

• In particular we thought about sorting numbers, but we can sort any collection of things where we put the things in order.

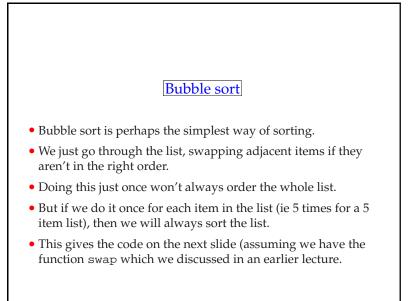
• So, we can sort:

- 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.

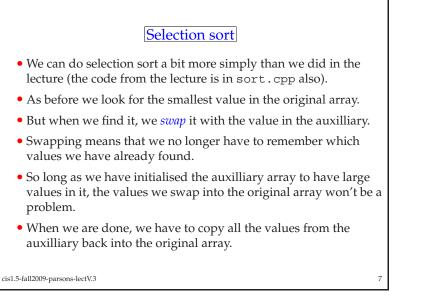
- The four sorting methods we considered last time were:
 - Blort sort;
 - Selection sort;
 - Insertion sort; and
 - Bubble sort.
- We won't look at blort sort (because it isn't one you'll ever need to know, and it is also a bit too complicated for you to deal with programming), but we'll look at the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

cis1.5-fall2009-parsons-lectV.3

3



cis1.5-fall2009-parsons-lectV.3



```
for(i = 0; i < size; ++i) {</pre>
```

 $\ensuremath{{\prime}}\xspace$ // For each position, compare that number with the one that $\ensuremath{{\prime}}\xspace$ // follows.

for(j = 0; j < (size - 1); j++){</pre>

// If the following number is smaller, swap the two.

```
if(a[j] > a[j+1]){
    swap(a[j], a[j+1]);
}
```

cis1.5-fall2009-parsons-lectV.3

```
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]);
      }
   }
}</pre>
```

cis1.5-fall2009-parsons-lectV.3

- 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.

```
cis1.5-fall2009-parsons-lectV.3
```

```
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;
}
cisl.5-fall2009-parsons-lectV.3
```

Insertion sort

- We already saw this in Lecture V.1
- 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.

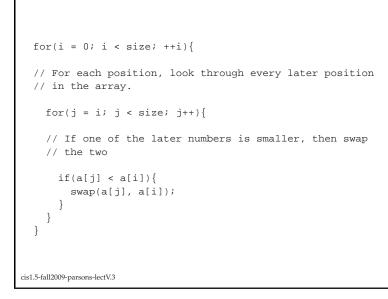
cis1.5-fall2009-parsons-lectV.3

Bonus — linear sort

- Here's another kind of sorting, *linear sort*.
- This is like selection sort, but without the auxilliary 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

cis1.5-fall2009-parsons-lectV.3

10



- 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.

cis1.5-fall2009-parsons-lectV.3

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.

cis1.5-fall2009-parsons-lectV.3

13

15

- 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 $O(N^2)$.

cis1.5-fall2009-parsons-lectV.3

16

14

