Recursion vs. Iteration continued
Thoughts on Assignment 3
Eliminating Recursion

CS 311 Data Structures and Algorithms
Lecture Slides
Monday, February 18, 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
- (part) Recursion vs. Iteration
- Eliminating Recursion
- Recursive Search with Backtracking

Review
Search Algorithms [1/2]

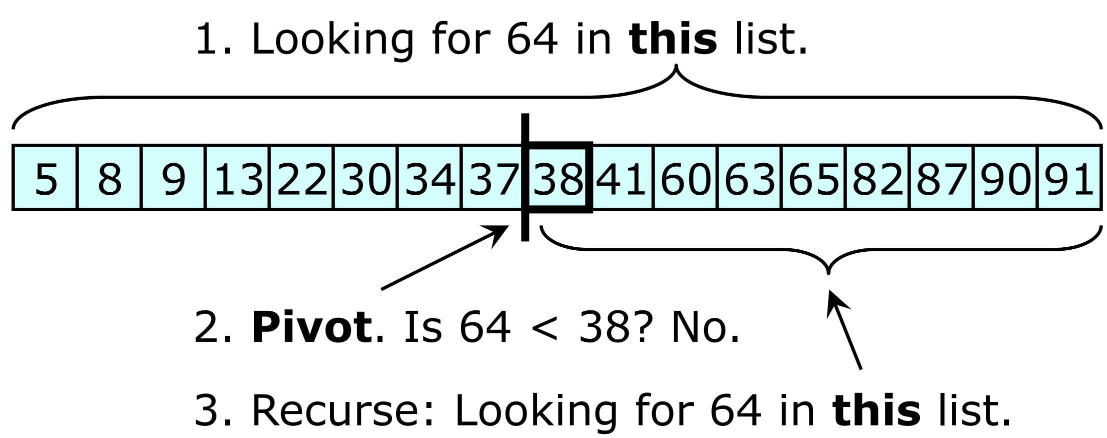
Binary Search is an algorithm to find a given **key** in a **sorted list**.

- 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 [2/2]

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

Number of Look-Ups We Have Time to Perform	Maximum Allowable List Size: Binary Search	Maximum Allowable 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.”
— Nick Trefethen

Recursion vs. Iteration
Drawbacks of Recursion

Two factors can make recursive algorithms inefficient.

- **Inherent inefficiency of <u>some</u> 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.

Interlude - Assignment 3

Six fairly simple topics

- Catching exceptions
- Throwing exceptions
- Recursion
- Iterators
- Calling the STL
- Simple Linked List Handling

Assignment 3 thoughts

- A: Don’t forget to make your exception message include the name of the function.
- B: You probably want to “catch all and rethrow.” See Exception slides for details.
- C: The example code for binary search is a good place to start if you’re having trouble wrapping your head around passing and using iterators.
- D: This is trivial, 10 seconds with Google search and you should be done.
- E: Don’t get confused between the number in the sequence you are computing (n) and the number of steps (which you don’t even need to name…) Look at the recursive version of the linked list size() function – it never names the size, but it still correctly returns it, without global variables!

Eliminating Recursion In General [1/2]

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

- Think: How does the system help you do recursion?
  - It provides a **Stack**, used to hold return addresses for function calls, and values of automatic local variables.
- We can implement such a Stack ourselves. We need to be able to store:
  - Values of automatic local variables, including parameters.
  - The return value (if any).
  - Some indication of where we have been in the function.
- Thus, we can eliminate recursion by mimicking the system's method of handling recursive calls using Stack frames.

Eliminating Recursion In General [2/2]

"Brute-force" method

To rewrite **any** recursive function in iterative form:

- Declare an appropriate Stack.
  - A Stack item holds all automatic variables, an indication of what location to return to, and the return value (if any).
- Replace each automatic variable with its field in the top item of the Stack.
  - Set these up at the beginning of the function.
- Put a loop around the *rest* of the function body: "**`while (true) { … }`**".
- Replace each recursive call with:
  - Push an object with parameter values and current execution location on the Stack.
  - Restart the loop ("**`continue`**").
  - A label marking the current location.
  - Pop the stack, using the return value (if any) appropriately.
- Replace each "**`return`**" with:
  - If the "return address" is the outside world, really **`return`**.
  - Otherwise, set up the return value, and skip to the appropriate label ("**`goto`**"?).

This method is primarily of theoretical interest.

- *Thinking* about the problem often gives better solutions than this.
- We will look at this method further when we study **Stacks**.

Eliminating Recursion
Tail Recursion [1/4]

A special kind of recursion is **tail recursion**: when a recursive call is the last thing a function does.

Tail recursion is important because it makes the recursion $\rightarrow$ 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++.
- When a compiler does this conversion, it is called **tail call optimization** (**TCO**). This is common in functional languages like Lisp, Scheme, and Haskell.

Eliminating Recursion
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
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
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.