CMPSCI 187: Programming With Data Structures

Lecture #16: Thinking About Recursion
David Mix Barrington
12 October 2012

Thinking About Recursion

- Review of the `Grid` Class
- Recursion on Linked Structures
- Printing a Linked List Backwards
- Recursion at the Machine Level
- Removing Recursion
- Whether to Use Recursion
- Very Bad Recursive Algorithms

Review of the `Grid` Class

- DJW present a class called `Grid` in Section 4.4, which we will adapt in Project #3. A `Grid` object has a two-dimensional array of booleans, called `grid`. I, not DJW, call a pair (i, j) with `grid[i][j]` true a “land square”, and one with `grid[i][j]` false a “water square”. They have a constructor to set the grid values randomly, with the average percentage of land squares as a parameter.

- A “continent” (or “blob” for DJW) is a set of squares such that from one of them, you can get to any of the others by an NESW path on land squares, but you cannot get to any land square off the continent. DJW present a method `markBlob` that marks all the squares on a continent, and a method `countBlobs` to determine how many continents are in a given grid.

- The `markBlob` method operates recursively -- it can be summarized as “mark the target square, then call `markBlob` on any unseen land neighbors it has”. This *implicitly* manages the search with a stack.

Recursion on Linked Structures

- We can define a linked list recursively once we have the definition of the `LLNode<T>` class. A linked list of `T` objects is either empty (when its `head` pointer is `null`) or consists of an `LLNode` whose info field `contains` a T object and whose `link` field is the head of a linked list.

- If we can define what we want to do recursively, we can easily write recursive code. To add an element to the tail of the list, we make a new node and attach either at the beginning (if the list is now empty) or to the tail of the list at head.link. We can check our three questions -- there is a base case, we will get to it, and our result is correct if the recursive call is correct.

```
public void addToTail (LLNode<T> here, T elem) {
// add node containing elem to tail of list starting at here
   LLNode newNode = new LLNode<T> (elem);
   if (head == null) head = newNode;
   else addToTail (head.getLink( ));}
```

Printing a Linked List Backwards

- DJW's example of a recursive procedure on a linked list is one that prints out the contents of each node to `System.out`, but in *reverse order*, with the tail element first and the head element last.

- The idea is simple given the recursive definition. If the list is empty we have nothing to do. If it is not, we *first* print the list starting with `head.link`, *then* print out the contents of the head node. Their example fits within their `LinkedStack` class. As is common in such cases, they use a **helper method**. The general method takes any node pointer as argument, then the specific method calls the general one on the head pointer of the list. Note this is O(n^2).

```
public void revPrint (LLNode<T> listRef) {
   if (listRef != null) {
      revPrint (listRef.getLink( ));
      System.out.println (" " + listRef.getInfo( ));}}

public void revPrint ( ) {
   revPrint (top);}
```

Recursion at the Machine Level

- I've mentioned before how method calls are handled at the machine level. When method A calls method B, the execution of A is suspended while B runs, and the machine stores an **activation record** that will allow it to restore A's context when B finishes. If B then calls C, the activation record for B goes on top of A's on the **call stack**.

- It's no different when a method calls itself, directly or indirectly. We get multiple activation records on the call stack for the same method. Note that each of these is independent, with, for example, separate copies of each local variable. Running our `factorial(n)` eventually gets a stack of n records.

- Our Project #2 operated recursively in a sense, and the explicit stack could have been replaced by recursive calls to a method that carried out the search starting with a particular `Move`. The `markBlob` method, contrariwise, could also have been implemented non-recursively with an explicit stack.

Removing Recursion

- Many of the recursions we have seen so far are examples of **tail recursion**. They make one call to themselves in the general case, and it occurs at the end of the code. Thus the recursive calls are first all made, and then each returns a a value to the method that called it.

- We can replace this sequence by a non-recursive loop. In the factorial case, we have a value `retValue` which we set to 1 at the beginning, so that we will correctly return 1 if n is 0 and there is no recursion. We then change `retValue` to reflect the action of each version of the method, also keeping the value of n so that we will know when we reach the base case. (Note that DJW's example has reversed the order of the multiplications of the recursive version, which does not matter because multiplication is commutative.)

```
private static int factorial (int n) {
   int retValue = 1;
   while (n > 0) {
      retValue *= n; n--;}
   return retValue;}
```

Stacking to Remove Non-Tail Recursion

- In the recursive revPrint method we saw earlier, each run of the method had at most one recursive call to revPrint, but we did not have a tail recursion because there was code to run after the recursive call.

- To simulate this without recursion, we note that the print statement will execute on the elements successively, operating on what would then be the top element of the calling list. We need to record, on the way down the list, the information that this last statement will need. An explicit stack does this.

```
public void prentReversed ( ) {
   USI<T> stack = new LinkedStack<T>( );
   LLNode<T> listNode = top;
   while (listNode != null) {
      stack.push (listNode.getInfo( ));
      listNode = listNode.getLink( );}
   while (!stack.isEmpty( )) {
      System.out.println (“ “ + stack.top( ));
      stack.pop( );}}
```

Whether to Use Recursion

- Using recursion incurs costs, as do the techniques we could use to replace the recursion. Whether to use recursion for a given problem depends on the tradeoffs between these costs.

- A recursive method uses space on the call stack, which may be a limited resource. It uses some additional time to create and dispose of activation records, and these actions may use slower memory than is needed by the iterative version.

- The iterative version, on the other hand, may require more lines of code for the same job, and may be harder to debug because the algorithm is further removed from the definition of the problem. (Our Three Questions are enormously powerful for verifying completion and correctness.) We'll also see on the next slide that recursive versions can sometimes be far slower. The ideal situation would be to write and debug a recursive version and have it automatically converted to an iterative version by a compiler.

Very Bad Recursive Algorithms

- I can define the function $f(x) = 2^x$ recursively, by the rules $f(0) = 1$ and $f(x) = f(x-1) + f(x-1)$. If I use this definition to write a recursive algorithm, it will make 2^x different calls to f(0) and add their results.

- DJW's example (foreshadowing CMPSCI 240) is to count the number of subsets of a group with a certain number of members. The recursive algorithm below is correct, but horribly slow, again because it reaches its result by adding 1's. You'll eventually see better ways to calculate this.

```
public static int twoToThe (int n) {
   if (n == 0) return 1;
   else return twoToThe (n-1) + twoToThe (n-1);}

public static int combinations  (int group, int members) {
   if (members == 1) return group;
   else if (members == group) return 1;
   else return (combinations (group - 1, members - 1) +
                combinations (group - 1, members));}
```