retrieved 15 May 2022, from <https://www.geeksforgeeks.org/insertion-sort/>

### **Radix Sort - GeeksforGeeks**

Radix Sort - GeeksforGeeks. (2013). Retrieved 15 May 2022, from <https://www.geeksforgeeks.org/radix-sort/>

### **Merge Sort - GeeksforGeeks**

Merge Sort - GeeksforGeeks. (2013). Retrieved 15 May 2022, from <https://www.geeksforgeeks.org/merge-sort/>

### **Time and Space complexity of Radix Sort**

Time and Space complexity of Radix Sort. (2021). Retrieved 15 May 2022, from <https://iq.opengenus.org/time-and-space-complexity-of-radix-sort/>