The Bubble Sort Dance

Age group: 8 – adult (some extensions more suited for older groups)

Abilities assumed: None

Time: 20-30 minutes for a basic version depending on depth of explanation

Size of group: anything from 5 to hundreds

Focus
Sort algorithms: bubble sort

Syllabus Links
This activity can be used as a general introduction to sort algorithms at various key stages.

Summary
Demonstrate Bubble Sort by having students role play the data being sorted.

Aims
This activity aims to teach bubble sort as well as to explore how the efficiency of an algorithm can be improved

Technical Terms
Sort algorithms, bubble sort, efficiency.

Materials
A set of 5 A4 cards with different numbers on written as large as possible,
Fasten a loop of ribbon to each to go round a student’s neck (optionally just have the person hold the numbers for a less theatrical version).
A hat (e.g., a top hat suitable for a dance). Other props can be used instead - e.g. a cane.
What to do

Explain you are going to demonstrate a simple way that computers sort data. This is quite a common task for computers to do. For example, you might sort your music collection, say alphabetically by artist but then later putting them in a different order by song title. You are also going to do it as a dance showing how computer programmers choreograph the movement of data in memory a little like dance choreographers choreograph human dancers. The difference is that computer dances do useful work at the same time.

Line up 5 students and give them the 5 cards with ribbon to put over their heads, so the cards are on their backs. They will be the ‘dancers’ as well as the data being sorted. To help when you come to explain, make sure that the rightmost dancer as you face them has the smallest number. A largish number should be at the left hand end, with the other cards mixed randomly. The dancers should be turned so that the actual numbers are concealed from the audience.

Have all 5 dancers spin round so their card is visible. Point out they are in the wrong order - you want them in ascending order, running smallest on the left to largest on the right. Have them all spin back so the numbers are hidden again.

Point out that while they might glance at a few numbers and mentally put them in the right order, if there were a million numbers they would need a better way. A computer needs detailed instructions and can only compare pairs of numbers at a time. It can look at any two numbers and decide if one is bigger than the other. You are going to demonstrate a way that sorts things by repeatedly doing such comparisons.

Give the dancer on the left the hat to wear. Explain that if we are going to put the numbers in to order, then we need to keep track of which pair we want to compare next. In a program we would use a variable storing a position to do this. For our dance, the hat will keep track of the next people to compare.

The person with the hat should take the hand of the person to their left (the audience's right), and the pair step forward. They are the pair to be compared. Have them spin round so everyone can see their numbers. The largest needs to end up on the audience’s right. If they are in the wrong order the audience shout “SWAP” and the two dancers swap positions. Otherwise the dancer with the hat just passes the hat to their partner. Either way the hat has moved on. They then step back in to the line.

Before continuing with the dance, explain what the pair have done can be written as a series of instructions (in pseudocode) as follows:

```
Pass hat to first dancer
Dancer with hat and next dancer step forward
if dancer with hat > next dancer
   Dancer with hat swaps places with next dancer
else
   Dancer with hat passes the hat to next dancer

The pair step back in to the line
```

Running down the line

Now obviously doing that once isn’t enough to sort everything. We need to check each pair, so we now do it again. As the hat has been passed on, when we follow the same instructions a second
time a new pair will step forward and be compared. Talk them through the process. They swap or not and step back. Then the next pair step forward and so on, until the hat is at the end of the row.

If the class has done coding ask them what construct we’ve just done: we have just done a loop. Passing the hat to the first dancer is only done once so is outside the loop. Everything else is repeated so inside the loop (i.e., so inside the curly brackets in the instructions below). Explain the pseudocode for what has just happened:

Pass hat to first dancer

Repeat 4 times
{
    Dancer with hat and next dancer step forward

    if Dancer with hat > next dancer
        Dancer with hat swaps places with next dancer
    else
        Dancer with hat passes it to next dancer

    The pair step back in to the line
}

Notice that the number of times we did a comparison and the hat moved on (with or without a swap) is 4 times. It is one less than the number of dancers (i.e., amount of data) we were trying to sort. Why? Because the hat stops one before the end, and when in that position it leads to the last two dancers being compared. If we did another round of comparisons when the hat was with the last dancer, there would be no one to compare them with. That means that whatever number of dancers we have, we will need to repeat the instructions one less time than that number. We had 5 things to sort so needed a loop repeating 4 times. If we had a thousand dancers to sort we would need a loop that repeated 999 times.

The largest is in the right place

Now just running down the line isn’t enough to always guarantee that everyone is in the right order. The dance has carried the largest number to the end as you always moved the dancer with the biggest number found so far. You can demonstrate this by walking down the line with the hat pointing out as the comparisons were made the largest found so far moved along with the hat. The largest is now guaranteed to be in the right place. However, the others could still be mixed up. They may have just moved a little way towards their final destination. Of course you could be lucky and everything has just fallen in to the right place because of where they started, but an algorithm must not rely on luck. It must work every time.

Lots of passes

It must all be done again to guarantee that whatever the original order, the dancers end up in the right order. You need to do more passes, but how many? Ask the class for ideas. Is once enough? More? How long does the dance need to be. In fact you need one less pass than the number of dancers. For 5 dancers they need to do it all 4 times over.

Computer Science activities with a sense of fun: The Bubble Sort Dance V1.0 (22 March 2016)
This version created by Paul Curzon, Queen Mary, University of London for Teaching London Computing: http://teachinglondoncomputing.org working with the BBC.
Have them work though it a pass at a time, checking the largest value moves on each comparison. Between each pass, get the whole line to spin round so the class can see that they aren’t in order yet (they won’t be if the smallest number started at the wrong end). When the hat has passed down the line 4 times all together, the dancers should be in sorted order.

Ask if anyone can explain why it takes one less pass than the number of dancers. One way to see this is based on the fact that on each pass the largest value is carried by the hat in to its correct position. Similarly on the second pass the next biggest value is carried to its correct position in the second last spot. On each pass one more dancer ends up in the correct place. However when all but one is in the right place, the last dancer has no where else to go if everyone else is right. They must be in the right place too. You need one less pass than the number of dancers.

In our case that means we need another loop that repeats everything 4 times in total as we have 5 people to sort. With a thousand dancers to sort we would need to do 999 passes before they were guaranteed to be in the right order.

In pseudocode that just means putting a loop round the whole set of instructions so far:

```plaintext
Repeat 4 times
{
    Pass hat to first dancer
    Repeat 4 times
    {
        Dancer with hat and next dancer step forward
        if Dancer with hat > next dancer
            Dancer with hat swaps places with next dancer
        else
            Dancer with hat passes it to next dancer
        The pair step back in to the line
    }
}
```

We can generalise this to an algorithm for any amount of data. As we’ve seen the number 4 in the pseudocode is just one less than the number of dancers. So we can create a general sort algorithm, by introducing a variable n to stand for the number of dancers.

To sort n dancers:

```plaintext
Repeat n - 1 times
{
    Pass hat to first dancer
    Repeat n - 1 times
    {
        Dancer with hat and next dancer step forward
        if Dancer with hat > next dancer
            Dancer with hat swaps places with next dancer
        else
            Dancer with hat passes it to next dancer
        The pair step back in to the line
    }
}
```
Do the dance in full

Finish off by mixing up the order of the dancers again, and have them go through the steps from start to finish, doing a full bubble sort dance.

Variations and Extensions

Can we do better?

We now have an algorithm that is guaranteed to work, but could it be better? It is a long dance, and perhaps we don’t have that much time. In fact this is a fairly inefficient version of the bubble sort algorithm. A good extension activity for older groups is to explore improvements to the algorithm, both generally with the dancers and in pseudocode. Is it possible to come up with a dance that does the same job but involves less steps overall? Ask the class for ideas. Are we doing anything wasteful?

One clear way we are being wasteful is in our inner loop. We have already seen that after the first pass the biggest value is in the right place. That dancer is not going to move again on the later passes. So why waste time comparing against a dancer that we know isn’t going to move again? Similarly after 2 passes 2 dancers are in their final place (and so on). So on each pass there is one less dancer we really need to compare again. We can improve our inner loop. We can stop one place earlier on each pass.

- On pass 1 the hat stops at dancer 4 (doing 4 comparisons)
- On pass 2 the hat stops at dancer 3 (doing 3 comparisons)
- On pass 3 the hat stops at dancer 2 (doing 2 comparisons)
- On pass 4 the hat stops at dancer 1 (doing 1 comparison)

Another way of saying what we want to do is:

- On pass 1 we subtract 0 from the original stop point of position 4 giving 4 repetitions.
- On pass 2 we subtract 1 from the original stop point of position 4 giving 3 repetitions.
- On pass 3 we subtract 2 from the original stop point of position 4 giving 2 repetitions.
- On pass 4 we subtract 3 from the original stop point of position 4 giving 1 repetitions.

In general the pattern is that for any pass \( p \):

- On pass \( p \) we subtract \( p - 1 \) from the original stop point of position 4 giving \( 4 - (p - 1) \) repetitions.

If we have \( n \) dancers it means the inner loop doesn’t repeat \( n-1 \) times but \( (n - 1 - p - 1) \) or simplifying that: \( (n - p - 2) \).
Our general improved algorithm becomes:

To sort \( n \) dancers:

1. Set \( p \) to 1
2. Repeat \( n - 1 \) times
   - Pass hat to first dancer
3. Repeat \( n - p - 2 \) times
   - Dancer with hat and next dancer step forward
   - if Dancer with hat > next dancer
     - Dancer with hat swaps places with next dancer
   - else
     - Dancer with hat passes it to next dancer
   - The pair step back in to the line
4. Add one to \( p \)

This dance takes almost half the time to complete as the original but still guarantees to sort the dancers!

Another way to improve the algorithm is based on the idea that sometimes the dancers will be in sorted order long before the end. They may even be already sorted at the start. The trouble is how do we know? We can’t tell without checking and that means comparing every pair to see if they are in the right order. That would be a lot of extra work, and most of the time we will just find that they aren’t sorted so it was a waste. However, we are doing comparisons like that anyway. The thing to realise is that if we don’t swap any dancers during a whole pass, then that means they are in the right order. We can stop, having done just one extra pass after they became sorted to check. How to do this in pseudocode is described in the booklet ‘Bubblesort explained’ below.

**A version using for loops and arrays**

Our algorithm is in terms of a line of dancers. Write a version that talks about sorting an array of numbers. That would be closer to a program version as data to be sorted is stored in data structures such as an array. It may be easier to do this by switching to for loops instead of general repeat loops.

**Write the program**

We have written a pseudocode version of the algorithm. Write a program that implement it in the language of your choice.

**More dance**

Depending on the group, you can play up the idea of it being a dance to a greater or lesser extent. To be more formal just have people holding cards that they turn over to reveal the numbers and miss out the holding hands and twirls. To make it more of a dance have the volunteers do more dance-like steps with proper twirls. As a further extension, have them choreograph a dance version to music with more steps and spins. Add actions for those in the line while not their turn to step forward to be compared.
Turn other sort algorithms in to dances

There are other activities to introduce other sort algorithms such as divide and conquer algorithms. Can they be turned in to a dance? DO the dances look similar or different?

Further Reading

Bubble Sort Explained

A variation of this activity has been written up in a booklet. It also explains sorting and describes bubble sort in more detail. It is available from http://teachinglondoncomputing.org(sorting/)

Links to other activities

Divide and Conquer Sorting

This activity is a natural continuation from bubble sort showing how sorting can be done even faster still.

Punchcard Sorting

This is a simple demonstration of another divide and conquer sort algorithm (binary radix sorting) that was used as a way to sort punch cards on early computers. It shows a powerful use of binary.