n-squared Sorting Algorithms, Reading

As we noted in an earlier module, $O(n^2)$ does not work well for large data sets because it does not *scale* well. It's fine for small data sets, like the subject-count array we used in the DVC schedule history applications. But at some point we'll have to come up with something that scales well. Here's our starting point:

Nested For-Loop Sort

As developed in earlier studies, for an array "a" with its first "n" values (from index zero to index n-1) to be rearranged lo-to-hi, and with less-than defined for the ADT, we have this:

```
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    if (a[j] < a[i])
      swap(a[i], a[j]);
```

Note how the if-statement is written to use less-than. To arrange hi-to-lo, exchange the values being compared, instead of switching to a greater-than, so that we remain consistent with using only equals-equals and less-than for comparisons (like the C++ STL does).

Applying the same logic to linked structures, with a Node struct containing its ADT as an attribute named "value" and with a head pointer named "firstNode" it's:

```
for (Node* p = firstNode; p; p = p->next)
  for (Node* q = p->next; q; q = q->next)
    if (q->value < p->value)
      swap(p->value, q->value);
```

Note that the node linkages remain unchanged, swapping only the ADT values.

Bubble Sort

A slight variation of the nested for-loop sort compares *adjacent* values, swapping when the are not ordered correctly with respect to each other. Traversing the array from index zero to the end has the effect of moving the *highest* value to the end. That done, the process repeats from index zero, but stops before the last (correctly positioned) value. This continues in this way until there is only the index zero value to sort.

```
for (int i = n; i > 1; i--)
  for (int j = 1; j < i; j++)
    if (a[j] < a[j - 1])
      swap(a[j - 1], a[j]);
```

It does not work too well for singly-linked lists, because the inner loop has no good way of knowing where to end.

Insertion Sort

In the above algorithms, every possible combination of any two indexes gets compared. It seems logical and complete that you would have to do that. But in order to find solutions that are better that $O(n^2)$, we're going to have to find ways that do *fewer compares* and hopefully fewer swaps.

Once again we'll draw on human experience to see if we can find an efficient method. For this we go back to grade school when we learned to line ourselves up in alphabetical order. In case you don't remember back that far, the process went something like this:

1. The first student to arrive stands at the front of the line.
2. The next student stands behind the first, and compares their names. If the newly arriving belongs in front, the two swap positions. Otherwise the new

student remains at the end.
3. The next student stands at the end of the line, and through a series of name comparing exchanges, swaps with the student in front until they reach their correct place in line.
4. Repeat for each new arrival.

To see if this is any better than the either of the previous two methods, recall that we computed $n^2/2$ compares, and if half of those are swaps, then we have $3n^2/4$ operations.

In the above example with grade school students, the *first* student has to do zero compares and the last has to do anywhere from zero to n-1 compares. Zuthra would likely have to do zero, while Aaron would likely do n-1. On the average the *last* student would do about n/2 compares. The *average* student, arriving half-way through the process, then, would average n/4 compares.

Give n students averaging n/4 compares, that's $n^2/4$ *total* compare -- *half* that of the nested for-loop and the bubble sort. And probably the same savings for swaps. It has the same big oh scaling characteristics, but its constant multiplier is halved, and it at least shows that we don't have to compare every single pair of values.

Moving from the grade school classroom to C++ coding, let's consider that an array already contains all n values. Only the one at index zero is in the right place, if we ignore (for now) that positions 1 through n-1 are there. To model the arrival of new values at the end of the sorted part of the array, we simply traverse the array from 1 to n-1, considering each as a new arrival:

```
for (int i = 1; i < n; i++) // bring in each new value one-at-a-time
{
  for (int j = i - 1; j >= 0; j--) // traverse towards the front
  {
    if (a[j] < a[j + 1]) break;
    swap(a[j], a[j + 1]);
  }
}
```

There's an analogous linked list version, but that development is left to the student. Hint -- it *does* involve relinking nodes into a new list...

Insertion Sort, Best Case

Not only does the insertion sort seem to offer a better constant multiplier that nested for-loop and bubble, but it also has a "best case" scenario that the others don't -- what if the arriving students actually arrive in alphabetical order? Or what if the array is already in order? In those cases, each new arrival does *one* compare, not an average of n/4. Multiplying that by n values, we get O(n)!

Of course, why would we ever try to sort an array that's in order. But in reality, sometimes we may not know in advance that the array is already in order. And that if the array is *kind of* in order? Like the hash table probing and chaining that involved a small number of additional compares not proportional to n, maybe the small number of compares and swaps in our insertion sort will still make it behave like O(n) -- at least is should be closer to O(n) than $O(n^2)$.

Best Case, Average Case

Now that we have a term for "best case", the original big oh needs a name, and we call that the "average case". It's for when the data is randomly arranged. Best cases usually assume something about the arrangement that is not entirely random. Sometimes the best case is no better than the average case in terms of big oh. It might be better in terms of a constant multiplier, but it's *scaling* we're interested in here.