Search Algorithms II
Eliminating Recursion
Search in the C++ STL

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, September 25, 2023

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

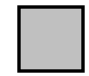
- ✓ Arrays & Linked Lists
- ✓ Introduction to recursion
- ✓ Search algorithms I
- ✓ Recursion vs. iteration
- Search algorithms II
- Eliminating recursion
- Search in the C++ STL
- Recursive backtracking

Review

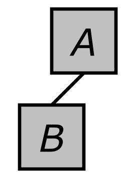
Review
Recursion vs. Iteration [1/5]

Use a **tree** to represent function calls some algorithm makes.

- A box represents making a call to a function.



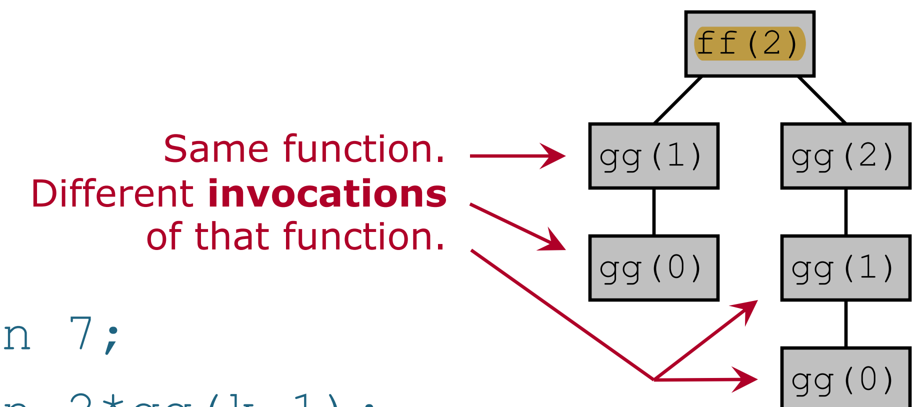
- A line from an *A* box down to a *B* box represents this call to function *A* making a call to function *B*.




```
int ff(int n)
{
    return gg(n-1) + gg(n);
}

int gg(int k)
{
    if (k == 0) return 7;
    else        return 2*gg(k-1);
}
```

Tree representing calls made by doing `ff(2)`



(Yes, our trees are upside-down.)

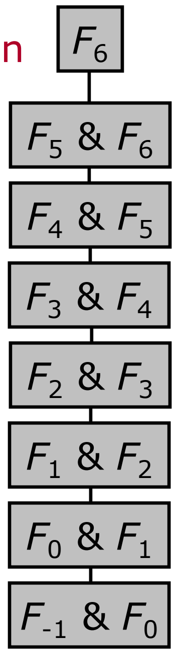
Review
Recursion vs. Iteration [2/5]

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

Computing F_6

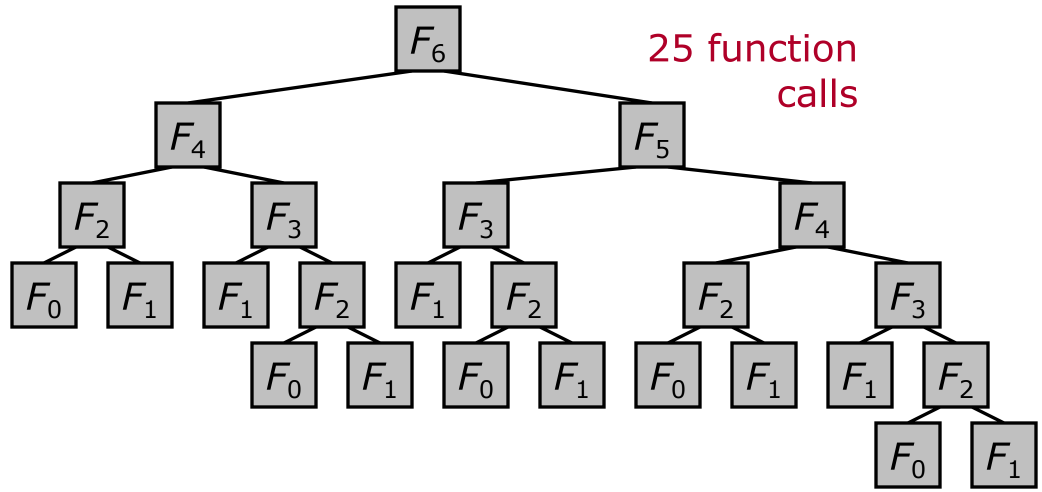
`fibo_recurse.cpp`

8 function calls



`fibo_first.cpp`

25 function calls



Fibonacci No.	`fibo_recurse.cpp`	`fibo_first.cpp`
F_7	9 calls	41 calls
F_{10}	12 calls	177 calls
F_{20}	22 calls	21,891 calls
F_{40}	42 calls	331,160,281 calls

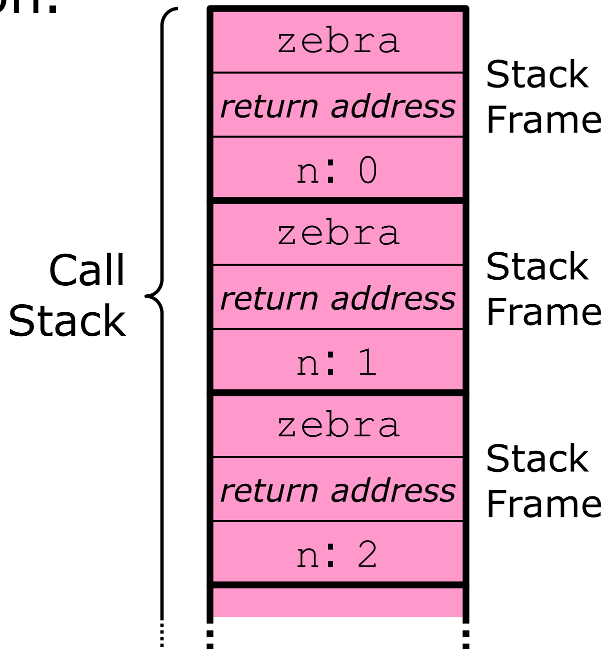
Review
Recursion vs. Iteration [3/5]

A running program uses a **call stack**, which holds **stack frames**.

- Each stack frame corresponds to an *invocation* of a function. It holds automatic variables and the function's **return address**.
- A function call **pushes** a new stack frame on the **top** of the stack.
- When the function exits, this stack frame is **popped** off the stack.

Recursion can result in many stack frames corresponding to different invocations of the same function.

```
void zebra(int n)
{
    if (n == 0)
        …
    …
    zebra(n-1);
}
```

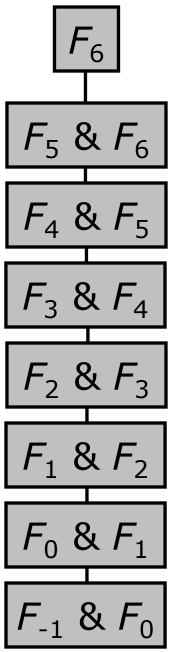


Review
Recursion vs. Iteration [4/5]

A function call’s **recursion depth** is the greatest number of stack frames on the call stack *at any one time* as a result of the call.

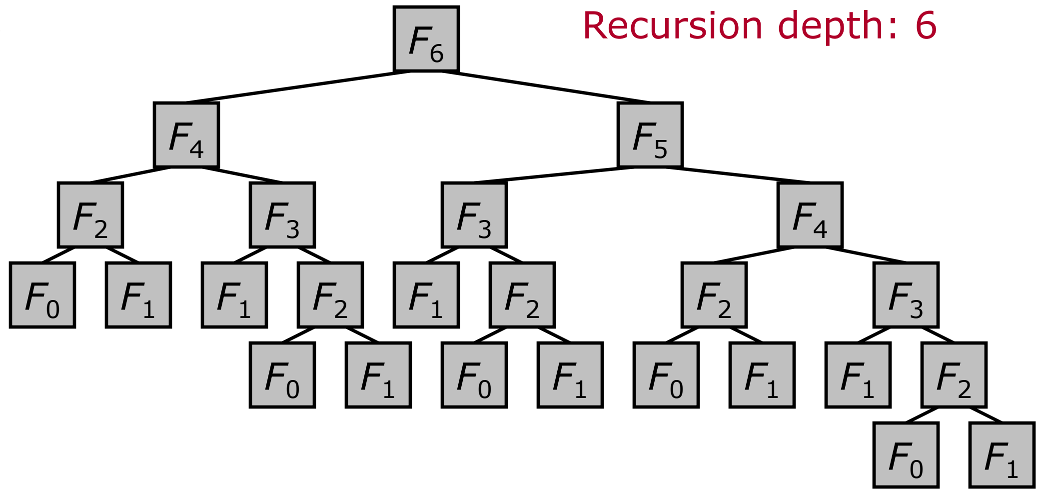
`fibo_recurse.cpp`

Recursion depth: 8



`fibo_first.cpp`

Recursion depth: 6



When analyzing *time* usage, the total number of calls is of interest.

When analyzing *space* usage, the recursion depth is of interest.

Review
Recursion vs. Iteration [5/5]

Two factors can make recursive code inefficient, compared to iterative code.

- Inherent inefficiency of some recursive algorithms
  - But there are efficient recursive algorithms.
- Function-call overhead
  - Making all those function calls requires work: pushing and popping stack frames, saving return addresses, creating and destroying automatic variables.

These two are important regardless of the recursive algorithm used.

And recursion has another problem.

- Memory-management issues
  - A high recursion depth causes the system to run out of memory for the call stack. This is **stack overflow**, and it generally cannot be dealt with using normal error-handling procedures. The result is usually a crash.
  - When we use iteration, we can manage memory ourselves. This can be more work for the programmer, but it also allows proper error handling.

Search Algorithms II

Sequential Search — Description

Sequential Search (also called **Linear Search**) is another algorithm for finding a given key in a list.

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.

65	82	41	5	87	22	9	38	63	60	90	13	91	37	34	8	30

...

Sequential Search — Comparison [1/3]

Binary Search needs some things to be true about its list.

- Binary Search *requires* its list to be sorted.
- For Binary Search to be efficient, the list should be random-access.

Sequential Search needs neither of these.

Still, we like Binary Search better than Sequential Search. Why?
See the next slide.

Search Algorithms II
Sequential Search — Comparison [2/3]

We like Binary Search better than Sequential Search, because it is much faster …

List Size	Lookups: Binary Search (worst case)	Lookups: Sequential Search (worst case)
1	1	1
2	2	2
4	3	4
100	8	100
10,000	15	10,000
1,000,000	21	1,000,000
10,000,000,000	35	10,000,000,000
n	*???*	*???*

Search Algorithms II
Sequential Search — Comparison [3/3]

… 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	*???*	*???*

"The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less." [From Lloyd N. Trefethen, "Maxims about Numerical Mathematics, Computers, Science, and Life", 1998.]

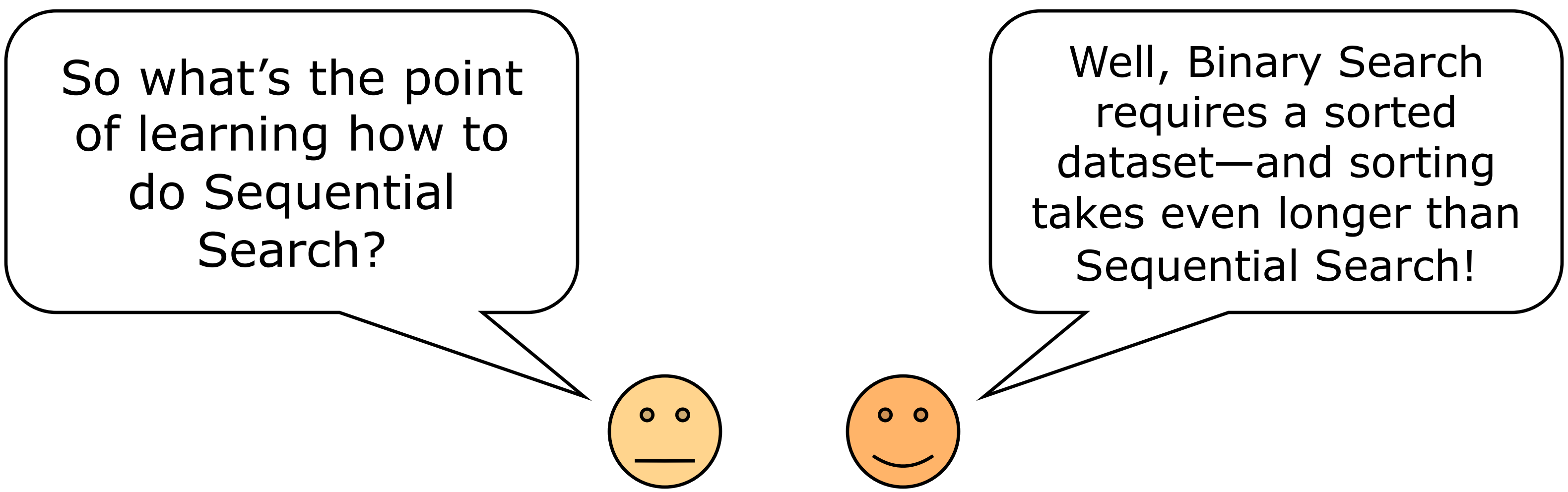
Search Algorithms II
Sequential Search — CODE

TO DO

- Write Sequential Search, and compare its performance with that of Binary Search.

> *See* `seqsearch_compare.cpp`*.*
>
> *This file contains the implementation of Binary Search from* `binsearch2.cpp`*.*

For large datasets, Sequential Search is much slower than Binary Search.



Eliminating Recursion

Eliminating Recursion In General [1/2]

While it is a useful algorithm-design tool, recursion can have serious drawbacks. Thus, it can sometimes be helpful to **eliminate recursion**—that is, to convert recursion to iteration.

Fact. *Every* recursive function can be rewritten as a non-recursive function that uses essentially the same algorithm.

This is true because we can simulate the call stack ourselves. We can eliminate recursion by mimicking the system's method of handling recursive calls using stack frames.

We *can* always eliminate recursion, but that does not mean that eliminating it is always a good idea.

Eliminating Recursion In General [2/2]

We discuss this method further when we cover *Stacks*, later in the semester.

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 Stack item.
  - Set these up at the beginning of the function.
- Put a loop around the *rest* of the function: `while (true) { … }`
- Replace each recursive call with:
  - Push an object with parameter values and current execution location.
  - Restart the loop (`continue`).
  - A label marking the current location.
  - Pop the stack. Make use of the return value (if any).
- Replace each `return` with:
  - If the "return address" is the outside world, then really `return`.
  - Otherwise, set the return value, and skip to the proper label (`goto` ?).

This method is rarely used. *Thinking* often gets better results.

Eliminating Recursion
Tail Calls [1/2]

When a calling some function is the last thing a function does, we refer to the call as a **tail call**.

For a `void` function, a tail call looks like this:

```
void foo(TTT a, UUU b)
{
    …
    gg(x);
}
```

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

```
SSS bar(TTT a, UUU b)
{
    …
    return gg(x);
}
```

Tail call

Returns whatever the final function call returns. Otherwise, it is not a tail call.

Eliminating Recursion
Tail Calls [2/2]

Some compilers—mostly *not* C++ compilers—perform **tail call optimization** (**TCO**), in which a tail call reuses the same stack frame as the function that makes the call.

Essentially, a tail call turns into a `goto`.

```
void foo(TTT a, UUU b)
{
    …
    gg(x);
}
```

If TCO is done, then this invocation of `gg` will reuse the same stack frame as the invocation of `foo`.

Automated TCO is not so common in C++ compilers because of the execution of destructors of automatic variables when a function exits; so what looks like a tail call may not actually be the *last* thing a function does.

Eliminating Recursion
Tail Recursion — Definition

When a tail call is a recursive call, we have **tail recursion**. A function that does this is said to be **tail-recursive**.

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);
}
```

Returns whatever the final function call returns. Otherwise, it is not a tail call.

Eliminating Recursion
Tail Recursion — Elimination

Even in C++, tail recursion allows us to do a kind of manual TCO. This *recursion* → *iteration* conversion is very easy.

We like tail recursion—ironically, because it is easy to eliminate.

Eliminating Tail Recursion

- Surround the function body with a big loop.
- 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.
- No changes are required in the base-case.

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

Eliminating Recursion
Tail Recursion — CODE

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 (tail recursion).

    New file: `binsearch3.cpp`.

  - Next, eliminate the tail recursion.

    New file: `binsearch4.cpp`.

Observation

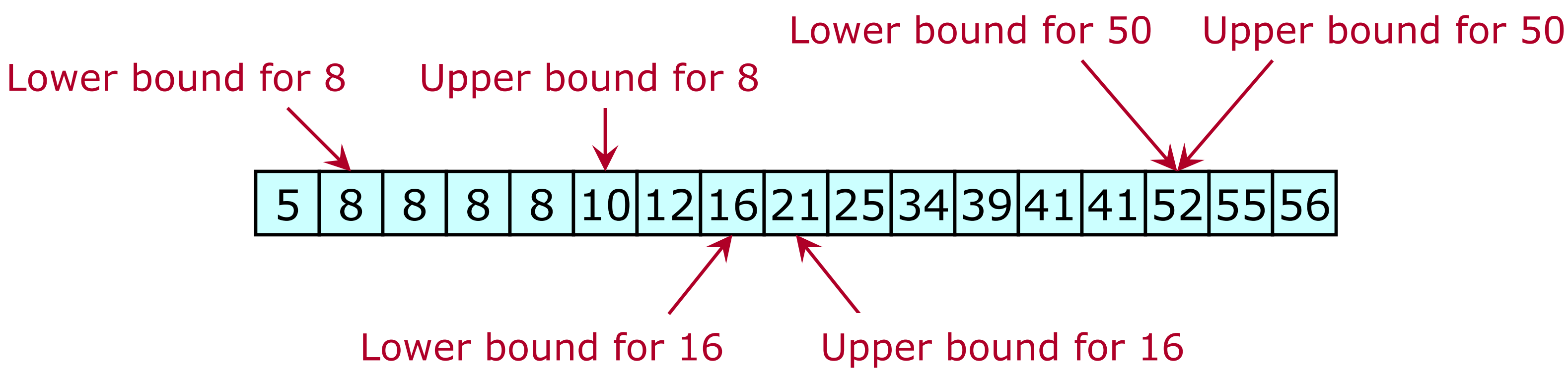
- We replace the tail-recursive call with: set parameters to their new values, and restart the loop—and that last part happens automatically.
- If the parameters already have their new values, then we replace the tail-recursive call with: *nothing!*

Search in the C++ STL

Search in the C++ STL
Binary Search [1/2]

The C++ Standard Template Library includes Binary Search.

- Function std::binary_search (<algorithm>) searches and returns a bool indicating success/failure.
- The following functions (also in <algorithm>) return iterators to where the value was found, or where it could be inserted.
  - std::lower_bound
  - std::upper_bound
  - std::equal_range



Search in the C++ STL
Binary Search [2/2]

These functions (`binary_search` in particular) are similar to ours:

- 3 parameters: 2 iterators specifying a range & a value to search for.
- They are templates, and they work for a wide range of types.
- They require the data to be sorted.
- They are faster on random-access data, but they do not require it.
- They search based on equivalence, not equality. Only `operator<` is used on the value type.

```
#include <algorithm>   // For std::binary_search

vector<int> v1 = …;    // Dataset, sorted
int key = …;           // Key to find

bool found =
    std::binary_search(begin(v1), end(v1), key);
```

Search in the C++ STL
Custom Comparison

All STL Binary Search algorithms have alternate forms that allow the client to specify a comparison other than `operator<`. This is done when the dataset to be searched is sorted differently.

```
#include <algorithm>    // For std::binary_search
#include <functional>   // For std::greater

vector<int> v2 = …;     // Dataset, sorted DESCENDING
int key = …;            // Key to find

bool found = std::binary_search(begin(v2), end(v2), key,
                                   std::greater<int>());
```

More on specifying custom comparisons later in the semester.

Search in the C++ STL
Sequential Search

Sequential Search is also available in the STL.

- It is called `std::find` (`<algorithm>`).
- It searches using equality (`==`), not equivalence.
- 3 parameters: 2 iterators specifying a range & a value to search for.
- Return value: an iterator to the first item found, or an iterator to just past the end of the range (the second parameter) if not found.
- A custom equality comparison can be specified.

```
vector<int> v3 = …;   // Dataset (unsorted?)
int key = …;          // Key to find

auto iter = std::find(begin(v3), end(v3), key);
if (iter == end(v3))  cout << "Not found";
else                  cout << "FOUND: " << *iter;
```

Search in the C++ STL
Algorithms for Specific Data Structures

Some data structures do not allow fast lookup by index—so Binary Search is slow—but still allow for fast lookup by key.

STL versions of such structures have their own search-by-key, as a member function `find`, which is used similarly to `std::find`.

You may be familiar with `std::map`. `std::binary_search` and `std::find` *can* be used with a `map`, but both are slow. The `map` member function `find` is much faster.

```
map<string, int> m = …;   // Dataset
string key = …;           // Key to find

auto iter = m.find(key);
if (iter == end(m))   cout << "Not found";
else                  cout << "FOUND: " << *iter;
```

Why is the `find` member faster? How much faster is it?

More on these when we cover *Tables*, later in the semester.