CS280, Spring 2001	Handout 14
February 21, 2001	

1. Reading: K. Rosen *Discrete Mathematics and Its Applications*, 3.3

2. The main message of this lecture:

The principle of mathematical induction can be used to define objects: functions, sequences, sets, etc., in a computationally friendly way.

Recursive definition of a function f defined on integers $n \geq 0$:

1. $f(0) = a$
2. $f(n+1) =$ *the value of some function G of $f(0), f(1), \ldots f(n+1)$*

"Computationally friendly" here means that the definition provides an algorithm for computing the values of f. Indeed, it is easy to see that if the step function G is computable, then f is computable. To evaluate $f(k)$ we consecutively compute $f(0) = a, f(1), \ldots, f(k-1), f(k)$ each time for $k \geq 1$ applying G to **some previously computed values of** f to obtain the next value of f.

Example 14.1.
1. $f(0) = 1$
2. $f(n+1) = f(n) + 3$

From this definitions we can immediately compute $f(0) = 1, f(1) = f(0)+3 = 1+3 = 4, f(2) = f(1) + 3 = 4 + 3 = 7, f(3) = f(2) + 3 = 7 + 3 = 10, \ldots$. It is easy to see that f is nothing but an arithmetical progression with the general term $f(n) = 1 + 3n$.

Example 14.2.
1. $f(0) = a$
2. $f(n+1) = f(n) + d$

Then $f(0) = a, f(1) = f(0) + d = a + d, f(2) = f(1) + d = (a + d) + d = a + 2d, f(3) = f(2) + d = (a + 2d) + d = a + 3d, \ldots, f(k) = f(k-1) + d = (a + (k-1)d) + d = a + kd$. This is the recursive definition of an arithmetical progression.

Example 14.3.
1. $f(0) = a$
2. $f(n+1) = f(n) \cdot q$

Then $f(0) = a, f(1) = f(0) \cdot q = aq, f(2) = f(1) \cdot q = (aq)q = aq^2, f(3) = f(2) \cdot q = (aq^2)q = aq^3, \ldots, f(k) = f(k-1) \cdot q = (aq^{(k-1)})q = aq^k$. This is the recursive definition of a geometrical progression.

Example 14.4. A recursive definition of multiplication via addition. For any given x we define $x \cdot y$ by a recursion on y:
1. $x \cdot 0 = 0$
2. $x \cdot (n+1) = x \cdot n + x$

Again, for any specific y to compute $x \cdot y$ one can consecutively compute $x \cdot 0 = 0, x \cdot 1 = x \cdot 0 + x = 0 + x = x, x \cdot 2 = (x \cdot 1) + x = x + x = 2x, \ldots$.

Example 14.5. Recursive definition of a **sum** $S(n)$ **and a product** $P(n)$ of the first terms of a given sequence $a_0, a_1, a_2, \ldots, a_n$

1. $S(0) = a_0$
2. $S(n+1) = S(n) + a_{n+1}$

1. $P(0) = a_0$
2. $P(n+1) = P(n) \cdot a_{n+1}$

It is easy to see that $S(m) = \sum_{i=0}^{m} a_i$ and $P(m) = \prod_{i=0}^{m} a_i$

Example 14.6. Fibonacci numbers: $f_0 = 0$, $f_1 = 1$, $f_{n+1} = f_{n-1} + f_n$, if $n \geq 1$. The first numbers from this sequence are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. This definition is recursive by spirit, and in can indeed be shaped as the standard recursive scheme. We use $f(n)$ for f_n:

1. $f(0) = 0$
2. $f(n+1) = \begin{cases} 1, & \text{if } n = 0 \\ f(n-1) + f(n), & \text{if } n > 0 \end{cases}$

Fibonacci numbers appear in many applications in Math, CS, Biology, etc. We will give a paradigmatic example of usefulness of Fibonacci numbers in complexity by calculating upper bounds of the Euclidean algorithm.

Lemma 14.7 *For all* $n \geq 3$ $f_n > [(1+\sqrt{5})/2]^{n-2}$.
Proof. Induction on n (see Chapter 3.3)

Theorem 14.8. (Lamé's theorem) *Let* $a \geq b > 0$. *Then the number of divisions used by the Euclidean algorithm to find gcd*(a, b) *is* $\leq 5 \cdot$ *"the number of decimal digits in b".*
Proof. See Chapter 3.3.

Note that the the number of decimal digits in b has the order of $\log_{10} b$. Therefore, the number of divisions (= the number of iterations) in the Euclidean algorithm running on a, b has the order of the decimal log of $\min(a, b)$, which is a pretty good efficiency!

Recursive definitions can define not only functions, but also sets. Naturally, the format of recursive definitions for sets is different from the one from functions. Such definitions are "induction friendly"; you can use induction to prove that all elements of a defined set satisfy certain property. A toy example:

Example 14.9. Define the set S of strings over alphabet a, b by conditions

1. $a \in S$ and $b \in S$,
2. if a string x is in S then $axa \in S$ and $bxb \in S$.

(There is a default assumption that nothing belongs to S unless it can be generated using the two statements in the recursive definition of S.)
Show that all strings from S have an odd length (i.e. the total number of symbols, notation $l(u)$ is the length of u). Induction on the recursive definition of S. We have to show that

A. the initial elements (1) in S have an odd length. Indeed, $l(a) = l(b) = 1$

B. each generating step (2) being applied to an odd length string produces odd length strings only. Indeed, if $l(x)$ is odd, then $l(axa) = 1 + l(x) + 1 = l(x) + 2$ is odd as well as $l(bxb) = 1 + l(x) + 1 = l(x) + 2$, which is odd since $l(x)$ is.

There are more examples of that kind in Chapter 3.3.

Homework assignments. (due Friday 03/02).

14A:Rosen3.3-6ac; 14B:Rosen3.3-10; 14C:Rosen3.3-30.