Selection Sort

The algorithm to sort n numbers is as follows:

For the i^{th} element (as i ranges from 0 to n-1)

1) **Determine the smallest element in the rest of the array to the right of the i^{th} element.**
2) **Swap the current i^{th} element with the element identified in step 1.**

In essence, the algorithm first picks the smallest element and swaps it into the first location, then it picks the next smallest element and swaps it into the next location, etc. In order to do #1, iterate through the array keeping track of the index that is currently storing the minimum value.

Let's look at an example of selection sort on the list:
8, 3, 1, 9, 5, 2

We iterate through this list starting at the first element all the way through until we determine that the smallest element is stored in index 2. We then swap this with the element at index 0 to yield:

1, 3, 8, 9, 5, 2. Similarly, we find the last element and swap it:
1, 2, 8, 9, 5, 3. Then,
1, 2, 3, 9, 5, 8. Followed by
1, 2, 3, 5, 9, 8, and finally
1, 2, 3, 4, 8, 9

The code for this (and all three sorts in this lecture is included in sort.c. Note that some printed versions of the algorithm select for the maximum first, not the minimum. This is a pretty minor modification to the overall idea.

In order to do this analysis, we see that the first time through, we will "go through" and compare every element in the array once. This is approximately n simple steps. The next time, we will only do n-1 simple steps because we no longer need to account for the first element because it is in the correct location.

After that, the next loop does n-2 steps, then n-3 steps, etc. The total number of simple steps (within a constant factor) that this algorithm executes is:

n+(n-1)+(n-2)+...+1

There is a formula for adding up the first n positive integers:

$$\mathbf{1+2+3+4+...+n} = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Consider the following derivation that Gauss supposedly came up with at the ripe old age of 7. (Note: This was known to people before Gauss, it's just amazing that he figured this out at such a young age...)

S = 1 + 2 + 3 + 4 + ... + n
S = n + (n-1) + (n-2) + (n-3) + 1

2S = (n+1)+(n+1)+... +(n+1)
2S = n(n+1), since (n+1) appears exactly n times above.
S = n(n+1)/2.

Thus, using order notation, we find that this algorithm runs in $O(n^2)$ time.

Insertion Sort

In this sort, we take each element one by one, starting with the second, and "insert" it into a sorted list. The way we insert the element is by continually swapping it with the previous element until it has found its correct spot in the already sorted list. Here is the algorithm for sorting n elements:

For the i^{th} element (as i ranges from 1 to n-1)

1) **As long as the current element is greater than the previous one, swap the two elements. Stop if there's no previous element.**

Consider using this algorithm to sort the following list:

3, 7, 2, 1, 5

Here are all the passes of the algorithm:

3, 7, 2, 1, 5, since 7 > 3, so the list 3, 7 is sorted.
3, 2, 7, 1, 5, since 2 < 7
2, 3, 7, 1, 5, since 2 < 3, so the list 2, 3, 7 is sorted.
2, 3, 1, 7, 5, since 1 < 7,
2, 1, 3, 7, 5, since 1 < 3,
1, 2, 3, 7, 5, since 1 < 2, so the list 1, 2, 3, 7 is sorted.
1, 2, 3, 5, 7, since 5 < 7 and now we can stop since 5 > 3.

You'll notice that the number of steps in this algorithm VARIES depending on the input...

Bubble Sort

The basic idea behind bubble sort is that you always compare consecutive elements, going left to right. Whenever two elements are out of place, swap them. At the end of a single iteration, the maximum element will be in the last spot. Now, just repeat this n times, where n is the number of elements being sorted. On each pass, one more maximal element will be put in place.

Consider the following trace through:

Original list:

6, 2, 5, 7, 3, 8, 4, 1

On a single pass of the algorithm, here is the state of the array:

2, 6, 5, 7, 3, 8, 4, 1 (The swapped elements are underlined)
2, 5, 6, 7, 3, 8, 4, 1
2, 5, 6, 7, 3, 8, 4, 1 (No swap occurs here, since the are in order)
2, 5, 6, 3, 7, 8, 4, 1
2, 5, 6, 3, 7, 8, 4, 1 (No swap)
2, 5, 6, 3, 7, 4, 8, 1
2, 5, 6, 3, 7, 4, 1, 8 (8 is now in place!)

On the next iteration, we can stop before we get to 8. Each subsequent iteration can stop one spot earlier on the list.

In some printed versions of the description of the algorithm, it stops if a whole iteration results on no swaps. Though this changes the best case run time to O(n), overall, it does not change either the average or worst case run times, in terms of order notation.

Limitation of Sorts that only swap adjacent elements

A sorting algorithm that only swaps adjacent elements can only run so fast.

In order to see this, we must first define an inversion:

An inversion is a pair of numbers in a list that is out of order. In the following list: 3, 1, 8, 4, 5 the inversions are the following pairs of numbers: (3, 1), (8, 4), and (8, 5).

When we swap adjacent elements in an array, we can remove at most one inversion from that array.

Note that if we swap non-adjacent elements in an array, we can remove multiple inversions. Consider the following:

8 2 3 4 5 6 7 1

Swapping 1 and 8 in ths situation above removes every inversion in this array (there are 13 of them total).

Thus, the run-time of an algorithm that swaps adjacent elements only is constrained by the total number of inversions in an array.

Let's consider the average case. There are $\binom{n}{2} = \frac{(n-1)n}{2}$ **pairs of numbers in a list of n numbers. Of these pairs, on average, half of them will be inverted. Thus, on average, an unsorted array will have** $\frac{(n-1)n}{4} = \Omega(n^2)$ **number of inversions, and any sorting algorithm that swaps adjacent elements only will have a** $\Omega(n^2)$ **run-time.**