Lecture Notes 7 – Recurrence relations
CSS 501 – Data Structures and Object-Oriented Programming

Reading for this lecture: Lecture notes

To be covered in this lecture:
- Recurrence relations:
  - Iteration
  - Master theorem
  - Examples

Recurrence relations
A recurrence relation is an equation for a sequence of numbers, where each number (except for the base case) is given in terms of previous numbers in the sequence. These are very useful for determining the complexity of recursive algorithms. We have already seen some examples of recurrence relations:

Fibonacci sequence: F(n) = F(n-1) + F(n-2) for n > 1, F(0) =0, F(1) = 1
Number of moves to solve the towers of Hanoi: M(n) = 2M(n – 1) + 1 for n > 1, M(1) = 1

Iteration
We will examine two methods of solving these relations, so that we have a closed form solution (not in terms of a previous value). The first method is called **iteration** (like an iterative algorithm). Let's look at a simple case:

f(n) = f(n – 1) + 10 for n > 1 f(1) = 5

For example, f(4) = f(3) + 10 = f(2) + 10 + 10 = f(1) + 10 + 10 + 10 = 5 + 10 + 10 + 10 = 35.

In general, f(n) = 10 + f(n – 1) = 10 + 10 + f(n – 2) = 10 + 10 + … + 10 + f(1).
Generalizing, we get: f(n) = k*10 + f(n – k).
This ends when we reach the base case n = 1, which occurs when k = n -1. At this point, we have:
f(n) = (n – 1)*10 + f(1) = 10n – 10 + 5 = 10n – 5, which is our solution.

We can apply this idea to analysis of the recursive factorial method:

```
// precondition: n >= 0
int factorial(int n) {
        if (n == 0 || n == 1) return 1;
        return n * factorial(n – 1);
}
```

Let's measure the running time in terms of the number of calls to the method, since each call requires O(1) time. The number of calls is:

f(n) = 1 + f(n - 1) for n > 1
f(1) = 1

Iteration yields: f(n) = 1 + f(n – 1) = 1 + 1 + f(n – 2) = 1 + 1 + 1 + f(n – 3) = 1 + 1 + 1 + 1 + f(n – 4)
From here, we can see that the pattern is: f(n) = k + f(n – k)
This will end when we reach the stopping case of n = 1, so we set n – k = 1 and get k = n – 1. Substituting this into the pattern we get f(n) = n – 1 + f (1) = n – 1 + 1 = n. Thus, we have n recursive calls for n > 0 and the overall running time is O(n).

We can do something similar for binary search. We can measure the number of recursive calls as:
B(n) = 1 + B(n/2)
B(1) = 1

Using iteration gives:
$B(n) = 1 + B(n/2) = 1 + 1 + B(n/4) = 1 + 1 + 1 + B(n/8)$.
This generalizes to:
$B(n) = k + B(n / 2^k)$
The ends when $n / 2^k = 1$ or $k = \log n$. Using $k = \log n$ yields:
$B(n) = (\log n) + B(1) = 1 + \log n$

Now, how about the number of move to solve the **towers of Hanoi** problem:
$M(n) = 2M(n - 1) + 1 = 2 * (2M(n - 2) + 1) + 1 = 4M(n - 2) + 2 + 1 = 4(2M(n - 3) + 1) + 2 + 1 =$
$8M(n - 3) + 4 + 2 + 1$

Generalizing, we get:
$M(n) = 2^k * M(n - k) + 2^{k-1} + 2^{k-2} + \ldots + 2 + 1$

This also ends when we reach $k = n - 1$, since $M(1)$ is our base case.
$M(n) = 2^{n-1}M(1) + 2^{n-2} + \ldots + 2^0$

Since $M(1)$ is 1, we get:

$$M(n) = \sum_{i=0}^{n-1} 2^i$$

We will use the following fact in order to simplify this summation.

Fact: $\sum_{i=0}^{x} k^i = 1 + k + k^2 + \ldots + k^x = (k^{x+1} - 1) / (k - 1)$

Substituting $k = 2$ and $x = n - 1$, we get:
$2^{n-1} + 2^{n-2} + \ldots + 2^0 = (2^n - 1) / (2 - 1) = 2^n - 1$

Master Theorem (optional)

If we have any recurrence relation of the form: $f(n) = a \cdot f(n/b) + c \cdot n^d$, then we can determine the solution (assumptions: a and b are integers such that $a \geq 1$, $b > 1$, and c and d are real number such that $c > 0$, $d \geq 0$.) This is called the **master theorem**. If this is true, then we have three cases:

Case 1: $a < b^d$: $f(n)$is $O(n^d)$
Case 2: $a = b^d$: $f(n)$is $O(n^d \log n)$
Case 3: $a > b^d$: $f(n)$is $O(n^{\log_b a})$

The proof of this is a little tricky!
(You start by using iteration and applying the summation fact from above to get:
$f(n) = a^{\log_b n} f(1) + ((a / b^d)^{\log_b n} - 1) / (a / b^d - 1) \cdot n^d$
It turns out that $a^{\log_b n} = n^{\log_b a}$.
A case-by-case analysis eventually leads to the above result.)

Powers

We can write the definition of computing x^n (x to the nth power) in multiple ways. The simplest is:

$x^0 = 1$
$x^n = x * x^{n-1}$ if $n > 0$

This leads to the following recursive method:

```
int power2(int x, int n) {
        if (n <= 0) return 1;
        else return x * power2(x, n-1);
}
```

The analysis of the running time is straightforward using another recurrence relation. Since, the time in each recurrence call is O(1), we can measure the running time by the number of recursive calls using the following recurrence relation:

$P(0) = 1$

$P(n) = 1 + P(n-1)$ if $n > 0$

The solution is simple and yields $P(n) = n + 1$ (for $n \geq 0$).

Another way of computing x^n uses the following recurrence:

$x^0 = 1$

$x^n = (x^{n/2})^2$ if $n > 0$ and n is even

$x^n = x * (x^{n/2})^2$ if $n > 0$ and n is odd

Here is the code to implement it:

```
int power(int x, int n) {
        if (n <= 0) return 1;
        int tmp = power(x, n/2);
        if (n%2 == 0) return tmp * tmp;
        else return x * tmp * tmp;
}
```

The running time of this version is more interesting.

$P(0) = 1$

$P(n) \leq 1 + P(n/2)$

This is just like binary search. We reduce the problem in half each time we make a recursive call. A similar analysis yields that this method requires O(log n) time.

Practice problems (optional):

Solve the following recurrence relations using iteration:

$F(n) = F(n-1) + 3$ $n>2$

$F(2) = 1$

$F(n) = F(n-1) + 2n$ $n>0$

$F(0) = 2$

$F(n) = F(n-1) + n^2$ $n>1$

$F(1) = 1$

Leaving the above in the form of summation is fine. Closed form solution for summations can be looked up, here we are more interested in going through with iteration and observing patterns for the summation.

Solve the following recurrence relations in terms of Big-O notation using the Master theorem:

$F(n) = 2F(n/2) + 6n$

$F(n) = 4F(n/3) + 3n$

$F(n) = F(n/4) + n^2$