Fibonacci numbers and Recursion
Announcements

• Reading Quizzes:
  – Out of 17
  – Grades are going to be updated
  – You still have until MidTerm

• Project 2 is out
Agenda

• The worst algorithm in the history of humanity

• An iterative solution

• A better iterative solution

• The repeated squaring trick
And the worst algorithm in the history of humanity

FIBONACCI SEQUENCE
Fibonacci sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- \( F[0] = 0 \)
- \( F[1] = 1 \)
• [http://www.youtube.com/watch?v=aB_KstBiou4](http://www.youtube.com/watch?v=aB_KstBiou4)
Recursion – fib1()

/**
 * the most straightforward algorithm to compute F[n]
 */
long fib1(int n) {
    if (n <= 1) return n;
    return fib1(n-1) + fib1(n-2);
}
Typical Runtime

Fib1 run time

Time in seconds

n

CSC 172, Fall 2017
On large numbers

• Looks like the run time is doubled for each n++

• We won’t be able to compute $F[140]$ if the trend continues

• The age of the universe is 15 billion years $< 2^{60}$ sec

• The function looks … exponential
  – Is there a theoretical justification for this?
A Note on “Functions”

- Sometimes we mean a Java function
- Sometimes we mean a mathematical function like $F[n]$
- A Java function can be used to compute a mathematical function
  - But not always! There are un-computable functions
  - Google for “busy Beaver numbers” and the “halting problem”, for typical examples.
- What we mean should be clear from context
Guess and induct strategy

Thinking about the main body

ANALYSIS OF FIB1()
Guess and induct

- For \( n > 1 \), suppose it takes \( c \) mil-sec in \( \text{fib1}(n) \) not counting the recursive calls
- For \( n = 0, 1 \), suppose it takes \( d \) mil-sec
- Let \( T[n] \) be the time \( \text{fib1}(n) \) takes
- \( T[0] = T[1] = d \)
- \( T[n] = c + T[n-1] + T[n-2] \) when \( n > 1 \)

- To estimate \( T[n] \), we can
  - Guess a formula for it
  - Prove by induction that it works
The guess

• **Bottom-up iteration**
  
  - \( T[0] = T[1] = d \)
  - \( T[2] = c + 2d \)
  - \( T[3] = 2c + 3d \)
  - \( T[4] = 4c + 5d \)
  - \( T[5] = 7c + 8d \)
  - \( T[6] = 12c + 13d \)

• **Can you guess a formula for \( T[n] \)?**
  
  - \( T[n] = (F[n+1] - 1)c + F[n+1]d \)
The Proof

- **The base cases:** \( n = 0, 1 \)
- **The hypothesis:** suppose
  - \( T[m] = (F[m+1] - 1)c + F[m+1]d \) for all \( m < n \)
- **The induction step:**
  - \( T[n] = c + T[n-1] + T[n-2] \)
    - \( = c + (F[n] - 1)c + F[n]d \)
    - \( + (F[n-1] - 1)c + F[n-1]d \)
    - \( = (F[n+1] - 1)c + F[n]d \)
How does this help?

$$F[n] = \frac{\phi^n - \left(-\frac{1}{\phi}\right)^n}{\sqrt{5}}$$

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6$$

The golden ratio

CSC 172, Fall 2017
So, there are constants $C, D$ such that

$$C \phi^n \leq T[n] \leq D \phi^n$$

This explains the exponential-curve we saw

$$T(n) = \Theta(\phi^n)$$
• Suppose $\text{fib1}(140)$ runs on a computer with $C = 10^{-9}$:

$$10^{-9} (1.6)^{140} \geq 3.77 \cdot 10^{19} > 100 \cdot \text{age of univ.}$$
- A Linear time algorithm using ArrayList
- A linear time algorithm using arrays
- A linear time algorithm with constant space
An algorithm using ArrayList

```java
long fib2(int n) {
    // this is one implementation option
    if (n <= 1) return n;
    ArrayList<Long> A = new ArrayList<>();
    A.add(new Long(0)); A.add(new Long(1));
    for (int i=2; i<=n; i++) {
        A.add(A.get(i-1) + A.get(i-2));
    }
    return A.get(n);
}

Guess how large an n we can handle this time?
```
## Data

<table>
<thead>
<tr>
<th>n</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
<th>$10^9$</th>
</tr>
</thead>
<tbody>
<tr>
<td># seconds</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>Eats up all my CPU/RAM</td>
</tr>
</tbody>
</table>
How about an array?

```java
long fib2(int n) {
    if (n <= 1) return n;

    long[] A = new long[n+1];
    A[0] = 0; A[1] = 1;
    for (int i=2; i<=n; i++) {
    }
    return A[n];
}
```

Guess how large an n we can handle this time?

CSC 172, Fall 2017
Data structure matters a great deal!

Some assumptions we made are false if too much space is involved: computer has to use hard-drive as memory

<table>
<thead>
<tr>
<th>n</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
<th>$10^9$</th>
</tr>
</thead>
<tbody>
<tr>
<td># seconds</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Stack Overflow</td>
</tr>
</tbody>
</table>
long fib3(int n) {
    if (n <= 1) return n;
    long a=0, b=1, temp = 0;
    for (int i=2; i<= n; i++) {
        temp = a + b; // F[i] = F[i-2] + F[i-1]
        a = b; // a = F[i-1]
        b = temp; // b = F[i]
    }
    return temp;
}
The answers are incorrect because $F[10^8]$ is greater than the largest integer representable by `int`.

But that’s ok. We want to know the runtime.
AN EVEN FASTER ALGORITHM

- The repeated squaring trick
Math helps!

• We can re-formulate the problem a little:

\[
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
0
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
1
\end{bmatrix}
= 
\begin{bmatrix}
2 \\
1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
2 \\
1
\end{bmatrix}
= 
\begin{bmatrix}
3 \\
2
\end{bmatrix}
\]
\[
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
F[n - 1] \\
F[n - 2]
\end{bmatrix} =
\begin{bmatrix}
F[n] \\
F[n - 1]
\end{bmatrix}
\]

\[
\begin{bmatrix}
F[n + 1] \\
F[n]
\end{bmatrix} =
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}^n
\begin{bmatrix}
1 \\
0
\end{bmatrix}
\]
How to we compute $A^n$ quickly?

- Want

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

- But can we even compute $3^n$ quickly?
First algorithm

```c
long power1(int n, int base) {
    long ret=1;
    for (int i=0; i<n; i++)
        ret *= base;
    return ret;
}
```

When n = $10^{10}$ it took 44 seconds
long power2(int n) {
    long ret;
    if (n == 0) return 1;
    if (n % 2 == 0) {
        ret = power2(n/2);
        return ret * ret;
    } else {
        ret = power2((n-1)/2);
        return base * ret * ret;
    }
}

When $n = 10^{19}$ it took < 1 second
Runtime analysis

• **First** algorithm $O(n)$

• **Second** algorithm $O(\log n)$

• We can apply the second algorithm to the Fibonacci problem: $\text{fib}4()$ has the following data

<table>
<thead>
<tr>
<th>n</th>
<th>$10^8$</th>
<th>$10^9$</th>
<th>$10^{10}$</th>
<th>$10^{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td># seconds</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Conclusion

- Recursion is powerful!

Source: https://i.ytimg.com/vi/Y2kenVSIV8U/maxresdefault.jpg
Recursion

Examples: From Textbook

1. Largest Number / Cumulative Sum
2. Greatest common divisor (GCD)
3. Log
4. Power
5. Many problems on List ADT

All these problems can be solved using either recursive or iterative algorithms.
Famous Quotations

• To err is human, to forgive divine.
  • Alexander Pope, An Essay on Criticism,
  • English poet and satirist (1688 - 1744)

• To iterate is human, to recurse, divine.
  • L. Peter Deutsch, computer scientist, or ....
  • Robert Heller, computer scientist, or ....
  • unknown ....
Objectives

• Thinking recursively

• Tracing execution of a recursive method

• Writing recursive algorithms

• Towers of Hanoi problem with recursion

• Backtracking to solve search problems, as in mazes
RECURSIVE THINKING
Recursive Thinking

• Recursion is:
  – A problem-solving approach, that can ...
  – Generate simple solutions to ...
  – Certain kinds of problems that ...
  – Would be difficult to solve in other ways

• Recursion splits a problem:
  • Into one or more simpler versions of itself
Recursive Thinking (cont.)

General Recursive Algorithm

if the problem can be solved directly for the current value of \( n \)

Solve it

else

Recursively apply the algorithm to one or more problems involving smaller values of \( n \)

Combine the solutions to the smaller problems to get the solution to the original problem
Examples
Recursive Algorithm for Finding the Length of a String

if the string is empty (has no characters) The length is 0

else

3. The length is 1 plus the length of the string that excludes the first character
public static int size(String str) {
    if (str == null || str.equals(""))
        return 0;
    else {
        int output = 1 + size(str.substring(1));
        return output;
    }
}
Tracing a Recursive function

```plaintext
3
size("ace")
  return 1 + size("ce");
2
  return 1 + size("e");
1
    return 1 + size(""");
0
```
public static void print_chars(String str) {
    if (str == null || str.equals("") ) {
        return;
    } else {
        System.out.println(str.charAt(0));
        print_chars(str.substring(1));
    }
}
Recursive Algorithm for Printing String Characters in Reverse Order

```java
public static void print_chars_rev(String str) {
    if (str == null || str.equals("")) {
        return;
    } else {
        print_chars_rev(str.substring(1));
        System.out.println(str.charAt(0));
    }
}
```
• What does this do?

```c
int mystery (int n) {
    if (n == 0)
        return 0;
    else
        return n * mystery(n-1);
}
```
Proving a Recursive Method Correct

• Recall **Proof by Induction**
• Prove the theorem for the base case(s): $n=0$
• Show that:
  – If the theorem is assumed true for $n$,  
    • Then it must be true for $n+1$
• Result: Theorem true for all $n \geq 0$
Proving a Recursive Method Correct (2)

• Recursive proof is similar to induction:

Show base case recognized and solved correctly

Show that

• If all smaller problems are solved correctly,
• Then original problem is also solved correctly

Show that each recursive case makes progress towards the base case $\Rightarrow$ terminates properly
The Stack and Activation Frames

• Java maintains a stack on which it saves new information in the form of an *activation frame*

• The activation frame contains storage for
  – function arguments
  – local variables (if any)
  – the return address of the instruction that called the function

• Whenever a new function is called (recursive or otherwise), Java pushes a new activation frame onto the stack
| Frame for size(""") | str: ""  
<table>
<thead>
<tr>
<th></th>
<th>return address in size(&quot;e&quot;)</th>
</tr>
</thead>
</table>
| Frame for size("e") | str: "e"  
|                   | return address in size("ce") |
| Frame for size("ce") | str: "ce"  
|                    | return address in size("ace") |
| Frame for size("ace") | str: "ace"  
|                      | return address in caller |

Run-time stack after all calls

Frame for size("e")  
str: "e"  
return address in size("ce")

Frame for size("ce")  
str: "ce"  
return address in size("ace")

Frame for size("ace")  
str: "ace"  
return address in caller

Run-time stack after return from last call
Run-Time Stack and Activation Frames

```
size("ace")
str: "ace"
"ace" == "" is false
return 1 + size("ce");

size("ce")
str: "ce"
"ce" == "" is false
return 1 + size("e");

size("e")
str: "e"
"e" == "" is false
return 1 + size(""");

size(""")
str: ""
"" == "" is true
return 0
```
Section 7.2

RECURSIVE DEFINITIONS OF MATHEMATICAL FORMULAS
Recursive Definitions of Mathematical Formulas

• Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms

• Examples include:
  – factorials
  – powers
  – greatest common divisors (gcd)
Factorial of $n$: $n!$

- The factorial of $n$, or $n!$ is defined as follows:

  $0! = 1$

  $n! = n \times (n - 1)! \ (\text{for } n > 0)$

- The base case: $n$ is equal to 0

- The second formula is a recursive definition
The recursive definition can be expressed by the following algorithm:

\[
\text{if } n \text{ equals } 0 \\
\quad \text{then } n! \text{ is } 1 \\
\text{else} \\
\quad n! = n \times (n - 1)!
\]

The last step can be implemented as:

```python
return n * factorial(n - 1);
```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
Recursive Algorithm for Calculating $x^n$

(double power(double x, int n) {
    if (n == 0)
        return 1;
    else if (n > 0)
        return x * power(x, n - 1);
    else
        return 1.0 / power(x, -n);
}
Recursive Algorithm for Calculating gcd (cont.)

```c
int gcd(int m, int n) {
    if (m < n)
        return gcd(n, m); // Transpose arguments
    else if (m % n == 0)
        return n;
    else
        return gcd(n, m % n);
}
```
Recursion and iteration are similar

**Iteration**
A loop repetition condition determines whether to repeat the loop body or exit from the loop

**Recursion**
the condition usually tests for a base case

You can always write an iterative solution to a problem that is solvable by recursion

**BUT**
A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read
Tail Recursion or Last-Line Recursion

- When recursion involves single call that is at the end...
- It is called **tail recursion** and it easy to make iterative

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

- It is a straightforward process to turn such a function into an iterative one
Iterative factorial function

```c
int factorial_iter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
        result = result * k;
    return result;
}
```
Efficiency of Recursion

Recursive method often slower than iterative; why?

- Overhead for loop repetition smaller than
- Overhead for call and return

If easier to develop algorithm using recursion,

Then code it as a recursive method:

- Software engineering benefit probably outweighs

...  

Reduction in efficiency

Don’t “optimize” prematurely!
Efficiency of Recursion (cont.)

• Memory usage
  – A recursive version can require significantly more memory than an iterative version because of the need to save local variables and parameters on a stack
Fibonacci Numbers (cont.)

Inefficient
An O(n) Recursive *fibonacci* function

```c
int fibonacci_start (int n) {
    return fibo(1, 0, n);
}
int fibo (int curr, int prev, int n) {
if (n <= 1)
    return curr;
else
    return fibo(curr+prev, curr, n-1);
}
```
Efficiency of Recursion: $O(n)$ fibonacci

```
fibonacci_start(5)
  return fibo(1, 0, 5);
    return fibo(1, 1, 4);
      return fibo(2, 1, 3);
        return fibo(3, 2, 2);
          return fibo(5, 3, 1);
```

Efficient
Examples

• Towers of Hanoi
Towers of Hanoi

• Move the three disks to a different peg, maintaining their order (largest disk on bottom, smallest on top, etc.)
  – Only the top disk on a peg can be moved to another peg
  – A larger disk cannot be placed on top of a smaller disk
Let’s Play

• [http://www.mathsisfun.com/games/towerofhanoi.html](http://www.mathsisfun.com/games/towerofhanoi.html)
## Problem Inputs

<table>
<thead>
<tr>
<th>Problem Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of disks (an integer)</td>
</tr>
<tr>
<td>Letter of starting peg: L (left), M (middle), or R (right)</td>
</tr>
<tr>
<td>Letter of destination peg: (L, M, or R), but different from starting peg</td>
</tr>
<tr>
<td>Letter of temporary peg: (L, M, or R), but different from starting peg and destination peg</td>
</tr>
</tbody>
</table>

## Problem Outputs

A list of moves
Recursive Algorithm for Towers of Hanoi

Recursive Algorithm for \( n \)-Disk Problem: Move \( n \) Disks from the Starting Peg to the Destination Peg

1. \textbf{if} \( n \) is 1
2. Move disk 1 (the smallest disk) from the starting peg to the destination peg

3. \textbf{else}

4. Move the top \( n - 1 \) disks from the starting peg to the temporary peg (neither starting nor destination peg)
5. Move disk \( n \) (the disk at the bottom) from the starting peg to the destination peg
6. Move the top \( n - 1 \) disks from the temporary peg to the destination peg
## Recursive Algorithm for Towers of Hanoi (cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>void show_moves(int n, char start_peg, char dest_peg, char temp_peg)</td>
<td>Builds a string containing all moves for a game with n disks on start_peg that will be moved to dest_peg, using temp_peg for temporary storage of disks being moved.</td>
</tr>
</tbody>
</table>
Backtracking

• Backtracking:
  – *Systematic* trial and error search for solution to a problem
  – *Example*: Finding a path through a maze

• In walking through a maze, probably walk a path as far as you can go
  – Eventually, reach destination or dead end
  – If dead end, must retrace your steps
  – Loops: stop when reach place you’ve been before

• Backtracking systematically tries alternative paths and eliminates them if they don’t work
Backtracking(2)

• If you never try exact same path more than once, and You try all possibilities
  – You will eventually find a solution path if one exists
• Problems solved by backtracking: a set of *choices*
• Recursion implements backtracking straightforwardly
  – Activation frame remembers choice made at that decision point
• A chess playing program likely involves backtracking
Finding a Path through a Maze

• Problem
  – Use backtracking to find a display the path through a maze
  – From each point in a maze you can move to the next cell in a horizontal or vertical direction if the cell is not blocked
Analysis (cont.)

(a) 

(b)