CS/Math 240: Introduction to Discrete Mathematics

Reading 8 : Recursion

Author: Dieter van Melkebeek (updates by Beck Hasti and Gautam Prakriya)a

8.1 Recursion

Recursion in computer science and mathematics refers to the idea of describing the solution of a problem in terms of solutions to easier instances of the same problem. This concept applies to definitions as well as to algorithms or programs. For example, inductive definitions can be thought of as a recursive definitions since they define more complex instances of a concept in terms of simpler instances. For programs, recursion means that we call the program from itself, but on a smaller input. Such a function call a known as a *recursive call*.

It is key that each recursive call to the program is a call that uses a smaller input, so that some call is eventually made with an input for which the problem solved by the program is trivial and for which the program can return the answer right away without any additional recursive calls. This causes an end to the chain of recursive calls, and the recursively called instances of the program terminate, one by one, with the instance called last returning first. If the recursive calls don't use smaller inputs, the program could keep calling itself forever and never terminate.

The next three sections give examples of recursive algorithms for some problems. The goal of these examples is to ease you into thinking in terms of recursion.

8.1.1 Fibonacci Numbers

We can view the definition of the n-th Fibonacci number as a recursive definition. The constructor rule says that $F_n = F_{n-1} + F_{n-2}$. In recursion terms, the "smaller instances" in the recursive definition of the n-th Fibonacci number are Fibonacci numbers with a lower index than n. The foundation rules $F_1 = 1$ and $F_2 = 1$ correspond to the base cases where we don't need another application of recursion.

We can turn the recursive definition of the n-th Fibonacci number into a recursive program that calculates the n-th Fibonacci number. Before we do so (and before we write any program), we should have a specification available. In this case, our program receives a positive integer, n, as input, and returns the n-th Fibonacci number. We give the program as a Function called Fib.

Algorithm Fib(n)

Input: n - A positive integer
Output: F_n - The n-th Fibonacci number

```
(1) if n = 1 or n = 2 then return 1
(2)
(3)                    else return Fib(n-1) + Fib(n-2)
(4)
```

8.1.2 Greatest Common Divisor

We can also rewrite the GCD algorithm using recursion. Observe that the behavior of the function GCD is the same as the behavior of iterative GCD algorithm from the previous reading.

Algorithm GCD(a, b)

Input: a, b - positive integers
Output: gcd(a, b) - their greatest common divisor

(1) **if** $a = b$ **then return** a
(2)
(3) **if** $a < b$ **then return** `GCD(`$a, b - a$`)`
(4)
(5) **if** $a > b$ **then return** `GCD(`$a - b, b$`)`
(6)

8.1.3 Grade School Multiplication Algorithm

Finally, we rewrite the grade school multiplication algorithm from last reading using recursion. We offer some intuition behind the recursive description.

Write $b = 2q + r$ where $q = \lfloor b/2 \rfloor$ and $r \in \{0, 1\}$. Then r corresponds to the last bit in the binary representation of b, and q is formed by the all the remaining bits. To get q out of the binary representation of b, we just "cut off" the last bit from the binary representation of b. Thus, it is possible to obtain binary representations of q and r from the binary representation of b. Also notice that to multiply a number x by 2, we just append a zero after the last bit of the binary representation of x.

If $b = 2q + r$, we can write $ab = a(2q + r) = 2(aq) + ar$. This reduces the multiplication ab into three multiplications, namely ar, aq, and multiplying aq by 2. Let's argue that these multiplication problems are easier so as to give ourselves some confidence that we don't just keep increasing the number of multiplications without end.

First, multiplication by r is easy because r is either 0 or 1. In the former case, $ar = 0$, and in the latter case, $ar = a$, so we don't need an additional recursive call here. Second, the multiplication aq is a multiplication of a by a number that is smaller than b, so this is a multiplication problem with a smaller input than the original problem. Finally, we saw in the previous paragraph how to multiply by 2 in binary: We just append a zero at the end of the binary representation of the number we multiply by 2 and return right away without another recursive call. Thus, we have reduced the problem of multiplying a and b into three easier multiplication problems.

Now that we have some intuition, let's write down the algorithm. It is not an exact transcript of our intuition, but is close. It also allows for multiplication by zero.

Algorithm MULT(a,b)

Input: a, b - integers
Output: ab - their product

(1) **if** $b = 0$ **then return** 0
(2)
(3) **if** b *is even* **then return** $2 \cdot \texttt{MULT}(a, \lfloor b/2 \rfloor)$
(4)
(5) **else return** $2 \cdot \texttt{MULT}(a, \lfloor b/2 \rfloor) + a$
(6)

8.2 Correctness of Recursive Programs

When we prove correctness, we show that the program meets its specification, i.e., a correct program satisfies the following:

For all valid inputs, the program produces the correct output.

We prove the correctness of recursive programs via induction. The breakup of correctness into partial correctness and termination remains when proving the correctness of recursive programs.

1. *Partial correctness*: For partial correctness, we need to show that for all valid inputs x , if the program terminates then the progam returns the correct output for x. To establish partial correctness for recursive programs, it is sufficient to establish the following:

   For all valid inputs x:

   - All recursive calls (if any) made by the program on input x are on valid inputs.
   - Assuming these recursive calls return the correct output and assuming the program terminates, the program returns the correct output on x.
     Here the assumption that the program terminates might seem redundant because we already assume that the recursive calls return something (which means the recursive calls terminate). But the assumption is necessary to handle other structures like while loops in the program. However most recursive programs we will see in this course won't have such additional structures.

   The structure of our proof of partial correctness will mimic the structure of the recursive program under consideration. That is, we first prove that the program returns the correct output on the simplest inputs (think of this as the base case in a proof by induction). After which we prove (a) the inputs to recursive calls are valid and (b) If the recursive calls return the correct output, then the program returns the correct output (think of this as the inductive step in a proof by induction).

   Now, *if* the program terminates on an input then the chain of recursive calls it makes will end at the simplest input. Our proof of partial correctness would then imply that the program returns the correct output (by induction).

2. *Termination*: For recursive programs, termination means that the chain of recursive calls eventually ends. Termination is proved by induction on some quantity (that depends on the inputs) which decreases with each recursive call.

We now give the correctness proofs of some of the recursive algorithms we presented in the previous section.

8.2.1 Greatest Common Divisor

We restate the recursive algorithm for computing greatest common divisors 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)
(3) **if** $a < b$ **then return** `GCD(`$a, b - a$`)`
(4)
(5) **if** $a > b$ **then return** `GCD(`$a - b, b$`)`
(6)

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. Let a, b be a valid input.

Case 1: $a = b$. In this case, the algorithm returns a, and this is the correct answer because $\gcd(a, b) = \gcd(a, a) = a$.

Case 2: $a < b$. In this case $\gcd(a, b) = \gcd(a, b - a)$ by a result from the previous reading. Since $b > a$, a and $b - a$ are both positive integers, $a, b - a$ is a valid input. That is the input to the recursive call is valid. Our assumption that the recursive calls return the correct output implies that the call `GCD(`a,b-a`)` returns $\gcd(a, b - a) = \gcd(a, b)$, and this is what the program returns. 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 the quantity $a + b$. (We choose of $a + b$ because it decreases in each recursive call)

Base case: $a, b \in \mathbb{N}$ since (a, b) is valid 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, b) is valid, 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 makes a recursive call on a valid input a', b' , and $a' + b' < a + b$ (one of a or b is reduced by at least 1 to obtain a' and b'). The recursive call halts by the induction hypothesis, and the algorithm returns terminates right after this.

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". For this reason, we can prove partial correctness of a program that doesn't always halt. For example, if our specification were to allow $a = -2$, $b = -3$, the chain of recursive calls would go on for ever (why?). But in this case, our assunmption that the recursive call returns the return the correct output would still imply that the algorithm returns the correct output. So partial correctness still holds. However, this doesn't tell us that the algorithm returns something on $(-2, -3)$, since that would require that it 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.

8.2.2 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.

Algorithm MULT(a,b)

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

(1) **if** $b = 0$ **then return** 0
(2)
(3) **if** b *is even* **then return** $2 \cdot$ `MULT(`$a, \lfloor b/2 \rfloor$`)`
(4)
(5) **else return** $2 \cdot$ `MULT(`$a, \lfloor b/2 \rfloor$`)` $+ a$
(6)

First let's prove partial correctness of this algorithm. Like in the previous proof, the proof structure follows the structure of the algorithm. Let (a, b) be a valid input.

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

$$ab = 2 \cdot a \cdot \frac{b}{2} = 2 \cdot \left(a \cdot \left\lfloor \frac{b}{2} \right\rfloor \right). \tag{8.1}$$

Note that a and $\lfloor b/2 \rfloor$ are valid inputs, so `MULT(`$a, \lfloor b/2 \rfloor$`)` returns $a \cdot \lfloor b/2 \rfloor$ by our assumption on recursive calls. We multiply the returned value by 2 and return the result, which is ab by (8.1).

Case 2.2: b is odd. Then

$$ab = a \cdot \left(2 \left\lfloor \frac{b}{2} \right\rfloor + 1 \right) = 2 \cdot \left(a \cdot \left\lfloor \frac{b}{2} \right\rfloor \right) + a. \tag{8.2}$$

Note that a and $\lfloor b/2 \rfloor$ are valid inputs, so `MULT(`$a, \lfloor b/2 \rfloor$`)` returns $a \cdot \lfloor b/2 \rfloor$ by our assumption on recursive calls. We multiply the returned value by 2, add a to the result, and return the result of the addition, which is ab by (8.2).

This completes the proof of partial correctness.

We prove termination by strong induction on b (b decreases with each recursive call). 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 end with another example of a recursive algorithm, the proof of correctness of which is left to the reader.

8.2.3 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 8.1. 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 8.1 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 subproblems involving fewer disks. A possible subproblem could be moving 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 8.1 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)

(3) **else return** `TOWERS(`*n-1,A,C,B*`)`
Move top disk of A to C
`TOWERS(`*n-1,B,A,C*`)`

(4)

We leave the full proof of correctness to th reader. 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 is by induction on the number of disks.

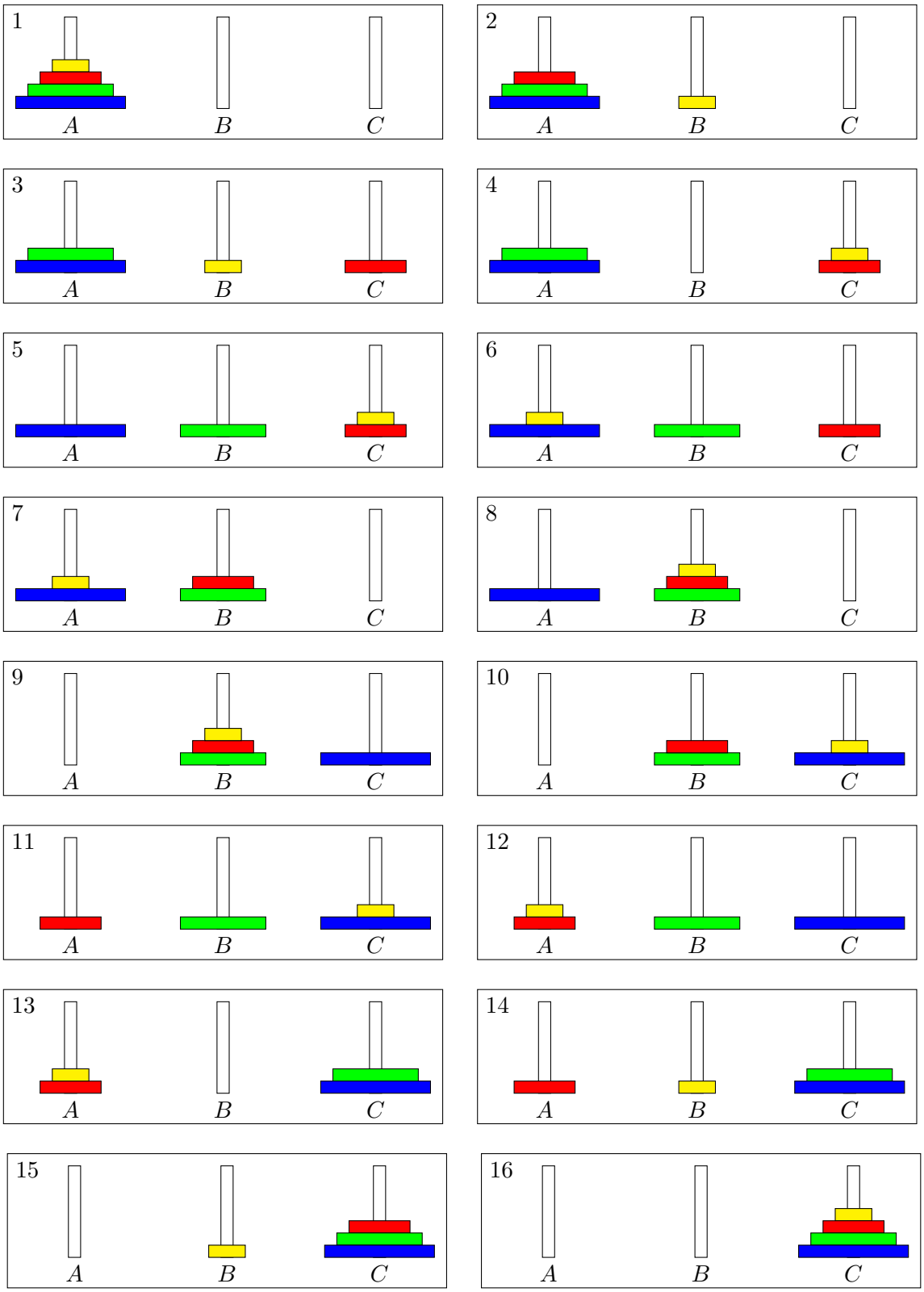

Figure 8.1: Solving the towers of Hanoi problem with 4 disks.