CS125 : Introduction to Computer Science

Lecture Notes #27
Sorting

Lecture 27 : Sorting

As you might expect, *sorting* a collection of values means to arrange the elements in a specific order – usually in increasing order (though not always). More generally, there could be duplicate values, so in that case we'd say that we are (usually) arranging values non-decreasing order, i.e. if our values are: x_1, x_2, x_3, ..., x_n, then we want to arrange them in some specific order x_{i_1}, x_{i_2}, x_{i_3}, ..., x_{i_n}, so that:

$$x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq, ..., \leq x_{i_n}$$

Sorting is one of the most common operations done on data, and as a result, it is a problem that has been exhaustively studied. We will be studying a number of sorting algorithms this semester and talking about their advantages and disadvantages. Right now, though, we are merely going to look at a few sorting algorithms, not perform any serious analysis of them.
First, we should turn our attention to swapping; if we are trying to rearrange values inside an array so that they are in order, then we will probably often need to exchange the position of two values in the array, from being in the wrong order to being in the correct order.

To swap the values at two indices, from a different method, we will pass in the array itself, and the two indices. Our `swap(...)` method will then read the two values at those indices and write each of them into the other cell.

```
public static void swap(int[] arr, int i, int j)
```

It is not enough to say

```
public static void swap(int[] arr, int i, int j)
{
   arr[i] = arr[j];
   arr[j] = arr[i];
}
```

because in the first line, we erase the value in `arr[i]` by overwriting it with the value in `arr[j]`. So, the end result of the above code will be that the value which is stored in `arr[j]` to start with gets written into `arr[i]` and then copied back into `arr[j]` as well. And we have lost the original `arr[i]` for good.

So, instead, we need to set up a temporary variable to store `arr[i]` so that we don't lose it. Then, we can store `arr[j]` in `arr[i]`, and finally we can write the value in the temporary variable into `arr[j]`. At this point, our swap is complete.

```
public static void swap(int[] arr, int i, int j)
{
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}
```

Given this `swap(...)` method, and the `findMinimum(...)` method from the last lecture, we can now design an algorithm known as *selection sort*. What does selection sort do? Well, one way you might sort things in real life is to find the smallest value in your collection and set it aside. Then, you are left with a pile of the remaining values – all of which come after that smallest value that you set aside – and you would need to sort that collection. Once you sort the remaining values, you just place the sorted collection *after* that smallest value which you had set aside, and you are done.

We already know how to find the smallest value in an array – that is what our `findMinimum(...)` method will do. The way we can "set the smallest value aside" once we've found it, is to swap it to the beginning of the array. For example, if this was our array:

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

then `findMinimum(arr, 0, 9)` would return the value `3`, since that is where the minimum (`6`) is located. Now, we can pull `6` out of the collection by moving it to the front of all the other remaining values, and the easiest way to do that is to just swap it with `93`, the value currently at the front of the collection:

```
arr[i]    6   11  71  93  39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

It's not a problem to move `93` to the middle of the array, since neither `93`, nor the values in the other cells indexed `1` through `9`, have been sorted yet anyway. We could also have figured out some way to *shift* the values between the beginning of the array, and `6`, to the right, to make room for `6`:

```
arr[i]    6   93  11  71  39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

but that can potentially take much longer (more array writes have to be done in that case), and for no benefit – once we have moved `6` to the front of the array, we don't care how the values in cells indexed `1` through `9` are arranged – since we don't know for sure they are sorted, we're going to sort them anyway. Which unsorted orderings they end up in along the way doesn't make any difference.

So, since shifting doesn't give us any benefit in this case, and will take longer, we prefer to swap. And that gives us the following basic outline for selection sort:

```
public static void selectionSort(int[] arr, int lo, int hi)
{
   if (recursive case)
   {
      // (1) Find index of minimum value of subarray
      int minLocation = findMinimum(arr, lo, hi);

      // (2) Swap minimum value into first cell of
      //   subarray (sometimes lo will equal minLocation;
      //   in that case, this is then a useless swap, but
      //   also a harmless one
      swap(arr, lo, minLocation);

      // (3) recursively sort the remainder of the array
      selectionSort(arr, lo + 1, hi);
   }
   else  // base case
      // what do we do here?
}
```

Now, as far as the base case goes, well, once we have only one value left, there's nothing to do. Any subarray of only one cell is sorted, by definition, since there's no other values with which our one value can be mis-ordered. And similarly, an empty subarray is also sorted, by definition. That means the base case would simply be an empty statement, and so we don't even need to bother coding that. That gives us our final selection sort code:

```
public static void selectionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)  // subarray is at least size 2
   {
      int minLocation = findMinimum(arr, lo, hi);
      swap(arr, lo, minLocation);
      selectionSort(arr, lo + 1, hi);
   }
}
```

We could also write a `selectionSort` wrapper method that will take only the array as an argument:

```
public static void selectionSort(int[] arr)
{
   selectionSort(arr, 0, arr.length - 1);
}
```

Here is an example run of the algorithm, on a sample array of size `10`. It is assumed we have made the call `selectionSort(arr, 0, 9);` that has started off the following example.

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #1, where we are
sorting the subarray with indices 0 through 9.
```

We are attempting to sort the subarray with indices 0 through 9. The minimum value over those indices is 6, which is located in cell 3. So, swap the value in cell 3 with the value at our low index – namely, index 0 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 1 through 9, and we will be done.

```
arr[i]    6   11  71  93  39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #2, where we are
sorting the subarray with indices 1 through 9.
```

We are attempting to sort the subarray with indices 1 through 9. The minimum value over those indices is 11, which is located in cell 1. So, swap the value in cell 1 with the value at our low index – namely, index 1 – so that the minimum is placed in the first cell in our subarray. (Note that this doesn't really do anything, since the minimum has stayed in the same cell. But in general, the minimum will be somewhere other than the first cell in our subarray, so we would have that swap in our code, and in the cases where the miniumum happens to already be in the first cell of our subarray, the swap is useless and thus wastes a bit of time, but does no harm.) Now, all that remains is to recursively sort the values in the subarray with indices 2 through 9, and we will be done.

```
arr[i]    6   11  71  93  39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #3, where we are
sorting the subarray with indices 2 through 9.
```

We are attempting to sort the subarray with indices 2 through 9. The minimum value over those indices is 23, which is located in cell 8. So, swap the value in cell 8 with the value at our low index – namely, index 2 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 3 through 9, and we will be done.

```
arr[i]    6   11  23  93  39  67  41  88  71  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #4, where we are
sorting the subarray with indices 3 through 9.
```

We are attempting to sort the subarray with indices 3 through 9. The minimum value over those indices is 39, which is located in cell 4. So, swap the value in cell 4 with the value at our low index – namely, index 3 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 4 through 9, and we will be done.

```
arr[i]    6   11  23  39  93  67  41  88  71  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #5, where we are
sorting the subarray with indices 4 through 9.
```

We are attempting to sort the subarray with indices 4 through 9. The minimum value over those indices is 41, which is located in cell 6. So, swap the value in cell 6 with the value at our low index – namely, index 4 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 5 through 9, and we will be done.

```
arr[i]    6   11  23  39  41  67  93  88  71  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #6, where we are
sorting the subarray with indices 5 through 9.
```

We are attempting to sort the subarray with indices 5 through 9. The minimum value over those indices is 54, which is located in cell 9. So, swap the value in cell 9 with the value at our low index – namely, index 5 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 6 through 9, and we will be done.

```
arr[i]    6   11  23  39  41  54  93  88  71  67
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #7, where we are
sorting the subarray with indices 6 through 9.
```

We are attempting to sort the subarray with indices 6 through 9. The minimum value over those indices is 67, which is located in cell 9. So, swap the value in cell 9 with the value at our low index – namely, index 6 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 7 through 9, and we will be done.

```
arr[i]    6   11  23  39  41  54  67  88  71  93
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #8, where we are
sorting the subarray with indices 7 through 9.
```

We are attempting to sort the subarray with indices 7 through 9. The minimum value over those indices is 71, which is located in cell 8. So, swap the value in cell 8 with the value at our low index – namely, index 7 – so that the minimum is placed in the first cell in our subarray. Now, all that remains is to recursively sort the values in the subarray with indices 8 through 9, and we will be done.

```
arr[i]    6   11  23  39  41  54  67  71  88  93
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #9, where we are
sorting the subarray with indices 8 through 9.
```

We are attempting to sort the subarray with indices 8 through 9. The minimum value over those indices is 88, which is located in cell 8. So, swap the value in cell 8 with the value at our low index – namely, index 8 – so that the minimum is placed in the first cell in our subarray. (Again, this is a swap that happens to be useless, but harmless.) Now, all that remains is to recursively sort the values in the subarray with indices 9 through 9, and we will be done.

```
arr[i]    6   11  23  39  41  54  67  71  88  93
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Start of selectionSort recursive call #10, where we are
sorting the subarray with indices 9 through 9.
```

We are attempting to sort the subarray with indices 9 through 9. In this case, our array is down to only one element, since the bounding indices are identical. Therefore, there is nothing to sort – certainly if you have only one element, it is in the "proper order" since there is nothing to be out of order with – and so we could simply return. In our selection sort code, the `lo < hi` condition would be false in this case.

Note that right before we return, we actually have 10 method calls to `selectionSort` active. The method `main()` (or whatever) called `selectionSort` the first time, which called `selectionSort` a second time, which called `selectionSort` a third time, and so on. We've never returned from any of those `selectionSort` calls. But now that we've completed our base case, we will start to return from those calls. There is never any more work to do when we return to a previous call – the recursive call is the last work each call does – and so basically at this point call #10 just returns to call #9, call #9 immediately returns to call #8, call #8 immediately returns to call #7, and so on, until finally call #2 returns to call #1, and at last, call #1 returns back to `main()` (or returns back to whatever called it, anyway). The array was completely sorted at the end of the base case call (call #10); the rest was just ending the methods we had started, one by one, so that we could return back to whatever method first called `selectionSort(...)` to begin with.

```
arr[i]    6   11  23  39  41  54  67  71  88  93
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

Final sorted array (same as array right before
returning from selectionSort call #10)

Next, let us consider what is involved in inserting a new value into an already sorted array, in sorted order. We can assume we already have `k` sorted values, indexed from `lo` through `lo + k - 1`. For example, if `lo` were `0`, and `k` were `12`, then we have `12` sorted values indexed from `0` through `11 (0 + 12 - 1 == 11)`. If we want to write a new value into that array, we need to make space for it – and thus we'd need to copy everything into an array whose length was one cell larger than our current array:

```
// assume arr is a reference to our current array

// create a new array, temp, whose length is one greater
int[] temp = new int[arr.length + 1];

// copy values from arr into temp
for (int i = 0; i < arr.length; i++)
   temp[i] = arr[i];

// now, the reference arr should point to our new array,
// meaning that nothing will point to the old array anymore,
// so the system collects it and eliminates it. Fortunately,
// we have copied all the values to our new array, so we don't
// lose any of our values.
arr = temp;
```

That code doesn't play a role in what we are about to discuss; we are just reminding you how you could "add an array cell to the end of your array". When we are done, we have `k` sorted values, stored in indices `0` through `k - 1` of an array of size `k + 1`. This leaves the last cell – the new cell, the cell at index `k` – empty and available for a new value.

For example, if this had been our initial array:

```
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93
  ------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11
```

then the above code will expand the size of the array by one (by copying the values – in their current, sorted order – into a new, larger array):

```
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
```

Once we have everything copied to the larger array, we then have room for our new value! That's where the fun begins – let's copy our new value into the last cell of the array – the extra cell we just created. For example, if we had wanted to insert `70` into our above array of size `12`, we first increase the size `12` array to a size `13` array, and then we write `70` into that new, last cell:

```
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  70
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
```

That might not be where the new value belongs! Certainly, above, if we want the entire array to be sorted, `70` belongs between `67` and `71`, not after `93`. But we will put the new value at the end just for a moment; since right now, that is the new, empty, available cell.

And that gives us the following situation (in general, our subarray would be indexed from `lo` through `hi`, not `0` through `12`):

```
arr[i]    |<------- this entire section is sorted ----->| newValue
  ------------------------------------------------------------
    i     lo  lo+1 lo+2  . . .                 . . . hi-1 hi
```

That is the setup we want to discuss – a sorted collection of values, with one new value at the end that is not necessarily in the correct order. And, as we described further above, if we want to insert a value into a sorted array, we can reach this setup point, by expanding the size of the sorted array by one cell, and then putting the new value in that new cell at the end of the array.

So, now we have this situation from above:

```
arr[i]    |<------- this entire section is sorted ----->| newValue
  ------------------------------------------------------------
    i     lo  lo+1 lo+2  . . .                 . . . hi-1 hi
```

and we would now like to move `newValue` to the correct spot in the array. In the case of our concrete example:

```
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  70
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
```

we would like to move `70` to its proper place between `67` and `71`. So, imagine smaller versions of this problem – i.e. imagine the same setup, just on smaller arrays:

```
// for example (hi in this picture is smaller than hi in the
// above picture)
arr[i]    |<- this entire section is sorted ->| newValue
  ------------------------------------------------
    i     lo  lo+1 lo+2  . . .       . . . hi-1 hi
```

That would be the sort of problem a recursive call would solve – still inserting a new value on the right into a sorted section on the left...only the sorted section is smaller. So which subproblem would help? Well, what if the subarray you send to the recursive call is only one cell smaller?

```
// current picture:
arr[i]    |<------- this entire section is sorted ----->| newValue
  ------------------------------------------------------------
    i     lo  lo+1 lo+2  . . .             hi-3 hi-2 hi-1 hi


// subarray one smaller in size, that we send to recursive call:
arr[i]    |<---- this entire section is sorted --->| newValue
  ----------------------------------------------------------
    i     lo  lo+1 lo+2  . . .             hi-3 hi-2 hi-1
```

If we had some way of converting the top picture into the bottom one, we have our basic recursive step. If in the top picture, `newValue` is at cell `hi`, then if we can move it to cell `hi-1`, while still keeping everything from `lo` through `hi-2` sorted, then the cells from `lo` to `hi-1` now form our subarray that we can send to the recursive call.

For example, if this were our original problem:

```
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  70
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
```

Then what we are looking for, is some way of getting this situation:

```
arr[i]    |<---- this entire section is sorted ---->| 70  ???
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
         |_____________________________________________|
           send this to recursive call
```

How could we obtain that situation with minimal work? And what about the value to the right of `70`? What should it be?

Well, in our example, `93` is greater than `70`, so it should be to the right of `70` anyway. So why not swap them?

For example, if this were our original problem:

```
// original problem...
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  70
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12


// ...but 93 > 70, so swap the two values...
arr[i]    6   11  17  23  39  41  47  54  67  71  88  70  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12

// ...and now we have subarray for recursive call!
arr[i]    6   11  17  23  39  41  47  54  67  71  88  70  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
         |_____________________________________________|
```

Since our algorithm is supposed to "insert new value at far right into sorted section to the left", then if we trust that it works, we could now run that algorithm recursively, and "insert new value at index `11` into sorted section from indices `0` through `10` inclusive". We solved the original problem, "insert new value at index `12` into sorted section from indices `0` through `11` inclusive", by swapping the values at indices `12` and `11`, leaving us with the problem, "insert new value at index `11` into sorted section from indices `0` through `10` inclusive", which we are claiming we can solve recursively. Our overall problem is to insert `70` into the sorted section of values from `6` through `93`; once we

swap 70 and 93, we can make a recursive call to that would insert 70 into the sorted section of values from 6 through 88.

Now, that would not work in all cases. What if instead of our new value being 70, it is 95?:

```
// original problem...
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  95
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12


// ...but 93 > 70, so swap the two values...
arr[i]    6   11  17  23  39  41  47  54  67  71  88  95  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12

// ...and now we have subarray for recursive call!
arr[i]    6   11  17  23  39  41  47  54  67  71  88  95  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
         |_____________________________________________|
```

In this case, we *could* insert 95 into the sorted section from 6 through 88, but even once that is done, we have 93 to the right and that is less than 95 – which we've now moved to its left – so there's at least part of the final array that will be out of order. So something's gone wrong here. Let's take a look at that original problem closely.

```
// original problem...
arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  95
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12
```

In this case, our new value, 95, is *greater than* the value to its left, 93. Remember that everything to the left of 95 is sorted – and thus 93 is the greatest value in the sorted range. If 95 is greater than *that*, it must also be greater than everything else in the sorted range...meaning 95 should be placed *after* the entire sorted range – exactly as it is right now!

Or in other words – if our new value is greater than everything in the sorted range, we can leave it right where it is. And it's easy to tell if the new value is greater than everything in the sorted range – just compare it to the value on its left, which is the greatest value in the sorted range. Above, since 95 is greater than 93, we know that 95 is also greater than everything that 93 is greater than.

This is one of our base cases, since once we establish that the new value is the greatest of the values we are dealing with, we know it is correctly placed (as it is at the end) and so we don't need to move any of the values anymore. So, so far, we have the following code:

```
// assume lo...hi-1 is a sorted range, and hi holds the new value
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if (arr[hi] >= arr[hi - 1])  // last value is greatest
      ;  // we don't need to do anything, as we said above
   else   // arr[hi] < arr[hi - 1]
   {
      swap(arr, hi - 1, hi);
      insertInOrder(arr, lo, hi - 1);  // the subproblem we discussed earlier
   }
}
```

Now, the above code assumes that we have two values to compare; we might not. If the sorted section to the left was completely empty, we can't compare `arr[hi]` and `arr[hi-1]` since `arr[hi-1]` would not exist!

```
// all we have here is arr[hi], not arr[hi-1]
arr[i]    newValue
  ------------------------------------------------
    i     lo == hi
```

If your original sorted array was empty, as it is above, then no matter *what* the new value is, placing it in cell `lo` is the correct way to go. You're inserting one new value into a subarray with one (empty) cell. It's the "trivial" case, where there's nothing else there yet and thus nothing that your new value could be out of order with. That's our other base case. So, let's add that to our code:

```
// assume lo...hi-1 is a sorted range, and hi holds the new value
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if (lo == hi)                     // subarray is of size 1
      ;                              //       we don't need to do anything
   else if (arr[hi] >= arr[hi - 1])  // last value is greatest
      ;                              //       we don't need to do anything
   else                              // lo < hi and arr[hi] < arr[hi - 1]
   {                                 //
      swap(arr, hi - 1, hi);         //   1) swap last two values
      insertInOrder(arr, lo, hi - 1);  // 2) run the subproblem
   }
}
```

Note that we are assuming the array is of at least size `1`. If our new value is in one of the cells, the array *has to* be at least size 1. So we don't need to check for `lo > hi`. Furthermore, note that we can eliminate the first two cases, since they don't actually do anything – all they have is empty statements. This gives us our final version of `insertInOrder(...)`:

```
// assume lo...hi-1 is a sorted range, and hi holds the new value
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if ((lo < hi) && (arr[hi] < arr[hi - 1]))
   {
      swap(arr, hi - 1, hi);           // 1) swap last two values
      insertInOrder(arr, lo, hi - 1);  // 2) run the subproblem
   }
   // else new value is already properly inserted; do nothing
}
```

If we ran this algorithm on our original example array, we'd need four `insertInOrder(...)` calls total, including the base case:

```
// original problem: lo == 0, hi == 12

arr[i]    6   11  17  23  39  41  47  54  67  71  88  93  70
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12

// 70 < 93, so swap, and recursive call has lo == 0, hi == 11

arr[i]    6   11  17  23  39  41  47  54  67  71  88  70  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12

// 70 < 88, so swap, and recursive call has lo == 0, hi == 10

arr[i]    6   11  17  23  39  41  47  54  67  71  70  88  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12

// 70 < 71, so swap, and recursive call has lo == 0, hi == 9

arr[i]    6   11  17  23  39  41  47  54  67  70  71  88  93
  ------------------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9   10  11  12

// 70 > 67, so we've hit our base case and we are done
```

We can use this algorithm as the basis for another sorting algorithm, called *insertion sort*. The insertion sort algorithm works in a slightly different manner than selection sort did. Both sorting algorithms build an increasingly-larger sorted array as the algorithm proceeds. The difference is that selection sort builds the array by *selecting* a specific value (the minimum) from the *unsorted* values, and then *sorting* the remainder of the *unsorted* values, whereas insertion sort will sort *most* of the values first, and then *insert* the last value into the sorted collection. That is, in selection sort, the recursive call is the last thing done; in insertion sort, it is the first thing done.

```
Description of insertion sort algorithm:

   If the array is of size 1, do nothing. Otherwise
   recursively sort all but the last cell of this subarray.
   Then, insert the value in the last cell, into the sorted
   collection to its left.
```

The code for this algorithm, basically just needs to check if the array size is greater than `1`, and if so, make two method calls – the first, a recursive call to `insertionSort(...)`, and the second, a call to the `insertInOrder(...)` method we wrote earlier:

```
public static void insertionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      insertionSort(A, lo, hi - 1);
      insertInOrder(A, lo, hi);  // everything from lo through hi-1 is
                                 //  sorted; arr[hi] is "new value"
   }
}
```

Here is an example, on an array of size `10` (the beginning of this gets a little repetitive, but I wanted to be complete so that everyone could see every detail if they wanted):

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Recursive call #1
Array as we start to sort elements at indices 0 through 9
```

Since the array is of size greater than 1, we decide to sort the values at indices 0 through 8 first. *Once this is done*, we can then insert the value `arr[9]` into its proper place among the sorted elements in cells 0 through 8. *But first we must recursively sort cells 0 through 8!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Recursive call #2
Array as we start to sort elements at indices 0 through 8
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 7 first. *Once this is done*, we can then insert the value `arr[8]` into its proper place among the sorted elements in cells 0 through 7. *But first we must recursively sort cells 0 through 7!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #3
Array as we start to sort elements at indices 0 through 7
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 6 first. *Once this is done*, we can then insert the value `arr[7]` into its proper place among the sorted elements in cells 0 through 6. *But first we must recursively sort cells 0 through 6!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #4
Array as we start to sort elements at indices 0 through 6
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 5 first. *Once this is done*, we can then insert the value `arr[6]` into its proper place among the sorted elements in cells 0 through 5. *But first we must recursively sort cells 0 through 5!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #5
Array as we start to sort elements at indices 0 through 5
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 4 first. *Once this is done*, we can then insert the value `arr[5]` into its proper place among the sorted elements in cells 0 through 4. *But first we must recursively sort cells 0 through 4!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #6
Array as we start to sort elements at indices 0 through 4
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 3 first. *Once this is done*, we can then insert the value `arr[4]` into its proper place among the sorted elements in cells 0 through 3. *But first we must recursively sort cells 0 through 3!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #7
Array as we start to sort elements at indices 0 through 3
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 2 first. *Once this is done*, we can then insert the value `arr[3]` into its proper place among the sorted elements in cells 0 through 2. *But first we must recursively sort cells 0 through 2!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #8
Array as we start to sort elements at indices 0 through 2
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 1 first. *Once this is done*, we can then insert the value `arr[2]` into its proper place among the sorted elements in cells 0 through 1. *But first we must recursively sort cells 0 through 1!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #9
Array as we start to sort elements at indices 0 through 1
```

Since the subarray is of size greater than 1, we decide to sort the values at indices 0 through 0 first. *Once this is done*, we can then insert the value `arr[1]` into its proper place among the sorted elements in cells 0 through 0. *But first we must recursively sort cells 0 through 0!!*

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Recursive call #10
Array as we start to sort elements at indices 0 through 0
```

Now that the array is down to size 1, that is our base case. Note that right now we have *ten* different method calls active at once! From `main()`, we called `insertionSort`, and from there we called `insertionSort` a second time, and from there we called `insertionSort` a third time, and so on. We've never returned from any of the `insertionSort` calls, because the first instruction in each call was to call `insertionSort` again on a smaller array. But *now* we will start to return to them one by one and do more work. For example, we are in recursive call #10 right now, but in a moment we will finish call #10, and return to call #9 to finish it up. Once we finish call #9 up, we will then return to call #8 and finish *it* up. And so on – one by one we will return to the previous

recursive call and finish it, until finally we are returning from the very first `insertionSort` call back to `main()`, and at that point the array will be sorted and we will be finished.

This recursive call #10 that we are in now is the case where making a further recursive call would either be pointless or would not make any sense. The subarray with indices 0 through 0 is already sorted – it is just one element and that element is of course in its proper order, since there is nothing to be out of order with. So, we are done with this method call (in the code, the `lo < hi` condition would be false here, since `lo == hi`) and we return from recursive call #10, back to recursive call #9.

```
arr[i]    93  11  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array after returning to recursive call #9.
```

At this point, we have finished recursive call #10, which sorted the elements at indices 0 through 0, and we are going to insert the value at index 1 into the sorted subarray consisting of the elements at indices 0 through 0. This results progressively swapping 11 with the element to its left until either 11 is less than the element to its left, or there are no more elements to 11's left. Since `arr[0]` is 93, we swap 11 with 93 and then 11 is at the far left so we stop.

```
arr[i]    11  93  71  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #9.
Thus, also array right after returning to recursive call #8.
```

And now that we have finished recursive call #9, note that we have sorted the elements in cells 0 through 1, as was the point of this method call. Now, when we return from call #9 back to call #8, the subarray consisting of cells 0 through 1 is sorted, and we need to insert the value in cell 2 into that sorted subarray in sorted order. Again, that consists of swapping our value – 71 in this case – with the value to its left until either the element to its left is less than 71, or there are no elements to the left. Here, we swap 71 with 93, but note that 11 is less than 71, and so we *don't* swap those two. And thus we are done with recursive call #8.

```
arr[i]    11  71  93  6   39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #8.
Thus, also array right after returning to recursive call #7.
```

And now that we have finished recursive call #8, note that we have sorted the elements in cells 0 through 2, as was the point of this method call. Now, when we return from call #8 back to call #7, the subarray consisting of cells 0 through 2 is sorted, and we need to insert the value in cell 3 into that sorted subarray in sorted order. Again, that consists of swapping our value – 6 in this case – with the value to its left until either the element to its left is less than 6, or there are no elements to the left. Here, we swap 6 with 93, and then 71, and then 11, and then realize there are no more elements to the left of 6, and so we stop. And thus we are done with recursive call #7.

```
arr[i]    6   11  71  93  39  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #7.
Thus, also array right after returning to recursive call #6.
```

And now that we have finished recursive call #7, note that we have sorted the elements in cells 0 through 3, as was the point of this method call. Now, when we return from call #7 back to call #6, the subarray consisting of cells 0 through 3 is sorted, and we need to insert the value in cell 4 into that sorted subarray in sorted order. Again, that consists of swapping our value – 39 in this case – with the value to its left until either the element to its left is less than 39, or there are no elements to the left. Here, we swap 39 with 93, and then 71, and then realize that 11 is less than 39 and so we stop. And thus we are done with recursive call #6.

```
arr[i]    6   11  39  71  93  67  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #6.
Thus, also array right after returning to recursive call #5.
```

And now that we have finished recursive call #6, note that we have sorted the elements in cells 0 through 4, as was the point of this method call. Now, when we return from call #6 back to call #5, the subarray consisting of cells 0 through 4 is sorted, and we need to insert the value in cell 5 into that sorted subarray in sorted order. Again, that consists of swapping our value – 67 in this case – with the value to its left until either the element to its left is less than 67, or there are no elements to the left. Here, we swap 67 with 93, and then 71, and then realize that 39 is less than 67 and so we stop. And thus we are done with recursive call #5.

```
arr[i]    6   11  39  67  71  93  41  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #5.
Thus, also array right after returning to recursive call #4.
```

And now that we have finished recursive call #5, note that we have sorted the elements in cells 0 through 5, as was the point of this method call. Now, when we return from call #5 back to call #4, the subarray consisting of cells 0 through 5 is sorted, and we need to insert the value in cell 6 into that sorted subarray in sorted order. Again, that consists of swapping our value – 41 in this case – with the value to its left until either the element to its left is less than 41, or there are no elements to the left. Here, we swap 41 with 93, and then 71, and then 67, and then realize that 39 is less than 41 and so we stop. And thus we are done with recursive call #4.

```
arr[i]    6   11  39  41  67  71  93  88  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #4.
Thus, also array right after returning to recursive call #3.
```

And now that we have finished recursive call #4, note that we have sorted the elements in cells 0 through 6, as was the point of this method call. Now, when we return from call #4 back to call #3, the subarray consisting of cells 0 through 6 is sorted, and we need to insert the value in cell 7 into that sorted subarray in sorted order. Again, that consists of swapping our value – 88 in this case – with the value to its left until either the element to its left is less than 88, or there are no elements to the left. Here, we swap 88 with 93, and then realize that 71 is less than 88 and so we stop. And thus we are done with recursive call #3.

```
arr[i]    6   11  39  41  67  71  88  93  23  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #3.
Thus, also array right after returning to recursive call #2.
```

And now that we have finished recursive call #3, note that we have sorted the elements in cells 0 through 7, as was the point of this method call. Now, when we return from call #3 back to call #2, the subarray consisting of cells 0 through 7 is sorted, and we need to insert the value in cell 8 into that sorted subarray in sorted order. Again, that consists of swapping our value – 23 in this case – with the value to its left until either the element to its left is less than 23, or there are no elements to the left. Here, we swap 23 with 93, and then 88, and then 71, and then 67, and then 41, and then 39, and then finally we realize that 11 is less than 23 and so we stop. And thus we are done with recursive call #2.

```
arr[i]    6   11  23  39  41  67  71  88  93  54
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9

Array right before returning from recursive call #2.
Thus, also array right after returning to recursive call #1.
```

And now that we have finished recursive call #2, note that we have sorted the elements in cells 0 through 8, as was the point of this method call. Now, when we return from call #2 back to call #1, the subarray consisting of cells 0 through 8 is sorted, and we need to insert the value in cell 9 into that sorted subarray in sorted order. Again, that consists of swapping our value – 54 in this case – with the value to its left until either the element to its left is less than 54, or there are no elements to the left. Here, we swap 54 with 93, and then 88, and then 71, and then 67, and then realize that 41 is less than 54 and so we stop. And thus we are done with recursive call #1.

```
arr[i]    6   11  23  39  41  54  67  71  88  93
  ------------------------------------------------
    i     0   1   2   3   4   5   6   7   8   9
```

```
Array right before returning from recursive call #1.
```

At this point, the array is sorted, and we return from recursive call #1 back to `main()`.

And that's insertion sort! The algorithm is named "insertion sort" because it basically consists of running an "insert new value into a sorted collection in sorted order" procedure over and over again, for each new value we see. We have a collection of size `1`. Then we insert a new value into that collection, in sorted order, so that we have a sorted collection of size `2`. Then we insert a new value into that collection, in sorted order, so that we have a sorted collection of size `3`. Then we insert a new value into that collection, in sorted order, so that we have a sorted collection of size `4`. And so on. This is in contrast to "selection sort", where we kept "selecting" the smallest value, out of the unsorted collection of values we still had left.

It works in either direction!

While we have described the two sorting algorithms we have seen so far, in their "standard" direction, please note that if we had done things from the other direction, it's still effectively the same idea, and thus the same algorithm.

For example, for selection sort, we kept selecting the minimum of what was left and moving it to cell `lo`, and then sorting `lo+1...hi`. Instead, we could always select the maximum of what was left, and move it to cell `hi`, and then sort `lo...hi-1`. It's the same basic idea, just from the other direction:

```
public static int findMaximum(int[] arr, int lo, int hi)
{
   if (lo == hi)
      return lo;
   else
   {
      int locOfMaxOfRest = findMaximum(arr, lo + 1, hi);
      if (arr[lo] >= arr[locOfMaxOfRest])     // arr[lo] is max value
         return lo;
      else // arr[locOfMaxOfRest] > arr[lo] -->  recursive result is max value
         return locOfMaxOfRest;
   }
}

public static void selectionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      int maxLocation = findMaximum(arr, lo, hi);
      swap(arr, hi, maxLocation);
      selectionSort(arr, lo, hi - 1);
   }
}
```

However, the "official" version of selection sort, and the one we'll use for this course, is the one where you select the minimum each time.

Similarly, for insertion sort, rather than always insert a new value into the sorted range to its left, we could always insert a new value into the sorted range to its right. We start by modifying `insertInOrder(...)` to assume the "new value" is at the far left, and that you are inserting it into a sorted range to the right:

```
// inserts "new value" at arr[lo] into sorted range from
// arr[lo+1] through arr[hi]
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if (lo == hi)                     // if one cell, you are done
      ;
   else if (arr[lo] <= arr[lo + 1])  // if arr[lo] is lowest, leave it
      ;
   else   // lo < hi  and arr[lo] > arr[lo + 1]
   {
      swap(arr, lo + 1, lo);          // arr[lo + 1] is lowest, swap...
      insertInOrder(arr, lo + 1, hi); // ...then insert ‘‘new value’’ into
                                      // smaller sorted subarray to the right
   }
}
```

Of course, that code can have the base case code eliminated, just as with our earlier version of `insertInOrder(...)`, and then we just rework the `insertionSort(...)` algorithm in the same way:

```
// inserts "new value" at arr[lo] into sorted range from
// arr[lo+1] through arr[hi]
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if ((lo < hi) && (arr[lo] > arr[lo + 1]))
   {
      swap(arr, lo + 1, lo);          // arr[lo+1] is lowest, swap...
      insertInOrder(arr, lo + 1, hi); // ...then insert ‘‘new value’’ into
                                      // smaller sorted subarray to the right
   }
   // else, arr[lo] is where it belongs; do nothing
}


public static void insertionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      insertionSort(arr, lo + 1, hi);
      insertInOrder(arr, lo, hi);     // everything from lo+1 through hi is
                                      // sorted; arr[lo] is "new value"
   }
}
```

However, the “official” version of insertion sort, and the one we’ll use for this course, is the one we went through in detail earlier.