Lecture 3.4: Recursive Algorithms

CS 250, Discrete Structures, Fall 2015

Nitesh Saxena

Adopted from previous lectures by Zeph Grunschlag

Course Admin

- Graded HW2
  - Will distribute today
- **HW3 posted**
  - Covers “Induction and Recursion” (Chapter 5)
  - Due 11am Nov 17 (Tuesday)
Course Admin

- **Mid Term 2**
  - Nov 10 (Tues)
  - Covers “Induction and Recursion” (Chapter 5)
  - Review Nov 5 (Thu)
- Study topics provided
- Sample exam will be provided

Outline

- Recursive Algorithms and Programs
- Proving the correctness of recursive programs
long factorial(int n){
    if (n==0) return 1;
    return n*factorial(n-1);
}

Compute 5!

f(5) = 5 \cdot f(4)
Recursive Algorithms Computer Implementation

```c
long factorial(int n){
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

<table>
<thead>
<tr>
<th>f(4)</th>
<th>f(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4·f(3)</td>
<td>5·f(4)</td>
</tr>
</tbody>
</table>

f(5) = 5·f(4)

Recursive Algorithms Computer Implementation

```c
long factorial(int n){
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

<table>
<thead>
<tr>
<th>f(3)</th>
<th>f(4)</th>
<th>f(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3·f(2)</td>
<td>4·f(3)</td>
<td>5·f(4)</td>
</tr>
</tbody>
</table>

f(5) = 5·f(4)
Recursive Algorithms Computer Implementation

```c
long factorial(int n)
{
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

<table>
<thead>
<tr>
<th>n</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 \cdot f(0)</td>
</tr>
<tr>
<td>2</td>
<td>2 \cdot f(1)</td>
</tr>
<tr>
<td>3</td>
<td>3 \cdot f(2)</td>
</tr>
<tr>
<td>4</td>
<td>4 \cdot f(3)</td>
</tr>
<tr>
<td>5</td>
<td>5 \cdot f(4)</td>
</tr>
</tbody>
</table>
Recursive Algorithms Computer Implementation

long factorial(int n)
{
    if (n==0) return 1;
    return n*factorial(n-1);
}

1·1 = 1
1 \rightarrow f(0)

f(0) = 1
f(1) = 1·f(0)
f(2) = 2·f(1)
f(3) = 3·f(2)
f(4) = 4·f(3)
f(5) = 5·f(4)
Recursive Algorithms Computer Implementation

```c
long factorial(int n){
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

2·1= 2→ f(3)= 3·f(2)
f(4)= 4·f(3)
f(5)= 5·f(4)

11/2/2015

Recursive Algorithms Computer Implementation

```c
long factorial(int n){
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

3·2= 6→ f(4)= 4·f(3)
f(5)= 5·f(4)

11/2/2015
Recursive Algorithms Computer Implementation

```c
long factorial(int n){
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

4\cdot6 = 24\rightarrow f(5) = 5\cdot f(4)

5\cdot24 = 120\rightarrow
Recursive Algorithms Computer Implementation

```c
long factorial(int n) {
    if (n==0) return 1;
    return n*factorial(n-1);
}
```

Return 5! = 120

From Recursive Definitions To Recursive Algorithms

In general, starting a recursive function:

\[
f(n) = \begin{cases} 
  \text{output}_1, & \text{if } n \text{ in range}_1 \\
  \vdots \\
  \text{output}_k, & \text{if } n \text{ in range}_k 
\end{cases}
\]

gives a recursive algorithm:

```c
output-type f(input-type n) {
    if (n in range_1)
        return output_1
    ...
    if (n in range_k)
        return output_k
}
```

11/2/2015
Efficiency of Recursive Algorithms

Examples

We can also turn the recursive definitions of the Fibonacci function, and the gcd into recursive algorithms, but in the case of Fibonacci, these algorithms are really bad (as far as running time is concerned).

Here's the pseudocode for these examples:

```
Recursive Algorithms
Fibonacci

integer f (non-neg. integer n){
    if (n ≤ 1) return n
    return f (n -1) + f (n -2)
}
```

We need to at most $2^n$ calls to the function $f()$ to be able to evaluate it for input $n$. Because going from $n$ to $n-1$ spawns off 2 method calls, and each of these, in turn, spawns 2 threads, and so on. Even if $n$ is moderately large, this algorithm will take a long time to compute.
Recursive Algorithms

Fibonacci

Q: Is there a better algorithm?
A: Use iterative algorithm instead of a recursive one, especially for large numbers

integer fibonacci(non-neg-integer n){
    f[0]=0, f[1]=1 //array or sequence
    for (i = 2 to n ) {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}

11/2/2015

Recursive Algorithms

gcd

integer gcd (positive integer x, positive integer y)
{
    if (y == 0 ) return x
    return gcd(y,x % y)
}

Running time: The algorithm is actually very efficient. It is linear with respect to the length of the largest input. Note that every time you recursively call the function, the size of inputs reduces.

11/2/2015
Program Correctness

Induction can be used to show that programs using iterative loops are correct.

Ex: Prove that the following program correctly calculates Fibonacci sequence:

```plaintext
integer fibonacci(non-neg-integer n) {
    f[0]=0, f[1]=1
    for (i = 2 to n) {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}
```

- Apply strong mathematical induction
  - Show that the program returns the correct output for n=0
  - Assume that the program returns the correct output for n=k, n=k-1, ..., n=0, and using this assumption, show that the program returns the correct output for n=k+1
  - Let's use the whiteboard to do this proof!
Program Correctness
Fibonacci Example

integer fibonacci(non-neg-integer n){
    for (i = 2 to n) {
        f[i] = f[i-1] + f[i-2]
    }
    return f[n]
}

What's wrong here?

Today’s Reading
- Rosen 5.4