[See the slides for figures to complement the discussion in these notes.]

One common problem that you need to solve in computing is to sort a list of N objects. Let's say we want the elements to be in *increasing* order. Here we are assuming that objects are comparable, in the sense that for any two objects A and B that we are considering, either $A < B$ or $A > B$. (It could also be that $A == B$, if there are multiple copies of the same element in the list.) For example, the elements in the list might be numbers, or they might be strings which can be ordered alphabetically. Today we will look a few simple algorithms. Later in the course, we'll look at more complicated algorithms which are much faster when the list is large.

We will discuss three algorithms today. These will be presented without Java code and without committing to a particular data structure e.g. array or array list or doubly or singly linked list. The algorithm could be implemented in principle with any of these, but the details would be distracting. Instead I would like you to think about the similarities and differences between the algorithms, which can be a bit subtle.

Bubble Sort

The first algorithm is perhaps the simplest to describe. You traverse through the list repeatedly, and whenever you find two neighboring elements that are out of order, you swap them. Elements gradually make their way to their correct position in the list. The algorithm is called bubblesort because one thinks of bubbles rising in a fluid [1]. Here it is:

```
ct = 0
repeat {
  continue = false
  for i = 0 to N - 2 - ct  {         //  N-1 is the last index
    if  list[ i ] > list[ i + 1 ]{
      swap( list[ i ], list[ i + 1 ] )
      continue = true
    }
  }
  ct++
} until continue == false
```

A few points to note: first, as you should have seen in COMP 202, swap is done using a temporary variable as follows. If you don't know why a temporary variable is needed, then see me.

```
swap(x,y){
   tmp = x
   x   = y
   y   = tmp
}
```

Second, what can we saw after one pass through the list? We can say that the largest element in the list will be at the end of the list. The reason is that the `for` loop will eventually hit this element and will then drag it to the end of the list via successive swaps.

[1] although this analogy doesn't really makes sense once you understand the algorithm, ...but it doesn't matter...

What can we say about the position of the smallest element in the list after the first pass? Not much, except that it won't be at the end of the list (unless all elements are equal). For example, if the smallest element of the list starts out at the end of the list, i.e. in position $N-1$, then in the first pass through the inner loop, the element will be moved only to position $N-2$.

How many passes through the list will we need to put all elements in order? We will need at most $N = 1$ passes. Take the case that the smallest element starts off at the end of the list at position $N-1$. In the first pass it moves to position $N-2$. In the second pass, it moves to position $N-3$. etc. Thus, it will take $N-1$ passes for the smallest element to arrive at position 0.

The `ct` variable counts the number of times through the outer loop. Each time through the outer loop, the largest element in positions from index 0 to `N-1-ct` is moved to position `N-1-ct`. See the example in the slides.

How long does the bubblesort algorithm take in the worse case that the outer loop is executed $N-1$ times ? There are two nested loops, so the operations within the inner loop are executed $(N-2)+(N-3)+\cdots+2+1$ times. This sum is $\frac{(N-2)(N-1)}{2}$ which is roughly $N^2/2$. Convince yourself that this *only* happens when the smallest element starts out at the end of the list.

Selection sort

The second algorithm we examine is called *Selection Sort.* The algorithm repeatedly finds the smallest element and moves it to its correct position in the list. For each i, the algorithm finds the minimum element in positions i to $N-1$. If this minimum element is at a position (`index`) different from i, then it swaps this minimum element with the element at position i.

```
  for  i = 0  to N-2   {
     index = i
     minValue = list[ i ]
     for k = i+1  to  N-1  {
        if ( list[k] < minValue ){
           index = k
           minValue = list[k]
        }
     }
     if ( index != i )
         swap(  list[i], list[ index ] )
}
```

Like bubblesort, this algorithm takes repeated passes through the array, but it covers fewer elements each time.

```
for (i = 0; i < N-1; i++)
        :
     for ( k = i+1; k < N; k++)
```

When `i` is 0, it takes `N-1` steps through inner loop (from k = 1 to N-1). When `i` is 1, it takes `N-2` steps through the inner loop, namely from k = 2 to N-1. The total number of times passing through the inner loop is

$$N-1+N-2+N-3+\ldots+3+2+1.$$

which is $\frac{N(N-1)}{2}$ which is roughly $N^2/2$.

Note a few differences with bubblesort. One difference is that in the best case bubble sort only takes one pass through the outer loop, whereas selection sort always takes $N-1$ pass through its outer loop. Thus bubble sort is faster in the best case. However, one advantage of selection sort over bubble sort is that selection sort does fewer swaps in the typical case. Indeed, selection sort does at most one swap in each pass in the outer loop, whereas bubblesort typically has to do a lot of swaps.

Insertion Sort

The third algorithm is similar to both the previous ones in that it uses nested loops. In particular, it is similar to selection sort in that it maintains a list of sorted elements at the front of the list, and then increases the size of the sorted list by one each time. However, whereas selection sort considers all the remaining unsorted elements and finding the smallest one, insertion sort considers only the next element in the list and finds where it belongs relative to the ones that have already been sorted. It then inserts this next element into the proper position. This is why it is called "insertion sort".

The algorithm goes through an outer loop $N-1$ times. In the kth pass through the loop (starting at $k=1$), the algorithm inserts element at index k into its correct position with respect to the elements up to and including position $k-1$, which are already in their correct order.

How does the algorithm put the element at index k (`list[k]`) into its correct position with respect to elements at indices 0 to $k-1$? The idea is to search backwards from index k until it finds the right place for this element. As it searches back, it shifts forward by 1 position any element that is bigger than element k. Once an element is found that less than or equal to `list[k]`, it inserts the value `list[k]` directly after that element. Here is the algorithm.

```
for k = 1 to N - 1 {       // don't need to consider k = 0
   elementK =  list[k]   //  copy current element so it doesnt get erased
   i = k
   while (i > 0) and ( elementK  < list[i - 1] ){
       list[i] =  list[i - 1]   // shift up larger elements
       i  = i -1
   }
   list[i] = elementK  // insert into correct position
}                         // (has no effect if elementK was in correct position)
```

What is the time complexity of insertion sort? As in the first two algorithms, we have nested for loops. The outer goes from k = 1 to N-1, and the inner goes from i = k down to 1 in the worst case. So the number of passes through the inner loop is:

$$1 + 2 + \ldots N - 1 = \frac{N(N-1)}{2}$$

which again is $O(N^2)$.

Note that the inner loop only continues as long as the while condition is met. If `elementK` is already greater than `list[k-1]` then the inner loop ends immediately because element **`k`** is already in the correct position.

What is the best and worst case scenario? In the best case, the array is already sorted from smallest to largest. Then the condition tested in the **while** loop will be false every time, and so the inner loop will take constant time. Since there are N passes through the **for** loop, the time taken is proportional to N in the best case.

Summary

In trying to understand these algorithms, it is best to start at a high level descrption and work ones way eventually to the code. Don't start with the code. It may also help to look at various website that give visualizations of different sorting algorithms. For example:
`https://www.toptal.com/developers/sorting-algorithms`.