Sorting Unordered Lists

Have you ever wondered how you put objects in order?
Could you explain your method clearly enough that a computer could follow it?
What thought-processes or calculations do you use (maybe subconsciously)?

When you are putting playing cards in order:
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 the cards 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 cards 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** Cognitive Working Load theory puts a fairly strict limit on the number of thing a human can hold in their 'working memory' at any one time. To cheat the system we write things down, or… 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. Some methods are too inefficient for large lists, however.

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. The number of comparisons and swaps is determined by 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, but it is also one of the least efficient options.*

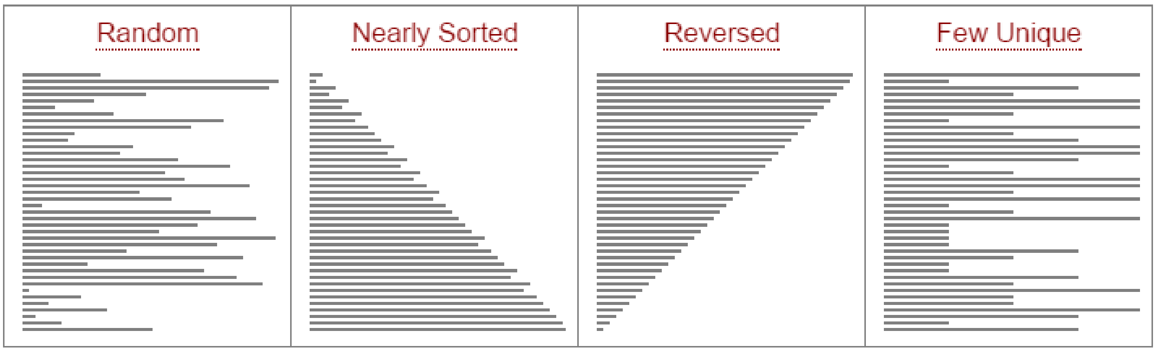
The largest value will 'bubble' to the top, hence the algorithm's name: **Bubble Sort**.
Due to the method chosen for this algorithm, we need a complete sweep of the whole list (comparing and, if need be, swapping successive pairs of items) to be certain the largest value is definitely in its correct position. Additional sweeps will successively ensure the placement of the next largest, and the next and so on. A full sweep with no swaps is required to verify that the list is indeed sorted at the end.

The worst case scenario for this algorithm is a list in perfect reverse order, when a list of n items will take a total of $\frac{n}{2}(n+1)$ comparisons to sort. Since this expression is quadratic, it indicates that doubling the length of a list will quadruple the computing time required.

Bubble Sort

Name:	The word 'bubble' is used because, when a list is to be sorted vertically in ascending order, starting from the bottom, the smallest terms 'bubble' to the top. Note: in D1 it is usually implemented horizontally from the left.
Summary:	Compare, and, if needed, swap successive pairs of items. Repeat until done.
Efficiency:	For nearly sorted lists: $O(n)$ (takes about $2n$ 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.

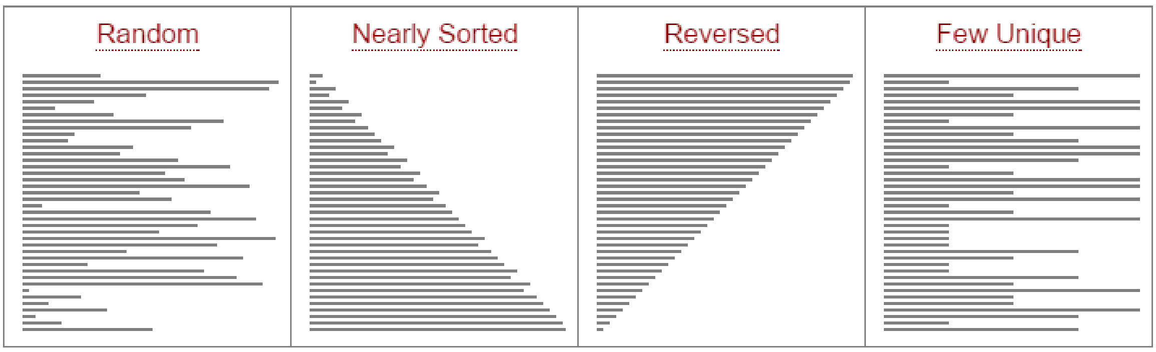
Example:	Sort the list 8 3 2 6 9 4 2 7 using bubble sort.
	`First pass:  3 2 6 8 4 2 7 9 (comparisons: 7, swaps: 6)`
	`Second pass: 2 3 6 4 2 7 8 9 (comparisons: 6, swaps: 4)`
	`Third pass:  2 3 4 2 6 7 8 9 (comparisons: 5, swaps: 2)`
	`Fourth pass: 2 3 2 4 6 7 8 9 (comparisons: 4, swaps: 1)`
	`Fifth 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)`
	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 (this does not have to be indicated in your solution as I have above by underlining, but should be taken into account as it reduces comparisons needed on subsequent passes). The algorithm concludes only after a 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 

Shuttle Sort

Name:	A ‘shuttle’ moves a number all the way along a sublist until it gets to the correct position. Also known as ‘insertion sort’ since each subsequent item is inserted into position.
Summary:	Make an ordered list of the first two items, then insert subsequent numbers in their correct position within this list. Continue until all items have been shuttled into position.
Efficiency:	For nearly sorted lists: $O(n)$ (takes about n comparisons to add a new item to a sorted list) For reverse order lists (worst case): $O(n^2)$ (but generally more efficient than bubble sort) Good enough with small lists to be used as part of larger ‘divide-and-conquer’ algorithms.
Algorithm:	Compare the first 2 items and swap if needed. Compare the next item successively to items in the already sorted sublist (items 1 and 2), swapping to move down the list until it occupies the correct position (this sublist will always be in order, and gradually grows as more items are added). Repeat for the fourth item, etc, until all items are inserted.

Example:	Sort the list 8 3 2 6 9 4 2 7 using shuttle sort.
	`The list:     8 3 2 6 9 4 2 7`
	`First pass:   3 8 2 6 9 4 2 7 (comparisons: 1, swaps: 1)`
	`Second pass:  2 3 8 6 9 4 2 7 (comparisons: 2, swaps: 2)`
	`Third pass:   2 3 6 8 9 4 2 7 (comparisons: 2, swaps: 1)`
	`Fourth pass:  2 3 6 8 9 4 2 7 (comparisons: 1, swaps: 0)`
	`Fifth pass:   2 3 4 6 8 9 2 7 (comparisons: 4, swaps: 3)`
	`Sixth pass:   2 2 3 4 6 8 9 7 (comparisons: 6, swaps: 5)`
	`Seventh pass: 2 2 3 4 6 7 8 9 (comparisons: 3, swaps: 2)`
	`Totals: comparisons: 19, swaps: 14`
	Note: After each pass, the underlined sublist from the **previous** swap is ordered, and one additional term is also underlined ready to be inserted during the upcoming pass (this **should** be indicated by underlining as shown). Therefore once a comparison shows a swap is not necessary, subsequent comparisons for that pass are not required. The algorithm concludes after every element has been inserted into the correct position.

Visual:	For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com 

Shell Sort

Name:	Shell sort is named after Donald Shell who published the version we use here in 1959.
Summary:	Split the data into sublists, shuttle sort each sublist, combine sublists and repeat.
Efficiency:	For nearly sorted lists: $O(n)$ (takes about n comparisons to add a new item to a sorted list) For reverse order lists (worst case): $O(n^2)$ (but this can be improved on slightly by varying the gap size (in D1 we always use powers of 2, but less regular gaps have been shown to increase efficiency to $O\left(n^{\frac{4}{3}}\right)$ or better). Inherits the efficiency of shuttle sort for the small sublists, and has the added advantage of being able to rapidly relocate items initially far from their correct position.
Algorithm:	Divide the data into $int\left(\frac{n}{2}\right)$ sublists (ie into sublists of 2 items each), by taking the 1^{st} and $\frac{n}{2}^{th}$ as one sublist, 2^{nd} and $\left(\frac{n}{2}+1\right)^{th}$ as another, etc. Shuttle sort each sublist and merge back together. For the next pass, divide into $int\left(\frac{n}{4}\right)$ sublists (ie sublists of 4 items) and repeat. When the sublist is the whole list, perform one final shuttle sort of the whole list.

Example:

Sort the list 8 3 2 6 9 4 2 7 using shell sort.

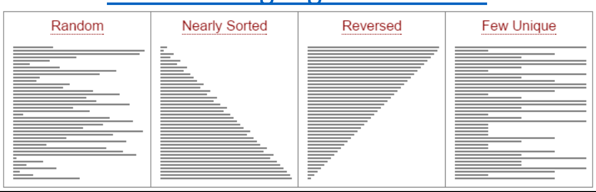
```
4 sublists: 8           9
               3           4
                  2           2
                     6           7
           ------------------------
Sort:       8           9
               3           4
                  2           2
                     6           7 (comparisons: 4, swaps: 0)
Merged:     8  3  2  6  9  4  2  7


2 sublists: 8     2     9     2
               3     6     4     7
           ------------------------
Sort:       2     2     8     9
               3     4     6     7 (comparisons: 5, swaps: 3)
Merged:     2  3  2  4  8  6  9  7


1 sublist:  2  3  2  4  8  6  9  7
           ------------------------
Sort:       2  2  3  4  6  7  8  9 (comparisons:11, swaps: 4)
```

Totals: comparisons: 20, swaps: 7

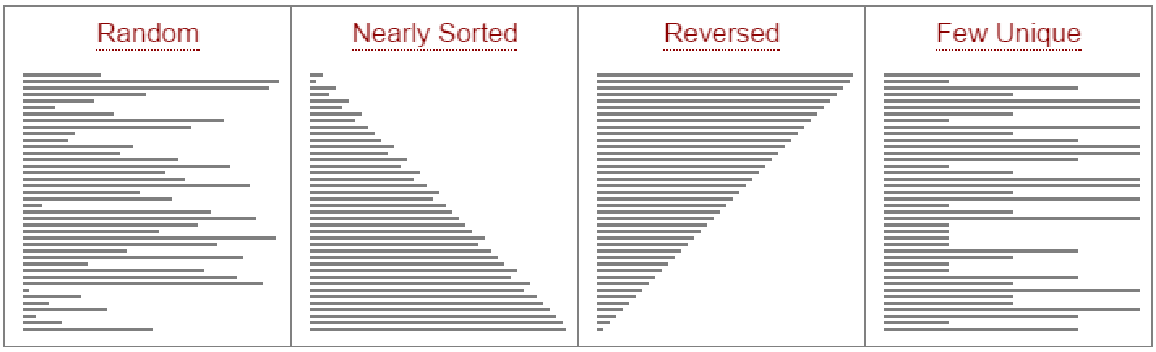
Note: The staggered layout is the clearest way to indicate sublists (each row is a sublist, but they maintain their position within the overall list this way). After each pass, merge again. The shuttle sorts completed within each sublist and at the end do not need to be shown.

Visual:

For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com



Quick Sort

Name:	Quick sort, like shell sort, is a 'divide-and-conquer' algorithm, designed to be really efficient. Also known as the 'partition-exchange' sort since each pivot partitions the remaining items.
Summary:	Choose a pivot and compare each item to it, forming two sublists. Recursively apply the algorithm to each sublist until all items have been pivots and are therefore in place.
Efficiency:	On average: $O(n \log n)$ For worst case: $O(n^2)$ (but this is fairly rare) Although traditionally (and in our implementation) the first element of any sublist is chosen as a pivot, in mostly sorted or reverse order lists this results in worst case behaviour, so a middle value is often chosen instead. Another common optimisation is to use shuttle sort for sublists that are sufficiently small, since it is a very cheap algorithm for small lists.
Algorithm:	Set the first item as a pivot and (without reordering) compare all subsequent numbers in the list with the pivot, adding the lower to a left-most list and the higher to a right-most list. Choose a pivot for each sublist and repeat the procedure on each sublist until all sublists contain only one element. Each pivot will end up in the correct place after use.
Example:	Sort the list 8 3 2 6 9 4 2 7 using quick sort. Pivots: **8** 3 2 6 9 4 2 7 First pass: 3 2 6 4 2 7 **8** 9 (comparisons: 7, swaps: 6) Pivots: **3** 2 6 4 2 7 **8** 9 Second pass: 2 2 **3** 6 4 7 **8** 9 (comparisons: 5, swaps: 2) Pivots: **2** 2 **3** **6** 4 7 **8** 9 Third pass: **2** 2 **3** 4 **6** 7 **8** 9 (comparisons: 3, swaps: 2) **Totals: comparisons: 15, swaps: 10** Note: Sublists of 1 can be ignored since they are automatically in the correct place.
Visual:	For more details and an animation of this algorithm for a number of different cases, see: www.sorting-algorithms.com 

Challenging Exam Question: Sorting Algorithms (D1 – Jan '12)

8 Four **distinct positive integers** are $(3x-5)$, $(2x+3)$, $(x+1)$ and $(4x-13)$.

(a) Explain why $x \geqslant 4$. *(2 marks)*

(b) The four integers are to be sorted into ascending order using a **bubble sort**.

The original list is	$(3x-5)$	$(2x+3)$	$(x+1)$	$(4x-13)$
After the first pass, the list is	$(3x-5)$	$(x+1)$	$(4x-13)$	$(2x+3)$
After the second pass, the list is	$(x+1)$	$(4x-13)$	$(3x-5)$	$(2x+3)$
After the third pass, the list is	$(4x-13)$	$(x+1)$	$(3x-5)$	$(2x+3)$

(i) By considering the list after the first pass, write down **three** inequalities in terms of x. *(3 marks)*

(ii) By considering the list after the second pass, write down **two** further inequalities in terms of x. *(2 marks)*

(iii) By considering the list after the third pass, write down **one** further inequality in terms of x. *(1 mark)*

(c) Hence, by considering the results above, find the value of x. *(2 marks)*

Sorting Algorithms (D1 – Jan '12)

8.

a)

Positive integers, so $4x - 13 > 0 \quad \Longrightarrow \quad 4x > 13 \quad \Longrightarrow \quad x > \frac{13}{4} \quad \Longrightarrow \quad \boldsymbol{x \geq 4}$

Note: we know x is an integer because $x + 1$ is an integer.

b)

i.

$$\boldsymbol{2x + 3 > 3x - 5 \qquad 2x + 3 > x + 1 \qquad 2x + 3 > 4x - 13}$$

This is because after the first pass, the largest number will have been placed at the end.

Note: it is not necessary to further simplify or analyse these inequalities, but if this were done they would yield the results: $x < 8$, $x > -2$ and $x < 8$ which, combined with the original inequality, gives: $4 \leq x \leq 7$.

ii.

$$\boldsymbol{3x - 5 > x + 1 \qquad 3x - 5 > 4x - 13}$$

This is because after two passes, the two largest numbers will be in their correct places.

Note: it is not necessary to further simplify or analyse these inequalities, but if this were done they would yield the results: $x > 3$, $x < 8$ which don't narrow down the range any further: $4 \leq x \leq 7$.

iii.

$$\boldsymbol{x + 1 > 4x - 13}$$

This is because after three passes, the three largest numbers will be in their correct places.

Note: it is not necessary to further simplify or analyse this inequality, but if this were done it would yield the result: $x < \frac{14}{3}$ which, combined with the other inequalities, gives: $4 \leq x \leq 4$.

c)

Starting from the last result, we have:

$$14 > 3x \quad \Longrightarrow \quad x < \frac{14}{3} \quad \Longrightarrow \quad x \leq 4$$

Since we already know from part a that $x \geq 4$ we have $4 \leq x \leq 4$ and therefore the only choice is $\boldsymbol{x = 4}$.

Checking:

This value of x gives an initial list of: 7 11 5 3

A first pass of: 7 5 3 11

A second pass of: 5 3 7 11

A third and final pass of: 3 5 7 11

These are consistent with a bubble sort.

Note: No further passes are required, despite not having zero swaps because $n - 1$ passes ensures the final $n - 1$ items are in the right place, which means the first item must be too.