Computational Thinking with Algorithms (46887)

Sorting Algorithm Report

Table of Contents

[Introduction: 3](#_Toc69645085)

# Introduction:

# Sorting Algorithms:

In this section the different sorting algorithms will be introduced and have their space and time complexity discussed. Each algorithm will be explained how it works and how it handles different kinds of inputs, there will also be a diagram present which will show the behaviour of algorithm.

The different types of sorting algorithm that were looked at and the specific ones chosen for this project are listed below, before expanding on each in more detail in their own sections.

1. A simple comparison-based sort - Bubble Sort

2. An efficient comparison-based sort – Quicksort

3. A non-comparison sort - **(Counting Sort, Bucket Sort or Radix Sort)**

4. Any other sorting algorithm of your choice - **TODO**

5. Any other sorting algorithm of your choice - **TODO**

## Bubble Sort

### Introduction:

Bubble Sort is a very simple sorting algorithm, it works by stepping through the input array and comparing each element against its neighbour, swapping their positions if the neighbour element is smaller than the current one. It is not a very practical algorithm due to it’s time complexity which will be discussed below but thanks to its relative simplicity, it’s a good example for teaching and as an intro into sorting algorithms as a concept. (https://en.wikipedia.org/wiki/Bubble\_sort)

### Space and Time Complexity:

The bubble sort function I defined is performed iteratively and not recursively, It has a space-complexity of O(1) or 1 as it performs it’s sort on the array in-place.

For Time Complexity bubble sort is in best-case O(n) – when the input array is already sorted.

On average it has O(n2)

It has time complexity in it’s worst case of O(n2) – when the input array is totally unsorted.

### Explanation:

How Bubble sort works is it makes use of loops to control stepping through the array, the outer loop iterates through each index of the array with the inner loop iterating over every index up till what is sorted. Per iteration of the outer loop the right most unsorted index get’s sorted, with the largest remaining unsorted value in the array ‘bubbling’ up to the top. For some moderate performance improvements, we take advantage of the fact that the largest value that is unsorted in the current loop of the array will be sorted by the end of that loop, which means we no longer need to compare against a sorted index once it has been sorted, so this means that we can reduce the number of iterations on the inner loop on each completed loop of the outer.

## QuickSort

### Introduction:

Quicksort is a recursive sorting algorithm and is a more practical algorithm (due to it having better space and time complexity in general) than the Bubble Sort and is still often used. At a high level it works using a pivot point (this can be picked a number of ways but for my implementation it is the first element in the array) to split the array to be sorted into left and right halves which are less than and greater than the pivot elements value. It then recursively does the same operation on each half until there are only 1 or 2 elements, at which point the array has been sorted.

### Space and Time Complexity:

The Space Complexity for this algorithm is O(n), meaning it takes more memory than the simple Bubble Sort to run, this is due to the recursive nature of quicksort, while the array is modified in place and so no auxiliary data structures are required, the stack is increasing on each recursive call.

Time Complexity in the best case is O(n log n).

Time Complexity in the average case is O(n log n).

Time Complexity in the worst case is O(n2).

Quicksort’s time complexity is the same on average as it is in the best case, which makes it a generally good and reliable method for sorting with it’s worst case happening only rarely, such as when the array has already been sorted, this makes it a far more efficient algorithm especially on larger arrays than something like Bubble Sort.

### Explanation:

The implementation of Quicksort used in my code has two functions, pivot and quicksort.

Pivot takes in three inputs, the array to be sorted and start and end which determine the scope within the array to look at. On a first call of pivot start and end will be the first and last elements of the array but on subsequent calls this will change. How the pivot function works is it chooses an element in the array between start and end that will be the pivot value (in my case the first element) and using this value it sorts the other values between start and end in the array into low and high halves, those lower than the pivot value go into the low and those higher go to the high, this is all done in-place and so it swaps elements within the array to achieve this. This function returns the index at which the pivot point is.

Quicksort function similarly takes in three inputs, the array to be sorted and start and end, again start and end determine the scope within the array to look at, however one part that differs here are that I have the quicksort function set up so that start and end are optional arguments and so if not specified supply default values (I have done so to allow reusability of my timer function which will be discussed in the later “Implementation and Benchmarking” section.). The function works by first checking start is greater than or equal to end, this is the base case and it returns True to denote that it has finished sorting. After this it calls the pivot function to sort the array into high and low and correctly position the pivot element, following this, the function recursively calls itself twice for the first it sends the array and the start and end for the left or low half of the array and for the second call it sends the array and the start and end for the right or high half of the array. These calls perform the same steps but on increasingly smaller parts of the array until each recursive call hits their base case and have sorted their respective parts of the array.