A discussion of sorting algorithms

by Mark Baker

Education Vision Consultancy

A discussion of sorting algorithms

Firstly the operation of the different algorithms is examined, then their relative performance is discussed.

This article has been translated into **Serbo-Croatian** *by* ***Jovana Milutinovich*** *from* Geeks Education*.*

Introduction

I grew up with the bubble sort, in common, I am sure, with many colleagues. Having learnt one sorting algorithm, there seemed little point in learning any others, it was hardly an exciting area of study. Mundane sorting may be, but it is also central to many tasks carried out on computer. Prompted by its inclusion in the AEB 'A' level syllabus, I looked at the process again in detail. The efficiency with which sorting is carried out will often have a significant impact on the overall efficiency of a program. Consequently there has been much research and it is interesting to see the range of alternative algorithms that have been developed.

It is not always possible to say that one algorithm is better than another, as relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted. In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order. Different algorithms will perform differently according to the data being sorted. Four common algorithms are the exchange or bubble sort, the selection sort, the insertion sort and the quick sort.

The selection sort is a good one to use with students. It is intuitive and very simple to program. It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons and exchanges, so its performance is predictable.

```
procedure SelectionSort ( d: DataArrayType; n: integer)
{n is the number of elements}
   for k = 1 to n-1 do
      begin
      small = k
      for j = k+1 to n do
            if d[ j ] < d[small] then small = j
      {Swap elements k and small}
      Swap(d, k, small)
      end
```

Exchange (Bubble) Sort

Element	1	2	3	4	5	6	7	8
Data	**27**	**63**	**1**	**72**	**64**	**58**	**14**	**9**
1st pass	**27**	**1**	**63**	**64**	**58**	**14**	**9**	**72**
2nd pass	**1**	**27**	**63**	**58**	**14**	**9**	**64**	**72**
3rd pass…	**1**	**27**	**58**	**14**	**9**	**63**	**64**	**72**

The first two data items (27 and 63) are compared and the smaller one placed on the left hand side. The second and third items (63 and 1) are then compared and the smaller one placed on the left and so on. After all the data has been passed through once, the largest data item (72) will have "bubbled" through to the end of the list. At the end of the second pass, the second largest data item (64) will be in the second last position. For n data items, the process continues for n-1 passes, or until no exchanges are made in a single pass.

Insertion Sort

Element	1	2	3	4	5	6	7	8
Data	**27**	**63**	**1**	**72**	**64**	**58**	**14**	**9**
1st pass	**27**	**63**	**1**	**72**	**64**	**58**	**9**	**14**
2nd pass	**27**	**63**	**1**	**72**	**64**	**9**	**14**	**58**
3rd pass…	**27**	**63**	**1**	**72**	**9**	**14**	**58**	**64**

The insertion sort starts with the last two elements and creates a correctly sorted sub-list, which in the example contains 9 and 14. It then looks at the next element (58) and inserts it into the sub-list in its correct position. It takes the next element (64) and does the same, continuing until the sub-list contains all the data.

Selection Sort

Element	1	2	3	4	5	6	7	8
Data	**27**	**63**	**1**	**72**	**64**	**58**	**14**	**9**
1st pass	**1**	**63**	**27**	**72**	**64**	**58**	**14**	**9**
2nd pass	**1**	**9**	**27**	**72**	**64**	**58**	**14**	**63**
3rd pass…	**1**	**9**	**14**	**72**	**64**	**58**	**27**	**62**

The selection sort marks the first element (27). It then goes through the remaining data to find the smallest number (1). It swaps this with the first element and the smallest element is now in its correct position. It then marks the second element (63) and looks through the remaining data for the next smallest number (9). These two numbers are then swapped. This process continues until n-1 passes have been made.

Quick Sort

Element	1	2	3	4	5	6	7	8
Data	**27**	**63**	**1**	**72**	**64**	**58**	**14**	**9**
1st pass	**1**	**9**	**63**	**72**	**64**	**58**	**14**	**27**
2nd pass	**1**	**9**	**14**	**27**	**64**	**58**	**72**	**63**
3rd pass	**1**	**9**	**14**	**27**	**58**	**63**	**72**	**64**
4th pass **sorted!**	**1**	**9**	**14**	**27**	**58**	**63**	**64**	**72**

The quick sort takes the last element (**9**) and places it such that all the numbers in the left sub-list are smaller and all the numbers in the right sub-list are bigger. It then quick sorts the left sub-list ({**1**}) and then quick sorts the right sub-list ({**63,72,64,58,14,27**}). This is a recursive algorithm, since it is defined in terms of itself. This reduces the complexity of programming it, however it is the least intuitive of the four algorithms.

Comparing the Algorithms

There are two important factors when measuring the performance of a sorting algorithm. The algorithms have to compare the magnitude of different elements and they have to move the different elements around. So counting the number of comparisons and the number of exchanges or moves made by an algorithm offer useful performance measures. When sorting large record structures, the number of exchanges made may be the principal performance criterion, since exchanging two records will involve a lot of work. When sorting a simple array of integers, then the number of comparisons will be more important.

It has been said that the only thing going for the bubble (exchange) sort is its catchy name. The logic of the algorithm is simple to understand and it is fairly easy to program. It can also be programmed to detect when it has finished sorting. The selection sort, by comparison, always goes through the same amount of work regardless of the data and the quick sort performs particularly badly with ordered data. However, in general the bubble sort is a very inefficient algorithm.

The insertion sort is a little better and whilst it cannot detect that it has finished sorting, the logic of the algorithm means that it comes to a rapid conclusion when dealing with sorted data.

The selection sort is a good one to use with students. It is intuitive and very simple to program. It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons and exchanges, so its performance is predictable.

The first three algorithms all offer O(n2) performance, that is sorting times increase with the square of the number of elements being sorted. That means that if you double the number of elements being sorted, then there will be a four-fold increase in the time taken. Ten times more elements increases the time taken by a factor of 100! This is not a problem with small data sets, but with hundreds or thousands of elements, this becomes very significant. With most large data sets, the quick sort is a vastly superior algorithm (although as you might expect, it is much more complex), as the table below shows.

Random Data Set: Number of comparisons made

Sort/Elements	50	100	200	300	400	500
Selection sort	**1225**	**4950**	**19900**	**44850**	**79800**	**124750**
Exchange sort	**1410**	**5335**	**20300**	**45650**	**79866**	**126585**
Insertion sort	**1391**	**5399**	**20473**	**44449**	**78799**	**123715**
Quick sort	**339**	**990**	**1954**	**3384**	**5066**	**6256**

It should be pointed out that the methods above all belong to one family, they are all internal sorting algorithms. This means that they can only be used when the entire data structure to be sorted can be held in the computer's main memory. There will be situations where this is not possible, for example when sorting a very large transaction file which is stored on, say, magnetic tape or disc. Then an external sorting algorithm will be needed.

Sorting for Teachers

Sorting for Teachers is a program that allows students to step through the four algorithms mentioned above. It shows them how different data sets would be sorted, by highlighting the appropriate lines of the algorithm and displaying the variable values, as they progress through it.

- *Software link: Sorting for Teachers* 

Reference

Data Structures with Abstract Data Types and Pascal by Stubbs and Webre, pub. Brooks/Cole