CS125 : Introduction to Computer Science

Lecture Notes #30
More Accumulator Recursion

Lecture 30 : More Accumulator Recursion

In this packet, we present some more examples of accumulator recursion. Our first example will be finding the minimum of a subarray.

```
// Implementation #1 of the findMinimum algorithm
// This is the algorithm in forward-recursion form
public static int findMinimum(int[] arr, int lo, int hi)
{
   if (lo == hi)
      return lo;
   else
   {
      int locOfMinOfRest = findMinimum(arr, lo + 1, hi);
      if (arr[lo] <= arr[locOfMinOfRest])
         return lo;
      else
         return locOfMinOfRest;
   }
}
```

Since it is the comparison we are doing after the recursive call, it is the comparison we want to move before the recursive call. That is, we want our recursive case to be set up like this:

```
public static int findMinimum(int[] arr, int lo, int hi)
{
   if (base case)
      // do something
   else
   {
      // compare arr[lo] to something

      int locOfMinOfRest = findMinimum(arr, lo + 1, hi);
   }
}
```

If we had a parameter that held our "index of the smallest value we've seen so far", we could compare to that, instead of the recursive result. (We don't know what this parameter would be initialized to, yet, but we'll talk about that in a moment.) Note that the following code does not correctly find the minimum! – we are now in the middle of modifying the code, and it won't be correct again until we are finished.

```
public static int findMinimum(int[] arr, int lo, int hi, int indexOfMinSoFar)
{
   if (base case)
      // do something
   else
   {
      if (arr[lo] <= arr[indexOfMinSoFar])
         // do something with lo
      else   // arr[lo] > arr[indexOfMinSoFar]
         // do something with indexOfMinSoFar
      int locOfMinOfRest = findMinimum(arr, lo + 1, hi);
   }
}
```

Now, we need to actually do something in this conditional we have written. What we can do, is send either `lo` or `indexOfMinSoFar` to the recursive call, as the argument for the new `indexOfMinSoFar` parameter we have added:

```
public static int findMinimum(int[] arr, int lo, int hi, int indexOfMinSoFar)
{
   if (base case)
      // do something
   else
   {
      int latestMinIndex;
      if (arr[lo] <= arr[indexOfMinSoFar])
         latestMinIndex = lo;
      else   // arr[lo] > arr[indexOfMinSoFar]
         latestMinIndex = indexOfMinSoFar;

      return findMinimum(arr, lo + 1, hi, latestMinIndex);
   }
}
```

We are performing the comparison, and then passing the index of the minimum value to the next recursive call. That call, in turn, will take that parameter index, and compare the value at that index – the minimum to far – to *that* recursive call's `arr[lo]`. The index of the smaller of those two values, will then be passed along to the next recursive call. And so on. At the very end of this procedure – i.e., when we reach the base case of the recursive method – we have only the last value left. That is, `lo == hi`. In our forward-recursive method, we simply returned this value, since if we had only one value, that value had to be the minimum. But now, when `lo == hi`, we have not just the value at `arr[lo]`, but also, the value at the parameter index `indexOfMinSoFar`. So, as our base case work, we should compare those two values – the overall minimum would be the minimum of those two values:

```
// Implementation #2 of the findMinimum algorithm
// This is the algorithm in accumulator-recursion form
public static int findMinimum(int[] arr, int lo, int hi, int indexOfMinSoFar)
{
   if (lo == hi)
   {
      if (arr[lo] <= arr[indexOfMinSoFar)
         return lo;
      else
         return indexOfMinSoFar;
   }
   else
   {
      int latestMinIndex;
      if (arr[lo] <= arr[indexOfMinSoFar])
         latestMinIndex = lo;
      else   // arr[lo] > arr[indexOfMinSoFar]
         latestMinIndex = indexOfMinSoFar;

      return findMinimum(arr, lo + 1, hi, latestMinIndex);
   }
}
```

To begin the recursion, it's necessary to have an initial value to send to `indexOfMinSoFar`. Presumably, our first comparison would be between `arr[lo]` and `arr[lo + 1]`, so let's send `arr[lo]` as the initial value, and actually have the first recursive call run from `lo + 1` through `hi`. That is, we could have a wrapper method as follows:

```
public static int findMinimum(int[] arr)  // assume arr.length >= 2
{
   return findMinimum(arr, 1, arr.length - 1, 0);
}
```

If we'd like the wrapper method to handle the `arr.length == 1` case as well, we could do this:

```
public static int findMinimum(int[] arr)  // assume arr.length >= 1
{
   if (arr.length == 1)
      return 0;     // only index that exists; must be the minimum
   else
      return findMinimum(arr, 1, arr.length - 1, 0);  // works for arr.length >= 2
}
```

Now, we can convert our accumulator-recursive version into a loop-based version:

```
public static int findMinimum(int[] arr, int lo, int hi, int indexOfMinSoFar)
{
   while (lo < hi)
   {
      int latestMinIndex;
      if (arr[lo] <= arr[indexOfMinSoFar])
         latestMinIndex = lo;
      else   // arr[lo] > arr[indexOfMinSoFar]
         latestMinIndex = indexOfMinSoFar;

      arr = arr;
      lo = lo + 1;
      hi = hi;
      indexOfMinSoFar = latestMinIndex;
   }
   if (arr[lo] <= arr[indexOfMinSoFar)
      return lo;
   else
      return indexOfMinSoFar;
}
```

Next, we can eliminate the `latestMinIndex` variable if we want:

```
public static int findMinimum(int[] arr, int lo, int hi, int indexOfMinSoFar)
{
   while (lo < hi)
   {
      if (arr[lo] <= arr[indexOfMinSoFar])
         indexOfMinSoFar = lo;
      else   // arr[lo] > arr[indexOfMinSoFar]
         indexOfMinSoFar = indexOfMinSoFar;

      arr = arr;
      lo = lo + 1;
      hi = hi;
      indexOfMinSoFar = indexOfMinSoFar;
   }
   if (arr[lo] <= arr[indexOfMinSoFar)
      return lo;
   else
      return indexOfMinSoFar;
}
```

and after removing the redundant assignments, we have:

```
// Implementation #3 of the findMinimum algorithm
// This is the algorithm in loop-based form
public static int findMinimum(int[] arr, int lo, int hi, int indexOfMinSoFar)
{
   while (lo < hi)
   {
      if (arr[lo] <= arr[indexOfMinSoFar])
         indexOfMinSoFar = lo;
      lo = lo + 1;
   }
   if (arr[lo] <= arr[indexOfMinSoFar)
      return lo;
   else
      return indexOfMinSoFar;
}
```

Finally, if we want, we can combine this with the wrapper method, by converting the last three parameters to local variables:

```
public static int findMinimum(int[] arr)
{
   int lo = 1;
   int hi = arr.length - 1;
   int indexOfMinSoFar = 0;
   while (lo < hi)
   {
      if (arr[lo] <= arr[indexOfMinSoFar])
         indexOfMinSoFar = lo;
      lo = lo + 1;
   }
   if (arr[lo] <= arr[indexOfMinSoFar)
      return lo;
   else
      return indexOfMinSoFar;
}
```

Next, we will do the same for `insertionSort`. Remember that we had two ways we could approach `insertionSort`; first, we could have the recursive call run on `lo...hi - 1`, in which case, `insertInOrder` assumes the sorted section is to the left:

```
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
   }
}


public static void insertionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      insertionSort(arr, lo, hi - 1);
      insertInOrder(arr, lo, hi);
   }
}
```

But we also had a version that runs the recursive call on `lo + 1...hi`, and for which `insertInOrder` assumes the sorted section is to the right:

```
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if ((lo < hi) && (arr[lo] > arr[lo + 1]))
   {
      swap(arr, lo + 1, lo);           // 1) swap first two values
      insertInOrder(arr, lo + 1, hi);  // 2) run the subproblem
   }
}


public static void insertionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      insertionSort(arr, lo + 1, hi);
      insertInOrder(arr, lo, hi);
   }
}
```

Both of these are "insertion sort", just from different directions. That said, of the two recursive versions, the first is the more common one. We will convert that into an accumulator-recursive version, and then into a loop.

We start by inserting `arr[hi]` somewhere other than into the sorted result of the recursive call – i.e. we move the `insertInOrder` idea from being after the `insertionSort` recursive call, to before the `insertionSort` recursive call:

```
public static void insertionSort(int[] arr, int lo, int hi)
{
   if (lo < hi)
   {
      // insert arr[hi] somewhere else, since now we haven't
      // sorted the range lo...hi-1 yet
      insertionSort(arr, lo, hi - 1);
   }
   else
      // base case; do something here
}
```

Where can we insert `arr[hi]`? Well, as with any accumulator recursion, we'd need an extra parameter of some kind. In this case, that parameter can be some other sorted collection. (Once again, it's not until our code is finished that it will actually work; our intermediate steps will just be pseudocode.)

```
public static void insertionSort(int[] arr, int lo, int hi, sortedCollection)
{
   if (lo < hi)
   {
      // insert arr[hi] into sortedCollection
      insertionSort(arr, lo, hi - 1, sortedCollection);
   }
   else
      // base case; do something here
}
```

When we reach just one value left (i.e. `lo == hi`), we will still have work to do. In the forward-recursive version, we had only one value, and that was already sorted, so we just returned. However, now we have that one value, plus the sorted collection that we sent along as an argument to our new last parameter. So, what we need to do is combine our final value, with the collection we have as a parameter. We could do this as follows:

- insert last value into sorted collection as well, creating a sorted collection of *all* the values we've been worried about throughout all the recursive calls
- replace the entire original subarray, with the sorted collection into which we just inserted, as the sorted collection is now the entire array that we want

So, we basically have this setup:

```
   // pseudocode of recursive method:
   public static void insertionSort(int[] arr, int lo, int hi, sortedCollection)
   {
      if (lo < hi)
      {
         // insert arr[hi] into sortedCollection
         insertionSort(arr, lo, hi - 1, sortedCollection);
      }
      else  // base case
      {
         // insert arr[hi] (which is the same as arr[lo]) into sorted collection
         // make sorted collection our actual answer, i.e. copy it into this
         //    array somehow
      }
   }



   // pseudocode of wrapper method:
   public static void insertionSort(int[] arr)
   {
      insertionSort(arr, 0, arr.length - 1, emptyCollection);
   }
```

Our initial value to the new parameter, will be an empty collection. Then, as we make our recursive calls, we will insert `arr[hi]` into that collection, and then `arr[hi - 1]`, and then `arr[hi - 2]`, and so on.

The last question to resolve is how to store this parameter collection and eventually copy it back to our actual array. It might seem that we need an external array to store this collection, but in fact, we can use the same array we are already manipulating. At any point in the algorithm, we have a range `lo...hi`, where `hi` is less than or equal to our original value of `hi`:

```
original subarray:


            <------------------------------------------>
            --------------------------------------------
            lo                                         hi


after many recursive calls:


            <---------------------->  ????????????????????
            --------------------------------------------
            lo                  current                 original
                                value of                value of
                                 hi                       hi
```

The area marked by question marks still exists! We've just decremented `hi` enough via our various recursive calls, that our current `insertionSort` call doesn't cover that area. But those cells are still there. So, why not use that area for our external collection? That is, we've already sent `arr` as a parameter; let's send our external collection simply by sending the index range of the "other" part of `arr`, the part that this recursive call isn't manipulating, but that the original recursive call *was* manipulating?

```
                        extraLo             extraHi
<-------------------->  ????????????????????
--------------------------------------------
lo                  current                 original
                    value of                value of
                     hi                       hi
```

Then, if we insert `arr[hi]` into this collection to the right, then the size of that collection grows by one, as the size of the subarray our recursive call is manipulating, shrinks by one:

```
                    extraLo                 extraHi
<---------------->  ????????????????????????
--------------------------------------------
lo              current                     original
                value of                    value of
                 hi                           hi
```

As our algorithm proceeds, `lo` and `extraHi` won't change, `hi` gets decremented once with each recursive call, and `extraLo` should also be decremented once with each recursive call, since it should always be the next cell over from `hi`.

That gives us our final accumulator-recursive version, which we see on the next page. The `insertInOrder` version that we use is the one that assumes the sorted range is on the right – since when we attempt to insert `arr[hi]` into the range indicated by question marks above, that range is to the right of `arr[hi]`.

```
// wrapper method
// the last two arguments are the lo and hi, respectively, of our
// extra collection -- which are stored in the parameters extraLo and
// extraHi in our recursive code. Since this extra collection should be
// empty to start with, we have extraLo > extraHi
public static void insertionSort(int[] arr)
{
   insertionSort(arr, 0, arr.length - 1, arr.length, arr.length - 1);
}


// this version of insertInOrder inserts the value at arr[lo]
//  into the sorted collection in lo+1...hi
public static void insertInOrder(int[] arr, int lo, int hi)
{
   if ((lo < hi) && (arr[lo] > arr[lo + 1]))
   {
      swap(arr, lo + 1, lo);           // 1) swap first two values
      insertInOrder(arr, lo + 1, hi);  // 2) run the subproblem
   }
}


// Implementation #2 of the insertionSort lgorithm
// This is the algorithm in accumulator-recursive form
// extraLo will always be one greater than hi; technically, we can
//   eliminate extraLo since it's not actually used anywhere
public static void insertionSort(int[] arr, int lo, int hi, int extraLo, int extraHi)
{
   if (lo < hi)
   {
      insertInOrder(arr, hi, extraHi);  // arr[hi] is inserted into the
                                        //  range hi+1...extraHi, i.e.
                                        //  into the range extraLo...extraHi
      insertionSort(arr, lo, hi - 1, extraLo - 1, extraHi);
   }
   else  // base case
   {
      insertInOrder(arr, hi, extraHi);
      // now the entire range lo...extraHi is sorted and that's the
      // original range our array was supposed to sort
   }
}
```

Converting this to a loop is then just a matter of using our usual technique:

```
public static void insertionSort(int[] arr, int lo, int hi, int extraLo, int extraHi)
{
   while (lo < hi)
   {
      insertInOrder(arr, hi, extraHi);
      arr = arr;
      lo = lo;
      hi = hi - 1;
      extraLo = extraLo - 1;
      extraHi = extraHi;
   }
   else  // base case
   {
      insertInOrder(arr, hi, extraHi);
      // now the entire range lo...extraHi is sorted and that's the
      // original range our array was supposed to sort
   }
}
```

and then eliminating the redundant assignments,

```
public static void insertionSort(int[] arr, int lo, int hi, int extraLo, int extraHi)
{
   while (lo < hi)
   {
      insertInOrder(arr, hi, extraHi);
      hi = hi - 1;
      extraLo = extraLo - 1;
   }
   else  // base case
   {
      insertInOrder(arr, hi, extraHi);
      // now the entire range lo...extraHi is sorted and that's the
      // original range our array was supposed to sort
   }
}
```

Note that `extraLo` isn't actually ever used anywhere, so we can get rid of that, too (we also could have gotten rid of it in the accumulator-recursive version):

```
// Implementation #3 of the insertionSort lgorithm
// This is the algorithm in loop-based form
public static void insertionSort(int[] arr, int lo, int hi, int extraHi)
{
   while (lo < hi)
   {
      insertInOrder(arr, hi, extraHi);
      hi = hi - 1;
   }
   else  // base case
   {
      insertInOrder(arr, hi, extraHi);
      // now the entire range lo...extraHi is sorted and that's the
      // original range our array was supposed to sort
   }
}
```

And you can see that the loop version, is basically just doing the same work that our "second" version of insertion sort did. The version of insertion sort where the recursive call was on the range `lo+1...hi` would, as the recursive calls returned, be sorting the right-hand side of the array. That's what our loop version does above.

And similarly, if you went through this process on that "second" version of insertion sort, you'd get a loop version that sorted the left-hand side of the array, just as our "first" version of insertion sort did on the return from the recursive call on the range `lo...hi-1`.

Or in other words, converting insertion sort from forward recursion to loop, reverses the direction of the sorting. The forward-recursive version that sorts left-to-right, becomes a loop version that sorts right-to-left, and the forward-recursive version that sorts right-to-left, becomes a loop version that sorts left-to-right. All four implementations are "insertion sort", though for both the forward-recursive and the loop-based versions, we tend to more commonly see the left-to-right code, than the right-to-left code.