Sorting is a way to order a bunch of items. There are different sorting methods that can be compared by how quick they are for a certain data size and whether they are stable or not. In this email I will be covering what Carey calls the "stupid sorts." A stable sorting algorithm takes into account the initial order when sorting. Here is an example:

There are two 2s.

2(a) 2(b) 1

A stable sort would sort them like this:

1 2(a) 2(b)

An unstable sort like this:

1 2(b) 2(a)

As I go  into each of the algorithms, you'll see why some sorts end up being stable or unstable.

Here are some numbers that we will be sorting

2 6 5 1 4 3

**Selection Sort**

The selection sort algorithm basically finds the smallest number (or whatever you are sorting) and swaps it with the first number. Then it looks at the rest of the numbers, ignoring the first, and swaps the smallest number of that selection.

1. Look at all n numbers and find the smallest number.

2. Swap it with the first number.

3. Look at the remaining n-1 numbers and find the smallest number.

4. And so on

Let's try using the algorithm on selection of numbers, and then write some code to do the same thing.

2 6 5 1 4 3

1. Smallest number is 1, swap 1 and 2

1 6 5 2 4 3

2. Now the smallest number is 2, swap 2 and 6

1 2 5 6 4 3

3. Swap 3 and 5

1 2 3 6 4 5

4. Swap 4 and 6

1 2 3 4 6 5

5. Swap 5 and 6

1 2 3 4 5 6

Notice that if the selection sort has gone through n iterations, the first n items *will* be in order.

Here is the code, with our selection of numbers an array:

void selection(int foo[], int n)

{

     for (int i = 0; i < n; i++)

     {

          int min = i;     // Save the index of the first item

          for (int j = i + 1; j < n; j++)     // Comparing every other item to the first item

          {

               if (foo[j] < foo[min])     // If it is less than the current min, that's the new min

                    min = j;

          }     // Once it exits the loop, min is equal to the smallest item between the ith item and n-1

          swap(foo[i], foo[min]);     // Swap the first item with the smallest item

}

So what's the big-o? There are two loops that each loop for n times, so O(n^2). As far as sorting methods go, this is pretty slow.

Is it stable or unstable? This is unstable. Why? Let's sort foo = {2, 2, 1}

2(a) 2(b) 1

The smallest number is 1, so we swap that with the first one. This results in

1 2(b) 2(a)

which is unstable.

**Insertion Sort**

The insertion sort compares an increasingly larger size of numbers. Every time it compares the last number of the selected size and if it is in the wrong order, remove it, shift all of the other numbers, and insert it into the right slot.

1. Start with a set size 's' equal to 2.

2. Look at the first s numbers.

3. If the last number in the set is in the wrong place

     i. Remove it

     ii. Shift how ever many numbers necessary before it to the right

     iii. Insert the number into the right slot (the first one in our case)

4. Increment s.

2 6 5 1 4 3     s = 2

1. 6 is in the right place (it is larger than 2)

2 6 5 1 4 3     s = 3

2. 5 is not in the right place. Remove it.

2 6   1 4 3

Shift 6 over.

2  6 1 4 3

Is 2 < 5? Yes, insert 5 there

2 5 6 1 4 3

2 5 6 1 4 3     s = 4

3. 1 is not in the right place. Remove it.

2 5 6   4 3

Shift 6 over.

2 5   6 4 3

Is 5 < 1? No, shift 5 over.

2   5 6 4 3

Is 2 < 1? No, shift 2 over.

  2 5 6 4 3

Now we have no choice but to put 1 in front

1 2 5 6 4 3

1 2 5 6 4 3     s = 5

4. 4 is not in the right place. Remove it.

...

1 2   5 6 3

Insert 4 there

1 2 4 5 6 3

1 2 4 5 6 3     s = 6

5. 3 is not in the right place. Remove it.

...

1 2   4 5 6

Insert 3 here

1 2 3 4 5 6

Here is the code:

void insertion(int foo[], int n)

{

     for (int s = 2; s <= n; s++)     // Start selection to look at 2

     {

          int last = foo[s - 1];     // Look at the last item in the selection

          int i = s - 1;               // We want to compare the last item to all of the other item(s)

          while (i >= 0 && last < foo[i])     // Keep on shifting the items until we found an item less than the selected item.

          // If the last item isn't less than the next-to-last one, we don't shift any items

          {

               foo[i + 1] = foo[i];     // Shift the items to the right

               --i;     // Going backwards

          }

         foo[i + 1] = last;     // Insert the last item in the correct place. If we didn't go through the loop, if will just reinsert into the same place

     }

}

What is the big-o? The outer loop will run at most n times. The inner loop will run at most n-1 times. Thus it is roughly O(n^2).

This is a stable sort. Let's look at our example again.

2(a) 2(b) 1     s = 2

2(b) is in the right place.

2(a) 2(b) 1     s = 3

1 is not in the right place. Remove it.

2(a) 2(b)

Shift 2(b) over.

2(a)   2(b)

Is 2(a) < 1? No, shift it over.

   2(a) 2(b)

Insert 1

 1 2(a) 2(b)

As you can see, the 2s have remained in the same initial order.

**Bubble Sort**

The bubble sort compares every two elements. If they are out of order they get swapped. If by the end it made at least one swap, it goes through the whole process again.

1. Compare the first two elements.

2. If they are out of order, swap them.

3. Advance one element

4. Repeat the process until the end of the array

5. If you made at least one swap, repeat from step 1

2 6 5 1 4 3

1. In order

2 6 5 1 4 3

2. Out of order, swap them

2 5 6 1 4 3

2 5 6 1 4 3

3. Out of order, swap them

2 5 1 6 4 3

2 5 1 6 4 3

4. Out of order, swap them

2 5 1 4 6 3

2 5 1 4 6 3

5. Out of order, swap them

2 5 1 4 3 6

6. Did you swap at least once this iteration? Yes, go back to the beginning

2 5 1 4 3 6

2 1 5 4 3 6     swap

2 1 4 5 3 6     swap

2 1 4 3 5 6     swap

2 1 4 3 5 6

3rd iteration

1 2 4 3 5 6     swap

1 2 4 3 5 6

1 2 3 4 5 6     swap. Notice at that this point we already have it in order, but because of the algorithm, we need to finish the iteration and do the next one

1 2 3 4 5 6

1 2 3 4 5 6

4th iteration (because we made at least one swap in the previous iteration)

1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5 6

1 2 3 4 5 6

Done!

Notice that this process isn't very efficient, but it's good for something that is basically already sorted, or for checking if something is sorted.

Here's the code

void bubble(int foo[], int n)

{

     bool swap;

     do

     {

          swap = false;

          for (int i = 0; i < (n - 1); i++)  // Don't want to go past the last element because of i+1

          {

               if (foo[i] > foo[i + 1])

               {

                    swap(foo[i], foo[i + 1]);

                    swap = true;

               }

          }

     }

     while (swap == true);     // Only keep on looping if you swapped at least one time

}

What is the big-o? The outer loop runs at most n times (it can go through the entire array at most n times) and the inner loop runs at most n times as well. the O(n^2).

This is a stable sort.

2(a) 2(b) 1

They are in order

2(a) 2(b) 1

They are not in order, swap them

2(a) 1 2(b)

We did at least one swap, go through array again

2(a) 1 2(b)

They are not in order, swap them

1 2(a) 2(b)

1 2(a) 2(b)

They are in order

We did at least one swap, go through array again

1 2(a) 2(b)

They are in order

1 2(a) 2(b)

They are in order

We did not do any swaps, so we are done!

The 2s have kept their initial ordering.

**H Sort**

The idea of h sorting is similar to that of the bubble sort, but instead of comparing every element with the next one, you compare every ith element with the i + hth element. H is some number that you choose to sort the array with, like 3. Note that an h sort (unless its a 1-sort, like a bubble sort) does not completely sort the array, but it is used for a different type of sorting, the shell sort.

1. Pick a value for h

2. If arr[i] and arr[i + h] are out of order, swap them

3. Continue until the end of the array (when arr[i + h] == arr[n - 1])

4. If you made at least one swap, repeat from step 1

Let's 3-sort these numbers (h = 3)

2 6 5 1 4 3

i = 0

i + h = 3

Compare 2 and 1. 1 < 2 so swap

1 6 5 2 4 3

1 6 5 2 4 3

i = 1

i + h = 4

Compare 6 and 4. Swap

1 4 5 2 6 3

1 4 5 2 6 3

i = 2

i + h = 5

Compare 5 and 3. Swap

1 4 3 2 6 5

We swapped at least once this loop, start again

1 4 3 2 6 5

i = 0

i + h = 3

Don't swap

1 4 3 2 6 5

i = 1

i + h = 4

Don't swap

1 4 3 2 6 5

i = 2

i + h = 5

Don't swap, didn't swap at all, done!

Here's the code. Notice that you just change every + 1 to + h. If you passed in 1 as h, it would be a bubble sort.

void h(int foo[], int n, int h)

{

     bool swap;

     do

     {

          swap = false;

          for (int i = 0; i < (n - h); i++)  // Don't want to go past the last element because of i+h

          {

               if (foo[i] > foo[i + h])

               {

                    swap(foo[i], foo[i + h]);

                    swap = true;

               }

          }

     }

     while (swap == true);     // Only keep on looping if you swapped at least one time

}

Is this stable? If it is a 1-sort (bubble sort) it is, but if it's anything other than that it is not.

Here's the example again (one more 2 added so we can do a 2-sort).

h = 2

2(a) 2(b) 2(c) 1

i = 0

i + h = 2

Don't swap

2(a) 2(b) 2(c) 1

i = 1

i + h = 3

Swap

2(a) 1 2(c) 2(b)

We did at least one swap, repeat

2(a) 1 2(c) 2(b)

Don't swap

2(a) 1 2(c) 2(b)

Don't swap, done!

As you can see, the 2s have not preserved their order.

**Shell Sort**

The idea of a shell sort is that you have a sequence of decreasing h-values that ends with an h-value of 1 (bubble). This will actually fully sort the array! There are different theories as to which sequence of h-values you should choose for the most efficiency, but let's do a simple 3-sort, 2-sort, 1-sort.

1. Choose a sequence of decreasing h-values, ending with 1

2. Do the first h-sort

3. Repeat as necessary for the next h-values

h = 3

2 6 5 1 4 3

i = 0

i + h = 3

Compare 2 and 1. 1 < 2 so swap

1 6 5 2 4 3

1 6 5 2 4 3

i = 1

i + h = 4

Compare 6 and 4. Swap

1 4 5 2 6 3

1 4 5 2 6 3

i = 2

i + h = 5

Compare 5 and 3. Swap

1 4 3 2 6 5

We swapped at least once this loop, start again

1 4 3 2 6 5

i = 0

i + h = 3

Don't swap

1 4 3 2 6 5

i = 1

i + h = 4

Don't swap

1 4 3 2 6 5

i = 2

i + h = 5

Don't swap, didn't swap at all, done!

h = 2

1 4 3 2 6 5

i = 0

i + h = 2

Don't swap

1 4 3 2 6 5

i = 1

i + h = 3

Swap

1 2 3 4 6 5

1 2 3 4 6 5

i = 2

i + h = 4

Don't swap

1 2 3 4 6 5

i = 3

i + h = 5

Don't swap

We swapped at least once this loop, start again

1 2 3 4 6 5

i = 0

i + h = 2

Don't swap

1 2 3 4 6 5

i = 1

i + h = 3

Don't swap

1 2 3 4 6 5

i = 2

i + h = 4

Don't swap

1 2 3 4 6 5

i = 3

i + h = 5

Don't swap, didn't swap at all, done!

h = 1

1 2 3 4 6 5

Don't swap

1 2 3 4 6 5

Don't swap

1 2 3 4 6 5

Don't swap

1 2 3 4 6 5

Don't swap

1 2 3 4 6 5

Swap

1 2 3 4 5 6

Repeat loop because we swapped once

[after loop]

1 2 3 4 5 6

Done!

As you can probably guess, the shell sort is unstable because it largely consists of the h-sort, which is unstable.

That's it!