Computational Thinking with Algorithms: Project 2021

## Eoin Lees – G

## Higher Diploma in data

# Introduction

Concept of sorting and sorting algorithms

Sorting is the action of “arranging a collection of items according to some pre-defined ordering rules.” (Mannion, 2021)

In computing it is an important concept as a large porting of computing work is spent on sorting. It has therefore since the beginning of modern computing been an important filed of study for optimization. By sorting information in advance, many tasks can be simplified greatly and take up less computational power. It is often an important part of other algorithms too, so its optimization to a specific task is important ??waffle

Sorting algorithms in computing began in 1945 with the development of merge sort by John von Neumann. Multiple other sorting methods were also developed in the early days of computing e.g Radix sort (1954), Counting sort (1954), Quick sort (1962) etc.

### Comparator Functions

Takes two elements of any type and compares one to the other.

Which input, a or b should appear first according to predefined definition of comparator function.

E.g < = or >

This project is focused on integer sorting. So greater than or less than will be used.

It is possible to sort strings using lexicographical methods. i.e how letters appear in alphabet, or items appear in dictionary.

### Comparison-based-sorts

“A comparison sort is a type of sorting algorithm which uses comparison operations only to determine which of two elements should appear first in a sorted list.” (Mannion, 2021)

A sorting algorithm is called ***comparison-based*** if the only way to gain information about the total order is by comparing a pair of elements at a time via the order ≤. (Mannion, 2021)

In comparison-based sorting no algorithm that sorts by this method can do better than n log n performance in the average or worst cases. This is reflected in table 1.

There are many sorting algorithms that are in this category (e.g. Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quicksort, Heapsort).

They are the most widely applicable sorts for a diverse type of input data.

### Non-comparison-based-sorts

“Under some special conditions relating to the values to be sorted, it is possible to design other kinds of non-comparison sorting algorithms that have better worst-case times (e.g. Bucket Sort, Counting Sort, Radix Sort)” (Mannion, 2021)

These methods can achieve better “worst case” times as shows in the table below however they are not as stable.

### Stability

Stability in a sorting algorithm as it is important to preserve the order of already sorted data. This is a desirable property in sorting algorithms.

### In-place sorting

Sorting algorithms have different memory requirements, which depend on how the specific algorithm works.

A sorting algorithm is called in place if it uses only a fixed additional amount of working space, independent of the input size.

Other sorting algorithms may require additional working memory, the amount of which is often related to the size of the input n

In place sorting is a desirable property if the availability of memory is a concern (Mannion, 2021)

Rewrite all this in my words

### complexity (time and space)

Write about it here… find notes

### Performance

Table shows run time effeciancy

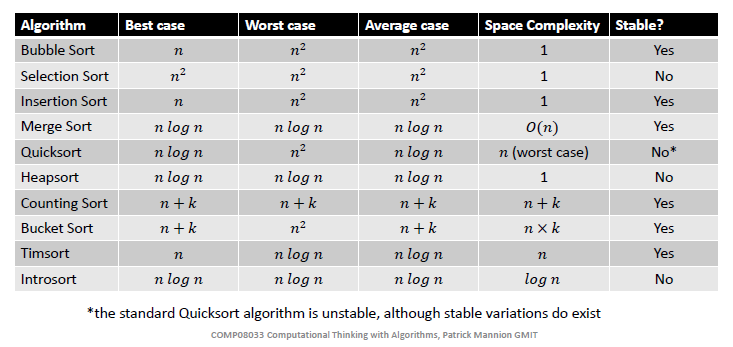


Table 1: Overview of sorting algorithms

Show graph from notes

# Sorting Algorithms

## Bubble Sort

Bubble sort takes its name from having larger values in a list “bubble up” to the end of the list as the sorting takes place. It was first analysed formally as early as 1956. It is a comparison-based sorting method.

It has a time complexity of n (linear) in best case and n2 (quadratic) in the worst and average cases. (table 1)

It is an in-place sorting algorithm. It uses only the size of the input data set plus a constant amount of additional working space to function. The additional working space is just a few variables required to make the swaps of the elements in different places. A temporary variable is used here.

It is a slow sorting method. It is impractical for most problems. It only really works well on small data sets, or on data that is already nearly sorted.

How it works:

* Compare each element to the element on it right (exclude last element)
  + If out of order, swap. Then repeat to next element.
  + Largest element ends up at end of list in its final position.
* Repeat process (exclude last two elements)
  + If out of order, swap. Then repeat to next element.
  + Second largest element ends up in its final position.
* Continue until there are no more remaining unsorted elements on the left.

The algorithm assumes every element on the left is unsorted, and the right is sorted. It then has two separate arrays of data. Sorted and un-sorted.

(Diagram of explination) \* Save until tomorrow

Different input instances – graph made already – single out each

## Merge Sort

Introduce algorithm

Discuss space and time complexity

Explain how it works

(Diagram of explination) \* Save until tomorrow

Different input instances – graph made already – single out each

## Counting Sort

Introduce algorithm

Discuss space and time complexity

Explain how it works

(Diagram of explination) \* Save until tomorrow

Different input instances – graph made already – single out each

## Insertion Sort

A simple comparison-based sorting algorithm. It is similar to the method used usually by card players sorting cards. It is intuitive to understand. It is easy to implement, is a stable sorting algorithm, it works in-place (efficient with memory) and is very good for small and partially sorted lists.

In best case scenarios it runs in n + d time, where d is the number of inversions in the input instance.

It is very inefficient for large lists that are randomly ordered.

It is an iterative approach. It splits the list into sorted and unsorted sublists. ( head and tail)

Complexity:

The total number of data comparisons is the number of inversions d plus n-1. On average

How it works:

* Starting on the left of the array, the key is set as the element at index 1.
  + Any elements on the left greater than the key get moved right by one position and the key gets inserted.
* Set the key as the element at index 2.
  + Any elements on the left greater than the key get moved right by one position and the key gets inserted.
* Set the key as the element at index 3.
  + Any elements on the left greater than the key get moved right by one position and the key gets inserted.
* Continue until the key is set at n-1 index.
  + Any elements on the left greater than the key get moved right by one position and the key gets inserted.
* The array is sorted.

(Diagram of explination) \* Save until tomorrow

Different input instances – graph made already – single out each

## Quick Sort

Introduce algorithm

Discuss space and time complexity

Explain how it works

(Diagram of explination) \* Save until tomorrow

Different input instances – graph made already – single out each

# Implementation and Benchmarking

Describe process followed when implementing the application

Present results:

Table of results

Results graphed against each other

Discuss how measured results differed

Were they similar to what you expected - Given time complexity of each chosen algorithm

Table and graph labelled appropriately

# References

Mannion, P., 2021. Sorting Algorithms, Part 1. In: *COMP08033 Computational Thinking with Algorithms,.* s.l.:GMIT.

Wikipedia, 2021. *Wikipedia.* [Online]   
Available at: https://en.wikipedia.org/wiki/Sorting  
[Accessed 12 May 2021].

**No table of figures entries found.**