CmpSci 187: Programming with Data Structures Spring 2015

Lecture #9

John Ridgway

February 26, 2015

1 Recursive Definitions, Algorithms, and Programs

Recursion in General

In mathematics and computer science we often have definitions of concepts that refer to themselves. We call them *recursive*. For example:

- A "directory" in most operating systems, for example, is something that contains things that are either files or directories.
- A "statement" in Java might be an "if statement", and an "if statement" is made up of the words `if` and `else`, a boolean expression in parentheses, and one or two "statements".
- A "postfix expression" is either an operand or two "postfix expressions" followed by an operator.

Recursive Algorithms

- Recursive algorithms are algorithms that call themselves, but only on a "smaller" version of the same problem.
- They are most useful when applied to recursively defined concepts.
- Much of CMPSCI 250 is devoted to the relationship of recursive definitions and algorithms to inductive proofs – here we will look at the definitions and algorithms.

Computing Factorials

- The factorial of a non-negative integer n, written $n!$, is the product of all the numbers from 1 through n, inclusive, or $1 \times 2 \times ... \times n$.

- (The factorial of 0 is 1, because multiplying together an empty set of numbers is like not multiplying at all, or multiplying by 1.)
- For example, $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.
- Here is a recursive definition of "factorial": $0! = 1$, and if $n > 0$, $n! = n \times (n-1)!$

Computing Factorials (continued)

- If we use this definition to try to calculate 5!, we get that $5! = 5 \times 4!$.
- If we apply the definition again to 4!, we get that $5! = 5 \times 4 \times 3!$.
- Continuing in the same way, we get

$$\begin{aligned} 5! &= 5 \times 4 \times 3 \times 2! \\ &= 5 \times 4 \times 3 \times 2 \times 1! \\ &= 5 \times 4 \times 3 \times 2 \times 1 \times 0! \\ &= 5 \times 4 \times 3 \times 2 \times 1 \times 1 \end{aligned}$$

  just as in the non-recursive definition.

Recursive Code for Factorials

The recursive definition is easy to turn into code, as is the original iterative definition:

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

public static int iterFact(int n) {
  int x = 1;
  for (int i=1; i<=n; i++) {x *= i;}
  return x;
}
```

Clicker Question #1

```
public int foo (int n) {
   if (n <= 1) { return 0; }
   return 1 + foo(n-2); }
```

What is returned by the above recursive method if called with a positive integer n as parameter?

A. n

B. 0

C. $n/2$

D. the method never returns a value

Recursive Factorial Example

- What happens when we run the recursive code with a parameter of 5?
- That call makes a call on `factorial(4)`, which calls `factorial(3)`, which calls `factorial(2)`, which calls `factorial(1)`, which calls `factorial(0)`.
- This returns 1 to the `factorial(1)`, which returns 1 to `factorial(2)`, which returns 2 to `factorial(3)`, which returns 6 to `factorial(4)`, which returns 24 to `factorial(5)`, which returns 120, the right answer.

2 The Three Questions

Three Fundamental Questions

For a recursive algorithm to work correctly on a particular input, we need positive answers to the three questions that DJW pose in Section 4.2:

- Does the algorithm have a base case?
- Does every recursive call make progress toward the base case?
- Can we show that the general call to the algorithm gets the right answer if we assume that all the recursive calls get the right answer?

Three Fundamental Questions (continued)

- A base case is where there is no further recursion. Our factorial algorithm has a base case of n = 0.
- We can often guarantee progress because some parameter gets smaller with every call. Our factorial algorithm always calls factorial(n-1), so we get smaller if we are positive.
- We can justify correctness, assuming correctness of the calls, by a recursive definition. This is clear for factorials.

Clicker Question #2

```
public void clear(Stack<Integer> s) {
   if (!s.empty()) {
      s.pop();
      clear(s); } }
```

What is the base case of this recursive method?

A. a `StackUnderflowException` is thrown

B. `s` has a finite number of elements

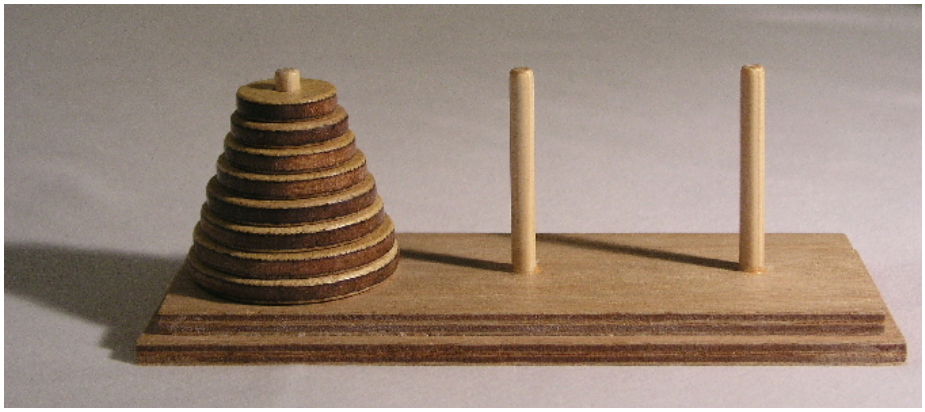
C. calling `clear` makes the stack smaller

D. **`s` is empty**

3 The Towers of Hanoi

Towers of Hanoi

In the Towers of Hanoi game, there are three pegs and n rings of different sizes. Originally all n rings are on the first peg, with the largest at the bottom and the others in order of size above it.



Towers of Hanoi

We want to move all the rings to another peg, following the rules:

- we move one ring at a time; and
- no larger peg may ever be put atop a smaller peg.

A Recursive Solution

We can recursively solve the puzzle by defining the following procedure to move `k` rings from peg `A` to peg `B`. This turns out to take $2^k - 1$ moves.

```
public void move(peg A, peg B, int k) {
  if (k == 1) {
    // move one ring from A to B
  } else {
    move(A, C, k-1);
    // move one ring from A to B
    move(C, B, k-1);
  }
}
```

A Recursive Solution

We can recursively solve the puzzle by defining the following procedure to move `k` rings from peg `A` to peg `B`. This turns out to take $2^k - 1$ moves.

```
public void move(peg A, peg B, int k) {
  if (k == 0) {
    // nothing to do
  } else {
    move(A, C, k-1);
    // move one ring from A to B
    move(C, B, k-1);
  }
}
```

Clicker Question #3

```
public void move (peg A, peg B, int k) {
  if (k != 0) {
    move (A, C, k-1);
   // move one ring from A to B
    move (C, B, k-1);
  } }
```

How do we know that this method will make progress toward its base case?

A. we move at least one ring with each call

B. **the parameter k decreases with each call**

C. the base case has k equal to 1

D. each call has only two recursive calls

4 Counting Blobs

Grids and Continents (Blobs)

- In games like CivilizationTM, the computer creates a map to play on, with land areas, water areas, different terrain for each area, and so forth.
- We're now going to look at a simple version of this where the world is a rectangular grid of squares, each one land or water. Squares are considered adjacent if they share a horizontal or vertical edge, not if they just meet at a corner.

Grids and Continents (continued)

- A continent (called a blob in DJW) is a set of land squares such that you may travel by land from any square on the continent to any other square on the continent, but not to any other square.
- Again, you cannot jump diagonally across a corner in this process.
- We're going to look at the computational problem of counting the continents in a particular rectangular map. We'll be greatly helped by the use of recursion.

Continent Example

- On the left, land squares have an `X` and water squares have a `-`.
- On the right, the five continents have been identified and each land square labeled with the name of its continent.

```
X-X--X--X-- A-A--B--C--
XXX--X----- AAA--B-----
--XX-X----- --AA-B-----
-----XX---- -----BB----
----X---XXX ----D---EEE
```

Clicker Question #4

How many continents does this grid have?

```
X-X--XXX---X-XXXX--X--
XXX--X--XXXXXX--X-----
--XXXX-------XX-X-----
-----XX--------XXXXX--
----XX-XXXX----X---XXX
```

A. 3 **B. 4** C. 5 D. 6

Continent Example

- DJW's code only counts the continents without naming or marking them. We could traverse the grid, counting the squares. But this over-counts each continent, counting one for each square in it.
- The right idea is to traverse the grid; the first time you find one of the squares of a continent increment your counter, then **mark** all of the squares of the continent.

Recursive Marking

- Use a `boolean` array `visited` that will hold true for every land square determined to be part of a continent.
- Set this `visited` value `true` for every square reachable from the target square by marking the target square then recursively marking any of its four neighbors that are in the grid and are unseen land squares.

Code to Mark Continents

```
private void markBlob(int row, int col) {
  visited [row][col] = true;
  if ((row - 1) >= 0 && grid[row - 1][col]
      && !visited[row - 1][col])
  {
    markBlob(row - 1, col);
  }
  // same for (row + 1, col)
  // same for (row, col - 1)
  // same for (row, col + 1)
}
```

Answering the Three Questions for `markBlob`

- Base case: when we call `markBlob` on a square that has no unseen land squares as neighbors, it just marks its target square and returns.
- Progress toward base case: every time we mark a square, we reduce the total number of unseen land squares in the grid, which started out as a finite number. And every call to `markBlob` marks at least one such square, because we only ever call it on an unseen land square.

The Third Question: Correctness

- General correctness: We want to be sure that when `markBlob(i,j)` is called, it marks `(i, j)` and every unseen land square that can be reached from `(i, j)` by using unseen land squares.
- Suppose we are sure that this property is true for all the calls to markBlob on neighbors of (i, j). Then if a square is reachable from `(i, j)`, it must be either be `(i, j)` itself or be reachable from one of its neighbors. So if the neighbor calls do their job, so does the first call.