Recursion continued
Search Algorithms

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, February 15, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- ✓ Silently written & called functions
- ✓ Pointers & dynamic allocation
- ✓ Managing resources in a class
- ✓ Templates
- ✓ Containers & iterators
- ✓ Error handling
- ✓ Introduction to exceptions
- ✓ Introduction to Linked Lists

Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- ✓ Testing
- ✓ Some principles

DONE

Unit Overview
Recursion & Searching

Major Topics

- (part) Introduction to Recursion
- Search Algorithms
- Recursion vs. Iteration
- Eliminating Recursion
- Recursive Search with Backtracking

Review
Introduction to Recursion

We looked at a recursive implementation of a function to return $1 + 2 + \ldots + n$, given n.

Some points to remember:

- Recursive code must have a **base case**. And every call to the recursive code must eventually reach a base case.
- **Recurrence relations** often turn naturally into recursive code.

$f(n) = f(n-1) + n \longrightarrow$ `return sumUpTo(n-1) + n;`

Review
Recursion: Sum Example — Coding

Now we write the actual code:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Recursive.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    if (n == 0)
        return 0;
    else
        return sumUpTo(n-1) + n;
}
```

How do we know we can make the recursive call?

- Hint: When we call a function, we must satisfy its preconditions.

Review
Sum Example — Invariants

We know we can make the recursive call because we have an **invariant** that makes the preconditions for the call true:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Recursive.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    if (n == 0)
        return 0;
    else  // Invariant: n >= 1. (Therefore n-1 >= 0.)
        return sumUpTo(n-1) + n;
}
```

Here we have an invariant that says *n* ≥ 0 (the precondition).

Here we leave if *n* is exactly 0.
Result: If we stay, then *n* ≥ 1.

This is the precondition for **this** function call.

Introduction to Recursion
Sum Example — Iterative Version

Often we do not really need recursion:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; ++i)
        sum += i;
    return sum;
}
```

This uses **iteration** (a loop) instead.

Introduction to Recursion
Sum Example — Formula Version

And sometimes there is a not-so-obvious way to do things *much* faster:

```
// sumUpTo
// Given n, return sum of integers 1 to n.
// Pre: n >= 0.
// Post: Return == 1 + ... + n.
int sumUpTo(int n)
{
    return n * (n+1) / 2;
}
```

Review: Search Algorithms
Binary Search Example — Method

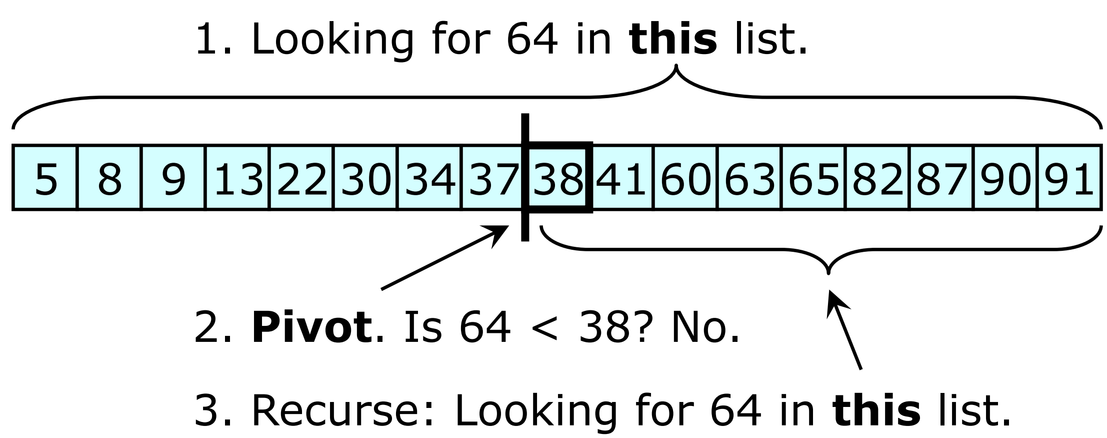
Binary Search is an algorithm to find a given **key** in a **sorted list**.

- Here, *key* = thing to search for. Often there is associated data.
- In computing, *sorted* = in (some) order.

Procedure

- Pick an item in the middle: the **pivot**.
- Use this to narrow search to top or bottom half of list. Recurse.

Example: Binary Search for 64 in the following list.



Review: Search Algorithms
Binary Search Example — Four Questions

How can you define the problem in terms of a smaller problem of the same type?

- Look at the middle of the list. Then recursively search the top or bottom half, as appropriate.

How does each recursive call diminish the size of the problem?

- It cuts the size of the list in half (roughly).

What instance of the problem can serve as the base case?

- List size is 0 or 1.

As the problem size diminishes, will you reach this base case?

- Yes.
  - A list cannot have negative size.

Review
Recursion vs. Iteration

The **Fibonacci numbers** are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …

- To get the next Fibonacci number, add the two before it.

They are defined formally as follows:

- We denote the nth Fibonacci number by F_n (n = 0, 1, 2, …).
- $F_0 = 0$.
- $F_1 = 1$.
- For $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$.

Another recurrence relation

As before, recurrence relations often translate nicely into recursive algorithms.

TO DO:

- Write a recursive function to compute a Fibonacci number: given n, compute F_n.

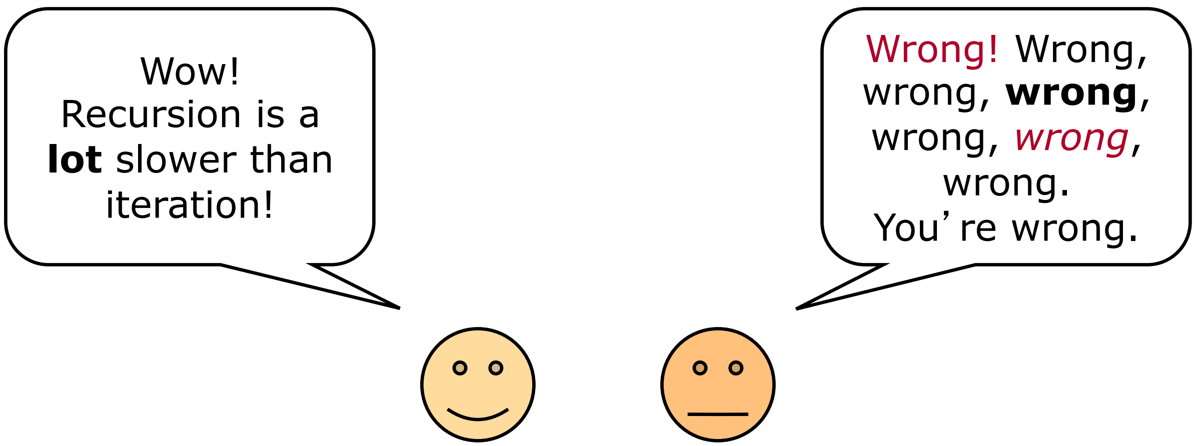
Recursion vs. Iteration continued
Fibonacci Numbers — Problem

For high-ish values of *n* (above 45, say) function **`fibo`** in **`fibo1.cpp`** is **extremely** slow.

- What can we do about this?

TO DO:

- Rewrite the Fibonacci computation in a fast iterative form.



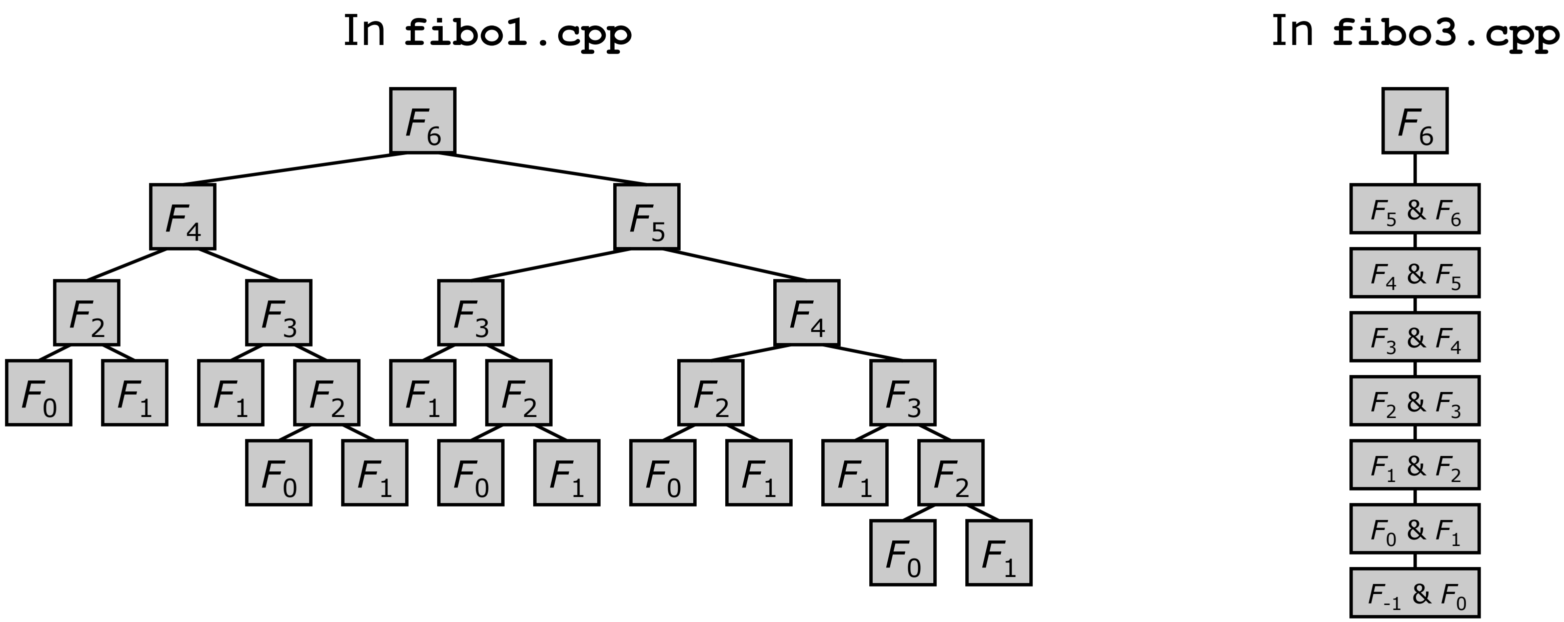
TO DO

- Figure out how to do a fast *recursive* version. Write it.

Recursion vs. Iteration
Fibonacci Numbers — Lessons [1/2]

Choice of algorithm can make a **huge** difference in performance.

Recursive calls made to compute F_6.



As we will see, data structures can make a similar difference.

Recursion vs. Iteration
Fibonacci Numbers — Lessons [2/2]

Some algorithms have natural implementations in both **recursive** and **iterative** form.

- *Iterative* means making use of loops.

A **`struct`** can be used to return two values at once.

- The template **`std::pair`** (declared in **`<utility>`**) can be helpful.

Sometimes we have a **workhorse** function that does most of the processing and a **wrapper** function that is set up for convenient use.

- Often the wrapper just calls the workhorse for us.
- This is common when we use recursion, since recursion can place inconvenient restrictions on how a function is called.
- We have seen this in another context. Remember **`toString`** and **`operator<<`** in the package from Assignment #1.