A Short Tutorial on Recurrence Relations

The concept: Recurrence relations are recursive definitions of mathematical functions or sequences. For example, the recurrence relation

```
g(n) = g(n-1) + 2n -1
g(0) = 0
```

defines the function f(n) = n^2, and the recurrence relation

```
f(n) = f(n-1) + f(n-2)
f(1) = 1
f(0) = 1
```

defines the famous Fibanocci sequence 1,1,2,3,5,8,13,....

Solving a recurrence relation: Given a function defined by a recurrence relation, we want to find a "closed form" of the function. In other words, we would like to eliminate recursion from the function definition.

There are several techniques for solving recurrence relations. The main techniques for us are the iteration method (also called expansion, or unfolding methods) and the Master Theorem method. Here is an example of solving the above recurrence relation for g(n) using the iteration method:

```
g(n) = g(n-1) + 2n - 1
     = [g(n-2) + 2(n-1) - 1] + 2n - 1
               // because g(n-1) = g(n-2) + 2(n-1) -1 //
     = g(n-2) + 2(n-1) + 2n - 2
     = [g(n-3) + 2(n-2) -1] + 2(n-1) + 2n - 2
               // because g(n-2) = g(n-3) + 2(n-2) -1 //
     = g(n-3) + 2(n-2) + 2(n-1) + 2n - 3
       ...
     = g(n-i) + 2(n-i+1) +...+ 2n - i
       ...
     = g(n-n) + 2(n-n+1) +...+ 2n - n
     = 0 + 2 + 4 +...+ 2n - n
               // because g(0) = 0 //
     = 2 + 4 +...+ 2n - n
     = 2*n*(n+1)/2 - n
               // using arithmetic progression formula 1+...+n = n(n+1)/2 //
     = n^2
```

Applications: Recurrence relations are a fundamental mathematical tool since they can be used to represent mathematical functions/sequences that cannot be easily represented non-recursively. An example is the Fibanocci sequence. Another one is the famous Ackermann's function that you may (or may not :-) have heard about in Math112 or CS14 [see CLR, pp. 451-453]. Here we are mainly interested in applications of recurrence relations in the design and analysis of algorithms.

Recurrence relations with more than one variable: In some applications we may consider recurrence relations with two or more variables. The famous Ackermann's function is one such example. Here is another example recurrence relation with two variables.

```
T(m,n) = 2*T(m/2,n/2) + m*n,    m > 1, n > 1
T(m,n) = n,   if m = 1
T(m,n) = m,   if n = 1
```

We can solve this recurrence using the iteration method as follows. Assume m <= n. Then

```
        T(m,n) = 2*T(m/2,n/2) + m*n
               = 2^2*T(m/2^2,n/2^2) + 2*(m*n/4) + m*n
               = 2^2*T(m/2^2,n/2^2) + m*n/2 + m*n
               = 2^3*T(m/2^3,n/2^3) + m*n/2^2 + m*n/2 + m*n
                 ...
               = 2^i*T(m/2^i,n/2^i) + m*n/2^(i-1) +...+ m*n/2^2 + m*n/2 + m*n
```

Let k = log_2 m. Then we have

```
        T(m,n) = 2^k*T(m/2^k,n/2^k) + m*n/2^(k-1) +...+ m*n/2^2 + m*n/2 + m*n
               = m*T(m/m,n/2^k) + m*n/2^(k-1) +...+ m*n/2^2 + m*n/2 + m*n
               = m*T(1,n/2^k) + m*n/2^(k-1) +...+ m*n/2^2 + m*n/2 + m*n
               = m*n/2^k + m*n/2^(k-1) +...+ m*n/2^2 + m*n/2 + m*n
               = m*n*(2-1/2^k)
               = Theta(m*n)
```

Analyzing (recursive) algorithms using recurrence relations: For recursive algorithms, it is convinient to use recurrence relations to describe the time complexity functions of the algorithms. Then we can obtain the time complexity estimates by solving the recurrence relations. You may find several examples of this nature in the lecture notes and the books, such as Towers of Hanoi, Mergesort (the recursive version), and Majority. These are excellent examples of divide-and-conquer algorithms whose analyses involve recurrence relations.

Here is another example. Given algorithm

```
         Algorithm  Test(A[1..n], B[1..n], C[1..n]);
            if n= 0 then return;
            For i :=  1 to n do
                C[1] := A[1] * B[i];
            call Test(A[2..n], B[2..n], C[2..n]);
```

If the denote the time complexity of Test as T(n), then we can express T(n) recursively as an recurrence relation:

```
         T(n) =  T(n-1) + O(n)
         T(1) = 1
```

(You may also write simply T(n) = T(n-1) + n if you think of T(n) as the the number of multiplications)

By a straighforward expansion method, we can solve T(n) as:

```
         T(n) =  T(n-1) + O(n)
              =  (T(n-2) + O(n-1)) + O(n)
              =  T(n-2) + O(n-1) + O(n)
              =  T(n-3) + O(n-2) + O(n-1) + O(n)
                 ...
              =  T(1) + O(2) + ... + O(n-1) + O(n)
              =  O(1 + 2 + ... + n-1 + n)
              =  O(n^2)
```

Yet another example:

```
         Algorithm Parallel-Product(A[1..n]);
            if n = 1 then return;
            for i := 1 to n/2 do
                A[i] := A[i]*A[i+n/2];
            call Parallel-Product(A[1..n/2]);
```

The time complexity of the above algorithm can be expressed as

```
            T(n) =  T(n/2) + O(n/2)
            T(1) = 1
```

We can solve it as:

```
            T(n) =  T(n/2) + O(n/2)
                 =  (T(n/2^2) + O(n/2^2)) + O(n/2)
                 =  T(n/2^2) + O(n/2^2) + O(n/2)
                 =  T(n/2^3) + O(n/2^3) + O(n/2^2) + O(n/2)
                    ...
                 =  T(n/2^i) + O(n/2^i) +...+  O(n/2^2) + O(n/2)
                 =  T(n/2^log n) + O(n/2^log n) +...+  O(n/2^2) + O(n/2)
                            // We stop the expansion at i = log n because
                               2^log n =  n //
                 =  T(1) + O(n/2^log n) +...+  O(n/2^2) + O(n/2)
                 =  1 + O(n/2^log n +...+  n/2^2 + n/2)
                 =  1 + O(n*(1/2^log n +...+  1/2^2 + 1/2)
                 =  O(n)
                           // because 1/2^log n +...+  1/2^2 + 1/2 <= 1 //
```

Using recurrence relations to develop algorithms: Recurrence relations are useful in the design of algorithms, as in the dynamic programming paradigm. For this course, you only need to know how to derive an iterative (dynamic programming) algorithm when you are given a recurrence relation.

For example, given the recurrence relation for the Fibonacci function f(n) above, we can convert it into DP algorithm as follows:

```
          Algorithm Fib(n);
              var f[0..n]: array of integers;
              f[0] := f[1] := 1;
              for i :=  2 to n do
                  f[i] := f[i-1] + f[i-2];
                      // following the recurrence relation //
              return f[n];
```

The time complexity of this algorithm is easily seen as O(n). Of course you may also easily derive a recursive algorithm from the recurrence relation:

```
          Algorithm Fib-Rec(n);
              if n = 0 or 1 then return 1;
              else
                  return Fib-Rec(n-1) + Fib-Rec(n-2);
```

but the time complexity of this algorithm will be exponential, since we can write its time complexity function recursively as:

```
          T(n) = T(n-1) + T(n-2)
          T(1) = T(0) = 1
```

In other words, T(n) is exactly the n-th Fibonacci nummber. To solve this recurrence relation, we would have to use a more sophisticated technique for linear homogeneous recurrence relations, which is discussed in the text book for Math112. But for us, here it suffices to know that T(n) = f(n) = theta(c^n), where c is a constant close to 1.5.