Recursive Search with Backtracking continued
Introduction to Analysis of Algorithms

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, February 22, 2013

Chris Hartman

Department of Computer Science

University of Alaska Fairbanks

cmhartman@alaska.edu

Based on material by Glenn G. Chappell

Unit Overview
Recursion & Searching

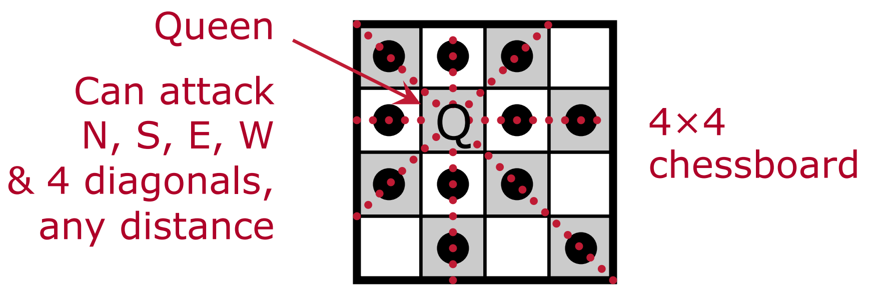
Major Topics

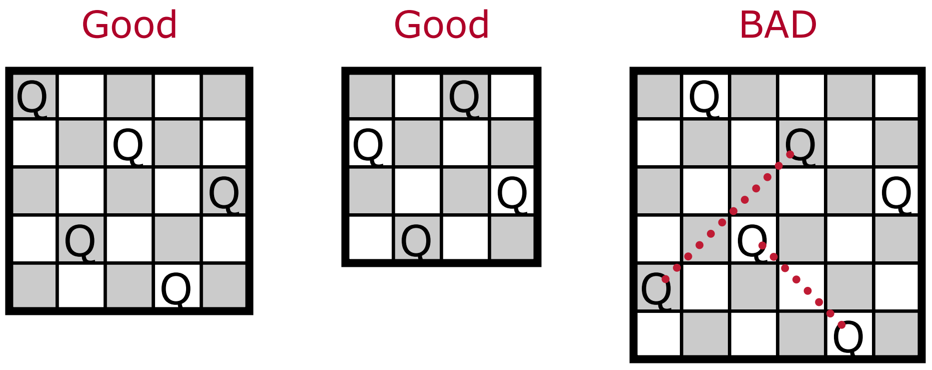
- ✓ Introduction to Recursion
- ✓ Search Algorithms
- ✓ Recursion vs. Iteration
- ✓ Eliminating Recursion
- (part) Recursive Search with Backtracking

Review
Recursive Search with Backtracking — Printing Solutions [1/3]

We looked 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.

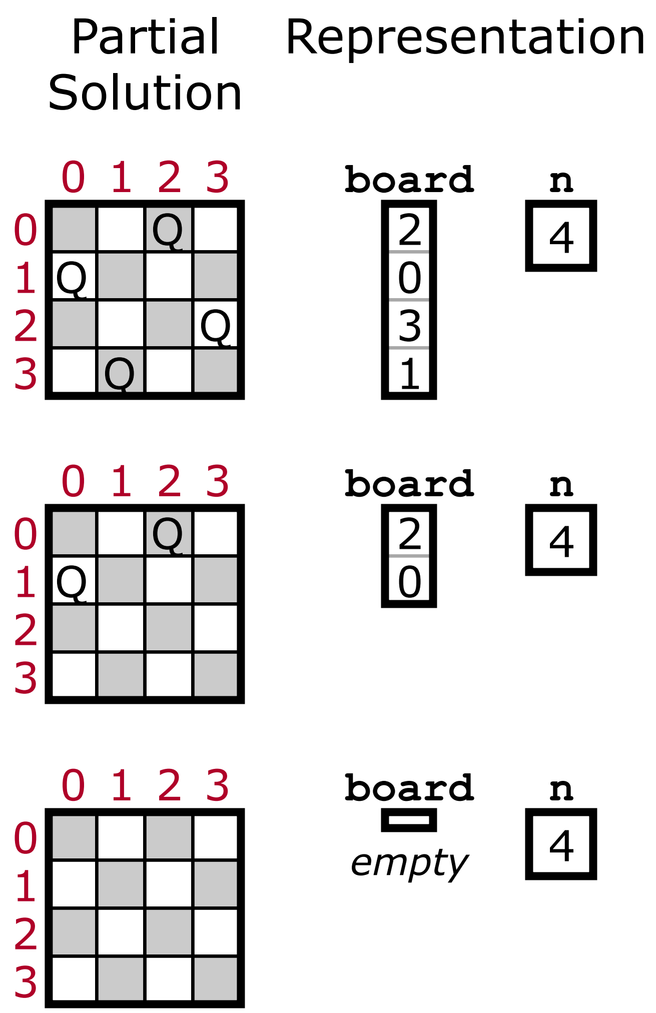




Review
Recursive Search with Backtracking — Printing Solutions [2/3]

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.



Review
Recursive Search with Backtracking — Printing Solutions [3/3]

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.

TO DO

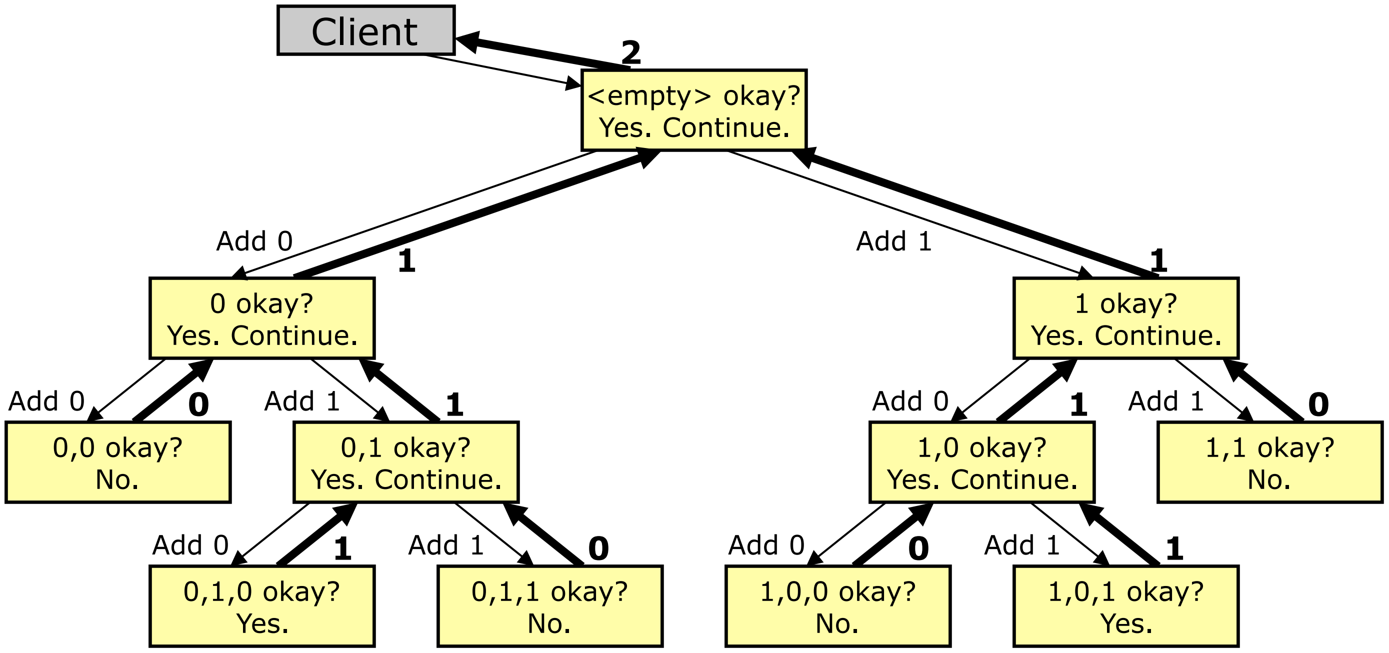
- Write a recursive function to print solutions to the *n*-Queens Problem.

Recursive Search with Backtracking Counting Solutions — Diagram

continued

We can use a similar approach to **count solutions**. Each recursive call returns the number of full solutions based on the given partial solution.

- Base Cases
  - “Found a solution” returns 1.
  - “Didn’t work” returns 0.
- Recursive Case
  - Make recursive calls, add their return values, and return the total.



Recursive Search with Backtracking
Counting Solutions — How to Do It

The Code

- **Nonrecursive wrapper function**
  - Creates an empty partial solution.
  - Calls the workhorse function with this partial solution.
  - Returns the return value of the workhorse function.
- **Recursive workhorse function** is given a partial solution, returns the number of full solutions that can be made from it.
  - Do we have a full solution?
    - If so, return 1.
  - Do we have a clear dead end?
    - If so, return 0.
  - Otherwise:
    - Set *total* to zero.
    - For each way of extending the current partial solution, make a recursive call, and add its return value to *total*.
    - Return *total*.

As before, this might be unnecessary.

Recursive Search with Backtracking
Counting Solutions — Write It

TO DO

- Modify the *n*-Queens code to **count** all possible non-attacking arrangements of *n* queens, instead of printing them.

Unit Overview
Algorithmic Efficiency & Sorting

We now begin a unit on algorithmic efficiency & sorting algorithms.

Major Topics

- Introduction to Analysis of Algorithms
- Introduction to Sorting
- Comparison Sorts I
- More on Big-*O*
- The Limits of Sorting
- Divide-and-Conquer
- Comparison Sorts II
- Comparison Sorts III
- Radix Sort
- Sorting in the C++ STL

During this unit will be the in-class Midterm Exam.

Introduction to Analysis of Algorithms
Efficiency [1/3]

What do we mean by an "efficient" algorithm?

- We mean an algorithm that **uses few resources**.
- By far the most important resource is **time**.
- Thus, when we say an algorithm is **efficient**, *assuming we do not qualify this further*, we mean that it can be executed **quickly**.

How do we determine whether an algorithm is efficient?

- Implement it, and run the result on some computer?
- But the speed of computers is not fixed.
- And there are differences in compilers, etc.

Is there some way to measure efficiency that does not depend on the system chosen or the current state of technology?

Introduction to Analysis of Algorithms
Efficiency [2/3]

Is there some way to measure efficiency that does not depend on the system chosen or the current state of technology?

- Yes!

Rough Idea

- Divide the tasks an algorithm performs into “steps”.
- Determine the maximum number of steps required for input of a given size. Write this as a formula, based on the size of the input.
- Look at the most important part of the formula.
  - For example, the most important part of “$6n \log n + 1720n + 3n^2 + 14325$” is “$n^2$”.

Next we look at this in more detail.

Introduction to Analysis of Algorithms
Efficiency [3/3]

When we talk about **efficiency** of an algorithm, without further qualification of what “efficiency” means, we are interested in:

- **Time** Used by the Algorithm
  - Expressed in terms of number of steps.
- How the **Size of the Input** Affects Running Time
  - Larger input typically means slower running time. How much slower?
- **Worst-Case** Behavior
  - What is the maximum number of steps the algorithm ever requires for a given input size?

To make the above ideas precise, we need to say:

- What is meant by a **step**.
- How we measure the **size** of the input.

These two are part of our **model of computation**.

Introduction to Analysis of Algorithms
Model of Computation

The **model of computation** used *in this class* will include the following definitions.

- The following operations will be considered a single **step**:
  - Built-in operations on fundamental types (arithmetic, assignment, comparison, logical, bitwise, pointer, array look-up, etc.).
  - Calls to client-provided functions (including operators). In particular, in a template, operations (i.e., function calls) on template-parameter types.
- From now on, when we discuss efficiency, we will always consider a function that is given a list of items. The **size** of the input will be the number of items in the list.
  - The "list" could be an array, a range specified using iterators, etc.
  - We will generally denote the size of the input by "*n*".

Notes

- As we will see later, we can afford to be *somewhat* imprecise about what constitutes a single "step".
- In a formal mathematical analysis of the properties and limits of computation, both of the above definitions would need to change.

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Definition

Algorithm *A* is *order* $f(n)$ [written $O(f(n))$] if

- There exist constants k and n_0 such that
- *A* requires **no more than** $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

We are usually not interested in the exact values of k and n_0. Thus:

- We don't worry much about whether some algorithm is (say) five times faster than another.
- We ignore small problem sizes.

Big-*O* is important!

- We will probably use it *every day* for the rest of the semester (the concept, not the above definition).

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Worst Case & Average Case

When we use big-*O*, unless we say otherwise, we are always referring to the **worst-case** behavior of an algorithm.

- For input of a given size, what is the **maximum** number of steps the algorithm requires?

We can also do average-case analysis. However, we need to say so. We also need to indicate what kind of average we mean. For example:

- We can determine the average number of steps required over all inputs of a given size.
- We can determine the average number of steps required over repeated applications of the same algorithm.

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Example 1, Problem

Determine the order of the following, and express it using “big-*O*”:

```
int func1(int p[], int n) // n is length of array p
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += p[i];
    return sum;
}
```

See the next slide.

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Example 1, Solution

I count 9 single-step operations in **`func1`**.

Strictly speaking, it is correct to say that **`func1`** is $O(4n+6)$. In practice, however, we always place a function into one of a few well-known categories.

ANSWER: Function **`func1`** is $O(n)$.

- This works with (for example) $k = 5$ and $n_0 = 100$.
- That is, $4n + 6 \le 5 \times n$, whenever $n \ge 100$.

What if we count "**`sum += p[i]`**" as one step? What if we count the loop as one?

- Moral: collapsing a **constant** number of steps into one step does not affect the order.
- This is why I said we can be *somewhat* imprecise about what a "step" is.

Operation	Times Executed
`int p[]`	1
`int n`	1
`int sum = 0`	1
`int i = 0`	1
`i < n`	$n + 1$
`++i`	n
`p[i]`	n
`sum += …`	n
`return sum`	1
TOTAL	$4n + 6$

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Scalability

Why are we so interested in the running time of an algorithm for **very large** problem sizes?

- Small problems are easy and fast.
- We expect more of faster computers. Thus, problem sizes keep getting bigger.
- As we saw with search algorithms, the advantages of a fast algorithm become more important at very large problem sizes.

Recall:

- “The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less.” — Nick Trefethen

An algorithm (or function or technique …) that works well when used with increasingly large problems & large systems is said to be **scalable**.

- Or, it **scales well**.
- This class is all about things that scale well.

This definition applies in general, not only in computing.

Introduction to Analysis of Algorithms
Order & Big-O Notation — Efficiency Categories

Know these!

An $O(1)$ algorithm is **constant time**.

- The running time of such an algorithm is essentially independent of the input.
- Such algorithms are rare, since they cannot even read all of their input.

An $O(\log_b n)$ [for some b] algorithm is **logarithmic time**.

- Again, such algorithms cannot read all of their input.
- As we will see, we do not care what b is.

An $O(n)$ algorithm is **linear time**.

- Such algorithms are not rare.
- This is as fast as an algorithm can be and still read all of its input.

An $O(n \log_b n)$ [for some b] algorithm is **log-linear time**.

- This is about as slow as an algorithm can be and still be truly useful (scalable).

An $O(n^2)$ algorithm is **quadratic time**.

- These are usually too slow for anything but very small data sets.

An $O(b^n)$ [for some b] algorithm is **exponential time**.

- These algorithms are *much* too slow to be useful.

$\uparrow$ Faster

Slower $\downarrow$

Notes

- Gaps between these categories are *not* bridged by compiler optimization.
- We are interested in the **fastest category** above that an algorithm fits in.
  - Every $O(1)$ algorithm is also $O(n^2)$ and $O(237^n + 184)$; but "$O(1)$" interests us most.
- **I will also allow $O(n^3)$, $O(n^4)$, etc.** However, we will not see these much.