CS/Math 240: Introduction to Discrete Mathematics 2/24/2011

Lecture 11 : Recursion

Instructor: Dieter van Melkebeek Scribe: Dalibor Zelený

DRAFT

Last time we started talking about recursion. It is convenient for solving problems whose solutions can be reduced to easier instances of the same problem. We've seen last time how to convert some earlier algorithms from class into recursive programs. Today we discuss correctness of recursive programs.

11.1 Correctness of Recursive Programs

When we show correctness, we show that the program meets its specification. We use the word *preconditions* for the conditions a valid input must satisfy, and the word *postconditions* for the conditions the output of the program must satisfy after receiving an input satisfying the preconditions. Using this terminology, a correct program must satisfy the following condition:

Condition ($*$): *For all possible inputs that satisfy the preconditions, the program produces an output that satisfies the postconditions, assuming it halts.*

We phrase the program correctness requirement this way because sometimes the program is designed to work only on some inputs. We saw an example of this last class when our greatest common divisor algorithm worked only on positive inputs, even though the greatest common divisor of zero and a positive number is also defined. Preconditions are also important when we argue correctness of recursive algorithms by induction. As part of the induction hypothesis, we assume that all recursive calls for which the input satisfies the preconditions terminate and return the correct value.

The breakup of correctness into partial correctness and termination remains when proving the correctness of recursive programs.

1. *Partial correctness*: The program produces correct output, assuming the input satisfies the preconditions and condition ($*$) holds for all the recursive calls.
2. *Termination*: On all inputs that satisfy the preconditions, the program produces some output.

We now give correctness proofs of the recursive algorithms we presented last lecture.

11.1.1 Greatest Common Divisor

We restate the recursive algorithm for computing greatest common divisors as function `GCD`, and then prove its correctness.

Algorithm GCD(a, b)

Input: $a, b \in \mathbb{N}$, $a > 0$, $b > 0$
Output: $\gcd(a, b)$

(1) **if** $a = b$ **then return** a
(2) **if** $a < b$ **then return** `GCD(`$a, b - a$`)`
(3) **if** $a > b$ **then return** `GCD(`$a - b, b$`)`

Let's start with partial correctness. The structure of the proof will follow the structure of the algorithm. We will have one case for each of the lines in the description of the algorithm.

Case 1: $a = b$. In this case, the algorithm returns a, and this is the correct answer because $\gcd(a, b) = \gcd(a, a) = a$. (Think of this case as the base case of an inductive proof.)

Case 2: $a < b$. In this case $\gcd(a, b) = \gcd(a, b - a)$ by a result from last lecture. Since $b > a$, a and $b - a$ are both positive integers, so the preconditions of `GCD` are satisfied. Hence, Condition $(*)$ implies that the call `GCD(`a`,`b`-`a`)` returns $\gcd(a, b - a) = \gcd(a, b)$, and that's what we return. Thus, we get partial correctness in this case as well.

Case 3: $a > b$. The argument for this case is similar to the one for Case 2, except the roles of a and b are interchanged.

This completes the proof of partial correctness.

Now let's argue termination. We give a proof by induction on $a + b$. Again, we break down the proof based on the three cases in the algorithm.

Since $a, b > 0$ by the preconditions, the smallest $a + b$ can be is 2. In that case $a = b = 1$, so the algorithm returns. This proves the base case.

For the inductive step, assume that when $a + b \leq n$ and a and b satisfy the preconditions, the algorithm terminates.

Case 1: $a = b$. In this case, $a = b$, so the algorithm returns right away and terminates.

Case 2: $a \neq b$. The algorithm calls `GCD` on input a', b' that satisfies the preconditions, and $a' + b' < a + b$ (one of a or b is reduced by at least 1 to obtain a' and b'). The recursive call then returns by the induction hypothesis, and the algorithm returns right after that happens.

The completes the proof that the algorithm terminates, and also concludes the proof of correctness of the algorithm.

When we prove partial correctness, we don't argue that the program actually halts at some point. We can take that as an assumption since we only prove the implication "if the program halts, the returned value is correct" when we prove partial correctness. For this reason, we can prove partial correctness of a program that doesn't always halt. For example, if our specification were to allow $a = 0$ or $b = 0$, the call `GCD(`a`,`b`-`a`)` would be the same as the call `GCD(`a`,`b`)`, so `GCD` would just be calling itself over and over again. But since both inputs to the recursive call satisfy the preconditions in this case, Condition $(*)$ gives us the postconditions, i.e., the result returned from the recursive call `GCD(`a`,`b`-`a`)` is correct. The result we return is then correct too. However, this argument doesn't tell us that we got closer to the solution by returning a correct answer since Condition $(*)$ assumes that the program terminates.

Finally, we remark that the kind of recursion used to describe function `GCD` is called *tail recursion*. In tail recursion, we only make a recursive call at the very end before we return, and return the value returned by the recursive call.

11.1.2 Grade School Multiplication Algorithm

Now we prove correctness of the recursive version of the grade school multiplication algorithm. This becomes more complicated because we need some more computation in order to produce the result after the recursive calls return.

Recall that we wrote $b = 2q + r$ where $q = \lfloor b/2 \rfloor$ and $r \in \{0, 1\}$ is the remainder after dividing b by 2. Now multiply both sides by a to get $ab = 2aq + ar$. The right-hand side expresses the multiplication ab as three simpler multiplication problems and one addition. See the notes from previous lecture for more detail on this. We just restate the algorithm here.

Algorithm MULT(a,b)

Input: $a, b \in \mathbb{N}$
Output: $a \cdot b$

(1) **if** $b = 0$ **then return** 0
(2) **if** b *is even* **then return** $2 \cdot$ `MULT(`$a, \lfloor b/2 \rfloor$`)`
(3) **else return** $2 \cdot$ `MULT(`$a, \lfloor b/2 \rfloor$`)` $+ a$

First let's prove partial correctness of this algorithm. Like in the previous proof, the proof structure follows the structure of the algorithm.

Case 1: $b = 0$. When $b = 0$, $ab = 0$ as well, and the program returns zero correctly in this case.

Case 2: $b \neq 0$. In this case there are two subcases depending on the parity of b.

Case 2.1: b is even. Then

$$ab = 2 \cdot a \cdot \frac{b}{2} = 2 \cdot \left(a \cdot \left\lfloor \frac{b}{2} \right\rfloor \right). \tag{11.1}$$

Note that a and $\lfloor b/2 \rfloor$ satisfy the preconditions, so `MULT(`$a, \lfloor b/2 \rfloor$`)` returns $a \cdot \lfloor b/2 \rfloor$ by Condition $(*)$. We multiply the returned value by 2 and return the result, which is ab by (11.1).

Case 2.2: b is odd. Then

$$ab = a \cdot \left(2 \left\lfloor \frac{b}{2} \right\rfloor + 1 \right) = 2 \cdot \left(a \cdot \left\lfloor \frac{b}{2} \right\rfloor \right) + a. \tag{11.2}$$

Note that a and $\lfloor b/2 \rfloor$ satisfy the preconditions, so `MULT(`$a, \lfloor b/2 \rfloor$`)` returns $a \cdot \lfloor b/2 \rfloor$ by Condition $(*)$. We multiply the returned value by 2, add a to the result, and return the result of the addition, which is ab by (11.2).

This completes the proof of partial correctness.

We prove termination by strong induction on b. When $b = 0$, our program returns right away, which proves the base case. For the induction step, assume the algorithm terminates whenever the second argument is at most b. Now consider a call to `MULT` with $b + 1$ as the second argument. The recursive call to `MULT` happens with $\lfloor (b+1)/2 \rfloor$ in this case. Note that $\lfloor (b+1)/2 \rfloor \leq b$, so the induction hypothesis implies that the recursive call terminates. After receiving the return value from the recursive call, our call to `MULT` with $b + 1$ performs a few mathematical operations and then returns as well. This completes the proof of termination.

We could also prove correctness of the recursive algorithm for computing the n-th Fibonacci number. We leave this to the reader, and instead focus on a new example, the towers of Hanoi problem.

11.1.3 Towers of Hanoi

In the towers of Hanoi problem, we have three pegs labeled A, B and C, and n disks of increasing size which are stacked on peg A with the largest disk on the bottom and the smallest disk on top. See the initial setup for $n = 4$ in the first frame in Figure 11.1. The goal is to bring all disks from peg A to peg C. In each step, we can move one disk from one peg to another peg, but we can never move a larger disk on top of a smaller disk. See Figure 11.1 for a sequence of moves that achieves this when $n = 4$.

We come up with a recursive procedure that tells us what sequence of moves to make. We want to reduce the problem into subtasks involving fewer disks. A possible subproblem could be moving

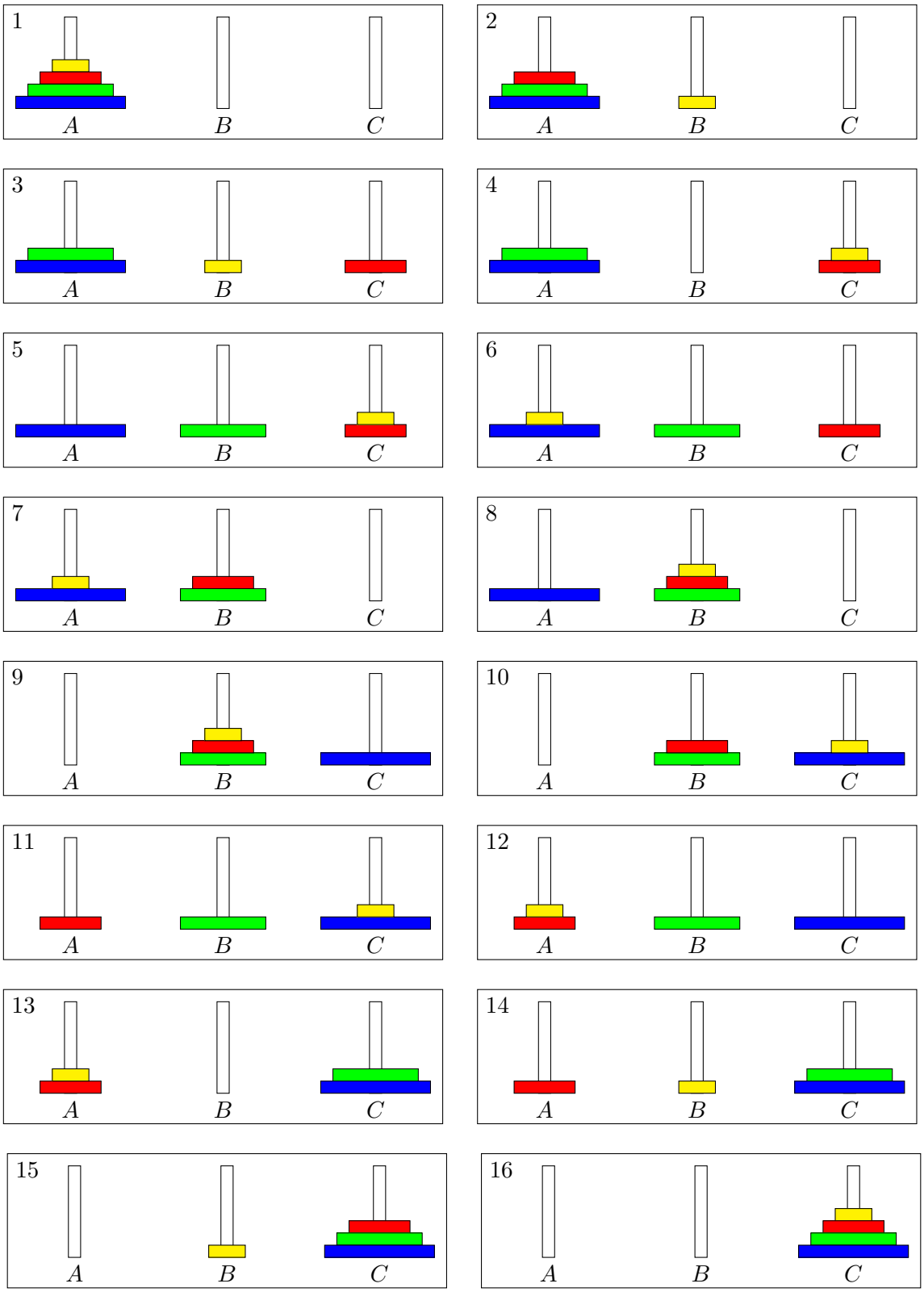

Figure 11.1: Solving the towers of Hanoi problem with 4 disks.

the top $n-1$ disks from peg A to peg B using only legal moves. Assuming this is possible, we can now move the largest disk from peg A to peg C, and then solve another subproblem involving $n-1$ disks to move the $n-1$ disks from peg B to peg C. The sequence of moves in Figure 11.1 achieves this for $n = 4$. Frames 1–8 show how to get the first three disks from peg A to peg B. Going from frame 8 to frame 9 moves the largest disk in its place. Finally, frames 9–16 show how to move all the remaining disks from peg B to their final destination.

We describe the strategy as function TOWERS. Observe that what's happening is that we are building a solution to a larger problem out of solutions to smaller versions of the problem.

Algorithm TOWERS(n, A, B, C)

Input: $n \in \mathbb{N}$, n disks in order on top of peg A, pegs B and C
Output: Sequence of moves that brings those disks to the top of C, using B as intermediate peg.

```
(1) if n = 0 then return
(2)      else return TOWERS(n-1,A,C,B)
                     Move top disk of A to C
                     TOWERS(n-1,B,A,C)
```

We leave the full proof of correctness for discussion sessions next week, and only give an outline. For partial correctness, we argue we don't violate any of the conditions for validity of moves. The key is that when we look at the subproblem of moving the top $n-1$ disks, the largest disk stays put and we can move any disk on top of it without risking that the move is invalid. The proof of termination goes by induction on the number of disks.

11.1.4 Counting Pairs

Our last application of recursion today is a procedure for counting the number of unordered pairs that can be formed by picking two out of n elements. We have n options for the first element in a pair, and $(n-1)$ for the second element since we can't pick the same element twice. Thus, the number of ordered pairs is $n(n-1)$. But we only want to count the number of unordered pairs, and right now we are counting every such pair twice because each element of a pair can be the one that's picked first. Thus, we divide by 2 to get $n(n-1)/2$ unordered pairs.

Now let's design a recursive algorithm that counts these pairs. Write the n elements in a row, and split the row into two, one consisting of k elements and one consisting of $n-k$ elements. There are now three ways in which we can pick a pair consisting of two elements. We can pick both elements from the first row, or we can pick them both from the second row, or we can pick one from the first row and one from the second row.

Let PAIRS(n) be the number of pairs of two out of n elements. Clearly, PAIRS$(1) = 0$. Now let's find a general expression for $n \geq 2$. There are PAIRS(k) pairs formed using just the elements of the first row, and PAIRS$(n-k)$ pairs formed using the elements of the second row. To get a pair consisting of one element from the first row and one element from the second row, we have k options for picking an element from the first row and $n-k$ options for picking an element of the second row, for a total of $n(n-k)$ pairs. Hence, the total number of ways to pick 2 elements out of n is PAIRS(k) + PAIRS$(n-k) + n(n-k)$.

This should remind you of the unstacking game we saw earlier. There, we showed that every strategy gives you the same score. Our recursive definition of PAIRS() provides an alternative explanation for why it is true. The number of unordered pairs formed by picking two elements out of n is unique.