CHAPTER 5

RECURRENCE RELATIONS

HOMEWORK

- Again, all homework is from the Exercises
  - No problems are from the Review Exercises
- Section 5.1, (p. 232), #4-8, 18-20, 37-40
- Section 5.2, (p. 244), #14-18, 34-36

RECURRENCE RELATIONS

- We actually saw these before
- Here is an example
  - $a_1 = 6$
  - $a_{n+1} = a_n - 3, n \geq 1$
- Here is another example
- This is the Fibonacci relation
  - $F_0 = 1, F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}, n >= 3$

AN EXAMPLE

- For example, suppose you do a job repeatedly
- It's hard
- The first time you do it, it takes one hour
- Each time you do it after that, it takes you one hour longer than the previous time
- You could describe the times like this:
  - $t_1 = 1$
  - $t_{n+1} = t_n + 1, n \geq 1$

THAT JOB JUST GOT EVEN HARDER!

- Now, suppose the job is even harder
- Each time it takes twice as long as the previous time
- Now you can describe the times as
  - $t_1 = 1$
  - $t_{n+1} = 2t_n, n \geq 1$

- Some vocabulary
  - The equations are called a recurrence relation
  - The "first equation" is called an initial condition

A THIRD EXAMPLE

- This example is similar to an example from the book
- You deposit $1000 into the bank
- The bank pays 12% interest annually, compounded monthly
  - This means that 1% is paid each month
- After n months, how much money will you have?

THE RECURRENCE RELATION

- This time I'm starting at 0
- The recurrence relation is
  - $a_0 = 1000$
  - $a_{n+1} = a_n + 0.01a_n, n \geq 0$

- Let's rewrite this by factoring a_n out of the right side of the recurrence relation:
  - $a_0 = 1000$
  - $a_{n+1} = (1 + 0.01)a_n, = 1.01a_n, n \geq 0$

RECURRENCE RELATIONS COME FROM RECURSIVE CODE

- Recurrence relations often come from recursive code.
- For example, to compute the interest, you could use this algorithm:

```
procedure compute_interest (months)
    if months = 0 then
        return 1000
    else
        return 1.01*compute_interest (months–1)
end compute_interest
```

STILL ANOTHER EXAMPLE

- Let S_n = the number of n-bit strings that do not contain 111
- It's easy to find a recurrence relation for this
- We break all n-bit strings that do not contain 111 into three categories
  - Those that start with 0 and do not contain 111
  - Those that start with 10 and do not contain 111
  - Those that start with 11 and do not contain 111
- Notice that these three categories are disjoint (they have nothing in common)
- Also, notice that these three categories cover all possibilities

CONTINUING THE EXAMPLE

- The first case: The string starts with 0 and doesn‘t contain 111
  - It’s easy to see that the number of strings is S_{n-1}
- The second case: The string starts with 10 and doesn’t contain 111
  - It’s easy to see that the number of strings is S_{n-2}
- The third case: The string starts with 11 and doesn’t contain 111
  - Suppose a string starts 11_ and doesn’t contain 111
  - Then the blank cannot contain 1, and so must contain 0
  - So, the number of strings is S_{n-3}
- So we get the recurrence relation $S_n = S_{n-1} + S_{n-2} + S_{n-3}$

FINISHING THE EXAMPLE

- Notice that this is for n >= 4
- For initial conditions, we need to know
  - $S_1 = 2$
  - $S_2 = 4$
  - $S_3 = 7$
- So the recurrence relation is
  - $S_1 = 2.\quad S_2 = 4.\quad S_3 = 7.$
  - $S_n = S_{n-1} + S_{n-2} + S_{n-3}, n >= 4$

THE TOWERS OF HANOI

- Here is another example of recursive code
- Suppose you code the Towers of Hanoi problem directly
- You want to know how much faster a recursive program would run
  - You are ignoring some implementation details
  - For example, it does take time to process activation records
- The code appears on the next slide

THE CODE

- procedure move_disks (n, start_pole, spare_pole, end_pole)
- move_disks (n-1, start_pole, end_pole, spare_pole);
- Move top disk from start_pole to end_pole
- move_disks (n-1, spare_pole, end_pole, start_pole);
- end move_disks

COUNTING MOVES

- We want to know how many moves it takes to move n disks
  - Let's call this m_n
- If there is one disk, $m_1 = 1$
- Otherwise, we get the recurrence relation
  - $m_1 = 1$
  - $m_{n+1} = m_n + 1 + m_n$, or $m_{n+1} = 2m_n + 1, n \geq 1$

THE COBWEB IN ECONOMICS

- Model demand
  - $p = a - bq$, where p = unit price, q = quantity purchased, a, b are constants $>= 0$
  - This equation shows that as the price increases, people will buy less of the product
- Model supply
  - $p = kq$, where p is as above, q = quantity produced, and k is a positive constant
  - This equation shows that as the price increases, the producer will make more
- We will assume that the two q's are the same
  - This means we sell every item we produce
- We want to model this over a period of time
  - This means we have $p_0, p_1, p_2, \ldots$
- We also will assume that there is a one-unit time lag to get the supply to consumers

GENERATING THE MODEL

- So, we use the demand equation $p_n = a - bq_n$
  - The quantity produced at time n will be sold at the price p_n
- We also use the supply equation $p_n = kq_{n+1}$
  - It takes one time period for the producer to adjust the quantity (from q_n to q_{n+1}) for price p_n
- Combining these equations gives $p_{n+1} = a - (b/k)p_n$
- We also need p_0, but we get that from data

INTERPRETING THIS GRAPHICALLY

- If $b < k$, we get the picture shown in Fig. 5.1.5 on p. 230
  - The price stabilizes
- If $b > k$, we get the picture shown in Fig. 5.1.7 on p. 231
  - The price keeps spiraling out of control
- If $b = k$, then the equation changes to $p_{n+1} = a - p_n = a - (a - p_{n-1}) = p_{n-1}$
  - This will oscillate between p_0 and p_1, where $p_1 = a - p_0$

ANOTHER EXAMPLE

- This is n example developed to check on recursion
  - The idea here was to convert a recursive program to one using only for loops
- Ackermann's original function looked different from this one, but is the same
  - See the note before Problem 44 on p. 233 for his original function
- The function is
  - $A(0, n) = n + 1, n \geq 0$
  - $A(m, 0) = A(m-1, 1), m \geq 1$
  - $A(m, n) = A\,(m-1, A(m, n-1)), m \geq 1, n \geq 1$

CHECKING ON THE RECURRENCE RELATION

- This function gets complicated quickly
- Let's calculate A(2, 1)
- We have, by the last recurrence
  - A(2, 1) = A(1,A(2,0))
  - = A(1,A(1,1))
- This process continues
- Details are given on the next slide

THE FULL CALCULATION OF A(2,1)

	Calculation	Justification	Formulas discovered
1	A(2, 1)		
2	= A(1, A(2,0))	because (3) A(2,1) = A(1,A(2,0))	A(2,0) = 3 (from Step 7)
3	= A(1, A(1,1))	because (2) A(2,0) = A(1,1)	A(1,1) = 3 (from Step 7)
4	= A(1, A(0,A(1,0)))	because (3) A(1,1) = A(0,A(1,0))	
5	= A(1, A(0,A(0,1)))	because (2) A(1,0) = A(0,1)	A(1,0) = A(0,1) = 2 (Step 6)
6	= A(1, A(0, 2))	because (1) A(0,1) = 2	A(0,1) = 2
7	= A(1, 3)	because (1) A(0,3) = 3	A(0,2) = 3
8	= A(0,A(1,2))	because (3) A(1,3) = A(0,A(1,2))	
9	= A(0, A(1,2))	Just rewriting the line above with spaces	A(1,2) = 4 (from Step 12
10	= A(0, A(0, A(1,1)))	because (3) A(1,2) = A(0,A(1,1))	
11	= A(0, A(0, 3))	because A(1,1) = 3 from the work above	
12	= A(0, 4)	because (1) A(0,3) = 4	
13	= 5	because (1) A(0,4) = 5	

HOMEWORK

- Now you should be able to complete the homework from Section 5.1
- Again, all homework is from the Exercises
  - No problems are from the Review Exercises
- Section 5.1, (p. 232), #4-8, 18-20, 37-40