# 122COM: Sorting algorithms

David Croft
david.croft@coventry.ac.uk

November 20, 2015

## 1 Introduction

This week we are looking at the problem of sorting data into order. Sorting is a classic problem in computer science and there are a wide range of different sorting algorithms available. You will be understanding and implementing various sorting algorithms. This is also an opportunity to use the profiling skills that you learnt during the previous weeks to examine code performance in a more realistic setting.

This week will be taught in Python and C++. It should be clear by this point which language you are expected to use for your answers.

## 2 Sorting attributes

Different sorting algorithms have different attributes which affect how suitable they are for various tasks. The fastest algorithm may not always be appropriate so it's important that you understand these factors so that you can make informed decisions.

### 2.1 Stable sort

Stable sorting algorithms do not change the order of equivalent elements i.e. elements with the same value not have their relative orders changed after a stable sorting. With an unstable sorting algorithm the relative orders or equivalent elements can be changed.

For some applications this is an important consideration. Imagine a queue in an emergency room. You want to treat the most serious conditions first, so you sort the people based on how bad their injury is. However, if two or more people have the same injury then they should get seen based on when they entered the queue.

### 2.2 In place sort

Certain algorithms are able to sort a sequence without requiring additional space (or only a small amount of additional space). In place algorithms are much more memory efficient than other algorithms. Bubble sort is an example of an in place sorting algorithm. In place sorting can be an important feature when dealing with large amounts of data or limited computational resources.

## 2.3 Divide and conquer

These algorithms work by dividing the sequence to be sorted in sub sections and working on each subsection separately. The results of each subsection are then combined to produce the final sequence.

Divide and conquer algorithms are naturally suited to parallel processing and so can be highly efficient when multiple processors/cores are available. Quicksort and mergesort are both examples of divide and conquer algorithms.

## 2.4 Random access

The nature of the data structure being sorted will have a big effect of which algorithms are most effective. An obvious example of this are arrays and linked lists. As we've seen in previous weeks arrays and linked lists are both linear data structures. Linked lists have the advantage that we can easily add or remove elements from the middle of the list, but in order to access a random location in the list we have to step through it one element at a time. Arrays have the advantage that we can access any element directly, but adding or deleting elements means shuffling the rest of the array elements.

## 2.5 Complexity

Complexity is often used to describe sorting algorithms @@@. Sorting algorithm complexity is more complex than some other examples as the amount of time required to sort a sequence depends heavily on the starting order of that sequence.

For example and depending on the algorithm, a sequence that is already sorted may take less time to sort than and a sequence which is in reverse order. Sorting algorithms are typically given 3 complexity values which describe their best case, worst case and average performance. The worst and average cases are the most important as these control the guaranteed completion time for an algorithm and the typical performance.

# 3 Introduction

## 3.1 Bubble sort

The most basic sorting algorithm is bubblesort. It's introduced here to give you an introduction to sorting algorithms and has been superseded by better algorithms. While the average performance of bubblesort is often very poor and the worst case is terrible, it has good performance if the sequence is almost sorted before it starts.

Bubblesort uses a single cursor to iterate over the list one element at a time, in each iteration the element at the cursor is compared to the next element in the list. If the elements are the correct order the cursor moves on, if the elements are in the wrong other than the elements are swapped and the cursor moves on. When the cursor reaches the end of the list the cursor moves back to the beginning and the process repeats until the list is in order.

Under the best case scenario[1] it requires a single iteration. The best case complexity is, therefore, $O(n)$. Under the worst case scenario[2] the complexity is $O(n^2)$.

[1] The sequence is already sorted.

[2] The sequence is in reverse order.

**Pre-lab work:**

1. Implement bubblesort.
    - Some basic code is provided for you in pre_bubblesort along with some code to test your implementations.

### 3.2 Selection sort

Selection sort uses a cursor to divide the list into two sections, sorted and unsorted. Everything behind the cursor is sorted, everything in front of the cursor is sorted. It does this by iterating over the list one element at a time, in each iteration the lowest value in the unsorted region is swapped with the value at the cursor. The cursor then moves to the next element and the process repeats.

**Lab work:**

2. Using the bubblesort code as a starting point, implement a selection sort function.
3. What are the best and worst case complexities for selection sort?

## 4 Quicksort

Quicksort is, as it's name suggests, a very fast sorting algorithm. It was designed by Tony Hoare in 1960. It normally has an $O(n\ log\ n)$ complexity but occasionally reaches $O(n^2)$.

Quick sort works by splitting the list into smaller and smaller sub lists using a pivot.

1. Select a pivot element from the list.
2. Split list into two sub lists, one sub list contains elements $<$ pivot and the other elements that are $\geq$.
3. Repeat from step 1 on each sub list until list size is $\leq 1$.

Figure 1 on the following page shows a visualisation of quicksort on ten random numbers. In this figure pivot values are shown as circles, lists as rectangles. In this figure the middle element of each list is selected as the pivot.

Any element in the list can be selected as the pivot. Ideally we want to pick a pivot value that will cause the two sub lists to be of equal size. When quicksort was originally proposed the first element in the list was used as the pivot but this causes problems if the list is already ordered. The same applies to using the last element.

Suggested approaches include selecting a random element, the middle element or the median of the first, last and middle element.

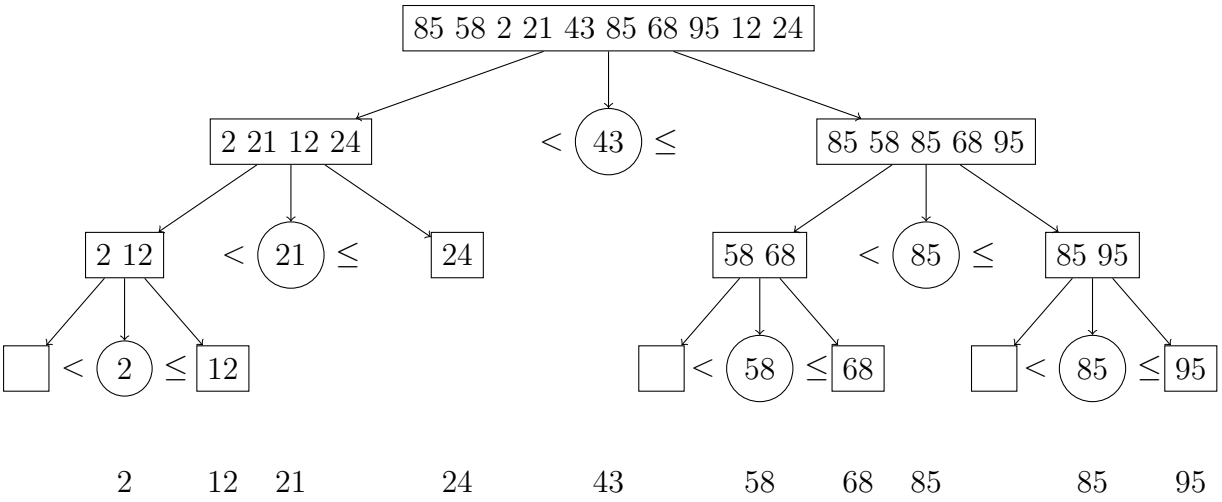


Figure 1: Visualisation of Quicksort on ten random numbers.

**Lab work:**

4. Implement quicksort. You can use the bubblesort code as a starting point but you will need to use recursion.
5. Compare the performance of bubblesort to quicksort by profiling both.
    - 110 resit students can skip this question.

# 5 Advanced work

**Extended work:**

6. See if you can write an in place implementation of quicksort.
7. Investigate merge sort, see if you can implement it. You may find it helpful to use the quicksort code as a starting point.
8. Investigate some of the joke sorting algorithms.
    - I.e. Bogo and Sleep sort, make sure you understand why they are bad algorithms.
    - Feel free to test them out but never use them in a real program.