Sorting: Reread Chapter 2.2 and read chapter 2.3 in the text.

1. QUADRATIC SORTING ALGORITHMS

Suppose that you have an array of random numbers called theArray, of size n, and you want to sort it, smallest to largest.

If we just built a sorted array at the front of our unsorted array, would that pay off? That's the idea behind the first algorithm, insertion sort, which you remember from the first weeks of school. It keeps adding new items as it builds a the sorted array at the front. Here's the logic for insertion sort.

```
for (int unsorted = 1; unsorted < n; unsorted++)
{
  // Find the right position (loc) in theArray[0..unsorted]
  // for theArray[unsorted], which is the first entry in the
  // unsorted region; shift, if necessary, to make room
  ItemType nextItem = theArray[unsorted];
  int loc = unsorted;
  while ((loc > 0) && (theArray[loc - 1] > nextItem))
  {
    // Shift theArray[loc - 1] to the right
    theArray[loc] = theArray[loc - 1];
    loc--;
  } // end while

  // At this point, theArray[loc] is where nextItem belongs
  theArray[loc] = nextItem; // Insert nextItem into sorted region
}
```

Another way to tackle this is bubble sort: write a loop to compare each item in the array to the one after it; if the two items are out of order, swap them.

```
bool sorted = false; // False when swaps occur
int pass = 1;
while (!sorted && (pass < n))
{
  sorted = true; // Assume sorted
  for (int index = 0; index < n - pass; index++)
  {
    int nextIndex = index + 1;
    if (theArray[index] > theArray[nextIndex])   // look for out of order entries
    {
      // Exchange entries
      std::swap(theArray[index], theArray[nextIndex]);
      sorted = false; // Signal exchange
    }
  }
  pass++;
```

```
  }
```

At the end of the loop, the biggest item is at the end of the array. This loop must run, at most, n-1 times for an array of size n. It must do (at worst) n-1 swaps each time. (Check this on an array that is sorted exactly backwards.) This algorithm is called bubble sort.

What kind of time will this take? In the worst case, we must still make n-1 passes through the array and all n-1 swaps will be needed. But we might do better. We could stop if we found no items out of order in any pass, so we might only make one starting pass if the array was already sorted.

So bubblesort gives us O(n^2) in the worst case, and O(n) in the best case.

Selection sort tackles this by finding the largest item in the unsorted part of the array and moving it into place at the sorted end:

```
  for (int last = n - 1; last >= 1; last--)
  {
    // Select the largest entry in theArray[0..last]
    int largest = findIndexofLargest(theArray, last+1);       // this makes another loop

    // Swap the largest entry, theArray[largest], with
    // theArray[last]
    std::swap(theArray[largest], theArray[last]);
  }
```

Time: O(n-1 + n-2 + … + 2 + 1) = O(n * (n-1) / 2) = O(n^2). Quadratic.

This might look faster—but it's not fast enough to break the quadratic O(n^2) barrier. We pay in rearrangements of the array; every time we write a new item into the sorted part, we have to shift the other elements one slot forward to make room.

None of the sort methods we'd think of quickly would probably work better than these. That worst case O(n^2) is hard to break. And that quadratic time bound is too slow!

Next, we'll discuss 3 algorithms that achieve O(n * log(n)) performance, which is much faster. These algorithms are mergesort, quicksort, and heapsort. All of these use recursion (and divide and conquer logic) to do the job. We'll cover these in the next lecture or two. Your programming exam next week will involve these sorting algorithms.