Recursive Backtracking

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, September 25, 2024

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Recursion & Searching

Topics

- ✓ Arrays & Linked Lists
- ✓ Introduction to recursion
- ✓ Search algorithms I
- ✓ Recursion vs. iteration
- ✓ Search algorithms II
- ✓ Eliminating recursion
- ✓ Search in the C++ STL
- Recursive backtracking

Review

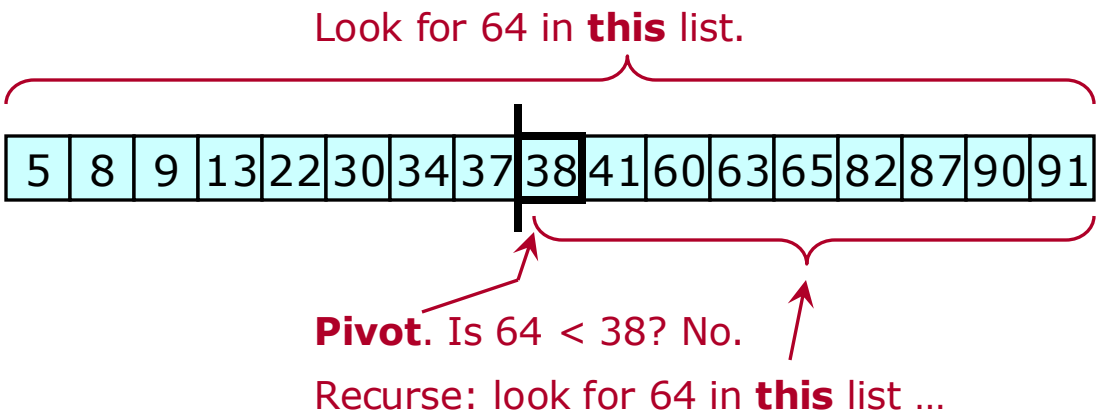
Review
Search Algorithms I/II [1/3]

The **Binary Search** algorithm finds a given **key** in a **sorted list**.

Procedure

- Pick an item in the middle of the list: the **pivot**.
- Compare the given key with the pivot.
- Using this, narrow search to top or bottom half of list. Recurse.

Example. Use Binary Search to search for 64 in the following list.

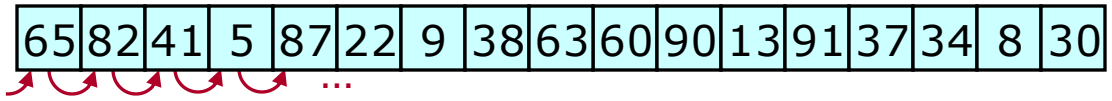


Review
Search Algorithms I/II [2/3]

The **Sequential Search** (a.k.a. **Linear Search**) algorithm finds a given key in a list—which is not required to be sorted.

Procedure

- Start from the beginning, checking each item, in order.
- If the desired key is the one being checked, then stop: FOUND.
- If we are past the end of the list, then stop: NOT FOUND.



Review
Search Algorithms I/II [3/3]

Binary Search is much faster than Sequential Search, so it can process much more data in the same amount of time.

Number of Look-Ups We Have Time For	Maximum List Size: Binary Search	Maximum List Size: Sequential Search
1	1	1
2	2	2
3	4	3
4	8	4
10	512	10
20	524,288	20
40	549,755,813,888	40
k	*Roughly* 2^k	*k*

"The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less." [Lloyd N. Trefethen]

Eliminating Recursion

It can sometimes be helpful to **eliminate recursion**—converting it to iteration.

We can eliminate recursion by mimicking the call stack. This method always works, but it is rarely used; better results are usually gotten by *thinking* about the problem to be solved.

More on this method when we cover *Stacks*.

If a recursive call is a **tail call** (the last thing a function does), then we have **tail recursion**.

Eliminating tail recursion is easy and practical.

See `binsearch2.cpp`, `binsearch3.cpp`, `binsearch4.cpp`.

Review
Search in the C++ STL

The STL includes four function templates that do Binary Search:

- `std::binary_search`
- `std::lower_bound`
- `std::upper_bound`
- `std::equal_range`

All are called the same way, but they return different information. All allow for optional specification of a custom comparison.

The STL also includes Sequential Search:

- `std::find`

This returns an iterator to the first item found, or the end iterator for the given range, if nothing is found.

Some STL containers, like `std::map`, have their own search by key: a member function `find`.

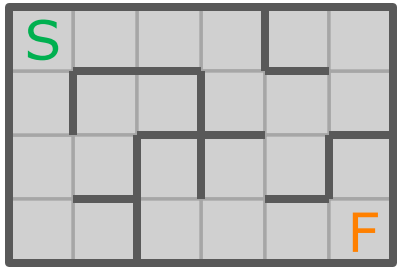
More on this when we cover *Tables*.

Recursive Backtracking

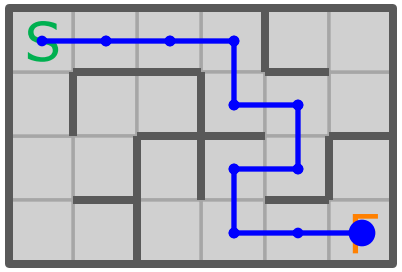
Basics — Backtracking [1/2]

In most programming, we proceed directly toward a goal. Work never needs to be undone. But what if it does?

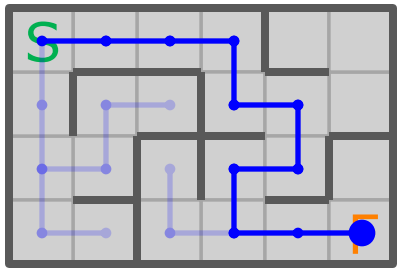
By way of illustration, consider a maze with specified **start** and **finish** squares.



The goal is to find a path through the maze that goes from the **start** to the **finish**, without passing through any walls. This is a *solution*.



To find a solution, we begin at the **start** square and try moving in various directions. If we hit a dead end, then we must *backtrack*.



Basics — Backtracking [2/2]

Now we cover a different kind of search—one that proceeds along the lines just discussed.

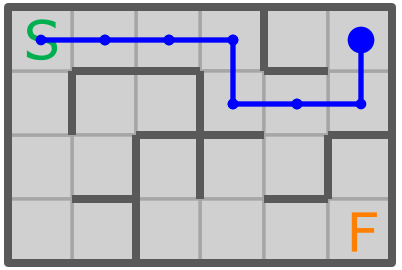
Sometimes we **search** for a solution to a problem.

- We attempt to build up a solution bit by bit.
- We might need to undo some work and try something different.
- Restoring to a previous state is called **backtracking**.

It is often convenient to implement backtracking using recursion. However, such recursive programming can require different ways of thinking from the recursion we have discussed so far.

Recursive Backtracking
Basics — Partial Solutions

Recursive solution search works well when we have a notion of a **partial solution**: something that *might* be a step on the way to a finished **full solution**.

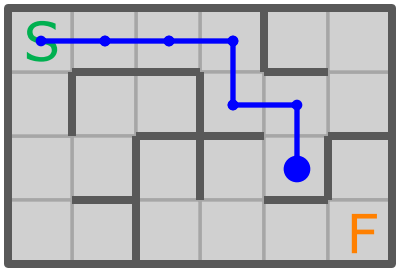


Partial Solution

Each recursive call says, "Look for full solutions based on this partial solution."

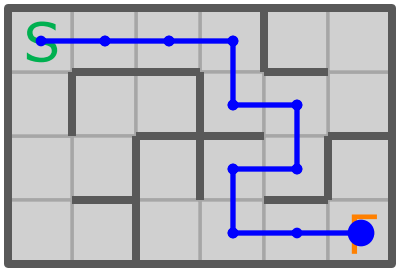
For each possible more complete solution, a recursive call is made.

To backtrack, we might simply return from a function.



Partial Solution

We usually have a wrapper function, so client code does not deal with partial solutions.



Full Solution

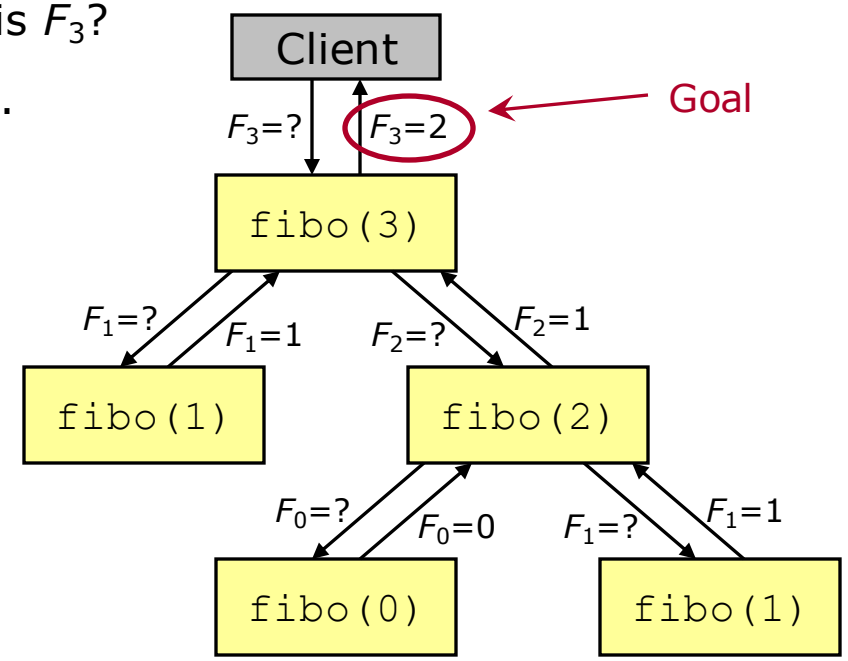
In the recursion we studied earlier:

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

The diagram below shows the information flow in `fibo_first.cpp`.

Q. What is F_3?

A. $F_3 = 2$.



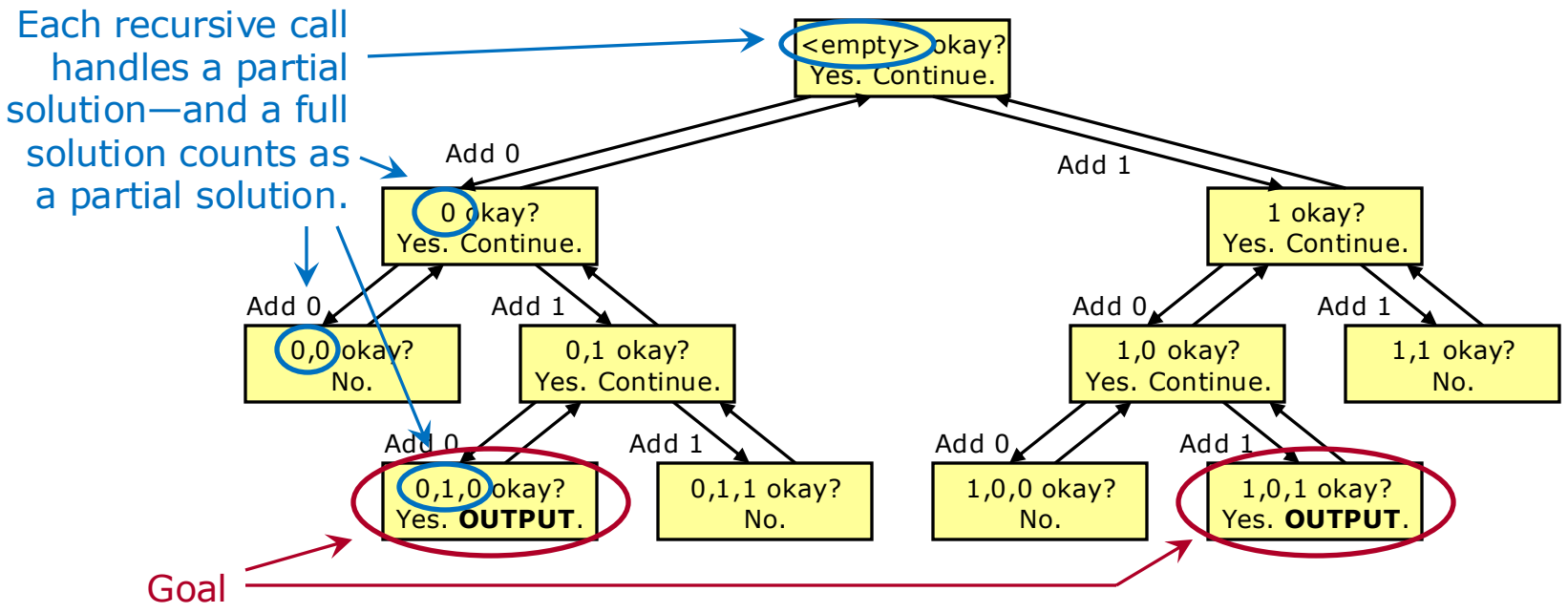
Recursive Backtracking
Basics — Backtracking Diagram

In recursive backtracking:

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

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

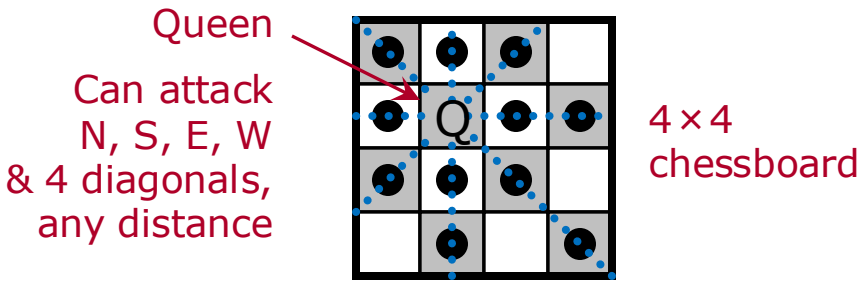
On finding a solution, stop. Or—continue, finding all solutions.

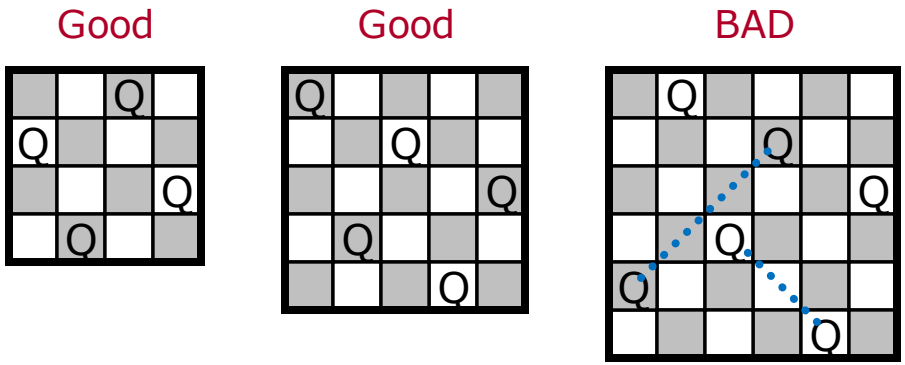


Recursive Backtracking
Example — Description

We now look at how to solve the ***n*-Queens Problem** using recursive backtracking search.

Problem. Place n queens on an $n \times n$ **chessboard** so that none of them can attack each other.





Recursive Backtracking
Example — How to Do It [1/4]

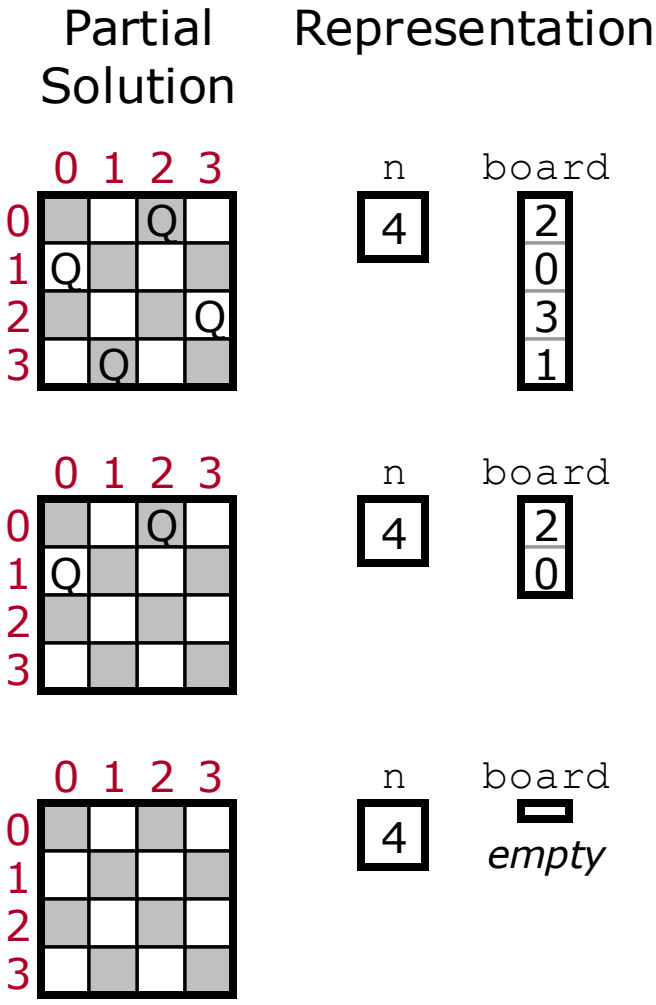
To Figure Out

1. What is a partial solution for the problem we wish to solve?
2. How should we represent a partial solution in a program?
   - 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.
3. How should we output a full solution?

1. Partial Solution. A nonattacking placement of k queens on the first k rows of an $n \times n$ chessboard , where $0 \leq k \leq n$.

2. Representing a Partial Solution

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



3. Outputting a Full Solution. I will simply print the array items.

Recursive Backtracking
Example — How to Do It [3/4]

The Code

- **Nonrecursive wrapper function**
  - Create an empty partial solution.
  - Call 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.

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 Backtracking
Example — How to Do It [4/4]

Notes

- We often need to check the validity of a proposed way to extend a partial solution. It can be convenient to have a separate function that does this checking.
- When backtracking, we need to make sure we 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, undo any changes made.

Recursive Backtracking
Example — CODE

TO DO

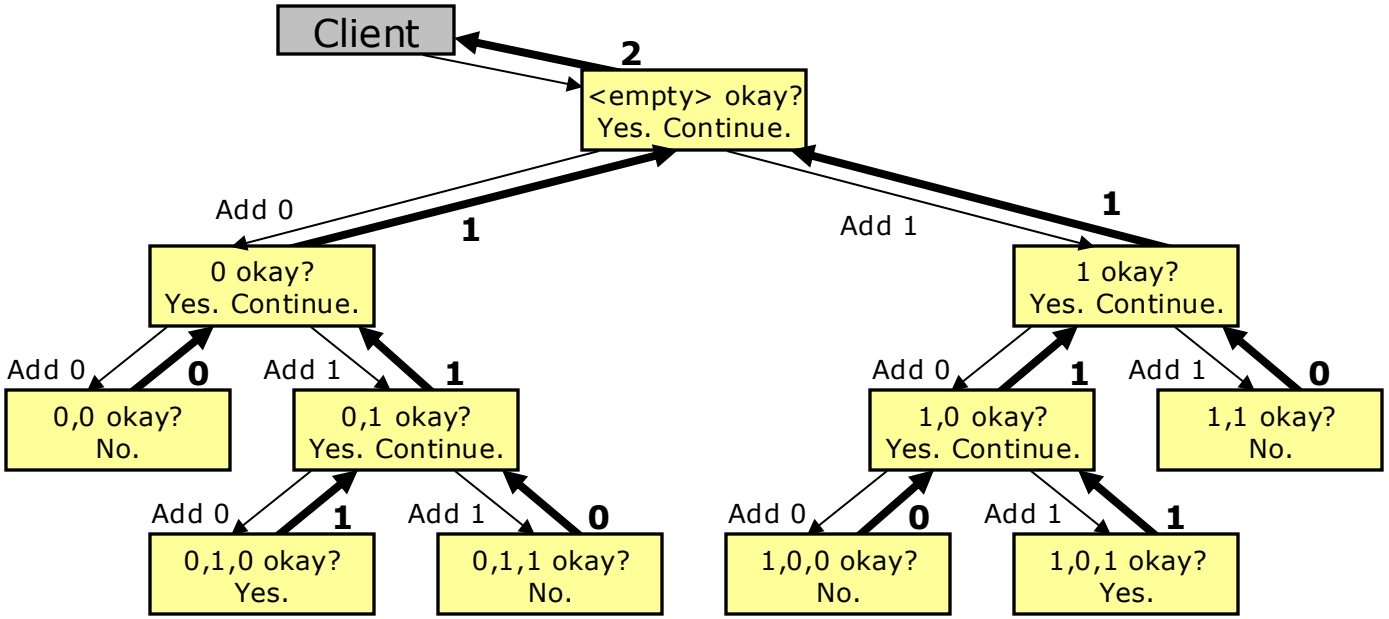
- Write a function that uses recursive backtracking to print all solutions to the *n*-Queens Problem, for a given chessboard size.

Done. See `nqueen.cpp`*.*

Recursive Backtracking
Counting Solutions — Diagram

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

- Base Cases
  - "Found a solution" returns 1.
  - "Dead end" returns 0.
- Recursive Case
  - Make recursive calls, add their return values, and return the total.



Counting Solutions — How to Do It

The Code

- **Nonrecursive wrapper function**
  - Create an empty partial solution.
  - Call the workhorse function with this partial solution.
  - Return 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, then return 1.
  - Do we have a clear dead end?
    - If so, then return 0.

  As before, **this** might be unnecessary.
  - 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*.

Recursive Backtracking
Counting Solutions — CODE

TO DO

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

Done. See `nqueen_count.cpp`.

This n-Queens counting code is similar to what you will write for Assignment 4.

Unit Overview
Algorithmic Efficiency & Sorting

Our next unit covers analyzing the efficiency of algorithms, along with a number of algorithms for sorting.

Topics

- Analysis of Algorithms
- Introduction to Sorting
- Comparison Sorts I
- Asymptotic Notation
- Divide and Conquer
- Comparison Sorts II
- The Limits of Sorting
- Comparison Sorts III
- Non-Comparison Sorts
- Sorting in the C++ STL

The Midterm Exam will be given at the end of this unit.