

# SORTING ALGORITHMS

## Today

- Last time we started looking at how to sort.
- Today we'll look at sorting in more detail.
- We'll think about the code that we'll need to sort things.
- We will also think a bit about the *computational complexity* of the sorting process.

- In particular we thought about sorting numbers, but we can sort any collection of things where we put the things in order.
- So, we can sort:
  - Presidential candidates you could vote for.
  - Games you might play when you are done with your CS homework.
  - Meals you might eat for dinner.
- Of course, to sort these you'd need more than just the  $>$  operator.
- Before we described some sorting methods rather abstractly.
- Now we'll think about the code we need to do these things.

- The four sorting methods we considered last time were:
  - Blort sort;
  - Selection sort;
  - Insertion sort; and
  - Bubble sort.
- We won't look at blort sort (because it isn't one you'll ever need to know, and it is also a bit too complicated for you to deal with programming), but we'll look at the code for all the others.
- The code is all in the program `sort.cpp` which you can download from the course website.

## Bubble sort

- Bubble sort is perhaps the simplest way of sorting.
- We just go through the list, swapping adjacent items if they aren't in the right order.
- Doing this just once won't always order the whole list.
- But if we do it once for each item in the list (ie 5 times for a 5 item list), then we will always sort the list.
- This gives the code on the next slide (assuming we have the function `sort` which we discussed in an earlier lecture).

```
for(i = 0; i < size; ++i) {  
  
    // For each position, compare that number with the one that  
    // follows.  
  
    for(j = 0; j < (size - 1); j++){  
  
        // If the following number is smaller, swap the two.  
  
        if(a[j] > a[j+1]){  
            swap(a[j], a[j+1]);  
        }  
    }  
}
```

## Selection sort

- We can do selection sort a bit more simply than we did in the lecture (the code from the lecture is in `sort.cpp` also).
- As before we look for the smallest value in the original array.
- But when we find it, we *swap* it with the value in the auxilliary.
- Swapping means that we no longer have to remember which values we have already found.
- So long as we have initialised the auxilliary array to have large values in it, the values we swap into the original array won't be a problem.
- When we are done, we have to copy all the values from the auxilliary back into the original array.

```
for(i = 0; i < size; ++i){  
  
    // Go through the array we have to sort, and find the  
    // smallest element.  
  
    for(j = 0; j < size; ++j){  
  
        // When we find it, swap its value with that in the  
        // auxillary array  
  
        if(a[j] <= aux[i]){  
            swap(aux[i], a[j]);  
        }  
    }  
}
```



- Comparing the two versions of selection sort, illustrates an important idea.
- The one from the lecture is a pretty direct implementation — it does exactly what we said selection sort does when we described it last time.
- The one above is a bit different, but a lot simpler (less variables, less lines of code).
- You often find this tradeoff.

## Insertion sort

- For insertion sort we take each element from the list and put it into the auxilliary.
- We put it in place in order.
- To find the right place we have to look through the auxilliary list.
- Once we have found the right place, we have to move all the remaining items in the auxilliary list down to make room.
- Before we run the code on the next slide, we have to set up the auxilliary array so that every item has a large value.
- After the code on the next slide has run, we have to copy the values from the auxilliary array back into the original array.

```
for(i = 0; i < size; ++i){  
    // Look through the auxilliary  
    for(j = 0; j < size; ++j){  
        // until the element of the array is smaller than an element of the auxilliary.  
        if(a[i] < aux[j]){  
            // then move all the remaining elements in the auxilliary down to make room  
            for(k = size-1; k >= j; --k){  
                aux[k] = aux[k-1];  
            }  
            // and copy the element of the original array across  
            aux[j] = a[i];  
            // At this point we don't need to look through the auxilliary array any more.  
            j = size;  
        }  
    }  
}
```

## Bonus — linear sort

- Here's another kind of sorting, *linear sort*.
- This is like selection sort, but without the auxiliary array.
- To linear sort, we look in turn at each member of the array in turn.
- For each of these we look at all the members *later* in the array.
- If the later member is smaller, we swap the two.
- This is now very close to what we do in bubble sort.
- The next slide has the code. Again this is in `sort.cpp`

```
for(i = 0; i < size; ++i){  
  
    // For each position, look through every later position  
    // in the array.  
  
    for(j = i; j < size; j++){  
  
        // If one of the later numbers is smaller, then swap  
        // the two  
  
        if(a[j] < a[i]){  
            swap(a[j], a[i]);  
        }  
    }  
}
```

## Computational complexity

- With many different ways to solve a problem like sorting a list, we are interested in which way is “best”.
- “Best” can be measured in at least two ways:
  - Which method uses least computer time.
  - Which method uses least computer memory
- We will think about time (“time complexity”).
- Because we want to think about complexity without taking variables like the speed of the computer into account, we think in terms of the number of operations a computer has to carry out.
- We also think about how this time varies as the size of the problem we are solving changes.
- Here, naturally, the size of the problem is the length of the list.

- What is the complexity of bubble sort?
- Well, if we have an array with  $N$  elements, the outer `for` loop will be executed  $N$  times.
- Each of those executions of the loop will execute the inner `for` loop  $N - 1$  times.
- In total, that is  $N(N - 1)$  or:

$$N^2 + N$$

executions of the innermost `if` statement and is comparison of values.

- The most significant part of this number is the  $N^2$ , and we write the number of comparisons as  $O(N^2)$ .
- This is known as “Big O” notation.

- What about linear sorting?
- Here we have to do  $N$  comparisons followed by  $N - 1$  comparisons, followed by  $N - 2$  comparisons, followed by ... all the way down to  $N - N$  comparisons.

- In total, that is:

$$\frac{N(N + 1)}{2}$$

comparisons.

- Thus the complexity of the algorithm is also  $O(N^2)$ , and it turns out that all the complexity of all of the approaches to sorting that we have looked at here are  $O(N^2)$ .



## Summary

- We looked in more detail at the different types of sorting.
- We also gave the code for them.
- Finally, we considered their complexity.