CS280, Spring 2001 **Handout 15**
February 23, 2001

1. Reading: K. Rosen *Discrete Mathematics and Its Applications*, 3.4
2. The main message of this lecture:

> **A recursively defined function f admits iterative (straight) computation $f(0), f(1), \ldots, f(n)$ as well as recursive (backward) when the algorithm computing $f(n)$ invokes $f(n-1)$ which invokes $f(n-2)$, etc. Efficiency of those methods can be very different.**

Consider an example of a function *power* defined recursively:
1. $power(0) = 1$
2. $power(n+1) = 2 \cdot power(n)$

We can use either of two approaches. If we want to find $power(12)$, for example, we can begin with $power(0) = 1$ and from here compute $power(1) = 2 \cdot power(0) = 2 \cdot 1 = 2$, $power(2)$, $power(3)$, and so on, until we finally get to $power(12)$. A pseudocode algorithm using this approach is shown below.

procedure *power*(n : *nonnegative integer*)
$x := 0$ **for** $i = 0$ **to** n
  $x := x \cdot 2$
$\{x$ is $power(n)\}$

The second approach to computing $power(n)$ uses the recursive definition of *power* directly. Algorithm $rpower(n)$

1. **procedure** *rpower*(n : *nonnegative integer*)
2. **if** $n = 0$ **then**
3. $rpower(n) := 1$
4. **else**
5. $rpower(n) := 2 * rpower(n-1)$

To understand how the algorithm $rpower(n)$ works, let us consider how we could compute $rpower(4)$, for example. We can find the value of $rpower(4)$ if we know the value of $rpower(3)$, but to compute $rpower(3)$, we must first compute $rpower(2)$, and to do this we must first compute $rpower(1)$, therefore, we must first compute $rpower(0)$. Aha!-this we can do, by the basis step. Knowing the value of $rpower(0)$, we can then find the value of $rpower(1)$, then $rpower(2)$, then $rpower(3)$, and finally $rpower(4)$.

Now suppose we start to execute algorithm $rpower(n)$ with the input value $n > 0$. lines 2 and 3 are passed over because $n > 0$. When the algorithm gets to the line 5. it temporarily suspends activity on computing *rpower* with an input value n and invokes itself with a smaller input value. The execution of algorithm *rpower* with an input value of $n-1$, if $n-1 > 0$, will pass over lines 2 and 3 and then invoke algorithm *rpower* with the input value of $n-2$. This [process will continue , with successive invocations, until the input value is finally 0 and the output value, 1, can be computed by the basis step,lines 2 and 3. This final invocation of the algorithm will then give this output value to the second-to-last invocation, and so on. Finally, the original invocation of *rpower* can be completed. Note that in some sense writing algorithm *rpower* is easier then *power*: the former itself carrying out the work we had to do in writing the loop of algorithm *power*. Algorithm *rpower*(n) is an example of a **recursive algorithm**, one that invokes itself. Many programming languages allow such recursion, and it is very natural to use a recursive algorithm to compute a sequence that has been defined recursively.

Definition 15.1. An algorithm is called **recursive** if it works by reducing to its own value on smaller inputs. Recursive algorithms are usually performed backwards.

Example 15.2. A recursive algorithm for computing $gcd(a, b)$

procedure $gcd(a, b :$ *nonnegative integers with* $a < b)$
if $a = 0$ **then** $gcd(a, b) := b$
else $gcd(a, b) := gcd(b$ **mod** $a, a)$

Example 15.3. A recursive algorithm for computing Fibonacci numbers.

procedure *rfibonacci*$(n :$ *nonnegative integer*$)$
if $n = 0$ **then** *rfibonacci*$(n) = 0$
else if $n = 1$ **then** *rfibonacci*$(n) = 1$
else *rfibonacci*$(n) :=$ *rfibonacci*$(n - 1) +$ *rfibonacci*$(n - 2)$

Here the recursive process of invoking algorithm with less input values **branches** which causes an exponential blow-up of its complexity (= the number of additions performed for computing *rfibonacci*(n)): this algorithm requires $f_{n+1} - 1$ additions to find f_n. (Note, that f_n grows faster than $[(1 + \sqrt{5})/2]^n > 1.6^n$.) Let us prove that by induction on n.

Base: $n = 0$ (the number of additions is 0 which is equal to $f_1 - 1$). $n = 1$ (the number of additions is still 0 which is equal to $f_2 - 1$).

Step: The number of additions $\sharp$ in *rfibonacci*(n) equals
$1+ \sharp$ of additions in *rfibonacci*$(n - 1) + \sharp$ of additions in *rfibonacci*$(n - 2) =$
$= 1 + (f_n - 1) + (f_{n-1} - 1) = f_n + f_{n-1} - 1 = f_{n+1} - 1$.

Surprisingly, the straightforward *iterational* algorithm for Fibonacci numbers takes only $n - 1$ additions to find f_n. Such a huge difference in favor of the iterational algorithm (linear vs. exponential) has an easy explanation: the recursive algorithms branches each time it invokes itself and it computes the same sums $f_{k-1} + f_k$ independently along each of the branches. To the contrary, the iterational algorithm computes each sum $f_{k-1} + f_k$ only once and then just uses this sum as many times as necessary.

procedure *ifibonacci*$(n :$ *nonnegative integer*$)$
if $n = 0$ **then** $y = 0$ **else**
begin $x := 0,\ y := 1$
  for $i := 1$ **to** $n - 1$
  begin $z := x + y,\ x := y,\ y := z$
  end
end
$\{y$ is the nth Fibonacci$\}$

Claim: for $n > 1$ this algorithm requires only $n - 1$ additions to compute f_n. Indeed, it takes $n - 1$ loops to get to f_n, each making only one addition.

Homework assignments. (due Friday 03/02).

15A:Rosen3.4-2; 15B:Rosen3.4-8.