Bubble Sort Explained

Paul Curzon

Queen Mary University of London

How do you put things in to order? You need a sort algorithm. Bubble sort is a very simple way of sorting data.

What is Sorting?

There are many meanings of the verb ‘to sort’. It could mean to fix things, it could mean to put things in to groups.To a computer scientist ‘**sorting**’ means something very specific. It means taking a series of pieces of data and putting them into a set order, one after the other, either from smallest to largest or from largest to smallest.

For example, sorting numbers means putting them in to numerical order. If we are given the numbers:

5, 7, 2, 99, 4

then after sorting them in to ascending order we have the list:

2, 4, 5, 7, 99

If we were to sort them into descending order then, instead, we would get the list:

99, 7, 5, 4, 2

We can sort other kinds of data. If instead we were given the words

“cat”, “hat”, “ant”

then sorting them in to (ascending) order would give the list:

“ant”, “cat”, “hat”

A **sort algorithm** is a set of instructions that, if followed, is guaranteed to put data in to order in this sense. There are many different sort algorithms: many different ways of putting the same data in to order.

Sorting is important because it is used as a way of making things easier to find. For example why is a printed telephone directory in sorted order? Because people will search for numbers in it frequently. It is only worth putting the effort in to sort data, if someone is going to search through it lots of times. A telephone directory is sorted in order of name not in order of telephone number, because people normally know a name they want to look up and want to find the corresponding number. Having sorted data makes it possible

to search faster. Here we focus on sorting itself though not on search algorithms, which is a separate topic.

There are many sort algorithms some more efficient than others. Bubble sort is a very simple algorithm for putting things in to order, and is a good place to start thinking about sort algorithms. We will explain it, starting with a simple version, and building up to a better version. First we need to know about arrays.

Arrays

The data we are to sort needs to be put somewhere. Programming languages provide all sorts of data structures to store things, though the simplest for sorting data is an **array**. An array is like a big table that can hold a series of pieces of data one after the other (think spreadsheet column). Each piece of data is stored in the next position in the table. We can refer to any particular piece of data by its position in the table. We call the position the **index**. There is one twist to the way arrays are normally numbered. Computer scientists start counting from 0. So the 'first' position of the array has index 0.The next has index 1 and so on. Counting from 0 isn't actually that strange. It is how (in the UK at least) we number the floors of a hotel or office tower block. The floor you enter on is the Ground floor, labelled G, not the first floor. The next floor up is floor 1 and so on. It's all spelled out in the lift where the column of numbers labelling the floor buttons goes: G, 1, 2, … Think of G as a malformed 0 and that office block is like an array, numbered from 0, with each floor holding the next piece of data.

We give arrays of data names, so we know which array we are talking about, and then refer to a particular piece of data stored in the array by the name together with the index giving the position. We will use the common notation, used by programming languages of putting the index in square brackets after the name. So if we have an array called a, then **a[0]** refers to the very first piece of data. **a[1]** refers to the next piece of data and so on. If we were to label our buildings like this, then if our tower block is called **TheShard**, then **TheShard[0]** refers to whatever is on the ground floor. **TheShard[1]** refers to the next floor up, and so on all the way to the top. If Bloggs and Co have offices on floor 21 then **TheShard[21]** is referring to their offices.

We often want to refer to a general entry in an array (or floor in a building), the particular one changing depending on the context. We will refer to the **i**-th position of array a as **a[i]**, whatever **i** currently is. Similarly, the nth floor of the Shard would be referred to as **TheShard[n]**. As i or n changes we can refer to different data (or floors).

Comparing and Swapping

Sorting is about putting things into order. To do that we need to compare all the different pieces of data some way or other. Standard computers, can't in general compare lots of things at once. Instead they have ways to compare pairs of things. Programming languages reflect this in the commands they give to control the computer. What do we mean by putting two things in order? For numbers we need to check if one number is greater than the other. For

letters we need to check if one letter is later in the alphabet than another. More generally, whatever the data we just need an operation that determines if one piece of data is ‘greater than’ another for some consistent meaning of ‘greater than’. We will use > to mean ‘greater than’, whatever the data, though as we will use sorting numbers as our running example. Substitute > with the appropriate operator to compare data and it will sort that data.

Let’s use a concrete example. Imagine a row of people, standing in a line. Each has written their age on a piece of paper that they hold but with the number hidden. Each can be referred to by their position in the line counting from 0 from one end. They are an array of ages. We want to sort them by age, putting the youngest at one end and the oldest at the other. How can we do it? We can’t see all the ages at once, all we can do in any one step is compare two people’s ages. If they are in the wrong order we can swap them. We could compare any two people at any time, but to keep things organised, let’s just compare ones next to each other.

In our notation the following:

```
(age [position] > age [position + 1])
```

tests if the number in position ‘position’ is greater than the value next to it in position: position+1.

To check if the earlier is the bigger of the two, and if so swap them, we can write:

```
IF (age [position] > age [position + 1])
THEN
    swap (age, position, position + 1)
```

Here

```
swap (age, position, position + 1)
```

just means swap the values in the place referred to as ‘position’ with the value next to it in array, age. In other words, the two people in those positions should just swap places. If we were writing a program to do this we would need to write instruction as to how to do that swap. We will ignore the detail of how swapping is done, though, to focus on the sorting itself.

Keep Going to the End

Of course doing that just once isn’t enough. We will need to do it over and over again before the whole is sorted. We can do that in an organised way by scanning down the array from one end to the other. Going back to our row of people, imagine having someone keep track of which pair we must compare. They are acting as position in the above code. They start pointing to the first person (in position 0). That person and the person next to them reveal their ages. We check them and if they are in the wrong order they swap positions. If they are in the right position relative to each other, then we leave them alone. That is what the IF statement in the code above is describing.

The person keeping track of the position then moves on one place, and points to the second person (in position 1). We do the same again swapping their positions if they are in the wrong order, before moving on. We do this with the

person keeping track of the position all the way from one end of the line of people to the other. The pointer doesn't need to go all the way to the end in fact, because there is no one to compare the very last person with. The pointer can stop one place before the end. So suppose we have 5 people in the line. That person needs to start from position 0 and go to position 3. Why position 3? Because we are counting from 0: the 5 positions of the people run from position 0 to position 4, and the pointer stops one position before the end at position 3. We always run the pointer from 0 up to two less than the number giving the amount of data we are trying to sort.

To write this in code we use a for loop - it is just a programming construct that says we should do something repeatedly, keeping a counter to tell us when to stop. In the following, the special 'keyword' **FOR** is used to indicate we are going to do something repeatedly. The counter we are using is called 'position' (though we could call it anything as long as we are consistent). We indicate where we start counting from (0) and to (3). We will use curly brackets to show what instructions are to be repeated.

```
FOR position FROM 0 UPTO 3
{
   IF (age [position] > age [position + 1])
   THEN
     swap (age, position, position + 1)
}
```

We have done one pass over the array. Is this enough to guarantee the array is sorted, whatever the order the numbers were in at the start? Actually it isn't.

More passes

With our single pass over the data, only numbers that end up next to one another get swapped. If two numbers are a long way apart they may not be brought together in a single pass, and so not be swapped. We need to repeat the process.

The person keeping track of the position needs to go back to the start and do it all over again. We need to do a second pass. Are two passes enough? No we must do it all repeatedly. In fact to sort 5 numbers, we need 4 whole passes: we need to run from the start to the end of the line of people (or array) 4 times.

In code, we need to put another for loop round the first, to say do the original loop over and over again 4 times.

```
FOR pass FROM 0 UPTO 3
{
    FOR position FROM 0 UPTO 3
    {
        IF (age [position] > age [position + 1])
        THEN
            swap (age, position, position + 1)
    }
}
```

How many passes?

How can we be sure that 4 passes is actually enough? We need some **logical thinking**. Consider what happens on a single pass. Think of the person acting as the pointer. They carry the highest numbered person they have seen so far with them. Whenever a new highest number is encountered, it swaps places and is the one carried forward from that point. What that means is that when you get to the end of the pass, the number deposited in that last position is the largest. After one pass, one number is guaranteed to be in the right place.

On the next pass, the same thing happens, we take the next biggest number with us. After two passes, two numbers are in the right place. This pattern continues, so that after the fourth pass, four pieces of data are in the right place. If there are only five pieces of data all together (and five places) at that point the last piece of data must be in the right place too because there is no where else for it to go. So, for five pieces of data, four passes are enough to guarantee everything is in the right place. Similarly, if you have to sort 100 pieces of data, you need 99 passes, 1000 pieces of data need 999 passes. How ever much data you have, you need one less pass to guarantee it is sorted. That is what our outer loop has to count up to. If you count from 0 then that means you count up to two less than the amount of data. To sort n pieces of data using this algorithm (whatever n is), we need n−1 passes, and that means we need to count from 0 up to n−2.

Even in the worst situation (which for Bubble sort happens to be when the data is in exactly the opposite order to the sorted order) n−1 passes will guarantee it all ends up sorted.

With this information we can create a **generalised** version of our algorithm to work in any situation. It is exactly the same as the one we worked out for five pieces of data but with n−2 as the number to count up to in each loop. For the outer loop that means we do enough passes. For the inner loop it means we go to just before the end of the array. If we give our algorithm a name bubblesort, and give the array to be sorted a more general name, array, and refer to its length as n, then we get the final algorithm:

```
bubblesort (array, n):
    FOR pass FROM 0 UPTO (n − 2)
    {
            FOR position FROM 0 UPTO (n − 2)
            {
                IF (array [position] > array [position + 1])
                THEN
                    swap (array, position, position + 1)
            }
    }
```

Can we do better?

In devising the algorithm, we have been doing **algorithmic thinking**. We have been thinking through the steps we need to follow so that we can always guarantee being able to sort things. In future, we won't have to do all this

thinking again, we will be able to just follow the algorithm. Another part of algorithmic thinking though is seeing if we can do better. We now have an algorithm, but perhaps we can improve it. Perhaps we can come up with a faster algorithm!

In fact, the version of bubble sort we have devised so far is actually quite a slow version. We can do better. We have already seen that after the first pass the biggest value is in the right place. It is not going to move again on the later passes. So why waste time comparing against something that we know isn't going to move? Similarly after 2 passes 2 entries are right (and so on). So on each pass there is one less thing we really need to compare. We can improve our inner loop.

It was:

```
FOR position FROM 0 UPTO 3
{
   …
}
```

This says on every pass go up to the 3rd position one before the end. What our logical thinking has shown is that we can stop one place earlier on each pass.

- On pass 0 we go to position 3 (doing 4 comparisons)
- On pass 1 we go to position 2 (doing 3 comparisons)
- On pass 2 we go to position 1 (doing 2 comparisons)
- On pass 3 we go to position 0 (doing 1 comparison)

We need to change that end point of the for loop. How can we do that? Well another way of saying what we want to do is:

- On pass 0 we subtract 0 from the stop point of position 3 giving 3
- On pass 1 we subtract 1 from the stop point of position 3 giving 2
- On pass 2 we subtract 2 from the stop point of position 3 giving 1
- On pass 3 we subtract 3 from the stop point of position 3 giving 0

In other words we just need to subtract the value of pass from the stop point.

We can do this by subtracting pass from the number the inner loop goes to.

```
FOR position FROM 0 UPTO (3 − pass)
{
   …
}
```

As the outer loop changes, the inner loop then adapts its end point to match. This idea generalises whatever amount of data we have. In our general algorithm where our stopping point for the inner loop was (n−2), it now becomes (n − 2 − pass):

```
FOR position FROM 0 UPTO (n − 2 − pass)
```

```
{
   …
}
```

The full algorithm becomes:

```
bubblesort (array, n):
    FOR pass FROM 0 UPTO (n − 2)
    {
            FOR position FROM 0 UPTO (n − 2 − pass)
            {
                IF (array [position] > array [position + 1])
                THEN
                    swap (array, position, position + 1)
            }
    }
```

How good is that?

So we came up with a better algorithm, but a Computer Scientist would want to know how much better. We can use a little more computational thinking, backed by maths to put some numbers on it. (If you don't like maths, don't Panic! Work through it slowly a step at a time.)

Let's go back to our row of 5 people with their ages. With our original algorithm:

- On the 1st pass we did 4 comparisons
- On the 2nd pass we did 4 comparisons
- On the 3rd pass we did 4 comparisons
- On the 4th pass we did 4 comparisons

Altogether we did 4×4 or 16 comparisons. With the same reasoning we would see that for 10 pieces of data to sort we would do 9×9 or 81 comparisons. To sort 100 pieces of data would take 99 passes each doing 99 comparisons so would need 99×99 comparisons in total. There is a pattern here. If we have n pieces of data to sort we will need to do $(n-1)$ passes each doing $(n-1)$ comparisons or $(n-1) \times (n-1)$ comparisons in total i.e. $(n-1)^2$. That gives us a generalised equation for how much work was needed to do the sorting for our slower version of the algorithm:

$$\textit{comparisons needed} = (n-1)^2$$

The question then is how much better is the improved algorithm. Let's follow similar reasoning.

- On the 1st pass we did 4 comparisons
- On the 2nd pass we did 3 comparisons
- On the 3rd pass we did 2 comparisons
- On the 4th pass we did 1 comparison

This time the total number of comparisons is 4 + 3 + 2 + 1 or 10 comparisons, rather than the 16 comparisons of the original.

If we had 10 pieces of data to sort we would do:

9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 or 45 comparisons, rather than 81 comparisons of the original. It is working out as just over half as many comparisons being needed for our new algorithm.

We can work out the general case. If we had *n* pieces of data to sort we would need to do

$n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$

i.e. whatever amount of data, the number of comparisons is the sum of the numbers from 1 to one less than the amount of data. It turns out there is an equation for that (see the end of the booklet if you are interested in how to come up with it). The sum of numbers from 1 to $(n-1)$ is given by the equation:

$$sum = n\,(n-1)\,/\,2$$

So in our case that means:

$$comparisons\ needed = n\,(n-1)\,/\,2$$

Let's just check that works and gives the same numbers as the calculations we did above. If *n* is 5 pieces of data, then we calculate

(5 x 4) / 2 = 10 comparisons.

If *n* is 10 pieces of data, then we calculate

(10 x 9) / 2 = 45 comparisons.

So instead of $(n-1)^2$ comparisons we need only $n\,(n-1)\,/\,2$ comparisons, which is a little over half as many as the $(n-1)^2$ comparisons of the original (as we saw with the actual numbers).

Can we do better still?

Computer Scientists are never satisfied. Perhaps we can do better still. With a bit more thought we can!

What happens with our algorithm if the array is already sorted before we even start? Over and over again we do comparisons, never changing anything. We do lots of work, having no effect at all. So if we were to try and sort 100 pieces of data that was already sorted we would still do 4950 comparisons, none of them leading to any change at all. That is pretty wasteful way of working.

To make that a bit more real. Suppose I gave you a stack of exam papers and told you that I needed them in alphabetical order (I do that a lot!). You could get going doing bubble sort. You would compare nearly five thousand pairs of names before you were done. Now suppose what neither of us knew was that someone else had already sorted them for us before you started… All that effort was for nothing. You have a brain though, so I am betting you would in reality pretty quickly realise they were sorted and stop. In fact you would know for sure at the end of the first pass, when you had checked every adjacent

pair. You might suspect it long before that, but to be sure you would still need to do the full first pass. After all there could be just one pair out of order somewhere near the end. In fact at any point in doing bubble sort the data could become sorted early. However, if we ever do a whole pass and change nothing on that pass, we know they are sorted and can stop.

How can we embed that behaviour in to our algorithm? Well in our head we would have been keeping a note “I’ve not changed anything yet on this pass”. If you ever changed anything you would know you would have to do another pass. Why? Well, the number you just moved back a place might be very small and need to move all the way back to near the start of the pile. On each pass the biggest thing moves with us, but small things just get bumped back one place towards their final destination as we pass over them. So we need some kind of ‘**flag**’ to keep track of whether we have changed anything. A flag to a computer scientist is just something that is either true or false.

Let’s call our flag ‘*changed*’. It will be true if we know we changed something on the current pass, and false if so far we haven’t. Every time we do an actual swap, as a result of a comparison, we reset *changed* to true to say we changed something.

```
IF (array[position] > array [position + 1]
THEN
{
    swap (array, position, position + 1)
    SET changed TO true
}
```

We set it to false at the start of every pass, to say at the start, at least, that nothing has been changed on this pass.

```
SET changed TO false
FOR position FROM 0 UPTO (n − 2 − pass)
{
   …
}
```

If it is still false at the end of the pass, then we will know that the pass didn’t swap anything, and the data is sorted. So each pass does the following:

```
SET changed TO false
FOR position FROM 0 UPTO (n − 2 − pass)
{
    IF (array[position] > array [position + 1]
    THEN
    {
            swap (array, position, position + 1)
            SET changed TO true
    }
}
```

Now controlling when to stop the pass is a bit more complicated than before. We can now keep going as long as pass has not reached the end point and also changed is still true. In programming terms, for this kind of more

complicated stopping condition we use what is called a WHILE loop rather than a FOR loop. For loops are used when we are using a counter (like position or pass) to keep track of how many times to repeat the instructions. A WHILE loop is more general.

We replace:

```
FOR pass FROM 0 TO (n − 2)
```

with

```
WHILE ((pass <= n − 2) AND (changed = true))
```

However, using a while loop means we need to keep track of pass with explicit instructions. We set it to 0 at the start before we enter the loop, and then at the end of each pass increase it by 1. We also need to set changed to true at the very start or the condition of the loop will fail at the outset and we will stop straight away.

The final version of our algorithm becomes:

```
bubblesort (array, n):
    SET changed TO true
    SET pass to 0
    WHILE ((pass <= n − 2) AND (changed = true))
    {
        SET changed TO false
        FOR position FROM 0 to (n − 2 − pass)
        {
            IF (array[position] > array [position + 1]
            THEN
            {
                swap (array, position, position + 1)
                SET changed TO true
            }
        }
        SET pass TO pass + 1
    }
```

In the worst case this takes just as long as the previous version, but it will now stop sooner if it can. It doesn't stop the moment the data is sorted - it needs to do one more full pass where it changes nothing, just to check there are no more changes needed, but that is all.

Can we do even better?

We have improved things a lot but we ought to still ask: "can we do better even than this?" In fact we can. Bubble Sort, which is the algorithm we have just devised, is actually a really slow algorithm, even with these improvements. There are many other algorithms for sorting that are far, far faster. For sorting small numbers of things that won't matter, but if you are sorting millions or billions of things (like the census data for a country, for example) then it will matter a lot. It takes some new ideas, new algorithms and so a new story…and so we will save it for a new booklet.

Computational Thinking

We have been looking at how to sort data and in particular how Bubble Sort works. Along the way we have done a lot of **computational thinking**. We have of course been doing **algorithmic thinking**. Rather than just trying to put the data in order in some haphazard way, we have come up with a series of algorithms to do it. Once we have worked them out and written them down, sorting is easy. Any time in future we want to sort anything, we just have to follow our instructions. We don't have to think it all through again. We could give the instructions to someone else (or a computer) and they could sort the data without any idea of how or why it worked by blindly following them.

Coming up with the algorithm involved a lot of clear, **logical thinking**, both to come up with the steps of the algorithm, and to check they worked as well as to see how much faster one algorithm was than another. We were of course **evaluating** our algorithms in doing that - both in terms of whether they were correct and in terms of how efficient they were. We also used **decomposition** in that rather than thinking about the whole problem at once we thought about it a bit at a time. We first worried about swapping a pair, then looked at a single pass, and only then multiple passes. We also used a little bit of **abstraction**, in that we hid the details of exactly how to swap data round.

A very important step we did was to **generalise** our algorithm. We initially worked it out for sorting 5 pieces of data, but then we looked for the **pattern** involved and that showed us how to extend it to any amount of data. We used **pattern matching** again to work out the equation for how much work was involved in each algorithm. Finally we used **generalisation** and **abstraction** together in another way right at the start when we noted that > could stand for any appropriate comparison (e.g., numerical order, alphabetical order, …) so in fact the same algorithm works whatever kind of data we want to sort. All we have to do is provide the appropriate comparison operator.

Computational thinking, in all its facets, is at the core of how computer scientists solve problems.

Added Extra: The sum of numbers from 1 to n

How did we get that magical formulae for adding up a series of numbers like:

8 + 7 + 6 + 5 + 4 + 3 + 2 + 1?

It is easier to see if you reorder the numbers in the sum into pairs taking one from each end of the list to give:

8 + 1 +

7 + 2 +

6 + 3 +

5 + 4 +

We are still adding the same numbers so it will give the same answer. Each pair here adds up to 9 (i.e. one more than the largest number, 8). How many pairs are there? 4 (which is half of 8) as we were pairing them up. So the sum adds up to 9 × (8 / 2).

If we do the same thing for numbers up to 50, we get a similar pattern by pairing them. Each pair adds up to 51 and there are 25 pairs when we pair up 50 numbers. In that case the answer is 51 × (50 / 2). However many numbers we are adding, we can do the same trick. So to add the numbers from 1 up to (n - 1), for any number n, the answer is n (n - 1) / 2.

Use of this booklet

This booklet was created by Paul Curzon of Queen Mary University of London, cs4fn (Computer Science for Fun www.cs4fn.org) and Teaching London Computing (teachinglondoncomputing.org).
See the Teaching London Computing activity sheets in the Resources for Teachers Section of our website (http://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/) for linked activities based on this booklet.