After two passes, the second largest element will be in the second to last position as well. After n-1 passes, the entire list will be in order.

Thus the state of the list during the first pass will be (**bold** elements are about to be compared):

{**8,3**,2,1,4}, {3,**8,2**,1,4}, {3,2,**8,1**,4}, {3,2,1,**8,4**}, {3,2,1,4,8}

Notice that every comparison results in swapping the 8 to the right (since it is the largest item in the list and started in the first position. The states of the list during next pass will be the following:

{**3,2**,1,4,8}, {2,**3,1**,4,8}, {2,1,**3,4**,8}, {2,1,3,**4,8**},{2,1,3,4,8}

On this pass there are only two swaps as the 3 moves right twice. On the third pass there is even less movement, only the first two elements are swapped:

{**2,1**,3,4,8}, {1,**2,3**,4,8}, {1,2,**3,4**,8}, {1,2,3,**4,8**},{1,2,3,4,8}

Notice that the list is in order after only 3 passes; in general n-1 passes are required, since if the smallest element is in the last position, it can only move one position left on each pass.

No human would ever sort like this (one would hope!), but the machine does the mindless repetition, well, mindlessly, and this algorithm is very easy to code.

C. Algorithm/Pseudocode

The next step, after understanding the mechanics of an algorithm is to write pseudocode describing the operation of the algorithm to carry out that technique. Notice that for this level of description the representation of the list is not specified; it might be an array, a Vector, or some other list representation. To understand this pseudocode, you should get a pencil and paper and draw the states of the lists and keep track of the value of the indices as the loops repeat. Just glancing at the pseudocode is not likely to work; if that's what you're going to do, it might be time to put this down and do something constructive.

i) Insertion sort

```
create an empty list, called sorted
for each element of unsorted
    find where it goes in sorted*
```

```
    insert it there*
```

The *s indicate that this step requires additional specification. These subalgorithms are candidates for methods when the code is written.

```
:*: find where an element, insertMe, goes in sorted
for each element in sorted (call it current element)
    if insertMe < current element
        return location of current element
return one past the end of the list (since current >= all of them)

:*: insert an element at location i
shift the elements from i down, down by 1
store the element at i
```

ii) Selection sort

```
create an empty list, called sorted
iterate n times (with index, i, moving from first to last in unsorted
   find the location of the smallest item remaining in unsorted*
   remove it from unsorted and add it to sorted
```

Finding the minimum element is a list is something that must be done time and again. There are various ways to accomplish this. Here, we need to know where the minimum element is, as opposed to just what it is; otherwise, deleting it from the list will require finding it again. Thus, this algorithm keeps track of the index of the minimum value. Note that minIndex is initially set to the first location.

```
:*: find the smallest item remaining in unsorted
set minIndex to 0
for look=1 to last location of unsorted
    if elementAt(i) < elementAt(minIndex)
        minIndex = i
```

iii) Bubble sort

```
iterate n times
    do one pass*

:*: do one pass
for each element in the list (except the last)
    if it is > the next element
        exchange them
```