Sorting Unordered Lists: Bubble Sort

Have you ever wondered how you put objects in order? What methods do you use (perhaps without realising it)? Could you explain your process clearly enough for a computer to follow it?
Sorting algorithms are a staple of any course in computer science, and it's easy to see why: people can put things in order pretty quickly, but it's somewhat harder to describe exactly how we do it, and harder still to construct a precise and efficient set of instructions that allow the computer to do the same.

Stop. Shuffle. Sort.

Playing cards can be arranged in order by number, books alphabetically, or even random bits of junk on your desk in order of size or monetary value. Find a collection of unordered things, and have a go putting them in order. Pay attention to how you are doing it.
Do you search through all the cards until you find the first one? How do you identify it? Do you group them in heaps according to some criteria first? What comparisons did you make? How did you arrange or group them along the way? Does it make a difference if you know what items the list will contain before you begin? Does your method change if the items are already nearly sorted or in perfect reverse order?

Attribute	**Human** 	**Computer** 
***Speed**:* *How quickly a sorting procedure can be carried out and verified.*	**Slow** We are limited by our brain's processing speed and our working memory.	**Fast** With a large enough processor, a computer is incomparably quicker.
***Adaptability**:* *How well the procedure can be modified to take into account objects already being in order / almost in order / reverse order / from a recognizable set such as birthdays or surnames.*	**Good** Extremely adaptable: Can remove 'Mrs Bun' from a poker hand without freezing. Can spot and take advantage of pre-existing patterns or runs of objects already ordered, etc.	**Poor** Too specialist: Can very quickly sort lists it's designed to encounter, but needs increasingly complex code to take into account the more obscure variations, or to find the best method for otherwise slow lists.
***Accuracy**:* *How reliable is the final ordered list? What is the chance that some items were incorrectly compared, or that the final list has not been properly verified?*	**Poor** Especially for larger lists, the chance of an error is significant. Our adaptability comes at a price.	**Good** The lack of flexibility is a necessary trade-off when we require almost complete precision. Comparing and storing items is what they do.
***Capacity**:* *How large a list of objects can we deal with?* *Is there a limit to the total number, or to the efficiency with which they can be ordered?*	**Poor** There's a limit on the number of things a human can hold in their 'working memory' at any one time. To cheat the system we write things down. Or we use a computer!	**Good** This is only limited by the capacity of the computer's processor and running memory. They can handle awesomely large lists and keep going for months at a time if necessary.

A human sorting algorithm example
(designed for putting a stack of exam papers in alphabetical order)

Step 1: Pick up the top paper. If the surname starts with A, B, C, D or E, put it in pile 1. Surnames from F to L go in pile 2. M to R go in pile 3, S to Z in pile 4. *
Step 2: Repeat until all papers are 'bucketized'.
Step 3: With the first pile:
a) Pick up the first pile, and start a new pile with the first paper.
b) Pick up the next paper and insert it into the appropriate place in the new pile.
c) Repeat until the first pile is now ordered.
Step 4: Repeat step 3 with each pile until all piles are ordered.
Step 5: Place the four ordered piles back into one ordered stack.

* *Note: the apparently uneven sorting (5, 7, 6, 6) roughly reflects the frequency of initial letters. In the English language as a whole this splits roughly: 29%, 22%, 25% and 23%.*

Computers can't grab a bunch of papers and fan them out to get a feel for the list. They can, however, rapidly find objects, **compare**, and **swap** their positions as required.

What is efficient for a human is not necessarily so for a computer, and vice versa. When dealing with computer algorithms for sorting an unordered list, we try to minimise the total number of **comparisons** and **swaps** required. This number is determined by a few different things: the size of the list, the original state of the list and the algorithm used.

A computer sorting algorithm example
(designed for reordering a list of numbers)

Step 1: Compare the first two items in the list. If in order, move on. If not, swap.
Step 2: Compare the second and third items in the list. If in order, move on. If not, swap.
Step 3: Repeat steps 1 and 2 until all pairs of items have been compared.
Step 4: Return to the beginning of the list and repeat steps 1, 2 and 3. If no swaps were needed, move on. If swaps were needed, repeat the whole process again.

* *Note: this algorithm is one of the most basic – there are more efficient alternatives.*

During each pass (a single run through the list), the algorithm will cause the next largest value to 'bubble' to the end, hence the algorithm's name: **Bubble Sort***.

Due to the method chosen for this algorithm, we need a complete sweep of the whole list (comparing and, if need be, swapping successive pairs of items) to be certain the largest value is definitely in its correct position. Additional sweeps will successively ensure the placement of the next largest, and the next and so on. Once we carry out a full sweep with no swaps that shows us the list is sorted and we can stop.

*Note: the version given here is incomplete. There are a couple of further improvements that are usually made: for a start, after the first pass, since the final element is in the right place, we can make successive passes shorter (one shorter each time). Read on for more…

Modelling it with Python

No matter how simple the concept of 'compare, swap, move on' might seem, it can quickly get out of hand if we try putting it all together at once into a program. So we'll start small, with a manageable list, and only gradually abstract out to more general examples.

Sorting a list of 5 items

We will start with a simple list: **[3, 5, 1, 2, 4]**. We'll give it a one-letter name because, even though this breaks the normal conventions of Python, it'll make it easier to understand some of the lines…

```
1   a = [3, 5, 1, 2, 4]
2   print(a)
3
4   print("First pass:")
5   for i in range(4):
6     if a[i] > a[i + 1]:
7       a[i], a[i + 1] = a[i + 1], a[i]
8     print(a)
```

The loop repeats 4 times so we compare items 0 & 1, 1 & 2, 2 & 3 and finally 3 & 4.

If the first item is greater than the second, they get swapped (using Python's simultaneous assignment trick).

Note: as you will see when you run this, the list is not sorted. What we have done here is one pass through the list, comparing every pair and, if necessary, swapping them. The largest element has 'bubbled' to the end of the list, but the others are not right yet.

Because the final item is in the right place, our next pass can be more efficient:

```
10  print("Second pass:")
11  for i in range(3):
12    if a[i] > a[i + 1]:
13      a[i], a[i + 1] = a[i + 1], a[i]
14    print(a)
15
16  print("Third pass:")
17  for i in range(2):
18    if a[i] > a[i + 1]:
19      a[i], a[i + 1] = a[i + 1], a[i]
20    print(a)
21
22  print("Fourth pass:")
23  for i in range(1):
24    if a[i] > a[i + 1]:
25      a[i], a[i + 1] = a[i + 1], a[i]
26    print(a)
```

We only need to compare items 0 & 1, 1 & 2 and 2 & 3, so this pass loops through a range of length 3.

By the same logic as before, the second-to-last element must now be in its correct position, so each subsequent pass can be shorter.

The final pass is hardly a loop at all (recall, **range(1)** goes from 0, the default starting value, up to *but not including* 1), so it simply compares items 0 & 1.

*If you run this code now, you should notice a couple of things: 1) the items are now in order. 2) they were already in order before the third pass! What happens if you change your list to something with even more initial order, like **[1, 2, 3, 5, 4]**?*

A human following this algorithm would notice that the elements are all in order, and stop crunching through the steps. The computer needs to be told if the job is done early, and to do that, we'll need to start keeping track of the number of **swaps** that take place…

Nested loops

Before we start implementing improvements to our code, notice how much repetition we have! It's no bad thing to start out like this, when we are trying to get our heads around a multi-layered process, but once you can see the higher structure, you should be finding a more efficient way to write your code. For one thing, this set-up will only work for lists of length 5!

Get rid of this…

```
a = [3, 5, 1, 2, 4]
print(a)

print("First pass:")
for i in range(4):
    if a[i] > a[i + 1]:
        a[i], a[i + 1] = a[i + 1], a[i]
    print(a)

print("Second pass:")
for i in range(3):
    if a[i] > a[i + 1]:
        a[i], a[i + 1] = a[i + 1], a[i]
    print(a)

print("Third pass:")
for i in range(2):
    if a[i] > a[i + 1]:
        a[i], a[i + 1] = a[i + 1], a[i]
    print(a)

print("Fourth pass:")
for i in range(1):
    if a[i] > a[i + 1]:
        a[i], a[i + 1] = a[i + 1], a[i]
    print(a)
```

And replace it with this…

```
a = [3, 5, 1, 2, 4]
print(a)

for p in range(1, 5):
    print(f"Pass number {p}:")
    for i in range(5 - p):
        if a[i] > a[i + 1]:
            a[i], a[i + 1] = a[i + 1], a[i]
        print(a)
```

The nested loop still works the same way, with the range goes from $5-1$ to $5-4$. Since there are 5 items in the original list, the number of passes we require is 4, and **range(1, 5)** gives us values 1, 2, 3 and 4 as required. We could even generalise pretty easily to work with a list of any length:

```
a = [3, 5, 1, 2, 4]
print(a)
n = len(a)

for p in range(1, n):
    print(f"Pass number {p}:")
    for i in range(n - p):
        if a[i] > a[i + 1]:
            a[i], a[i + 1] = a[i + 1], a[i]
        print(a)
```

Counting swaps

Now we're ready to start optimising things. If the algorithm makes a complete pass without making any swaps, it must be because the list is already in order. That's the definition of being in order: if every person in line is shorter than the person behind them, then the entire line is in height order. We can build in a counter to keep a tally of the total swaps for each pass:

```
a = [3, 5, 1, 2, 4]
print(a)
n = len(a)

for p in range(1, n):
    swaps = 0
    print(f"Pass number {p}:")
    for i in range(n - p):
        if a[i] > a[i + 1]:
            swaps += 1
            a[i], a[i + 1] = a[i + 1], a[i]
            print(a)
    print(f"Pass {p} done.  Made {swaps} swaps.")
```

The counter **swaps** is reset to 0 at the beginning of each pass, incremented by 1 every time a swap takes place, and is printed to the screen at the end of each pass.

The next step will be to break out of the loop if an entire pass completes without any swaps…

Breaking early

```
1   a = [3, 5, 1, 2, 4]
2   print(a)
3   n = len(a)
4
5   for p in range(1, n):
6     swaps = 0
7     print(f"Pass number {p}:")
8     for i in range(n - p):
9       if a[i] > a[i + 1]:
10        swaps += 1
11        a[i], a[i + 1] = a[i + 1], a[i]
12        print(a)
13    print(f"Pass {p} done.  Made {swaps} swaps.")
14    if swaps == 0:
15      print("Pass completed with no swaps. Done!")
16      break
```

The added lines here (14 to 16) are within the main (outer) **for** loop, so if the number of swaps is 0, the explanatory statement is printed and then the code is instructed to **break** out of the loop.

Tracking comparisons

The efficiency of a sorting algorithm can be measured by timing the computer, but the results will vary depending on the machine being used and the efficiency of the CPU, etc. In order to reliably compare sorting algorithms, the best thing to do is keep track of the number of actions we require of the computer. For sorting, this is **comparisons** (checking to see whether one number is larger than another) and **swaps** (switching the position of two elements within a list). In terms of processing time, I estimate swaps cost about twice as much as comparisons (around 1ns for swaps, and 0.5ns for comparisons), so we could even build up an overall **cost** for a given list sort, which would allow effective comparison between different initial list configurations. And, when we start investigating some alternative sorting algorithms, between one algorithm and another.

```
1   a = [3, 5, 1, 2, 4]
2   print(a)
3   n = len(a)
4
5   total_swaps, total_comps = 0, 0
6   for p in range(1, n):
7     swaps, comps = 0, 0
8     print(f"Pass number {p}:")
9     for i in range(n - p):
10      comps += 1
11      if a[i] > a[i + 1]:
12        swaps += 1
13        a[i], a[i + 1] = a[i + 1], a[i]
14        print(a)
15    print(f"Pass {p} done. Made {comps} comps, {swaps} swaps.")
16    total_swaps += swaps
17    total_comps += comps
18    if swaps == 0:
19      print("Pass completed with no swaps. Done!")
20      break
21
22  cost = total_swaps + 0.5 * total_comps
23  print(f"Sorting complete: Total cost: {cost}ns")
24  print(f"({total_comps} comparisons, {total_swaps} swaps)")
```

We now have **total_swaps** and **total_comps** initialised at the very beginning, and after each pass the number is increased by the number of passes and comparisons that took place within that pass. At the end, the totals are printed to the screen.

Try this out with some different configurations for your starting list. What's the best? What's the worst?

More tests!

The way our program is set out, it gives us details at every point of the process. This is great for understanding the intricacies of the process, but obviously impacts on the efficiency of the algorithm substantially, and is probably too much information for when we start investigating bigger lists. If you **fork** your program at this point, you can make a version more suited to a testing environment which will be able to deal with many lists, only giving you the final details (total swaps, comparisons and overall cost).

```
1   import random
2
3   a = [random.randint(1, 100) for i in range(10)]
4   print(a)
5   n = len(a)
6
7   total_swaps, total_comps = 0, 0
8   for p in range(1, n):
9     swaps, comps = 0, 0
10    for i in range(n - p):
11      comps += 1
12      if a[i] > a[i + 1]:
13        swaps += 1
14        a[i], a[i + 1] = a[i + 1], a[i]
15    total_swaps += swaps
16    total_comps += comps
17    if swaps == 0:
18      break
19
20  print(a)
21  cost = total_swaps + 0.5 * total_comps
22  print(f"Sorting complete: Total cost: {cost}ns")
23  print(f"({total_comps} comparisons, {total_swaps} swaps)")
```

In addition to deleting almost all the **print** statements, I've modified the original list definition. Now, using the **random** module, the program generates a random list of 10 integers between 1 and 100. Each time we run this program, we'll get a different list, and a different number of swaps and comparisons.

Try a few runs, and see if you can identify what makes a particular list slow to sort.

Get the stats

By now you're probably curious about the average, worst and best performance for the bubble sort. You guessed it – we're going to need another loop! An elegant way to do this is to use functions to contain the code we want to repeatedly run:

```
1   import random
2
3   def sort_random_list(n):
4     a = [random.randint(1, 100) for i in range(n)]
5     total_swaps, total_comps = 0, 0
6     for p in range(1, n):
7       swaps, comps = 0, 0
8       for i in range(n - p):
9         comps += 1
10        if a[i] > a[i + 1]:
11          swaps += 1
12          a[i], a[i + 1] = a[i + 1], a[i]
13      total_swaps += swaps
14      total_comps += comps
15      if swaps == 0:
16        break
17    return total_comps, total_swaps
```

This creates and sorts a random list, of the length required, producing only the information we want (number of comparisons and swaps).

By enclosing this in a loop, we can sort hundreds or thousands of random lists, and keep track of the number of swaps, comparisons or both as we go along…

The big loop

```
25  trials = 10000
26  length = 10
27  print(f"Sorting {trials} random lists of size {length}...")
28  num_comps, num_swaps, costs = [], [], []
29  for i in range(trials):
30      c, s = sort_random_list(length)
31      cost = s + 0.5 * c
32      num_comps.append(c)
33      num_swaps.append(s)
34      costs.append(cost)
```

This code sets up the loop with the number of trials to run, and the length of each list to be sorted. The empty lists created are to keep track of the values produced.

Displaying the data

When the code above has finished executing, we'll have three (big) lists containing the details of comparisons, swaps and total cost for each list sorted. We'll want to easily view a summary of this data rather than simply printing all the numbers to the screen. A simple approach is to print the key stats, but since we want to do something very similar for each one, a helper function might save some time:

```
36  print_stats(num_comps, "Comparisons:")
37  print_stats(num_swaps, "Swaps:")
38  print_stats(costs, "Costs:")
```

This code was written first – I decided I wanted a function whose job it was to present nicely the stats for each set of data I'd collected.

The following code was then written, to display the basic information as required:

```
19  def print_stats(values, label):
20      print(label)
21      print(f" - Lowest: {min(values)}")
22      print(f"   - Mean: {round(sum(values) / len(values), 1)}")
23      print(f"     - Highest: {max(values)}")
```

This calculates and prints a few basic summary stats. You can add to it to see more (like quartiles, etc).

Some extension ideas

- What happens with very large lists? Try a simulation with 10 items, then 20, 30, etc. See if you can identify a pattern in the number of comparisons or swaps required. How many comparisons does a list take if it has 100 items, or 1000? How does the time required by the computer change as the length of list increases?
- You can investigate larger lists, and see what happens with a long list which is only slightly out of order (eg start with a sorted list, choose two random values to swap, and see how efficiently the algorithm can fix it), and what happens when a list is in perfect reverse order. You could also construct a list with only a few different values (eg [3, 5, 5, 5, 3, 5] or similar) to see how well bubble sort deals with them.
- If you want to get more insight into what's going on to cause small or large numbers of comparisons, you can use the **itertools** module to cycle through all possible permutations of a list. It will let you analyse every single possible ordering of a list.

```
2  from itertools import permutations
3
4  original = [i for i in range(1, 6)]
5  for perm in permutations(original):
6      a = list(perm)
```

Caution: this will cycle through every permutation, so don't expect it to be quick if your list has more than a handful of items.

Additional info and how to write out an implementation of the Bubble Sort algorithm:

Name:	The word 'bubble' is used because the largest value 'bubbles' to the end of the list, and in the next pass, the next largest bubbles into its correct position.
Summary:	Compare, and, if needed, swap successive pairs of items. Repeat until done.
Efficiency:	For nearly sorted lists: $O(n)$ (takes $\approx 2n$ comparisons to add a new item to a sorted list) For reverse order lists (worst case): $O(n^2)$ (takes $\frac{n(n-1)}{2}$ comparisons to reverse a list) Generally speaking, very inefficient, particularly for large or very unordered lists.
Algorithm:	Compare the first 2 items and swap if needed. Compare the next two (items 2 and 3) and swap if needed. After going through all the numbers (one 'sweep'/'pass'), start again at the beginning. Repeat until you have completed one full sweep without any swaps.
Example:	Sort the list 8 3 2 6 9 4 2 7 using bubble sort. First pass: 3 2 6 8 4 2 7 9 (comparisons: 7, swaps: 6) Second pass: 2 3 6 4 2 7 8 9 (comparisons: 6, swaps: 4) Third pass: 2 3 4 2 6 7 8 9 (comparisons: 5, swaps: 2) Fourth pass: 2 3 2 4 6 7 8 9 (comparisons: 4, swaps: 1) Fifth pass: 2 2 3 4 6 7 8 9 (comparisons: 3, swaps: 1) Sixth pass: 2 2 3 4 6 7 8 9 (comparisons: 2, swaps: 0) *Note: the individual comparisons and swaps for a pass do not need to be listed, but you need to be able to count them.* **Totals: comparisons: 27, swaps: 14** Note: At the end of each pass, one additional number is definitely in the right place at the end of the list (it's helpful to indicate this by underlining, so you remember not to bother comparing within the already sorted section. The algorithm concludes either after the maximum number of passes ($n-1$) or after an entire pass is completed without any swaps being made.
Visual:	For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com