Sorting

- We won't be learning any more Python in this section, and will instead be focusing on algorithm design and problem-solving.

Lots of sorting happens behind the scenes of modern computing. Take e.g. lists of scores in videogames, or Google sorting search results.

Built-in Methods of Sorting

List Sorting (lst.sort())

Python can already use the `.sort()` list method:

```
lst = [5, 7, 1, 4, 2, 3]
lst.sort()
print(lst) => [1, 2, 3, 4, 5, 7]
```

Also the built-in function `sorted()`, which functions slightly differently, in that it doesn't alter the things it's applied to:

```
for key in sorted(d):
    pass
```

In this example, d remains unsorted, while sorted(d) returns a sorted version of it.

DIY Sorting

- If you do the sorting yourself, you can design customised sorting algorithms that do specific things.
- Also it's good practice of both problem-solving and Python.

Insertion Sort

Here's how to do it:

1. Look at the numbers in turn.
2. As you go, keep track of the numbers you've seen but in sorted order.
3. When you get to a new number, find where it goes in your sorted list of numbers so far, and put it there.

Example:

```
[5, 8, 2, 9, 7, 3]

5
5   8
2   5   8
2   5   8   9
2   5   7   8   9
2   3   5   7   8   9
```

In Python, it is possible to alter the list as you go, rather than creating a second list as you go.

Selection Sort

1. Go through the list, find the smallest number.
2. Move that to be first.
3. Go through the rest of the list, find the new smallest number.
4. Move that to be second.
5. etc

Here's an example in Python:

```
for i in range(len(lst)):
    imin = i
    for j in range(i+1, len(lst)):
        if lst[j] < lst[imin]:
            imin = j
    (lst[i], lst[imin]) = (lst[imin], lst[i])
```

The last line swaps the two values, like (x, y) = (y, x) will swap the values of y and x.

Merge Sort

- This won't be on the exam.

Take the list `[5, 7, 1, 4, 2, 6, 8, 9, 3]`.

1. Split the list in two (e.g. take the first 4 numbers and the last 5 numbers).
2. Sort each segment using recursion.
3. Recombine the segments by taking the smaller of the front element of each list repeatedly until you're done.

Merge sort seems complicated, but is much faster for large lists than either SelectionSort or InsertionSort. See the speed analysis below.

Quick Sort

Take the list `[5, 7, 1, 4, 2, 6, 8, 9, 3]`.

1. Take a number in the list (e.g. the first number).
2. Split the rest of the list into numbers that are smaller than our chosen number, and numbers that are bigger than our chosen number.
3. Sort each of those lists using recursion.
4. Recombine by putting your chosen number between the two sorted lists.

Quick Sort is also much faster for large lists than the first two sorting algorithms.

Speed Analysis

Let n = the number of items in the list.

Algorithm	Comparisons
Insertion	~n squared
Selection	~n squared
Merge	n.log_2(n)
Quick	n.log_2(n)

When the number of comparisons is n squared, doubling the length of the list increases the time to solve it by 4. n.log_2(n) is much faster for large numbers: If n is 1 million, n^2 is 1 million by 1 million, but n.log_2(n) is ~20 by 1 million, which is much smaller.

You can prove that no sorting algorithm can be designed which takes fewer than n.log_2(n) comparisons, which means that Merge Sort and Quick Sort are optimal algorithms.

Handouts & Assignments

- Handout 22
- Assignment 17