Section 5.1
Recurrence Relations

Definition: Given a sequence $\{a_{g(0)}, a_{g(1)}, a_{g(2)}, \ldots\}$, a *recurrence relation* (sometimes called a *difference equation*) is an equation which defines the nth term in the sequence as a function of the previous terms:

$$a_{g(n)} = f(a_{g(0)}, a_{g(1)}, \ldots, a_{g(n-1)})$$

Examples:

• The Fibonacci sequence $a_n = a_{n-1} + a_{n-2}$. (g(n) = n).

• If g(n) = m^n, the recurrence $a_n = ca_{n/m} + b$ describes the complexity of many divide and conquer algorithms.

• Pascal's recursion for the binomial coefficient is a <u>two variable</u> recurrence equation:

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$$

Normally, there are infinitely many sequences which satisfy the equation.

We distinguish them by the *initial conditions*, the values of $a_{g(0)}$, $a_{g(1)}, a_{g(2)}, \ldots$ to uniquely identify the sequence.

Examples:

• In the Fibonacci recurrence we must specify a_0 and a_1

• In the divide and conquer recurrence we must specify $a_{m^0} = a_1$.

• In Pascal's identity we must specify *C(1,0)* and *C(1,1)*.

Example:

If $f(n) = 3^n$, find a recursive definition for f:

• Initial condition: $f(0) = 3^0 = 1$

• Recurrence relation:

$$f(n + 1) = 3^{(n+1)} = 3(3^n) = 3\ f(n)$$

so

$$f(n+1) = 3\ f(n)$$

Modeling with Recurrence Relations

Many relationships are most easily described using recurrence relations.

Examples:

- EASY:

At the credit union interest is compounded at 2% annually. If we do not withdraw the interest, find the total amount invested after n years if the initial amount deposited is d.

Initial Condition: $a_0 = d$
Recurrence equation: $a_n = (1.02)a_{n-1}$

- HARD:

Find a recurrence relation for the number of bit strings of length n which contain 3 consecutive 0's.

Let S be the set of all such strings.

First define the set inductively BUT in such a way as to avoid counting the same string twice:

- *Basis*: 000 is in S

- *Induction* (1): if w is in S and u and v are in {0,1}* then uwv is in S.

- *Extremal clause*: (as usual)

The above definition is adequate to define S but NOT for counting (why?)

A better induction clause:

Induction (2): if w is in S and u is in {0,1}* then

- 1w
- 01w
- 001w
- 000u

are in S.

This yields the recurrence

$$a_n = a_{n-1} + a_{n-2} + a_{n-3} + 2^{n-3}$$

with initial conditions:

$$a_3 = 1,\ a_4 = 3,\ a_5 = 8$$

- a_3::
  - {000}
- a_4::
  - {1000, 0000, 0001} since 1w is in S

• a_5::

- 11000, 10000, 10001 since 1w is in S

- 01000, since 01w is in S

- 00000, 00001, 00010, 00011 since wu is in S

Check:

• $a_6 = a_5 + a_4 + a_3 + 2^3 = 8 + 3 + 1 + 8 = 20$

a_5:: 111000, 110000, 110001, 101000, 100000, 100001, 100010, 100011 = 8

a_4 : 011000, 010000, 010001 = 3

a_3:: 001000 = 1

2^3: 000000, 000001, 000010, 000011, 000100, 000101, 000110, 000111 = 8

Every string is present and nothing is counted twice.

More Examples:

Find a recursive program which determines if a bit string has at most a single 1.

Then determine the worst case number of recursive calls for a string of length n.

Recall the recursive definition:

• *Basis*:

0, λ , 1 are in S.

• *Induction*:

If w is in S then so are 0w and w0.

• *Extremal clause*: (as usual)

A recursive procedure:

procedure *at_most_one_1* (string, basis)

/* a recursive procedure to determine if a bit string has at most a single 1 based on the recursive definition.

- string: the input bit string;

- basis: a variable to remember a '1' has been found at a higher level in the recursion.

- If another '1' is found, the string is not in the language and we pass back 'NO'. Else 'YES' */

/* basis step: */

if string = λ or string = '0' **then**
return 'YES'

if string = ‘1’ and basis = 0 **then**
return ‘YES’

if string =‘1’ and basis = 1 **then return** ‘NO’
else

/* inductive step */

if first and last character of string = ‘0’ **then return** *at_most_one_1* (remainder of string, basis)

if first and last character of string = ‘1’ **return** ‘NO’

if first or last character of string = ‘1’ and
basis = 0 **then return** *at_most_one_1* (remainder of string, 1)

return ‘NO’

The procedure is called with the string to be tested and the basis variable set to zero.

At any level we check both the first and last character of the input string.

We assume the string is all zeros until we find a ‘1’. Then we set the basis variable to one.

If we find two ones at the same time, we return ‘NO’

If we find another ‘1’ and the basis variable is one, then we return ‘NO’, else we keep on stripping off zeros from

the front and the back until we get to the empty string or a single character.

Let f(n) be the number of recursive calls to the procedure (worst case) where n is the length of the input string.

- *Basis*: f(0) = f(1) = 0
- *Induction*: f(n) = f(n-2) + 1
