CS125 : Introduction to Computer Science

Lecture Notes #28
Tail Recursion and Loop Conversion

Lecture 28 : Tail Recursion and Loop Conversion

What we will do today is discuss the two types of recursion. The first is called *tail recursion.* Tail recursion is when the recursive call is the *last* computation you do in the recursive case of your algorithm. The only things you can do after your recursive call returns, are return with nothing, or perhaps return with an already-calculated value.

Many of the recursive methods we have already seen, are tail-recursive. One example, is our method for printing out the cells of an array:

```
// prints all values of arr whose indices are in range lo...hi inclusive,
//  in order from lowest to highest index
public static void print(int[] arr, int lo, int hi)
{
   if (lo <= hi)
   {
      System.out.println(arr[lo]);
      print(arr, lo + 1, hi);
   }
   // else you do nothing
}
```

Above, once you call `print(...)` recursively, you never need to do anything in that particular recursive call again. That is, after the `print(...)` call returns, the method itself returns (the compound statement ends, and thus the `if`-statement ends, and thus you reach the closing curly brace of the method, and thus return from the method).

Another example of a tail-recursive method would be our `linearSearch(...)` method:

```
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   int returnVal;
   if (lo > hi)
      returnVal = -1;
   else if (arr[lo] == key)
      returnVal = lo;
   else
      returnVal = linearSearch(arr, key, lo+1, hi);

   return returnVal;
}
```

In the case of `linearSearch(...)`, you do not have a return value of `void`, like the `print(...)` method did. Instead, we are returning a value of type `int`. So, once the recursive call returns, we still need to execute an assignment statement (to write the return value into the variable `returnVal`) and then we need to execute a return statement (to return the variable `returnVal`). That is okay, however, since once the recursive call returns a value to us, all we do after that, is copy the value into a local variable, and then return it. We don't actually perform any additional computation, nor do we copy the value to places outside the scope of this method. We just copy the value from being the result of a recursive-method-call expression, into `returnVal`, and then we copy that value from `returnVal`, to being the return value of this method. So the above code is basically just a longer form of this code:

```
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if (arr[lo] == key)
      return lo;
   else
      return linearSearch(arr, key, lo+1, hi);
}
```

In that case, as soon as the recursive call returns, we immediately turn around and return that value, without having written it into any variables in-between. The fact that in the first version, we wrote the result of the recursive call into a variable before we returned it, doesn't change the fact that the code is tail-recursive, since the assignment statement could easily be eliminated, as we see in the second version above.

Note that even if we re-wrote `linearSearch(...)` so that the cases were in a different order, it would still be a tail-recursive method:

```
// an alternate way to code linearSearch
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   if (lo <= hi)
   {
      if (arr[lo] != key)
         return linearSearch(arr, key, lo + 1, hi);
      else
         return lo;
   }
   else    // lo > hi
      return -1;
}
```

In that case, once the recursive call returns, we would immediately run a return statement, and so even though the actual line of code that triggers the recursive call, is near the top of the method, you still are done with the method once that recursive call returns. Where the recursive call is placed on the page doesn't have any particular affect on whether a recursive method is tail-recursive or not; the issue is how much computation is performed once the recursive call is over. And, in both the `print(...)` and `linearSearch(...)` examples, no additional computation needs to be done once the recursive call returns.

The second kind of recursion is known as *forward recursion*. This is recursion where you *do* need to do computation work after you return from the recursive call. An example was the factorial method from earlier, where we needed to still do a multiplication once we returned from the recursive call:

```
public static int fac(int n)  // n >= 0
{
   if (n > 0)
      return n * fac(n - 1);
   else  // n == 0
      return 1;
}
```

Basically, if a recursive method is not tail-recursive, then it is forward-recursive, and vice-versa.

Our focus today will be on tail-recursive methods. They are of special interest to us, because the fact that they are tail-recursive makes them very easy to convert into loop-based methods! The reason for this is, nothing significant gets done once you start returning from the recursive calls, so if you could somehow "end the recursive process" right at the base case, you'd be fine.

For example, consider our `print(...)` method again. Let's assume `arr` points to an array of size `4`, and so we've sent `0` and `3` as the initial arguments to `lo` and `hi`. Here's how the method calls would look, as we made our way down to the base case:

```
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|
|________________________________________________________________
| print  (arr: [pts to array])
|  (#1)  (lo: 0)     (hi:  3)
|
|  prints arr[0], then calls print(arr, 1, 3);
|
|_________________________________________________________________
| print  (arr: [pts to array])
|  (#2)  (lo: 1)     (hi:  3)
|
|  prints arr[1], then calls print(arr, 2, 3);
|
|_________________________________________________________________
| print  (arr: [pts to array])
|  (#3)  (lo: 2)     (hi:  3)
|
|  prints arr[2], then calls print(arr, 3, 3);
|
|_________________________________________________________________
| print  (arr: [pts to array])
|  (#4)  (lo: 3)     (hi:  3)
|
|  prints arr[3], then calls print(arr, 4, 3);
|
|_________________________________________________________________
| print  (arr: [pts to array])
|  (#5)  (lo: 4)     (hi:  3)
|
|  lo > hi, so this is base case
|
|_________________________________________________________________
```

Now, all that is left, is to return from each of the method calls. There is *no* more work to do. In fact, if while we are in the base case, the earlier method notecards "mysteriously vanished":

```
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|
|________________________________________________________________
|
|
|                    \ | /
|                  - POOF -
|                    / | \
|
|_________________________________________________________________
| print  (arr: [pts to array])
|  (#5)  (lo: 4)     (hi:  3)
|
|  lo > hi, so this is base case
|
|_________________________________________________________________
```

it doesn't affect the running of our code at all; all the work is already done. We could just return from the base case back to `main()`, and be done.

Contrast that, with a call to `fac(4)`:

```
-----------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need to calculate: fac(3), in order to calculate 4 * fac(3)
|
|_________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need to calculate: fac(2), in order to calculate 3 * fac(2)
|
|_________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need to calculate: fac(1), in order to calculate 2 * fac(1)
|
|_________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  need to calculate: fac(0), in order to calculate 1 * fac(0)
|
|_________________________________________________________________
| fac     (n: 0)
|  (#5)
|
|  base case!  (we have not returned yet)
|_________________________________________________________________
```

In this case, we need to return to the previous method calls, in order to complete the multiplications. If while we are in the base case, the other method notecards "mysteriously vanish":

```
-----------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|________________________________________________________________
|
|
|                   \ | /
|                 - POOF -
|                   / | \
|
|_________________________________________________________________
| fac     (n: 0)
|  (#5)
|
|  base case!  (we have not returned yet)
|_________________________________________________________________
```

then we are left only with our statement `return 1;`, and cannot complete the rest of the work – since we do not have the other method notecards anymore, we can't complete the multiplications. If we return from the base case, directly back to `main()`, then `main()` would think that the value of `fac(4)` is `1`, and of course, that's not true.

That is the reason we care if a method is a tail-recursive method or a forward-recursive method. Because there is no work done in a tail-recursive method once the recursive call is completed, it means we effectively have finished all the work we need to do, once we finish the base case; in a forward-recursive method, we will still be doing work as we return from each of the other non-base-case method calls as well.

The reason this is relevant, is as follows: if we won't ever need the earlier method notecards again, then there is no reason to save them. That is, right now, when we start the `print(...)` method, we make the first `print(...)` call, and from there, make the second `print(...)` call:

```
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#1)  (lo: 0)     (hi:  3)
|
|  prints arr[0], then calls print(arr, 1, 3);
|_________________________________________________________________
| print  (arr: [pts to array])
|  (#2)  (lo: 1)     (hi:  3)
|
|  prints arr[1], then calls print(arr, 2, 3);
|_________________________________________________________________
```

But if we won't ever "need" the first `print(...)` notecard again, then why save it? The only reason it is hanging around, is so that we can eventually return to it, but once we *do* return to it, we will immediately leave it and return back to `main()`. So, why make an entirely new notecard for call #2, when we have this notecard sitting around for call #1 that we'll never really need to make detailed use of anymore? Why not just *convert* the first notecard into the second one, by increasing `lo` right there on the first notecard?

FROM:

```
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#1)  (lo: 0)     (hi:  3)
|
|  prints arr[0], then calls print(arr, 1, 3);
|_________________________________________________________________
```

TO:

```
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#2)  (lo: 1)     (hi:  3)
|
|  prints arr[1], then calls print(arr, 2, 3);
|_________________________________________________________________
```

And similarly, we can keep converting that notecard for each subsequent call. First, we make call #3 from call #2:

```
FROM:
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#2)  (lo: 1)     (hi:  3)
|
|  prints arr[1], then calls print(arr, 2, 3);
|_________________________________________________________________

TO:
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#3)  (lo: 2)     (hi:  3)
|
|  prints arr[2], then calls print(arr, 3, 3);
|_________________________________________________________________
```

and next we make call #4 from call #3:

```
FROM:
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#3)  (lo: 2)     (hi:  3)
|
|  prints arr[2], then calls print(arr, 3, 3);
|_________________________________________________________________

TO:
-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#4)  (lo: 3)     (hi:  3)
|
|  prints arr[3], then calls print(arr, 4, 3);
|_________________________________________________________________
```

and finally, we make call #5 from call #4:

```
FROM:

-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#4)  (lo: 3)     (hi:  3)
|
|  prints arr[3], then calls print(arr, 4, 3);
|_________________________________________________________________


TO:

-----------------------------------------------------------------
| main
|   print(arr, 0, 3);
|________________________________________________________________
| print  (arr: [pts to array])
|  (#5)  (lo: 4)     (hi:  3)
|
|  lo > hi, so we are done
|_________________________________________________________________
```

At each step, rather than make a *new* method notecard for a new method call, we simply convert the notecard we already have (representing the old call), into the notecard we need (representing the new call). This works because *we never need the old notecard again.* So, it's okay to write over all the data for the old notecard, with new data for the new notecard. If we did need the old data again (like in `fac(...)`, when each non-base-case call needs to take a returned value and perform a multiplication with it), then we can't do this. But for tail-recursive methods, we will never need the old notecard again, so "erasing" it by writing over its data with data for the new notecard, is fine.

In the last picture above, the `print(...)` call representing the base case of the method, could then return straight back to `main()`, and our work is done.

What we have just done here is "invent" the loop!

That is, the above is all a loop is: you are making a recursive call, but rather than start a *new* notecard underneath the old one, you simply *convert* the existing (old) notecard *into* the new one. When we actually set up each notecard, what a recursive call does it to calculate some new values for the parameters (the expressions that serve as the arguments are these new values), and then a *new* notecard, with *new* versions of the parameters, is created, and those argument values are copied into those new versions of the parameters:

```
call from previous method: print(arr, lo + 1, hi);
        .                        ---  ------  --
        .                         |      |     \
        .                         |      |      \
        .                        \|/    \|/    \|/
 public static void print(int[] arr, int lo, int hi)
 {
```

Instead, we are now going to copy those arguments into the *existing* versions of the parameters. That is, rather than copy `arr`, `lo + 1`, and `hi`, into new versions of the parameters `arr`, `lo`, and `hi` in the next call, instead copy them into the versions of those parameters in *this* call, via assignments statements:

```
this is the call we ``want to'' make: print(arr, lo + 1, hi);



                                        // param = expression;
                                           arr   = arr;
                                           lo    = lo + 1;
                                           hi    = hi;

                              (yes, we know assigning a variable
                               to itself, as we did with arr and
                               hi, is useless; it doesn't do any harm,
                               either, so we'll keep it there for now
                               and get rid of it a bit later)
```

and then, just as we would copy the arguments into new parameters, and then repeat the algorithm (by starting the method again on a new notecard), we will here, copy the arguments into the existing parameters,a and then repeat the algorithm, via a loop:

```
  loop
  {
      ...
      arr = arr;
      lo = lo + 1;
      hi = hi;
      // now we repeat everything in loop all over again, with these
      //  new values stored in the parameters
  }
```

That is, we are changing our recursive case from this:

```
   if (lo <= hi)
   {
      System.out.println(arr[lo]);

      print(arr, lo + 1, hi);  // copy arr, lo + 1, and hi, into new
                               // version of parameters, and repeat
   }
```

to this:

```
   while (lo <= hi)   // same condition
   {
      System.out.println(arr[lo]);

      // param = argument;   // copy arr, lo + 1, and hi, into current
      arr = arr;             // version of parameters, and repeat
      lo = lo + 1;
      hi = hi;
   }
```

That is the core idea behind converting a tail-recursive method into a loop-based method. We want to keep repeating the method code over and over and over, but rather than continually making new method calls to accomplish this, we stay in the method call we are already in, and just repeat the method code via a loop. The glue that holds the loop version together is that we still need to "copy the arguments into the parameters" – we just copy the arguments into the *current* parameters that are already a part of our *current* notecard, rather than creating a new notecard to hold the parameters.

If we have a recursive method organized as follows:

```
  recursiveMethod(params)
  {
     if (recursiveCase)
     {
        code we run before recursive call
        recursiveMethod(args);
     }
     else // base case
     {
        base case code
     }
  }
```

then the loop-based version would rearrange the code as follows:

```
   loopBasedMethod(params)
   {
      while (recursiveCase)
      {
         code we run before recursive call
         params = args;
      }
      base case code
   }
```

where the line "`params = args;`" is a stand-in for all the assignments you need to write each of the arguments to the recursive call, back into the current method call's parameters.
For example, consider the `print(...)` method again:

```
public static void print(int[] arr, int lo, int hi)
{
   if (lo <= hi)
   {
      System.out.println(arr[lo]);
      print(arr, lo + 1, hi);
   }
   // else you do nothing
}
```

First of all, the "base case code" is basically nothing (or, if you wish, an empty statement). So when we write our loop, we'll have nothing after the loop finishes. The condition that indicates that we should enter the recursive case is `lo <= hi`, so that is the same condition that indicates we enter our loop. And, the only code in the recursive case that is performed before the recursive call, is the printing out of `arr[lo]`. That gives us the following rough draft of loop-based code:

```
// rough draft #1
public static void print(int[] arr, int lo, int hi)
{
   while (lo <= hi)
   {
      System.out.println(arr[lo]);
      // params = args;
   }
   // in the base case, we did nothing
}
```

and for the "`params = args;`" section of the code, we are writing the recursive call's arguments (`arr`, `lo + 1`, and `hi`) into the parameters (`arr`, `lo`, and `hi`):

```
// rough draft #2
public static void print(int[] arr, int lo, int hi)
{
   while (lo <= hi)
   {
      System.out.println(arr[lo]);
      arr = arr;
      lo = lo + 1;
      hi = hi;
   }
   // in the base case, we did nothing
}
```

And finally, two of those assignments just assign a variable to itself, so we can get rid of those:

```
// final loop-based version
public static void print(int[] arr, int lo, int hi)
{
   while (lo <= hi)
   {
      System.out.println(arr[lo]);
      lo = lo + 1;
   }
   // in the base case, we did nothing
}
```

and now we have a loop-based version of `print(...)`.

As a final tweak, we can recall our wrapper method:

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

rather than call a second method (our loop-based method) from the wrapper method, we can simply merge the two methods together if we like. (If you still wanted the method with three parameters, then you would not want to do this.) Rather than pass `0` and `arr.length - 1` to parameters `lo` and `hi`, we could simply make `lo` and `hi` local variables, and initialize them to `0` and `arr.length - 1`, respectively.

```
public static void print(int[] arr)
{
   // these two lines are the new ones
   int lo = 0;
   int hi = arr.length - 1;

   while (lo <= hi)
   {
      System.out.println(arr[lo]);
      lo = lo + 1;
   }
}
```

Since we never change `hi`, we don't even need that variable to begin with – we can just substitute `arr.length - 1` for `hi` in the `while`-loop condition:

```
public static void print(int[] arr)
{
   int lo = 0;
   while (lo <= arr.length - 1)
   {
      System.out.println(arr[lo]);
      lo = lo + 1;
   }
}
```

and we are done! Of course, the above code might seem a bit more familiar if we replace the variable name `lo` with the variable name `i`:

```
public static void print(int[] arr)
{
   int i = 0;
   while (i <= arr.length - 1)
   {
      System.out.println(arr[i]);
      i = i + 1;
   }
}
```

That's pretty much the standard code for printing out an array from beginning to end, using a `while`-loop.

As a second example, consider `linearSearch(...)`:

```
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   if (lo > hi)
      return -1;
   else if (arr[lo] == key)
      return lo;
   else
      return linearSearch(arr, key, lo + 1, hi);
}
```

First of all, let's rewrite the method in the form:

```
  recursiveMethod(params)
  {
     if (recursiveCase)
     {
        code we run before recursive call
        recursiveMethod(args);
     }
     else // base case
     {
        base case code
     }
  }
```

that we mentioned earlier. We hit the recursive case whenever the first two cases are both `false` – that is, whenever `lo <= hi` and `arr[lo] != key`. So, that's our recursive condition:

```
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   if ((lo <= hi) && (arr[lo] != key))
      return linearSearch(arr, key, lo + 1, hi);
   else  // lo > hi  or   arr[lo] == key
   {
      if (lo > hi)
         return -1;
      else   // lo <= hi and arr[lo] == key
         return lo;
   }
}
```

Now applying our loop-version template:

```
loopBasedMethod(params)
{
   while (recursiveCase)
   {
      code we run before recursive call
      params = args;
   }
   base case code
}
```

gives us the following:

```
// rough draft #1
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   while ((lo <= hi) && (arr[lo] != key))
   {
      // the statement
      //       return linearSearch(arr, key, lo + 1, hi);
      // is converted into ‘‘params = args;’’ form:
      arr = arr;
      key = key;
      lo = lo + 1;
      hi = hi;
   }
   // code from base case appears below:
   if (lo > hi)
      return -1;
   else   // lo <= hi and arr[lo] == key
      return lo;
   }
}
```

and since `lo` is the only one of the four parameters that actually gets reassigned, we can eliminate the other three meaningless assignments:

```
// rough draft #2
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   while ((lo <= hi) && (arr[lo] != key))
   {
      lo = lo + 1;
   }
   // code from base case appears below:
   if (lo > hi)
      return -1;
   else   // lo <= hi and arr[lo] == key
      return lo;
   }
}
```

and thus, we can eliminate the compound statement curly braces as well, if we want:

```
// final loop-based version
public static int linearSearch(int[] arr, int key, int lo, int hi)
{
   while ((lo <= hi) && (arr[lo] != key))
      lo = lo + 1;

   // code from base case appears below:
   if (lo > hi)
      return -1;
   else   // lo <= hi and arr[lo] == key
      return lo;
   }
}
```

And now we have our loop-based version of linear search! As with `print(...)`, we can also combine this with the wrapper method if we wanted to. This would be the wrapper method for `linearSearch(...)`:

```
public static int linearSearch(int[] arr, int key)
{
   return linearSearch(arr, key, 0, arr.length - 1);
}
```

so by combining the two, we get:

```
public static int linearSearch(int[] arr, int key)
{
   // these two lines are new
   int lo = 0;
   int hi = arr.length - 1;

   while ((lo <= hi) && (arr[lo] != key))
      lo = lo + 1;

   if (lo > hi)
      return -1;
   else   // lo <= hi and arr[lo] == key
      return lo;
   }
}
```

and after replacing `hi` with a hard-coded `arr.length - 1` in the code, and after replacing `lo` with `i`, we get the following code:

```
public static int LinearSearch(int[] arr, int key, int lo, int hi)
{
   int i = 0;
   while ((i <= arr.length - 1) && (arr[i] != key))
      i = i + 1;

   if (i > arr.length - 1)                           // we ran past end of array
      return -1;
   else   // i <= arr.length - 1 and arr[i] == key  // we found value in array
      return i;
   }
}
```

which should look familiar, being the sort of loop-based linear search you might have written earlier.

As a final example, we can take the `insertInOrder(...)` code from the last lecture packet:

```
// 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
}
```

and convert it the same way:

```
// rough draft #1
// assume lo...hi-1 is a sorted range, and hi holds the new value
public static void insertInOrder(int[] arr, int lo, int hi)
{
   while ((lo < hi) && (arr[hi] < arr[hi - 1]))
   {
      swap(arr, hi - 1, hi);           // 1) swap last two values
      arr = arr;
      lo = lo;
      hi = hi - 1;
   }
   // now, new value is already properly inserted; do nothing
}
```

And after removing the two useless assignments, we get:

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