

Eliminating Recursion, cont'd

Recursive Search with Backtracking

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, February 22, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Recursion & Searching

Major Topics

- ✓ • Introduction to Recursion.
- ✓ • Search Algorithms.
- ✓ • Recursion vs. Iteration.
- 1/2✓ • Eliminating Recursion.
 - Recursive Search with Backtracking.

Review

Recursion vs. Iteration [1/3]

We wrote four versions of the Fibonacci number computation program.

- `fibonacci1.cpp`: A recursive version using the “obvious” algorithm. ← *Very slow.*
- `fibonacci2.cpp`: An iterative version.
- `fibonacci3.cpp`: Based on `fibonacci2.cpp`, a recursive version that returns two values, instead of just one. We added a wrapper function, to allow for a consistent interface.
- `fibonacci4.cpp`: A version that used a formula.

Each of these is *much* faster.

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 (`fibonacci3.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.
- Thus, we can eliminate recursion 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 generally best to put some thought into the job.
- We will look at this method again in a few weeks when we study Stacks.

Eliminating Recursion, cont'd

Tail Recursion [1/4]

A special kind of recursion is **tail recursion**.

- *Tail recursion* is when a recursive call is the last thing a function does.

Tail recursion is important because it makes the recursion → iteration conversion very easy.

- That is, we like tail recursion because it is easy to eliminate.
- In fact, tail recursion is such an obvious thing to optimize that some compilers automatically convert it to iteration.
 - Note: we are speaking generally here, not specific to C++.

Eliminating Recursion, cont'd

Tail Recursion [2/4]

For a void function, tail recursion looks like this:

```
void foo(TTT a, UUU b)
{
    ...
    foo(x, y);
}
```

For a function returning a value, tail recursion looks like this:

```
SSS bar(TTT a, UUU b)
{
    ...
    return bar(x, y);
}
```

Eliminating Recursion, cont'd

Tail Recursion [3/4]

The reason tail recursion is so easy to eliminate is that we **never need to return** from the recursive call to the calling function.

- Because there is nothing more for the calling function to do.
- Thus, we can replace the recursive call by something essentially like a `goto`.

Eliminating Tail Recursion

- Surround the function body with a big loop, as before.
- Replace the tail recursive call with:
 - Set parameters to their new values (and restart the loop — which happens automatically, since we are already at the end of the loop).
- There is no need to make any changes to non-recursive `return` statements, that is, the base case(s).

If the *only* recursive call in a function is at the end, then eliminating tail recursion converts the function into non-recursive form.

Eliminating Recursion, cont'd

Tail Recursion [4/4]

TO DO

- Eliminate the recursion in `binsearch2.cpp`.
 - First, modify function `binSearch` so that it has exactly one recursive call, and this is at the end of the function.
 - This should be easy.
 - Next, eliminate tail recursion.
 - This is a little trickier to think about, but very easy to do.

*Done. See `binsearch3.cpp`,
on the web page.*

*Done. See `binsearch4.cpp`,
on the web page.*

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.

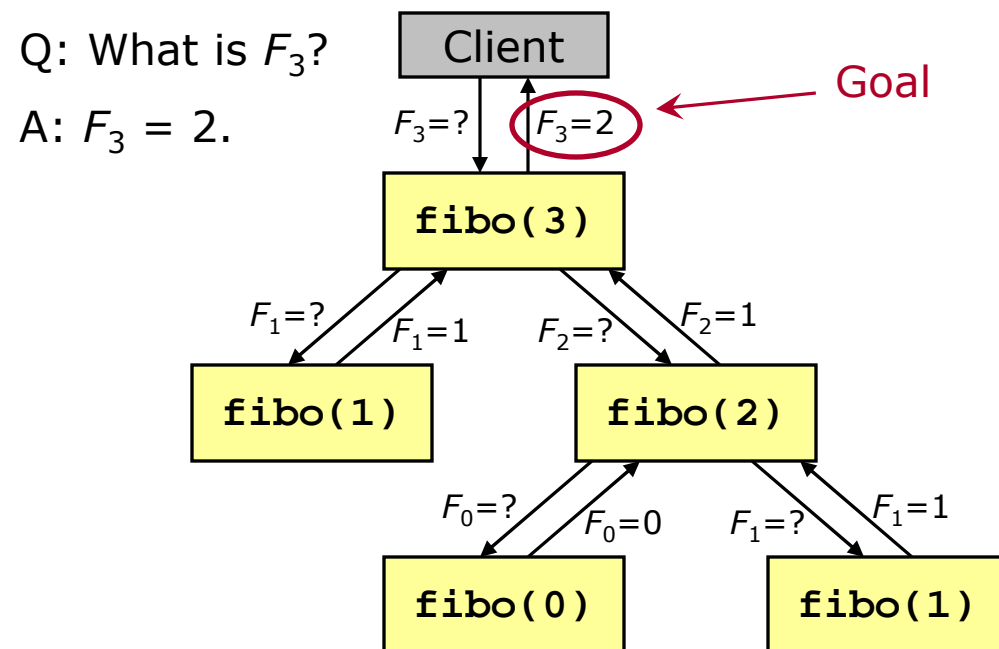
Recursive Search with Backtracking

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 `fibonacci`.



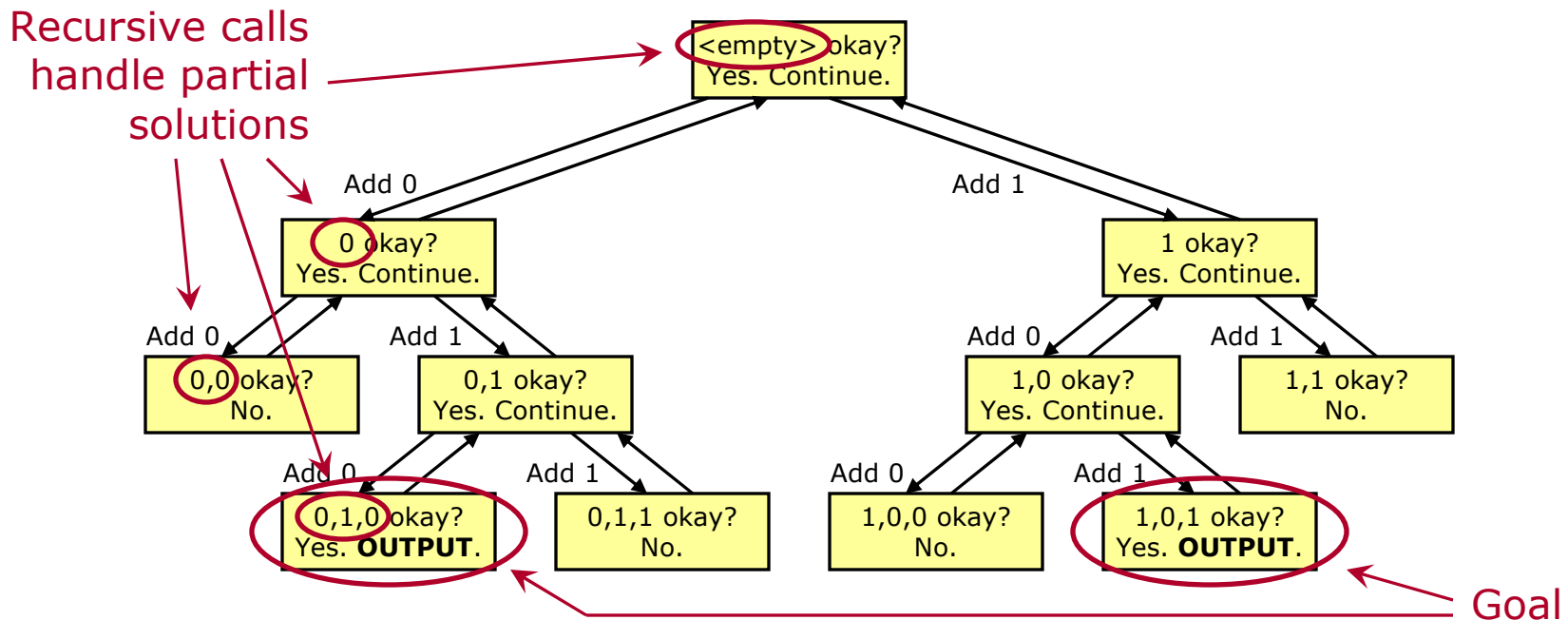
Recursive Search with Backtracking Printing Solutions — Diagram

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.

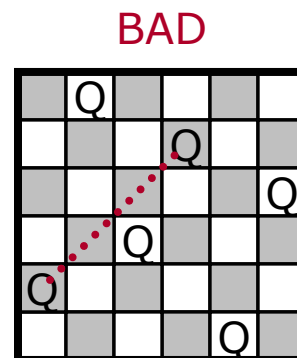
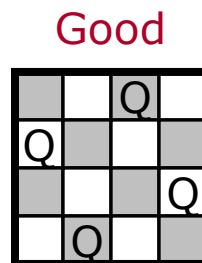
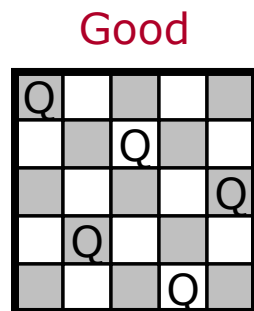
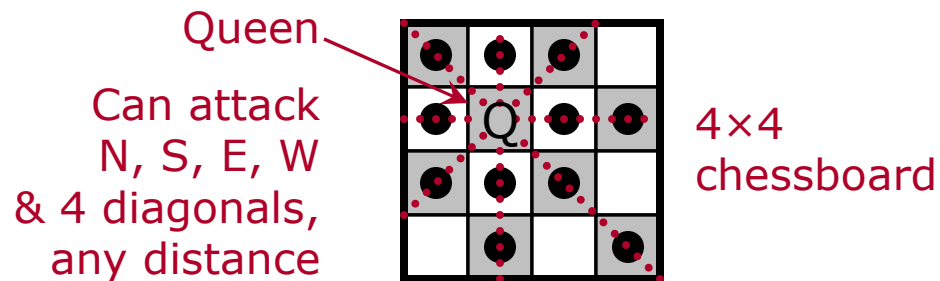
On finding a solution, we can stop. Or continue, finding **all solutions**.



Recursive Search with Backtracking Printing Solutions — n -Queens Problem

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

- Place n queens on an $n \times n$ 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

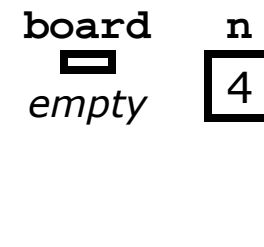
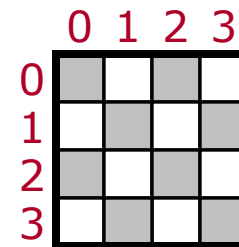
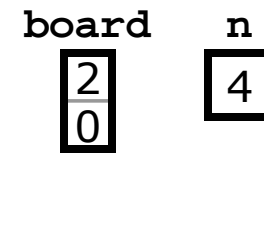
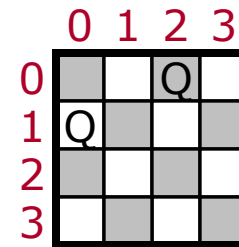
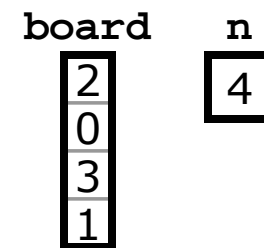
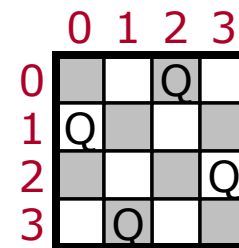
Printing Solutions — How to Do It [2/4]

Representing a Partial Solution

- Number rows and columns 0 .. $n-1$.
- Two variables:
 - Variable `board` (vector of `ints`).
 - 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.

Partial Solution

Representation



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 various ways 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.*