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

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.

## 2. 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.

## 3. Counting Sort

### Introduction:

Counting Sort is a non-comparison based sorting algorithm which is used on integers, performed iteratively and is not in-place. What makes this such an interesting type of algorithm is that unlike comparison based ones it doesn’t rely on comparing one value against another to see if it is larger and then swapping, instead it like the name implies counts how many times each value occurs and uses that to sort. Conceptually it is relatively simple, it steps through the input array to be sorted, using a second array it keeps track of the number of occurrences of each value using its index and the values at each index. Once the entire array has been iterated over and been unpacked into the counting array the values get unpacked from the count array where they are greater than 0, so if the first value greater than 0 was at index 1 and it had a value of 5, then the sorted array would have 5 1’s and so on through the rest of the count array until there is a sorted array.

There are however some limitations to counting sort algorithm and some assumptions it relies on in order to work. Regarding limitations, it doesn’t handle too large a range of values well and it can become a burden on memory and time to run, generally counting sort should only be applied on situations where the range of values or number of keys will not be too large. For assumptions counting sort expects non-negative numbers, and in a lot of cases expects to be given the maximum value of the array to be sorted as an input, in the case of my implementation I have this as an optional parameter which can be determined at run-time however this does impact run-time.

### Space and Time Complexity:

The Space Complexity for this algorithm is O(n+k) (k here represents the max key value), as both the size of the array to be sorted and the maximum value have an impact on the amount of memory needed.

Time Complexity in the worst case is O(n+ k), similar to the space complexity as the size of both the array and the maximum value in the array increase so too will the run-time.

### Explanation:

The counting sort function I wrote takes in two input arguments, the input array to be sorted and an optional argument for the max value in the input array, if no value supplied for the max value then this is determined using python’s max function.

A new list/array for counting the number of times each value occurs is initialised with all 0’s with a range of 0 to the max value from the input array plus 1, as the new array has indices for each possible value of the input array the value at these indices will represent a count for how many times each value occurs.

A second array is initialised as an empty array, this array will ultimately be used to store the sorted list and will be what the function returns, as this is not an in-place sorting algorithm the original input array will remain the same.

Incrementing through each item in the input array it uses that item to add 1 to the index representing that item in the count array.

Finally the code increments through the count list using enumerate (I did this to remove the need for a nested for loop and from testing this seems to have improved the run-time reasonably, I also experimented with using len(range(array) but saw no real difference in run-time over several tests, ultimately chose enumerate for readability), this allows the function to track the current index and the value at that index in two separate variables. As it loops through if the current item is 0 continue the loop as there’s nothing there to sort, if the value is greater than 0 then to the sorted list it appends a list object enclosing index multiplied by the value for the current index, this is a useful python way for getting multiple of the same values into a list and is the same logic as how the count array was created earlier. Once this loop finishes the sorted input array has been stored in the sorted array and this is returned.