Recursion

COMS W1007
Introduction to Computer Science

Christopher Conway
26 June 2003
The Fibonacci Sequence

The Fibonacci numbers are:

1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots

We can calculate the $n$th Fibonacci number ($n \geq 2$) using the formula:

$$F_n = F_{n-1} + F_{n-2}$$
Recursion

- Defining a function in terms of itself is called *recursion*. We call a method that calls itself a *recursive method*.

- We don’t have to do anything special to write a recursive method in Java; any method can call itself.

- Each recursive call has its own distinct set of parameters and local variables. A recursive call is a separate entry on the execution stack.
The Basis Case

- In order for recursion to work correctly, every recursive method must have a *basis case*.

- The basis case is an input for which the method does not make a recursive call. The basis case prevents *infinite recursion*.

- It’s not enough for there to simply *be* a basis case; the values of the input must reliably *approach* the basis value.
The Fibonacci Sequence Redux

In the Fibonacci sequence, the basis cases are \( n = 0 \) and \( n = 1 \). Since the sequence is only defined for nonnegative integers \( n \), the recursive definition will always approach 0.

Basis cases:

\[
F_0 = 1 \\
F_1 = 1
\]

Recursive case (\( n \geq 2 \)):

\[
F_n = F_{n-1} + F_{n-2}
\]
The Factorial Function

The factorial function is:

\[ n! = n(n - 1) \cdots 1 \]

We can define the factorial function recursively as:

Basis case:

\[ 1! = 1 \]

Recursive case \((n > 1)\):

\[ n! = n \cdot (n - 1)! \]
The Towers of Hanoi

The Towers of Hanoi problem is to move a stack of plates from the first post to the third. You may only move one plate at a time and a larger plate cannot be stacked on top of a smaller one.
The Towers of Hanoi

Step 1:

Step 2:
The Towers of Hanoi

Step 3:

Step 4:
The Towers of Hanoi

Step 5:

Step 6:
The Towers of Hanoi

Step 7:
The Towers of Hanoi

The Towers of Hanoi is a classic example of a recursive problem. To solve it for $n$ plates:

1. Move $n - 1$ plates from the first post to the extra post.
2. Move the largest plate to the destination post.
3. Move $n - 1$ plates from the extra post to the destination.

The basis case (1 plate) is trivial.
Merge Sort

Input: A list of numbers $X_1, X_2, \ldots, X_n$ and a range $i..j$ to sort. ($i$ and $j$ are initially 1 and $n$, respectively).

Output: A list in ascending order.

1. If $i = j$, goto Step 6.
2. $k := (i + j) \div 2$.
3. $Y := \text{sort } X_{i..k}$.
4. $Z := \text{sort } X_{k+1..j}$.
5. Merge $Y$ with $Z$ into $X'$.
6. Return sorted list $X'$. 
Merge Sort: Example

5 9 2 1 6
5 9 2 1 6
5 9 2 1 6
5 9 2 1 6
5 9 2 1 6
5 9 2 1 6
5 9 1 2 6
5 9 1 2 6
5 9 1 2 6
5 9 1 2 6
5 9 1 2 6
1 2 5 6 9
Merge Sort: Efficiency

From the top down, a merge sort of a list of length \( n \) will:

- Merge two lists of length \( n/2 \): approximately \( n \) operations.
- Merge four lists of length \( n/4 \): \( n \) operations.
- Merge eight lists of length \( n/8 \): \( n \) operations.
- And so on, until we have \( n \) lists of length one: \( n \) operations.

If each step takes \( n \) operations, the question is: how many steps until we reach the basis case?
Merge Sort: Efficiency, 2

- If we continually divide \( n \) by 2, it takes \( \lfloor \log_2 n \rfloor \) steps to reach 1.

- Since we use \( \log_2 n \) a lot in computer science, we like to abbreviate it \( \lg n \).

- Merge sort performs \( n \) operations in every one of \( \lfloor \lg n \rfloor \) steps. The running time is:

\[
T(n) \approx n \lg n
\]

- Recall that the other sorts we studied took approximately \( n^2 \) operations. \( n \lg n \) is much better than \( n^2 \). (Compare them for \( n = 100 \) or \( n = 1000 \).)
Binary Search

Input: A sorted list $X_1, X_2, \ldots, X_n$ and a number to find $a$.

Output: A boolean value $Found$ indicating whether $a$ is contained in $X$.

1. $Found := 0, \ i := 1, \ j := n.$
2. $k := (i + j) \div 2.$
3. If $j \leq i$, go to Step 6.
4. If $X_k > a$, $j := k - 1$, go to Step 2.
5. If $X_k < a$, $i := k + 1$, go to Step 2.
6. If $X_k = a$, $Found := 1$. 
Binary Search: Efficiency

- Each step of the binary search divides the list in 2. The number of steps it will take to find a number in a list of length $n$ is:

$$T(n) \approx \log_2 n$$

- The log function grows quite slowly:

$$\log 100 \approx 5$$

$$\log 1,000 \approx 10$$

$$\log 1,000,000 \approx 20$$
Calculating Fibonacci Numbers

Consider a recursive method for calculating the $n$th Fibonacci number:

```java
int fibo(int n) {
    if( n==0 || n==1 )
        return 1 ;
    else
        return fibo(n-1) + fibo(n-2) ;
}
```

What is the running time of `fibo` on an input $n$?
Calculating Fibonacci Numbers, 2

From the top down, a call to \texttt{fibo} will:

- Add the result of two recursive method calls: 1 operation.
- Each recursive call will add the value of two further recursive calls (4 in all): 2 operations.
- Each of those calls will add the value of two further recursive calls (8 in all): 4 operations.
- And so on, until we reach \texttt{fibo(1)} and \texttt{fibo(0)}.

It will take \( n - 1 \) steps to reach the basis case.
Calculating Fibonacci Numbers, 3

The running time of \texttt{fibo} grows exponentially with \( n \):

\[
T(n) = 1 + 2 + 4 + \cdots + 2^{n-2}
\]

\[
= \sum_{i=0}^{n-2} 2^i
\]

\[
= 2^{n-1} - 1
\]

\[
\approx 2^n
\]

This is bad. Exponential growth is worse than \( n^2 \). In fact, it's worse than \( n^c \) for any \( c \).
Iteration vs. Recursion

Consider an iterative method for calculating Fibonacci numbers:

```c
int fibo2(int n) {
    int n = 1, n2 = 1;
    for( int i=2 ; i < n ; i++ ) {
        int tmp = n2;
        n2 = n;
        n = tmp + n2;
    }
    return n;
}
```
Iteration vs. Recursion: 2

- `fibo2` performs one addition for each number in the sequence, from 2 to \( n \). Thus, it is linear in \( n \).

\[ T(n) \approx n \]

- Recursion is not always the best solution to a problem. Even when the problem itself is defined recursively.

- We can usually solve a problem iteratively (i.e., using loops) or recursively. Which one we choose depends on the particular problem and personal taste.
Orders of Magnitude

• When we say the running time of an algorithm is approximately $f(n)$, what we really mean is it is on the same order of magnitude as $f(n)$.

• We express orders of magnitude using the notation $\Theta(f(n))$. $T(n) = \Theta(f(n))$ means that an algorithm grows neither faster nor slower than $f(n)$.

• The orders of magnitude are related as follows:

\[ c \preceq \lg n \preceq n \preceq n \lg n \preceq n^c \preceq c^n \]

where $c$ is a constant.
Orders of Magnitude, 2

- Constant-time algorithms ($\Theta(1)$) are as good as it gets. That means we can calculate the result in a fixed number of steps, irregardless of the input.

- Exponential algorithms ($\Theta(c^n)$) are just about as bad as it gets. We call exponential algorithms *intractable*—it is not practical to solve them for anything but very small inputs.

- Quadratic and higher polynomial algorithms ($\Theta(n^c)$) are tractable but slow. Linear ($\Theta(n)$), logarithmic ($\Theta(lg\ n)$) and linear-logarithmic ($\Theta(n\ lg\ n)$) algorithms are what we shoot for.