Recursive Function Analysis

Here's a simple recursive function which raises one number to a (non-negative) power:

```cpp
//double power(double n, unsigned k)
if k == 0 return 1
return n * power(n, k-1)
```

What is the cost of power()? Note that this is no longer C++ code, but pseudocode. Most algorithms are described using pseudocode for compactness and clarity.

Analyzing Power

• First, note that we want to analyze power in terms of k, not n (why?)
• Now, ask the following:
  – How much work do we do within power(), excluding the recursive call?
  – How many calls do we make to power()?

Analyzing Power

We can think of this another way by visualizing our call stack, and ask these questions:

<table>
<thead>
<tr>
<th>How much work at each level?</th>
<th>How many levels?</th>
</tr>
</thead>
</table>
| One comparison, one multiplication | How many times can we subtract 1 before we get to \( k = 0 \)?
Analyzing Power

Analysis:
2 operations per level * k levels
= 2k operations

In “Big O”, we drop constants, so that’s O(k).

Analyzing Power 2

Suppose we try a different approach. This one is doubly-recursive:

```c
double power(double n, unsigned k)
{
    if k == 0 return 1;
    else if k == 1 return n;
    else return power(n, ⌈k/2⌉) * power(n, ⌊k/2⌋);
}
```

The expression ⌈x⌉ is called the ceiling of x, and means that we round up to the nearest integer. ⌊x⌋ is called the floor of x, and means we round down.

Analyzing Power 2

- Now things are more complicated, because each call to power turns into two more calls to power, etc.
- Instead of a stack, we can visualize this as a “call tree”:

```
   power()
  /     \
power()  power()  
 /     /     \   
power() power()  power()  
```

- How many calls to power here?

Analyzing Power 2

- For these kinds of problems, easier to approximate using an ideal case:
  - Assume k is power of 2: k = 2^p
  - Now we divide k evenly in half at each level

- How many levels are in our tree?
- How much work is done at each level?

Analyzing Power 2

We do 2 operations at most in power.
So our work is less than or equal to:

\[ 2 \cdot (1 + 2 + 4 + \ldots + k/2 + k) \]
\[ = 2k \cdot (1/k + \ldots + 1/4 + 1/2 + 1) \]

The sum \( 1 + 1/2 + 1/4 + \ldots + 1/k < 2 \), so our total is

\[ < 4k = O(k) \], same as before! \]
A Smarter Way

Here’s a better way:

```c
double power(double n, unsigned k)
    if k == 0 return 1
    double m = power(n, ⌊k/2⌋)
    if k is even
        return m * m
    else
        return m * m * n
```

Example

Does this work?

Try it: let k = 11

```
power (n, 11)
m = power (n, 5)
k is odd so
return (m * m * n) = (n^5 * n^5 * n) = n^{11}
```

Analyzing Power 3

Compare to previous version:

– Only 1 recursive call
– Still divide k in half at each step

Now our call “tree” is just a stack again...

But shorter than the first version’s stack!

Analyzing Power 3

How high is the stack?

How many times can you divide a number by 2 before getting to 1?

Suppose \( k = 2^p \). Then our answer is

\[
p = \log_2 2^p = \log_2 k.
\]

So the cost of this version is \( O(\log_2 k) \), much better than \( O(k) \).

Recurrence Relations

In previous slides, we just counted operations.

Recurrence relations are a mathematical expression that captures the cost of recursive functions.

Basically, it is shorthand to help with counting.

Recurrence Relations

Let \( T(n) \) be the cost of our function for an input of size \( n \).

For our three versions of power, we have the following:

1. \( T(k) = T(k-1) + O(1) \)
2. \( T(k) = 2T(k/2) + O(1) \)
3. \( T(k) = T(k/2) + O(1) \)

Solving these is something of an art form, and not a topic for this class. But you should be aware of their existence, anyway!
Analyzing Fibonacci

Here’s a popular function for analysis:

\[ F(n) = F(n-1) + F(n-2) \]

with

\[ F(0) = 0, \ F(1) = 1. \]

(Note this is already in the form of a recurrence relation!)

Analyzing Fibonacci

Let’s implement a recursive function to calculate the \( n \)th Fibonacci number:

```c
int fib(int n)
{
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

Analyzing Fibonacci

Just as in our second power() function, we make two recursive calls, so we get a tree:

```
fib(n)
   /\  
fib(n-2)  fib(n-1)
   /\  /\  
fib(n-3) fib(n-2) fib(n-3)
   /\  /\  /\  
fib(n-3) fib(n-4) fib(n-3) fib(n-4)
```

This is just the call tree for \( \text{fib}(4) \)!

Note that we do some constant amount of work on each call to \( \text{fib}() \). So we do work

\[ T(n) = T(n-1) + T(n-2) + O(1). \]

Recall

\[ F(n) = F(n-1) + F(n-2) \]

This means \( T(n) \geq F(n) \).

It turns out that \( F(n) \approx 2^{0.694n} \)

Thus we have an exponential cost – not good!

A Smarter Way

```c
int fib(int n)
{
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else
    {
        make array A of n integers
        A[0] = 0, A[1] = 1
        for i = 2 to n
        return A[n]
    }
}
```

What is the complexity of this function?