Recursive Search with Backtracking

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, October 2, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

Unit Overview
Recursion & Searching

Major Topics

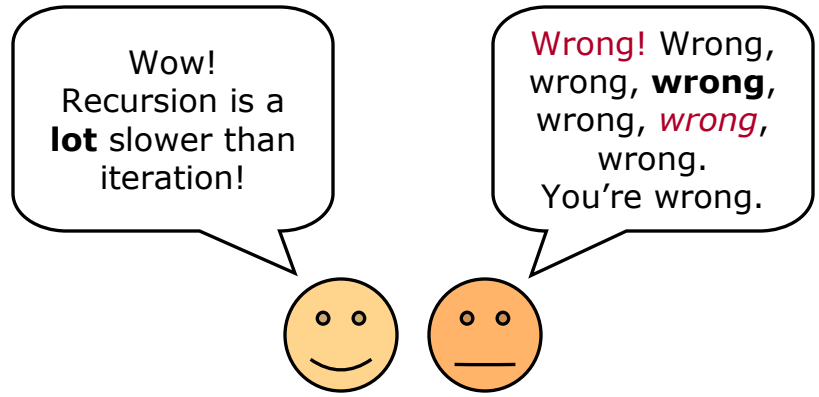
- ✓ Introduction to Recursion
- ✓ Search Algorithms
- ✓ Recursion vs. Iteration
- ✓ Eliminating Recursion
- Recursive Search with Backtracking

Review
Recursion vs. Iteration [1/3]

Five Fibonacci number computation programs.

- **`fibo1.cpp`**: Recursive. ← *Very* slow.
  Uses "obvious" algorithm.
- **`fibo2.cpp`**: Iterative. ← *Much* faster.
- **`fibo3.cpp`**: Inspired by previous. Recursive. Returns 2 values, instead of 1. Added wrapper.
- **`fibo4.cpp`**: Recursive memoizing.
  - *Memoizing* (which we do not really cover) is saving values of a function, to be returned when it is called with the same parameters. A special case of *caching*.
- **`fibo5.cpp`**: Uses a formula.

Also much faster.

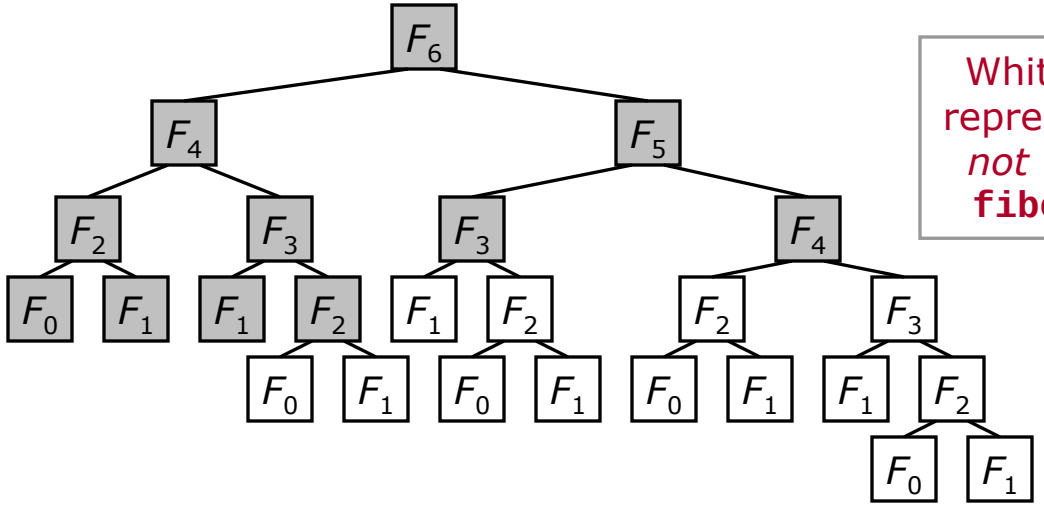


Review
Recursion vs. Iteration [2/3]

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

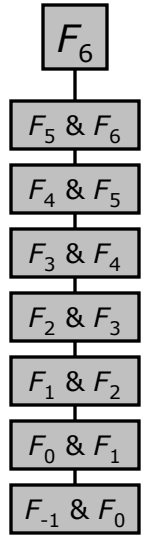
Recursive calls made to compute F_6.

In **`fibo1.cpp`**



White boxes represent calls *not* made in **`fibo4.cpp`**.

In **`fibo3.cpp`**



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

Review
Recursion vs. Iteration [3/3]

Two factors can make recursive algorithms inefficient.

- **Inherent inefficiency of some recursive algorithms**
  - However, there are efficient recursive algorithms (**`fibo3.cpp`**).
- **Function-call overhead**
  - Making all those function calls requires work: saving return addresses, creating and destroying automatic variables.

And recursion has another problem.

- **Memory-management issues**
  - Memory for automatic variables is allocated in a way that does not allow for normal error handling. Making too many recursive calls will cause stack overflow (resulting in a crash — or worse).
  - When we use iteration, we can manage memory ourselves. This is more work, but it also allows us to handle errors properly.

These two are important regardless of the recursive algorithm used.

Review
Eliminating Recursion

Every recursive function can be rewritten as an iterative function that uses essentially the same algorithm.

- The system helps you do recursion by providing a **Stack**, used to hold return addresses for function calls, and values of automatic local variables.
- We can eliminate recursion (convert it to iteration) by mimicking the system's method of handling recursive calls using Stack frames.
- This "brute force" method is primarily of theoretical interest. When eliminating recursion, it is usually better to do some thinking.
- We will look at this method again when we study Stacks.

Tail recursion is when the recursive call is the last operation a function does.

- Tail recursion is easy to eliminate.
- Some compilers (not C++) do this automatically: **tail call optimization**.

Recursive Search with Backtracking
Introduction — Backtracking

In most of the programming you have done, you have probably proceeded directly toward our goal.

- Work never had to be undone.
- But what if it does …

Sometimes we **search** for a solution to a problem.

- When we search for solutions, we may hit "dead ends" that do not work.
- Then we need to restore the program to a previous state. This is called **backtracking**.

Recursion is often a convenient technique for search with backtracking.

- However, such recursive programming can require rather different ways of thinking from "normal" recursive programming.

Recursive Search with Backtracking
Introduction — Partial Solutions

Recursive solution search works well when we have a notion of a **partial solution**.

- Each recursive call says, "Look for full solutions based on this partial solution."
- The function attempts to build more complete solutions based on the partial solution it was given.
- For each possible more complete solution, a recursive call is made.
- We usually have a wrapper function, so that the client does not need to deal with partial solutions.

In a recursive solution search, to backtrack, we often simply return from a function.

Introduction — No-Backtracking Diagram

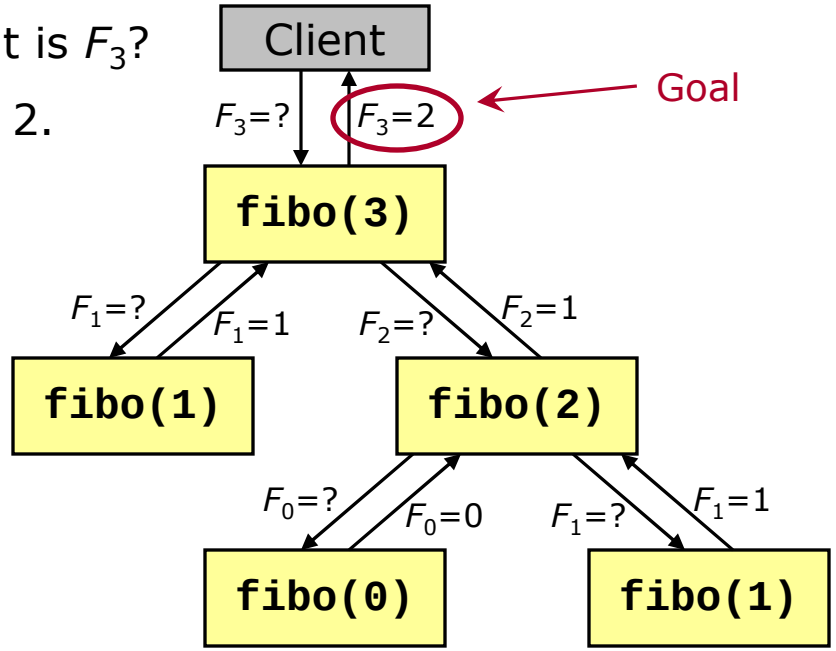
In the recursion we studied earlier:

- A recursive call is a request for information (or action).
- The return sends the information back (if any).

The diagram below shows the information flow in the first version of **`fibo`**.

Q: What is F_3?

A: $F_3 = 2$.



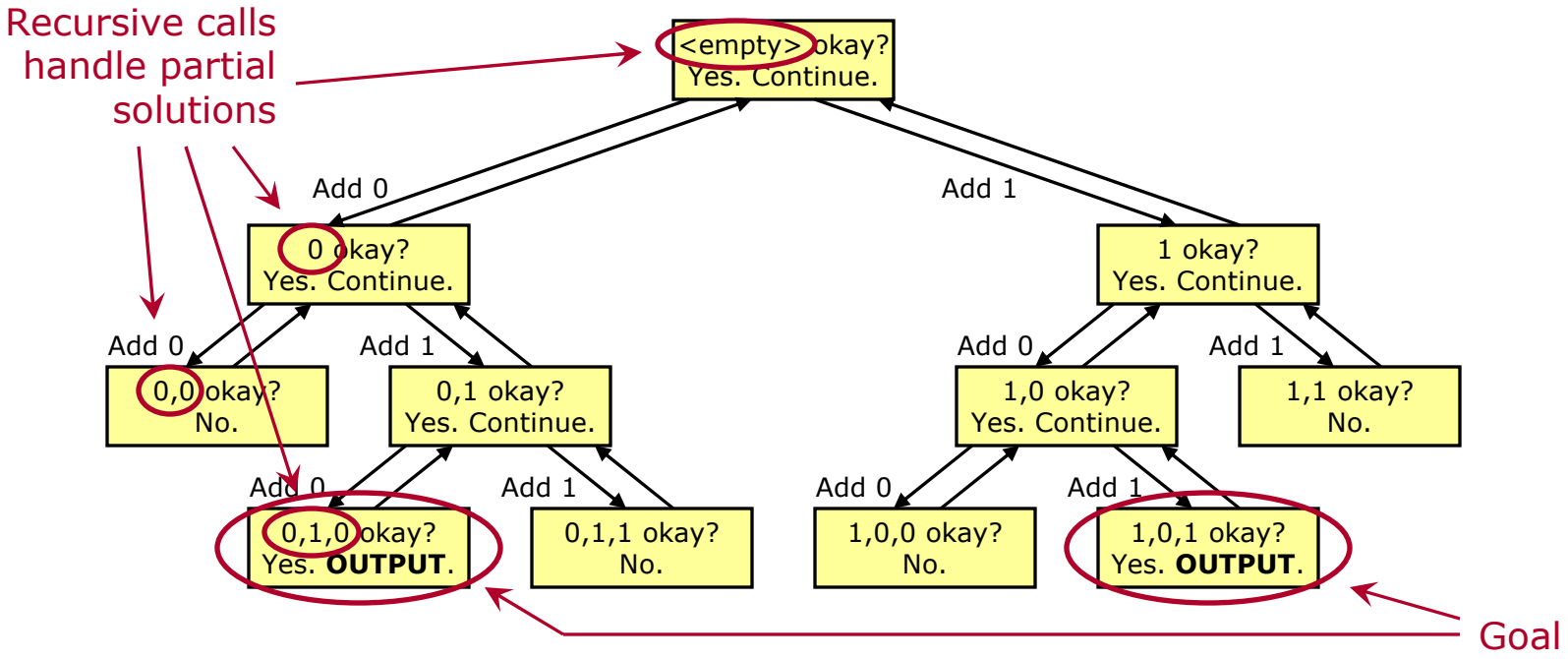
Recursive Search with Backtracking
Printing Solutions — Diagram

In recursive search with backtracking:

- A recursive call means "continue with the proposed partial solution".
- The return means "backtrack".

The diagram below illustrates a search for 3-digit sequences with digits in {0, 1}, in which no two consecutive digits are the same.

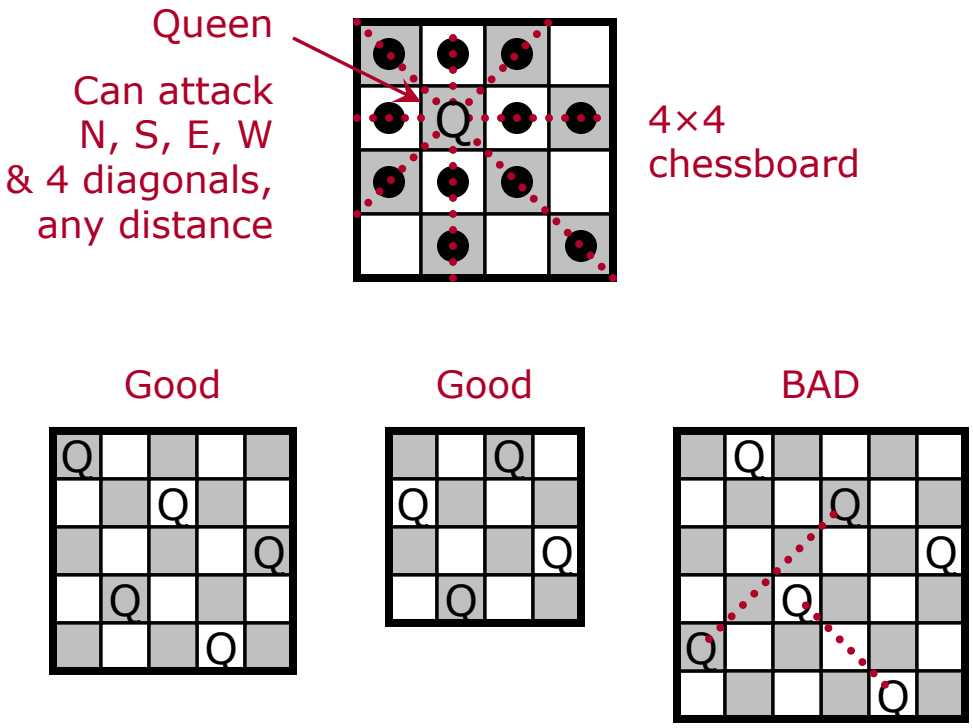
On finding a solution, we can stop. Or continue, finding **all solutions**.



Recursive Search with Backtracking
Printing Solutions — *n*-Queens Problem

We now look at how to solve the ***n*-Queens Problem**.

- Place *n* queens on an *n* × *n* chessboard so that none of them can attack each other.



Recursive Search with Backtracking
Printing Solutions — How to Do It [1/4]

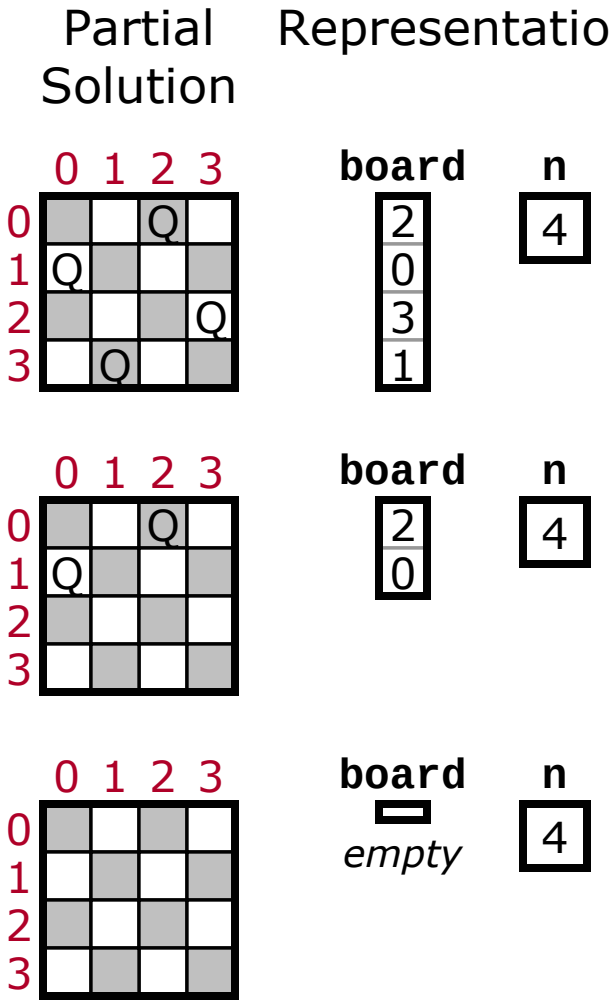
To Figure Out

- What is a **partial solution** for the problem we wish to solve? How should we represent a partial solution?
  - If possible, we should represent a partial solution in a way that makes it convenient to determine whether we have a full solution.
  - It is also nice if we can quickly determine whether we have a dead end.
- How should we output a full solution?

Recursive Search with Backtracking
Printing Solutions — How to Do It [2/4]

Representing a Partial Solution

- Number rows and columns 0 .. *n*–1.
- Two variables:
  - Variable **`board`** (**`vector`** of **`int`**s).
  - Variable **`n`** (**`int`**).
- Variable **`n`** holds the number of rows/columns in a full solution.
- Variable **`board`** holds the columns of the queens already placed, one per row.
- The size of **`board`** is the number of rows in which queens have been placed.



Recursive Search with Backtracking
Printing Solutions — How to Do It [3/4]

The Code

- **Nonrecursive wrapper function**
  - Creates an empty partial solution.
  - Calls the workhorse function with this partial solution.
- **Recursive workhorse function** is given a partial solution, prints all full solutions that can be made from it.
  - Do we have a full solution?
    - If so, output it.
  - Do we have a clear dead end?
    - If so, simply return.
  - Otherwise:
    - Make a recursive call for each way of extending the partial solution.

Note:
This part **might** not be necessary. Another way to handle dead ends is simply not to make any recursive calls when we get to this part.

Recursive Search with Backtracking
Printing Solutions — How to Do It [4/4]

Notes

- It is often convenient to have a separate function that checks the validity of a proposed way to extend a partial solution.
- When you backtrack make sure that you go back to the previous partial solution. Two ways to do this:
  - Each recursive call has its own copy of the current partial solution.
  - All use the same data. When backtracking, restore the previous state.

TO DO

- Write a recursive function to print solutions to the *n*-Queens Problem.

Done. See **nqueen.cpp**, *on the web page.*

Recursive Search with Backtracking
TO BE CONTINUED …

Recursive Search with Backtracking will be continued next time.