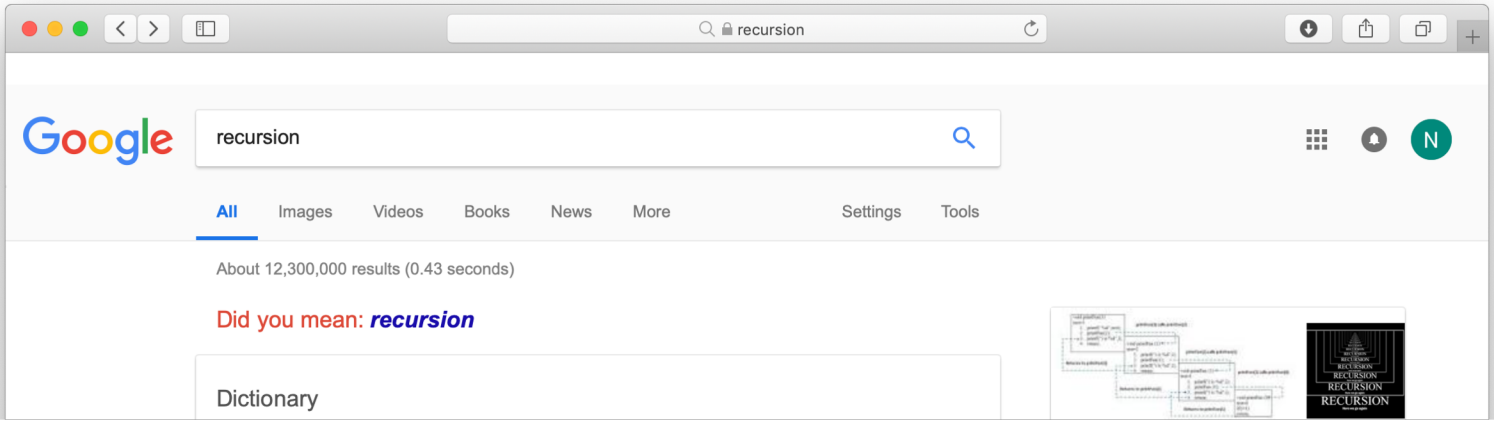
CS 106X, Lecture 8
More Recursion

reading:

Programming Abstractions in C++, Chapters 7-8



Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

Recursive Thinking

- **In code**, recursion is when a function in your program calls itself as part of its execution.
- **Conceptually**, a recursive problem is one that is *self-similar;* it can be solved via smaller occurrences of the same problem.

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

Example 1: Factorial

`n! = n * (n-1) * (n-2) * (n-3) * … * 1`

- Write a function that computes and returns the factorial of a provided number, recursively (no loops).
  - e.g. **`factorial(4)`** should return **`24`**
  - You should be able to compute the value of any non-negative number. (`0! = 1`).

Factorial

```
// Returns n!, or 1 * 2 * 3 * 4 * ... * n.
// Assumes n >= 0.
int factorial(int n) {
    if (n < 0) {                          // error handling
        throw "illegal negative n";
    if (n == 0) {                         // base case
        return 1;
    } else {
        return n * factorial(n - 1);      // recursive case
    }
}
```

Recursive Program Structure

```
recursiveFunc() {
  if (test for simple case) { // base case
    Compute the solution without recursion
  } else { // recursive case
    Break the problem into subproblems of the same form
    Call recursiveFunc() on each self-similar subproblem
    Reassamble the results of the subproblems
  }
}
```

Example 2: Fibonacci

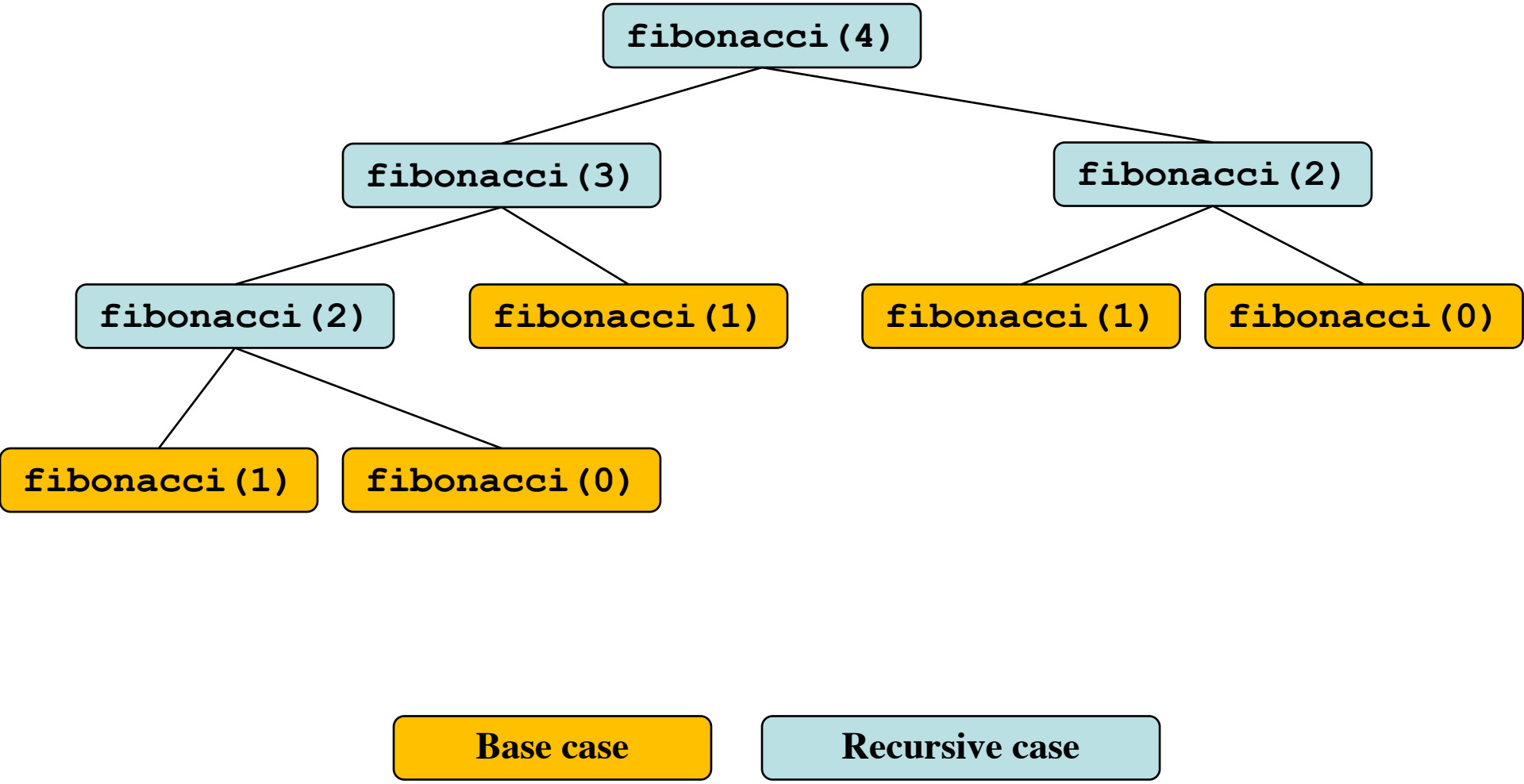
`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

- The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two previous numbers.
- Write a function that computes and returns the nth Fibonacci number, recursively (no loops).
  - e.g. **`fibonacci(6)`** should return **`8`**

Fibonacci

```
// Returns the i'th Fibonacci number in the sequence
// (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …)
// Assumes i >= 0.
int fibonacci(int i) {
    if (i < 0) {                          // error handling
        throw "illegal negative index";
    else if (i == 0) {                    // base case 1
        return 0;
    } else if (i == 1) {                  // base case 2
        return 1;
    } else {
        // recursive case
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```

Recursive Tree



isPalindrome exercise

- Write a recursive function `isPalindrome` accepts a `string` and returns `true` if it reads the same forwards as backwards.

```
isPalindrome("madam")                    → true
isPalindrome("racecar")                  → true
isPalindrome("step on no pets")          → true
isPalindrome("able was I ere I saw elba") → true
isPalindrome("Q")                        → true
isPalindrome("Java")                     → false
isPalindrome("rotater")                  → false
isPalindrome("byebye")                   → false
isPalindrome("notion")                   → false
```

isPalindrome solution

```
// Returns true if the given string reads the same
// forwards as backwards.
// By default, true for empty or 1-letter strings.
bool isPalindrome(string s) {
    if (s.length() < 2) {   // base case
        return true;
    } else {                // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
```

isPalindrome solution 2

```
// Returns true if the given string reads the same
// forwards as backwards.
// By default, true for empty or 1-letter strings.
// This version is also case-insensitive.
bool isPalindrome(string s) {
    if (s.length() < 2) {   // base case
        return true;
    } else {                // recursive case
        return tolower(s[0]) == tolower(s[s.length() - 1])
            && isPalindrome(s.substr(1, s.length() - 2));
    }
}
```

Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

reverseLines exercise

Write a recursive function `reverseLines` that accepts a file input stream and prints the lines of that file in reverse order.

Example input file:

```
I am not a person who contributes
And I refuse to believe that
I will be useful
```

Expected console output:

```
I will be useful
And I refuse to believe that
I am not a person who contributes
```

"I Have a Dream" by Antonia Lee, Sara Fung, Christy Fung, Rachel Lam
http://poets.spice.org.hk/index.php?option=com_content&view=article&id=45:my-family&catid=6:reverse-poem&Itemid=7

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

Reversal pseudocode

Reversing the lines of a file:

- If the input stream is empty:
  - Stop
- If there are lines left:
  - Read the next line L
  - Print the rest of the lines in reverse order
  - Print the line L.

reverseLines solution

```
void reverseLines(ifstream& input) {
    string line;
    if (!getline(input, line)) {
        // base case - stop
    } else {
        // recursive case
        reverseLines(input);
        cout << line << endl;
    }

}
```

```
if (x) {
    do nothing
} else {
    do something
}
```

```
if (not x) {
    do something
}
```

reverseLines solution

```
void reverseLines(ifstream& input) {
    string line;
    if (getline(input, line)) {
        // recursive case
        reverseLines(input);
        cout << line << endl;
    }
}
```

- Where is the base case?

Tracing our algorithm

- **call stack**: The function invocations running at any one time.

```
    reverseLines(input);
```

```
void reverseLines(ifstream& input) {  // call 1
    string line;
    if (getline(input, line)) {// "I am not a person who contributes"
        reverseLines(input);
        cout << line << endl;
    }
}
```

```
void reverseLines(ifstream& input) {   // call 2
    string line;
    if (getline(input, line)) {// "And I refuse to believe that"
        reverseLines(input);
        cout << line << endl;
    }
}
```

```
void reverseLines(ifstream& input) {  // call 3
    string line;
    if (getline(input, line)) {       // "I will be useful"
        reverseLines(input);
        cout << line << endl;
    }
}
```

```
void reverseLines(ifstream& input) {     // call 4
    string line;
    if (getline(input, line)) {          // fail
        reverseLines(input);
        cout << line << endl;
    }
}
```

```
input f
```

```
I am not a person who contributes
And I refuse to believe that
I will be useful
```

```
I will be useful
And I refuse to believe that
I am not a person who contributes
```

The "Recursive Leap of Faith"

- When writing a recursive function, pretend that someone has already written a function that can solve that problem for smaller inputs. How could you use it in your implementation?
- E.g. "what if we had a function that could print out a smaller file in reverse?"

```
void reverseLines(ifstream& input) {
    string line;
    if (getline(input, line)) {
        reverseLines(input);     // "leap of faith"
        cout << line << endl;
    }
}
```

The "Recursive Leap of Faith"

https://www.youtube.com/watch?v=VgolfyRO-RI

Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

Recursion & Big-O

```
void reverseLines(ifstream& input) {
    string line;
    if (getline(input, line)) {
        reverseLines(input);
        cout << line << endl;
    }
}
```

- What is the Big-O of the above function?
- (What is N?)

How many times is this function called in total? x **What is the runtime of each individual function call?**

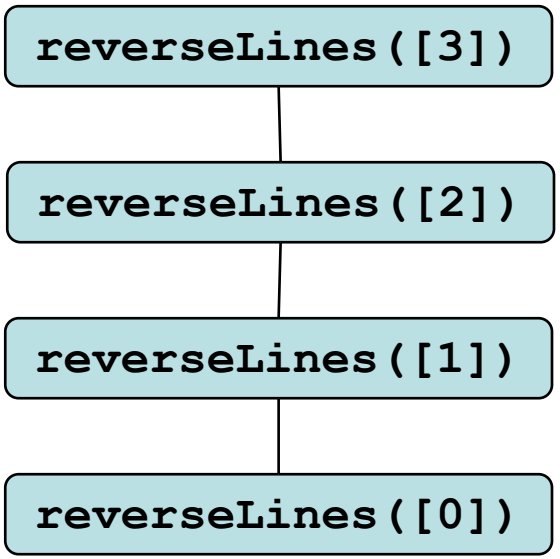
Recursion & Big-O

```
void reverseLines(ifstream& input) {
    string line;
    if (getline(input, line)) {
        reverseLines(input);
        cout << line << endl;
    }
}
```

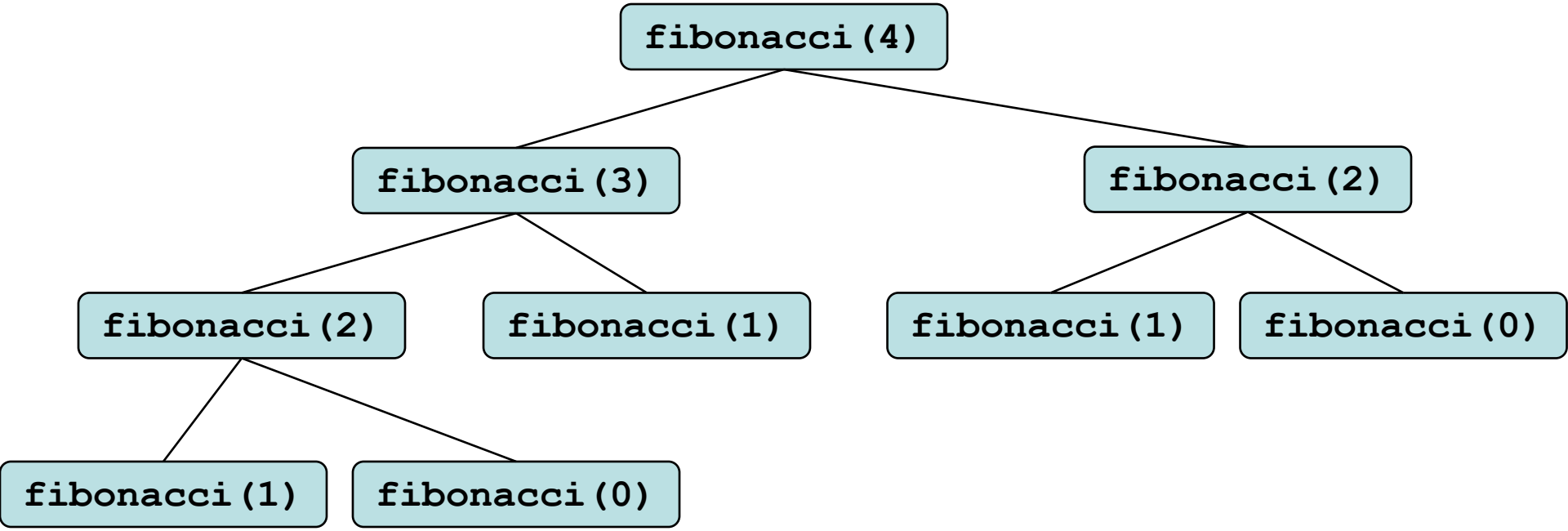
There is one recursive call for each line of the file.

The runtime of an individual function call is O(1).

Thus, the total runtime is O(N)*O(1) = O(N).



Fibonacci: Big-O



- Each recursive call makes 2 *additional* recursive calls.
- The worst-case depth of the recursion is the index of the Fibonacci number we are trying to calculate (N).
- Therefore, the number of total calls is $O(2^N)$.
- Each individual function call does O(1) work. Therefore, the total runtime is $O(2^N) * O(1) = O(2^N)$.

Recursion & Big-O

- The runtime of a recursive function is the number of function calls times the work done in each function call.
- The number of calls for a branching recursive function is usually

$$O(b^d)$$

where

  - **b** is the worst-case branching factor (# recursive calls per function execution)
  - **d** is the worst-case depth of the recursion (the longest path from the top of the recursive call tree to a base case).

Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

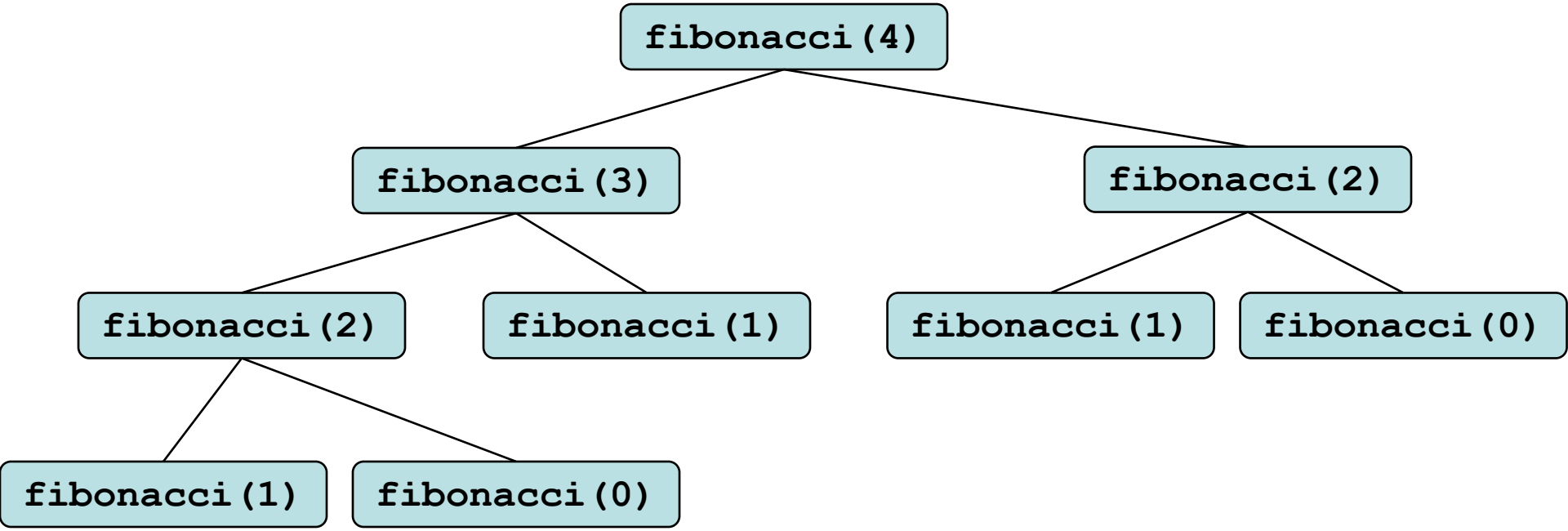
Announcements

- Computer Forum Career Fair **TODAY** until 4PM @ Packard Lawn
- Candy poll!
- HW1 IGs

Plan For Today

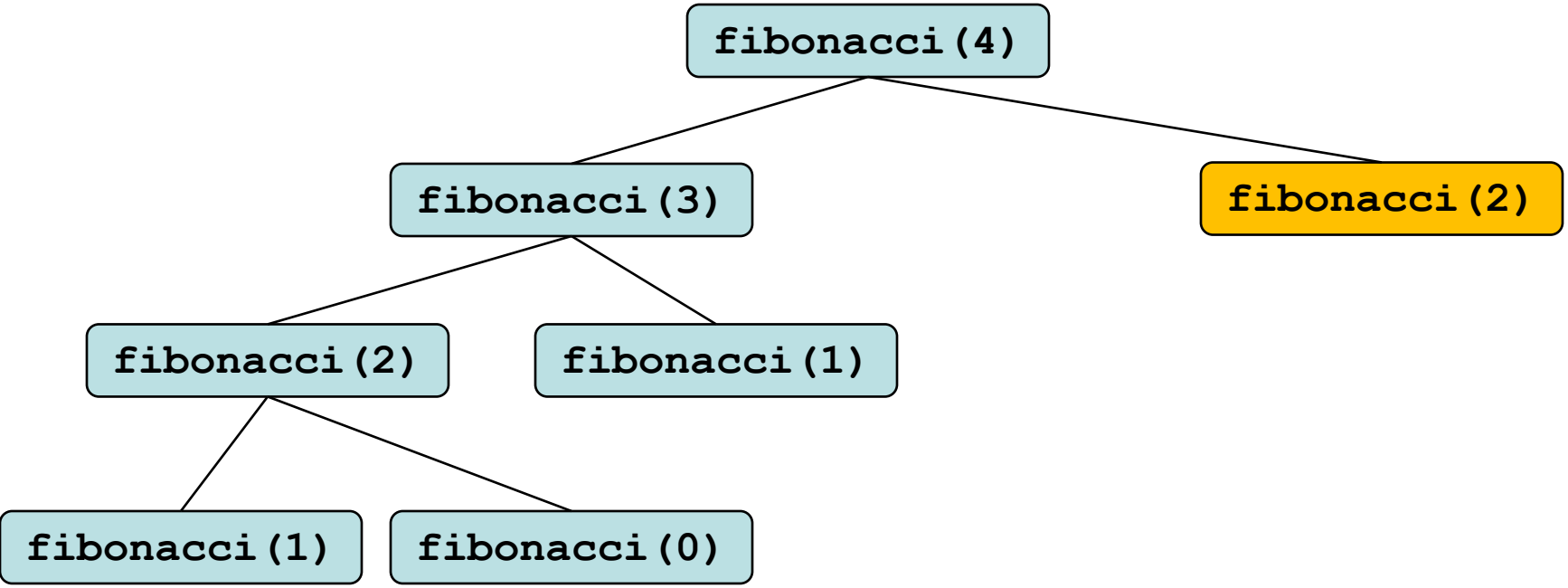
- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

Recursive Tree



As we recurse, we do extra work recomputing some Fibonacci numbers many times!

Recursive Tree



Is there a way to remember what we already computed?

Memoization

- **memoization**: Caching results of previous expensive function calls for speed so that they do not need to be re-computed.
  - Often implemented by storing call results in a collection.

- Pseudocode template:

```
cache = {}.          // initially empty

function f(args):
    if I have computed f(args) before:
        Look up f(args) result in cache.
    else:
        Actually compute f(args) result.
        Store result in cache.
    Return result.
```

Memoized Fibonacci

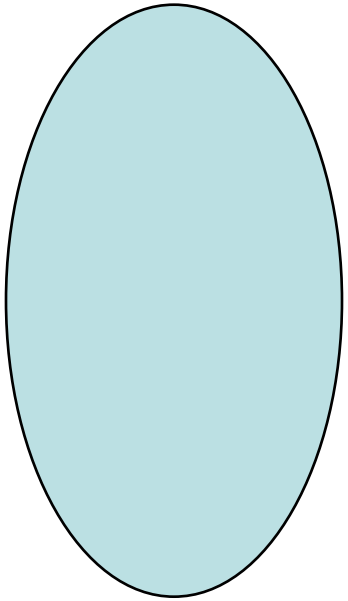
```
// Returns the nth Fibonacci number (no error handling).
// This version uses memoization.
int fibonacci(int i, Map<int, int>& cache) {
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

Memoized Stack Trace

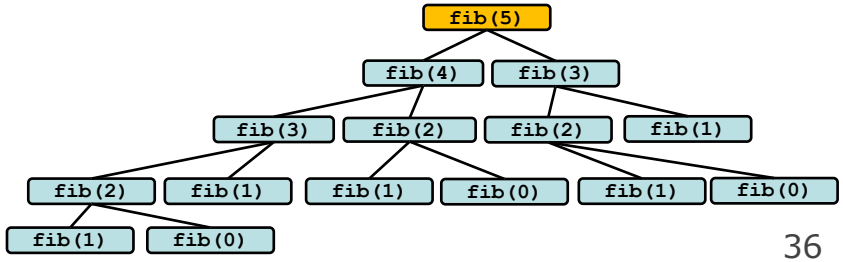
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

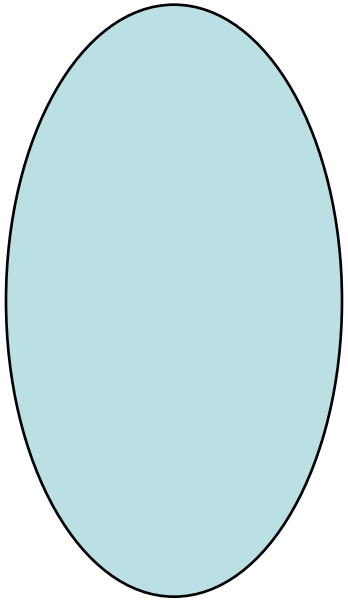


Memoized Stack Trace

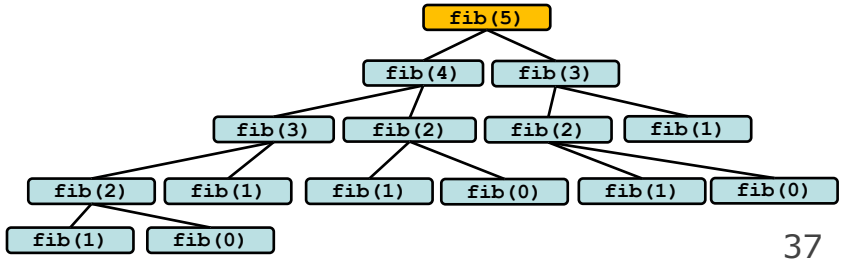
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree



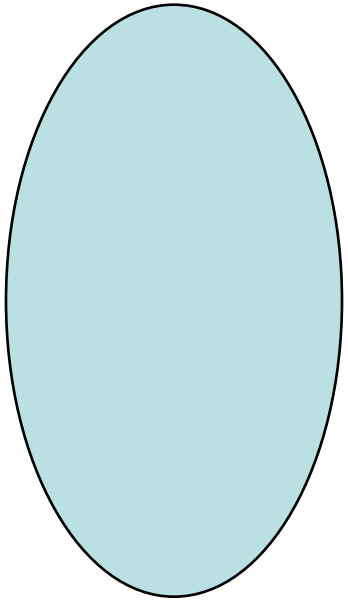
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

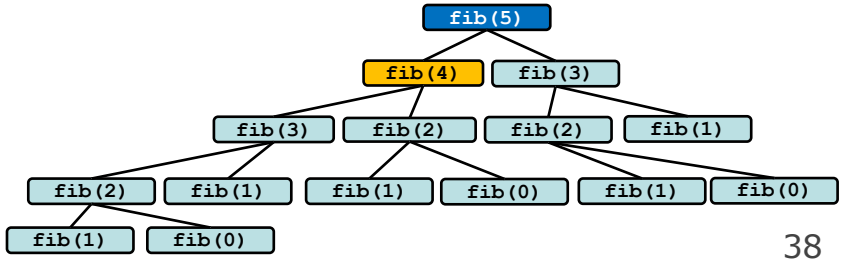
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=4
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree



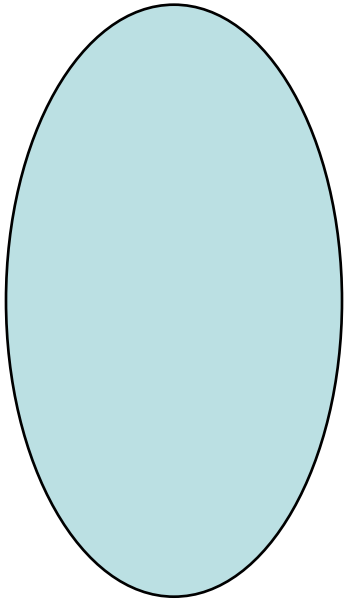
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

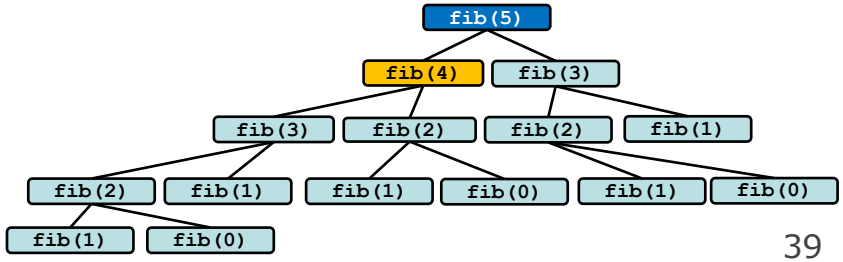
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=4
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

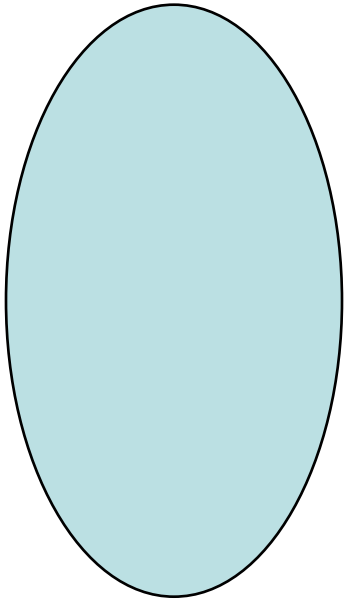


Memoized Stack Trace

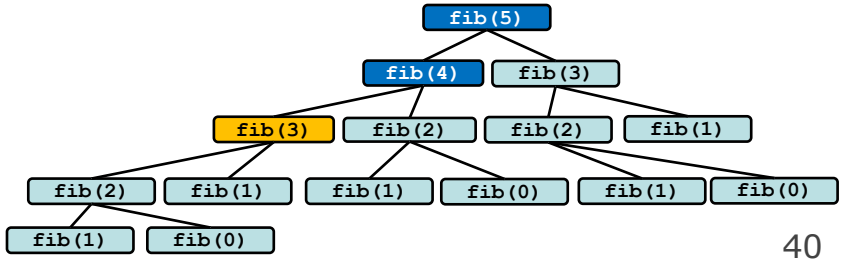
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

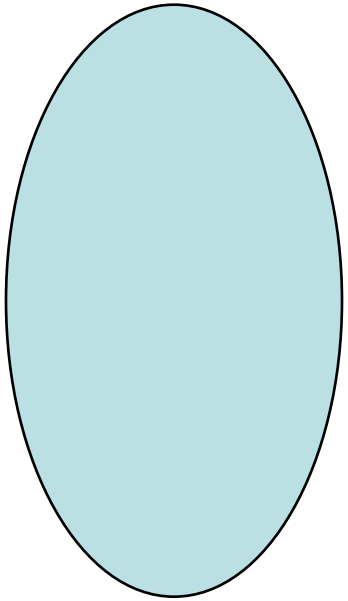


Memoized Stack Trace

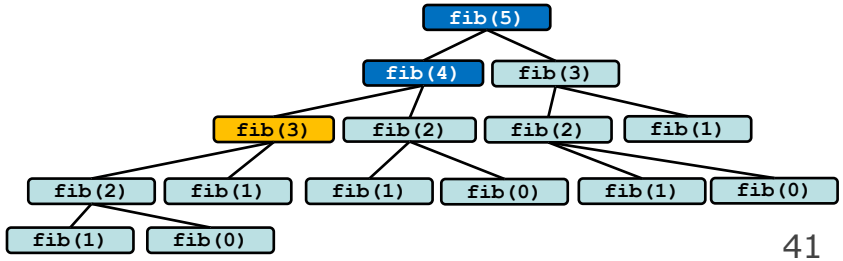
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

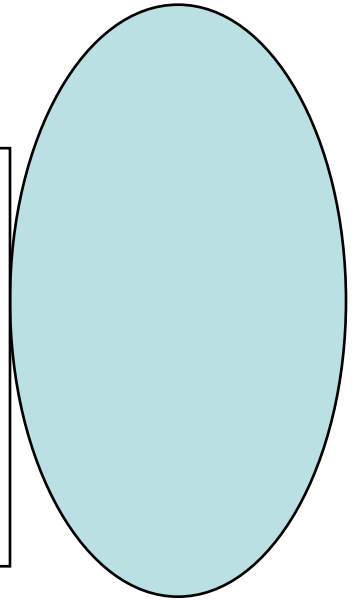


Memoized Stack Trace

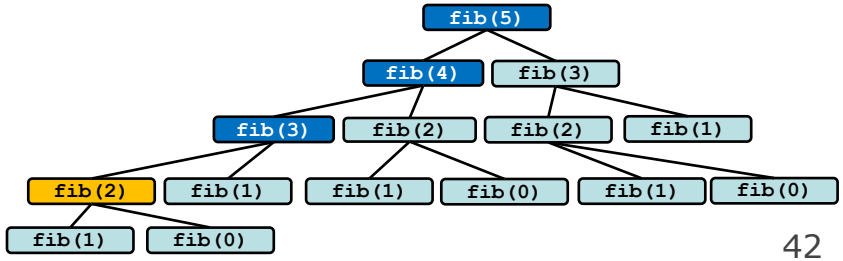
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

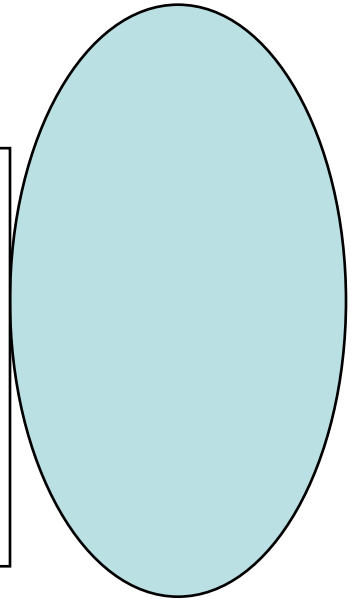


Memoized Stack Trace

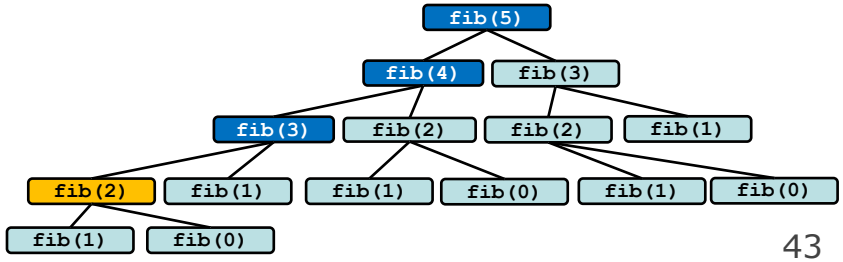
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



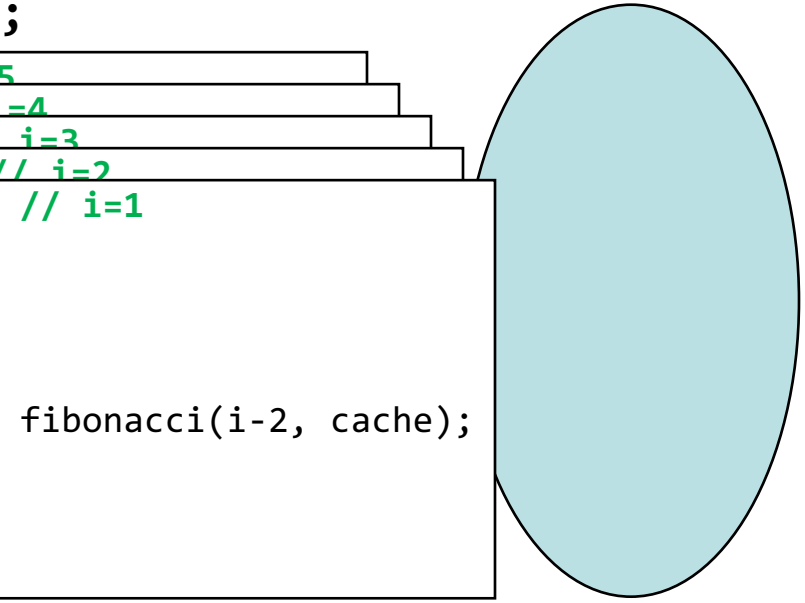
call tree



Memoized Stack Trace

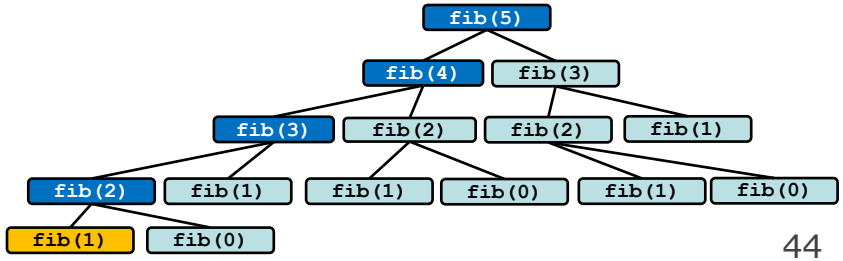
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

cache



```
int fibonacci(int i, Map<int, int>& cache) { // i=1
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

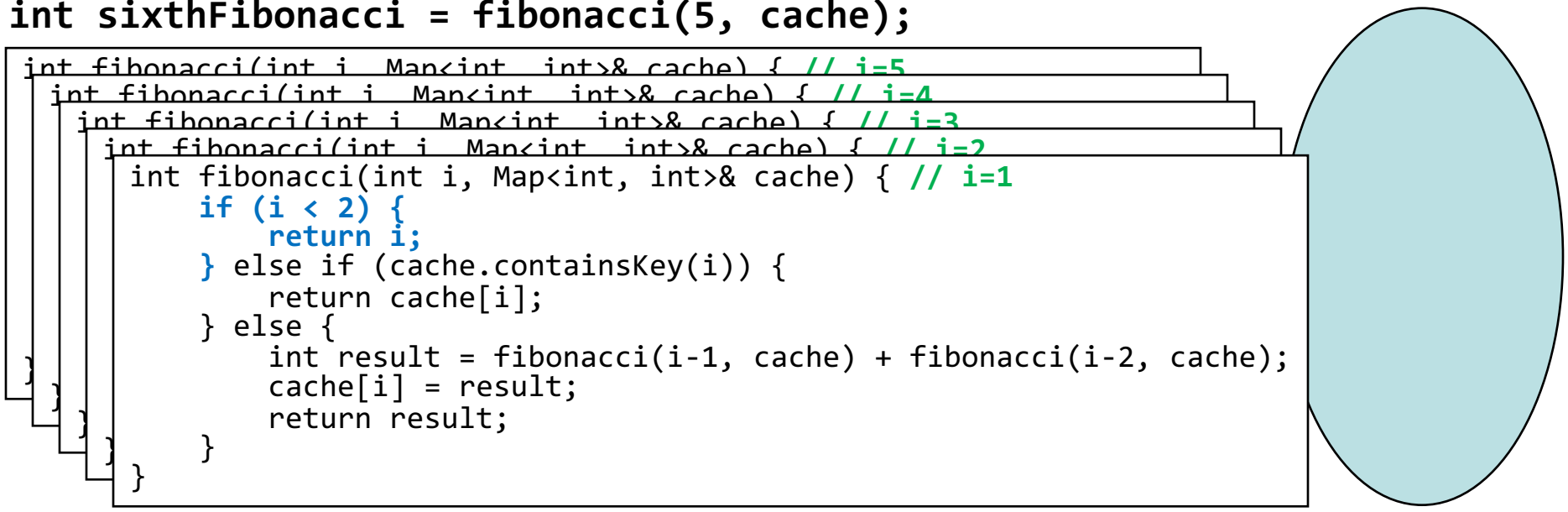
call tree



Memoized Stack Trace

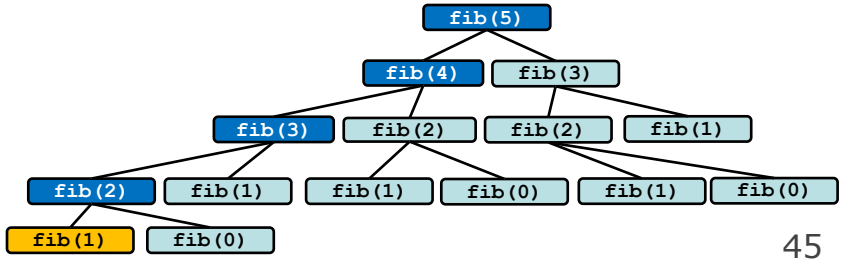
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

cache



```
int fibonacci(int i, Map<int, int>& cache) { // i=1
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

call tree



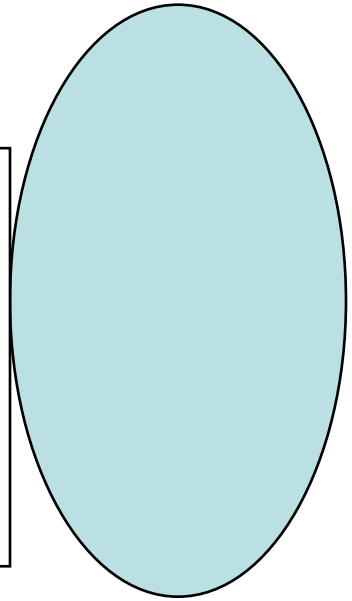
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

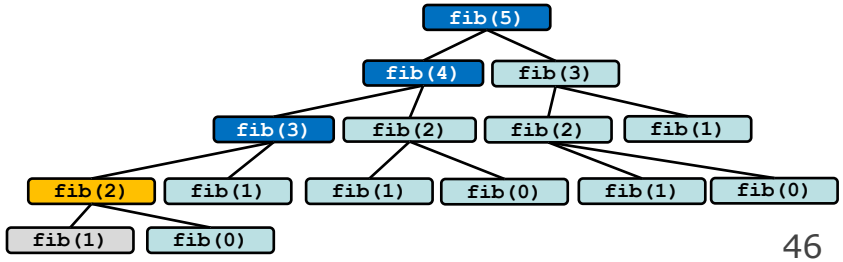
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

1

cache



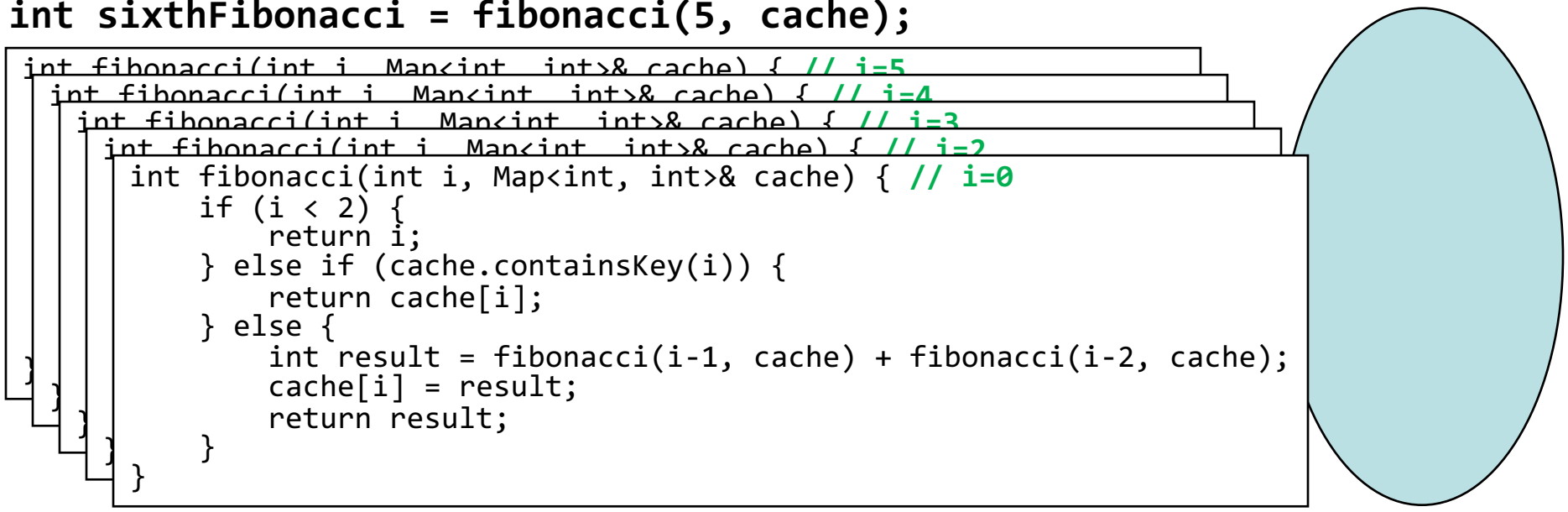
call tree



Memoized Stack Trace

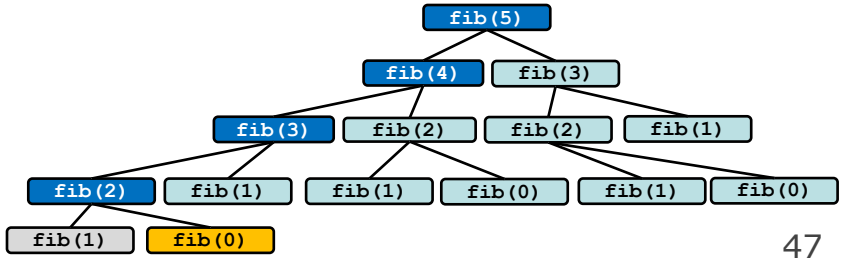
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

cache



```
int fibonacci(int i, Map<int, int>& cache) { // i=0
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

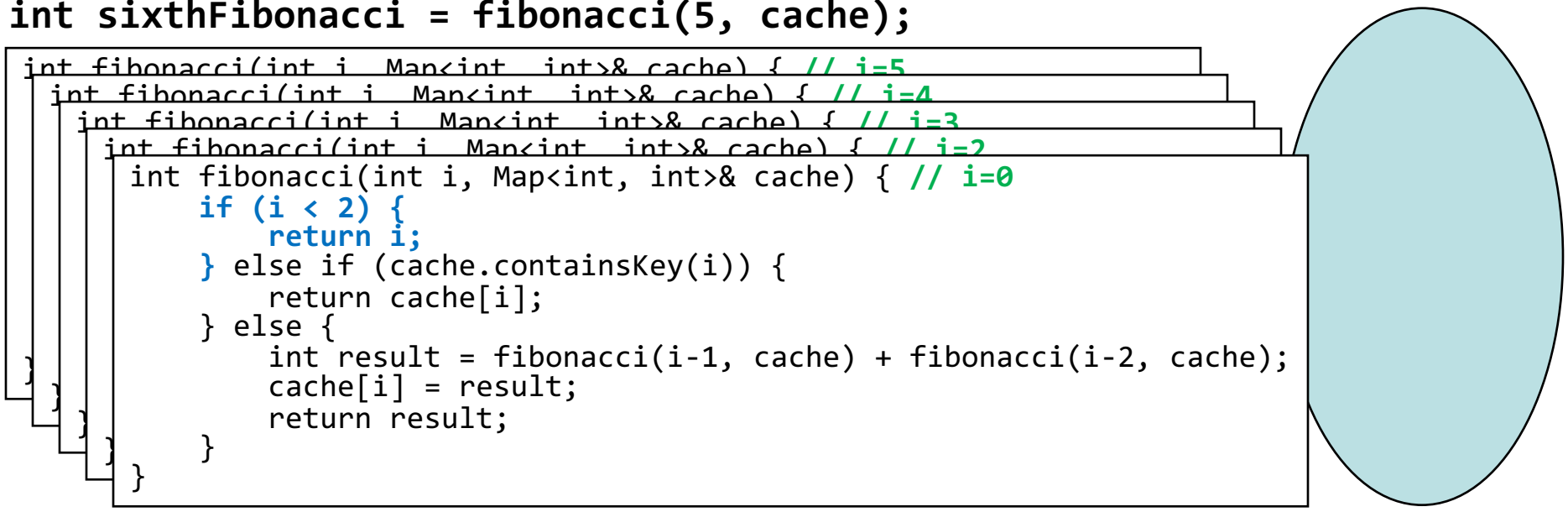
call tree



Memoized Stack Trace

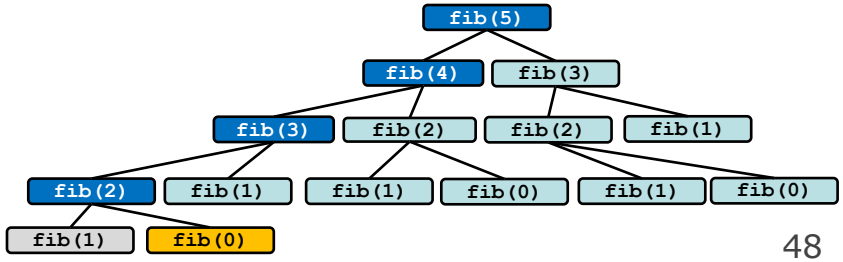
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

cache



```
int fibonacci(int i, Map<int, int>& cache) { // i=0
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

call tree



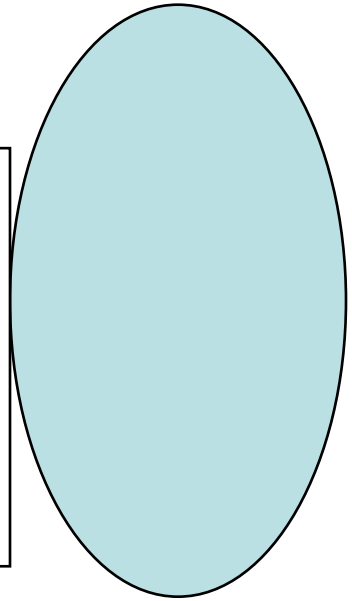
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

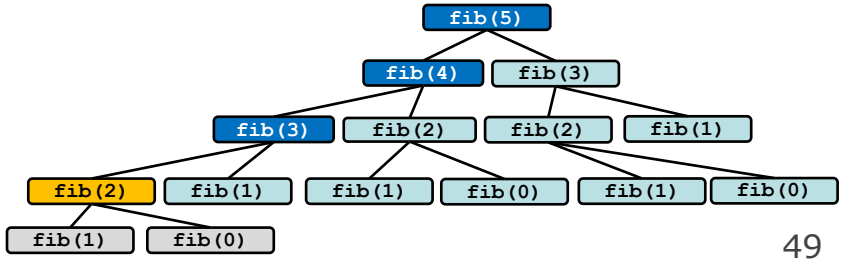
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

1 0

cache



call tree



Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

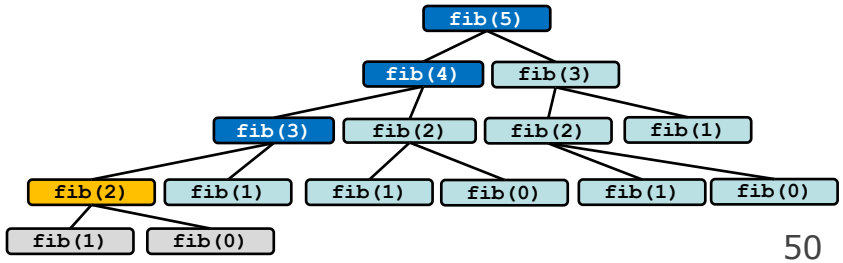
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

1 0

cache

2 -> 1

call tree



Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

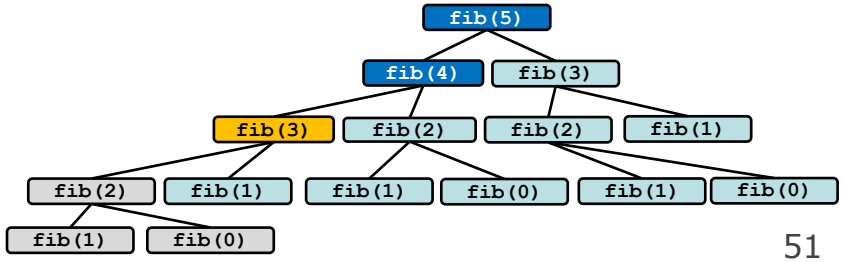
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

1

cache

2 -> 1

call tree

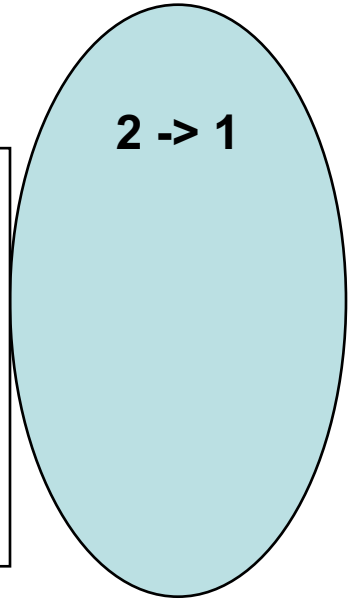


Memoized Stack Trace

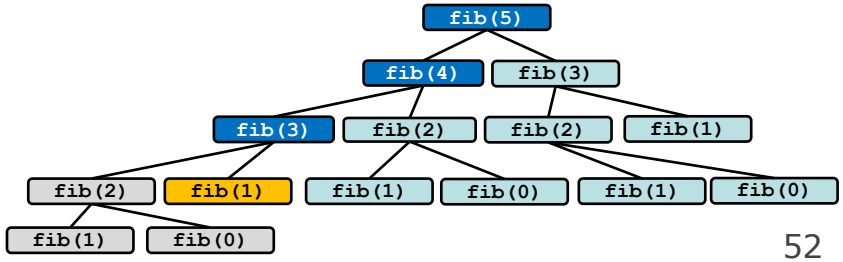
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=1
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

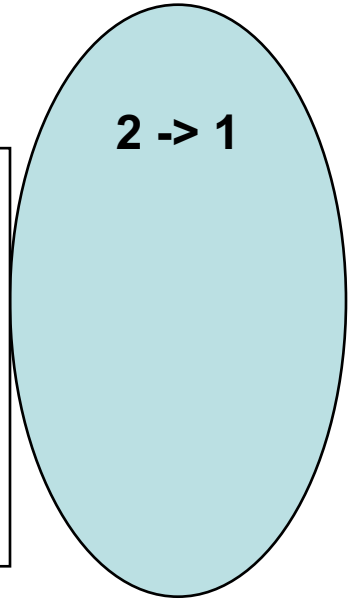


Memoized Stack Trace

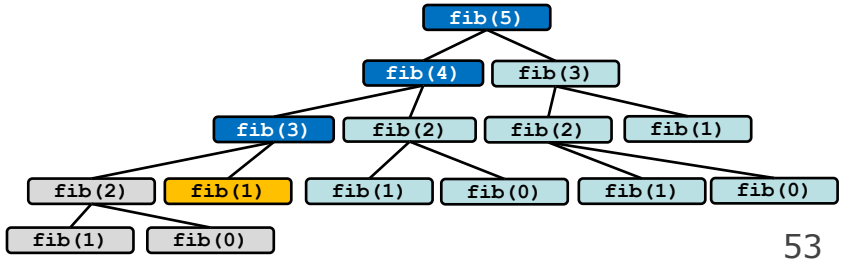
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
int fibonacci(int i, Map<int, int>& cache) { // i=1
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree



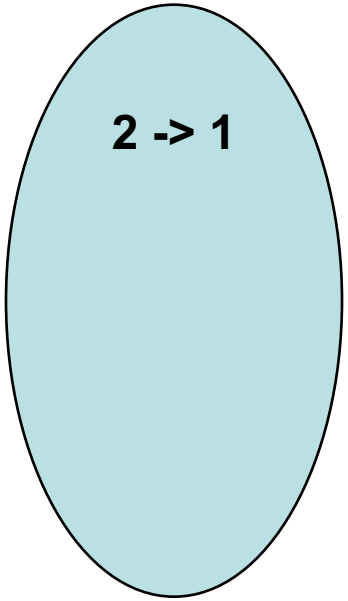
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

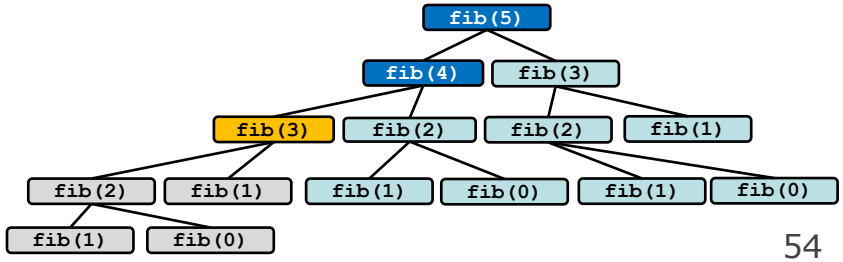
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

1 1

cache



call tree



Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

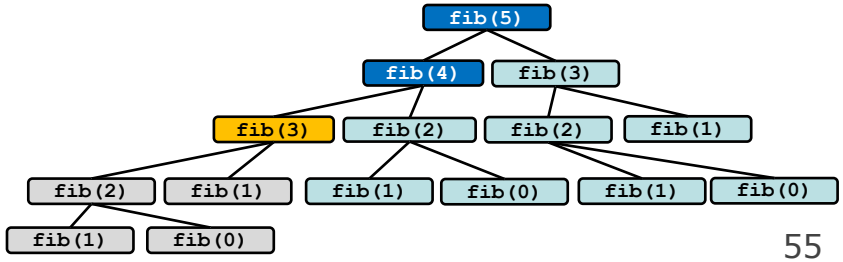
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

1 1

cache

2 -> 1
3 -> 2

call tree



Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

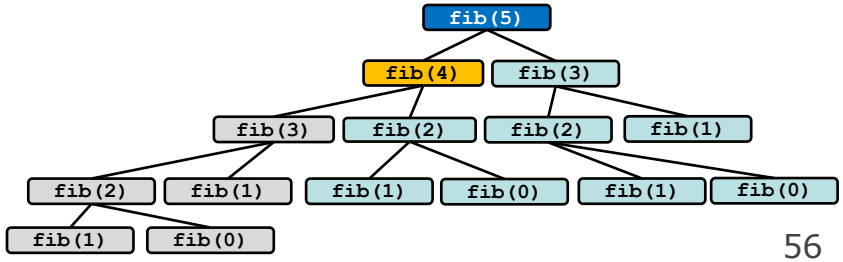
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

2

cache

2 -> 1
3 -> 2

call tree



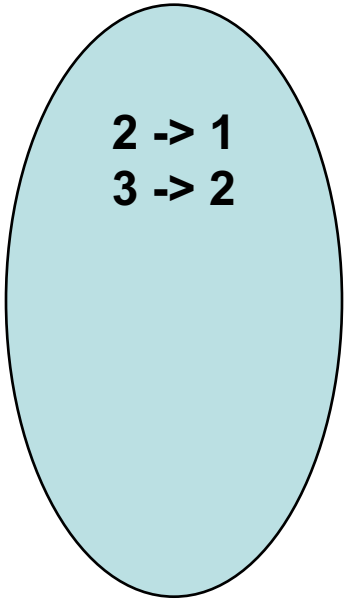
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

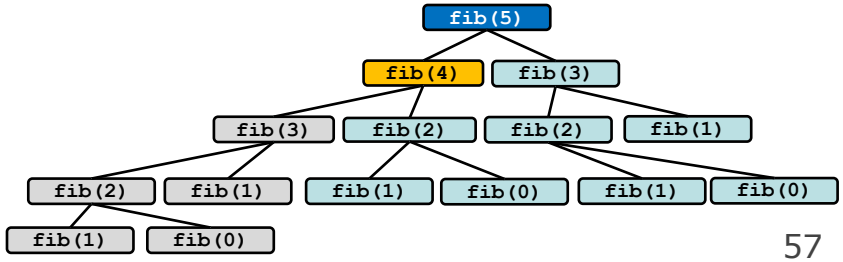
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

2

cache



call tree

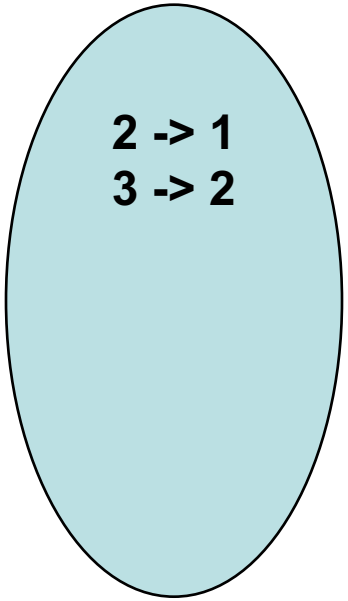


Memoized Stack Trace

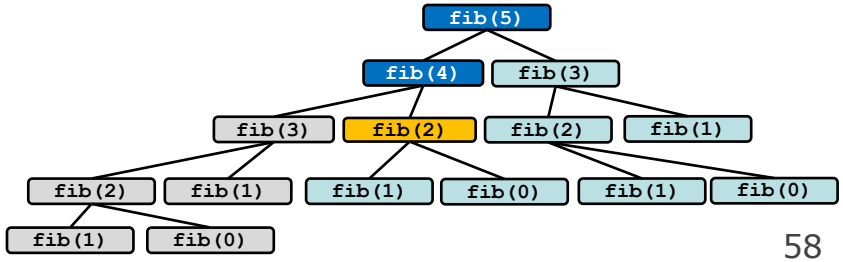
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

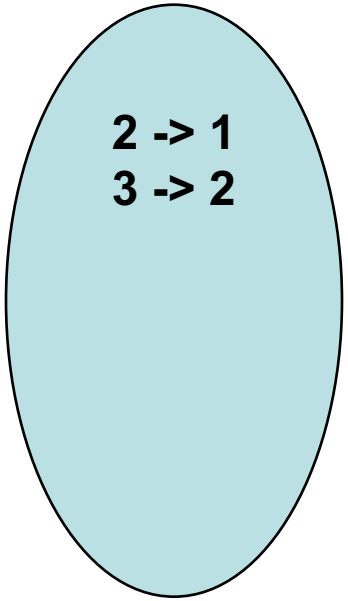


Memoized Stack Trace

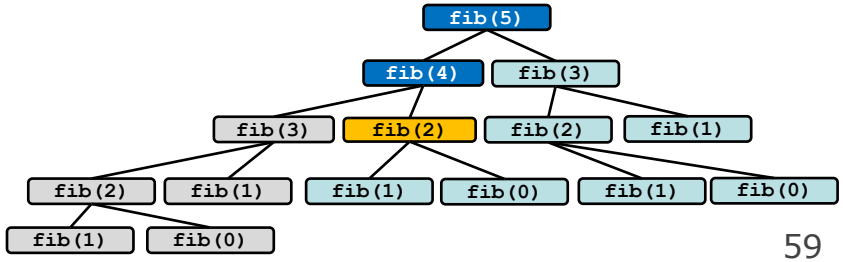
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
int fibonacci(int i, Map<int, int>& cache) { // i=2
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree



Memoized Stack Trace

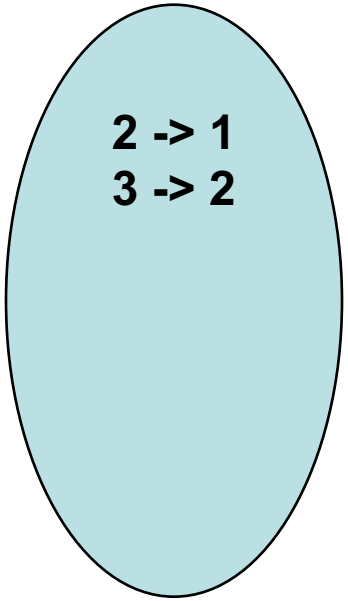
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

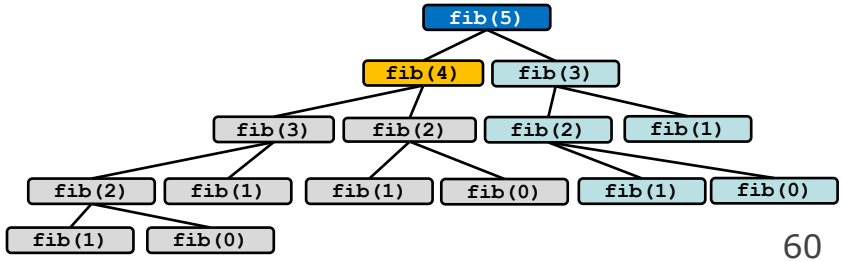
2

1

cache



call tree

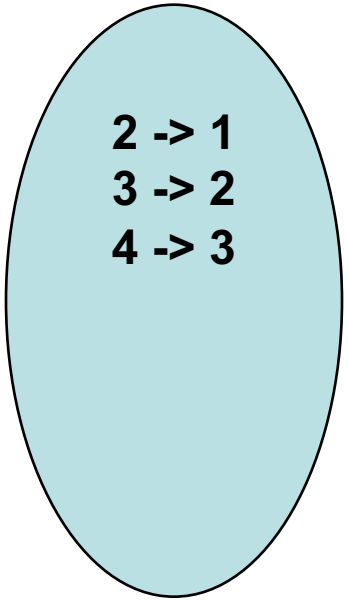


Memoized Stack Trace

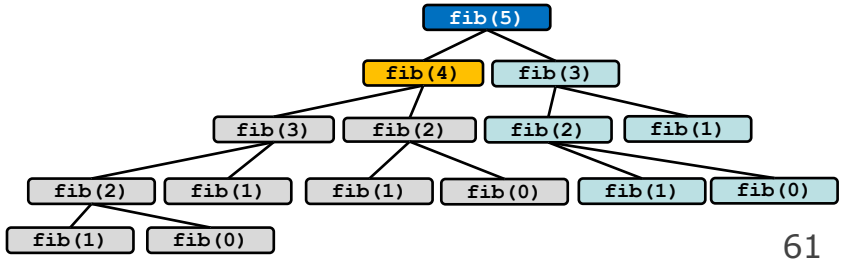
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=4
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {                                  2                    1
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

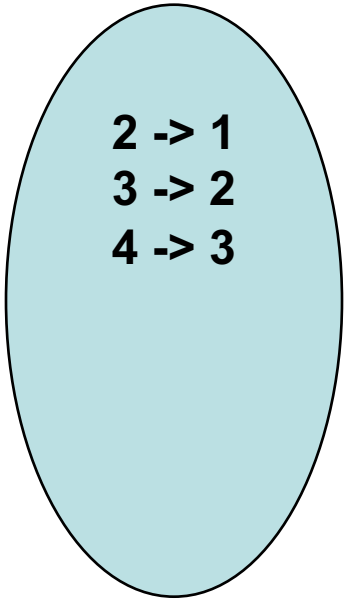


Memoized Stack Trace

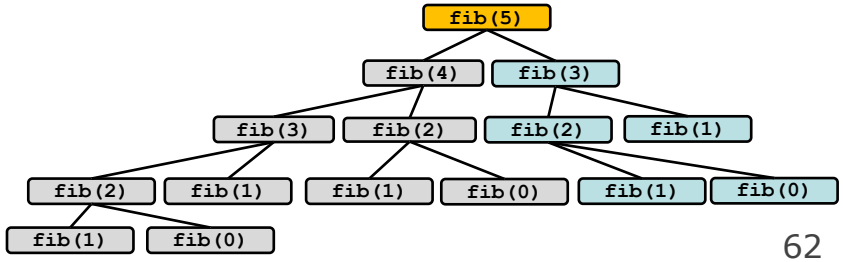
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {                                    3
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree



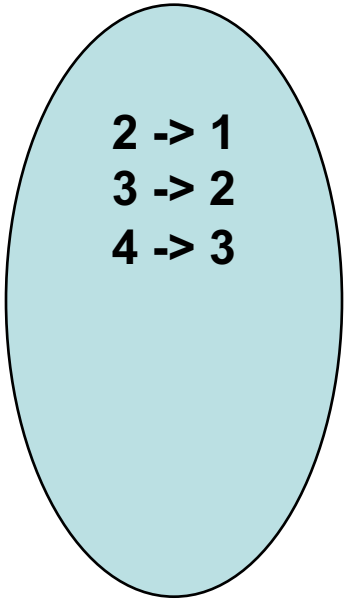
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

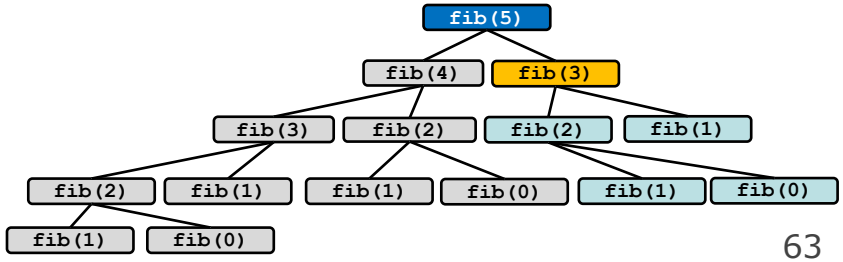
```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

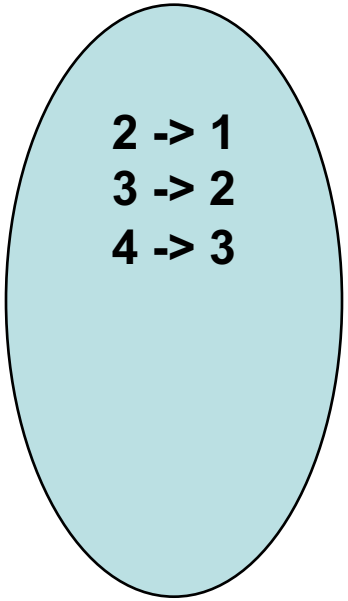


Memoized Stack Trace

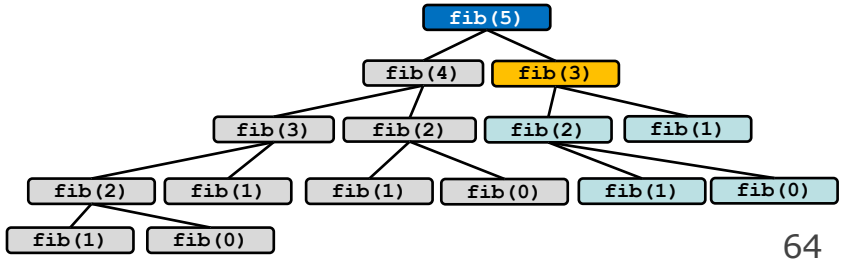
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
int fibonacci(int i, Map<int, int>& cache) { // i=3
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

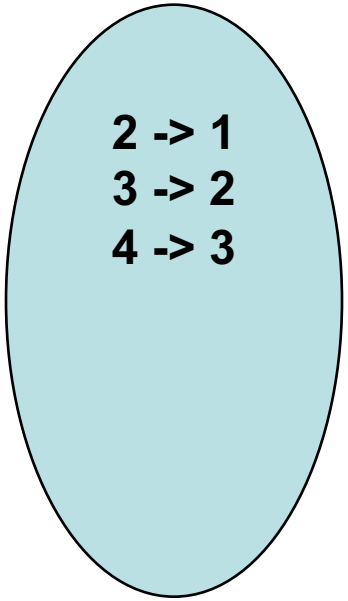


Memoized Stack Trace

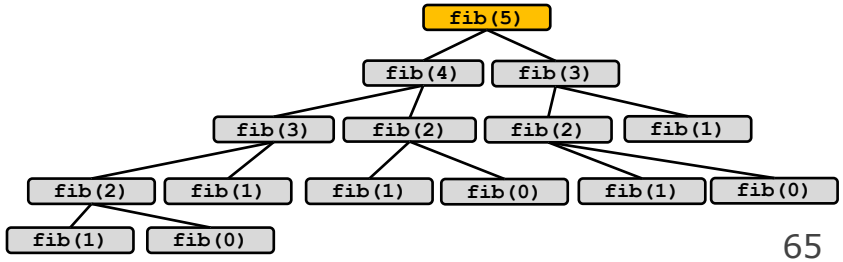
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {                          3                    2
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



call tree

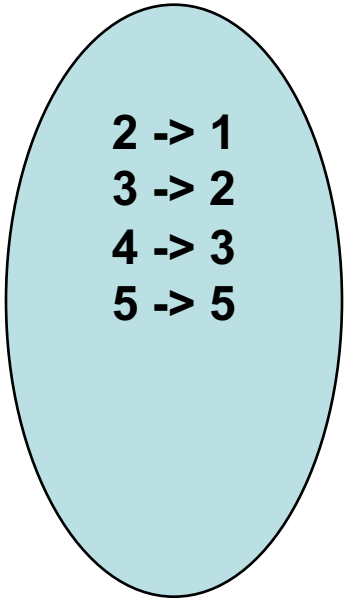


Memoized Stack Trace

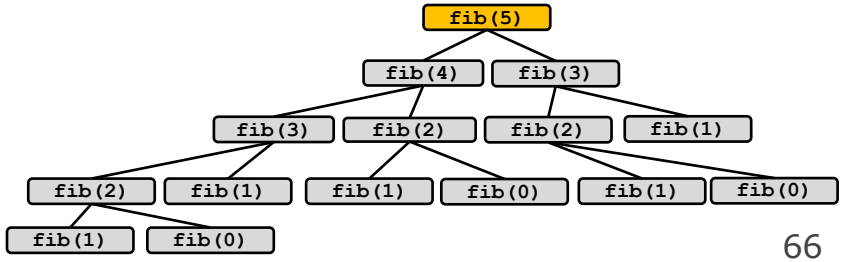
```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);
```

```
int fibonacci(int i, Map<int, int>& cache) { // i=5
    if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {                          3                    2
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

cache



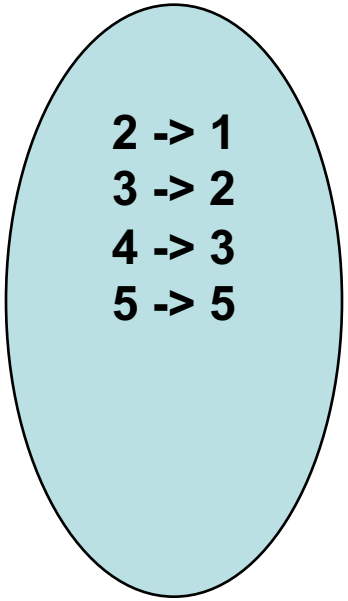
call tree



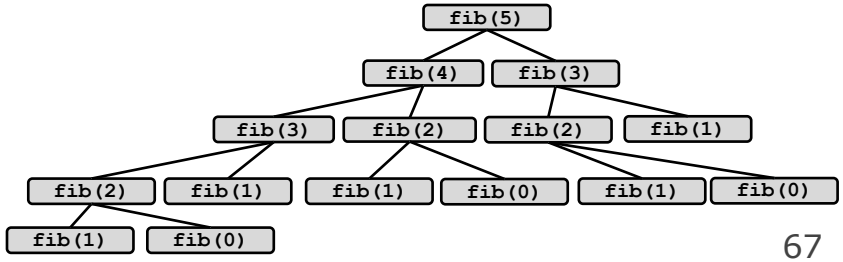
Memoized Stack Trace

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);  // 5
```

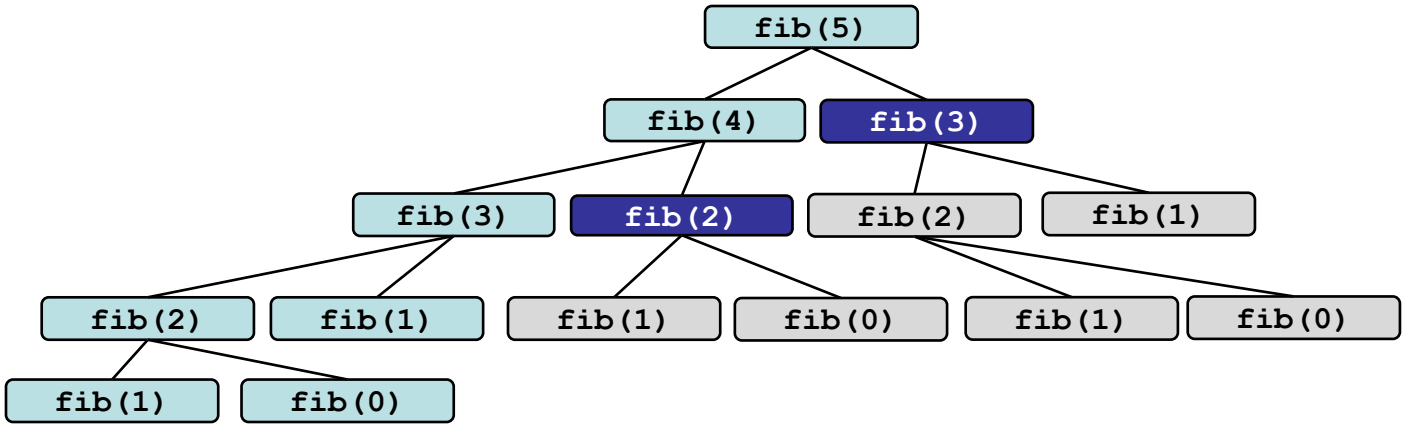
cache

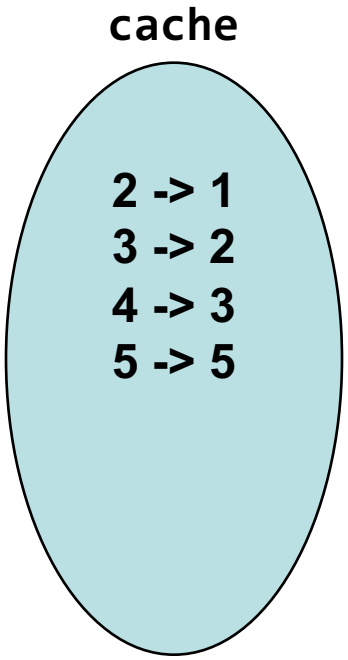


call tree



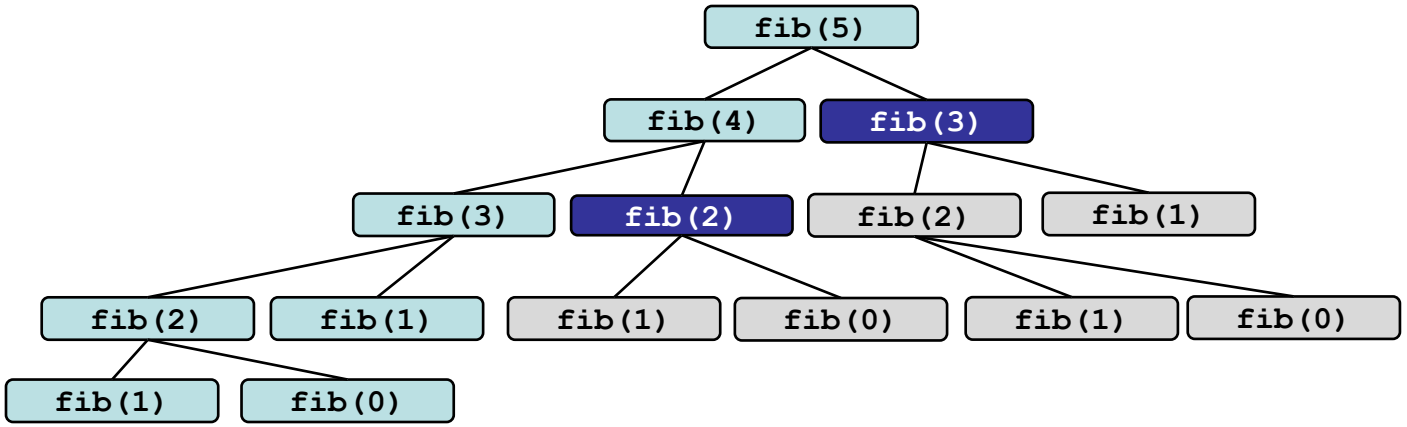
Memoized Fibonacci





Cached | Recurse | Skipped

Memoized Fibonacci



cache

2 -> 1
3 -> 2
4 -> 3
5 -> 5

What is the runtime of this function?

$O(2^N)$ is technically correct for worst-case, but we can be more precise.

The branch factor for the memoized version becomes 1, because once we calculate fib(n-1), fib(n-2) will always be cached. Thus, the runtime is O(n)!

Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

Wrapper Functions

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);  // 5
```

- The above function signature isn't ideal; it requires the client to know to pass in an (empty) map.
- In general, the parameters we need for our recursion will not always match those the client will want to pass.
- Is there a way we can remove that requirement, while still memoizing?
- **YES!** A "wrapper" function is a function that "wraps" around the first call to a recursive function to abstract away any additional parameters needed to perform the recursion.

That's a Wrap(per)!

```
// "Wrapper" function that returns the nth Fibonacci number.
// This version calls the recursive version with an empty cache.
int fibonacci(int i) {
    HashMap<int, int> cache;
    return fibonacci(i, cache);
}

// Recursive function that returns the nth Fibonacci number.
// This version uses memoization.
int fibonacci(int i, HashMap<int, int>& cache) {
    if (i < 0) {
        throw "Illegal negative index";
    } else if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

crawl exercise

- Write a function **`crawl`** that accepts a file name as a parameter and prints information about that file.
  - If the name represents a **normal file**, just print its name.
  - If the name represents a **directory**, print its name and information about every file/directory inside it, indented.

```
course
    handouts
        syllabus.doc
        lecture-schedule.xls
    homework
        1-gameoflife
            life.cpp
            life.h
            GameOfLife.pro
```

  - **recursive data**: A directory can contain other directories.

Stanford C++ files

```
#include "filelib.h"
```

Function	Description
`createDirectory(`***name***`)`	creates a a new directory with given path name
`deleteFile(`***name***`)`	removes file from disk
`fileExists(`***name***`)`	whether this file exists on the disk
`getCurrentDirectory()`	returns directory the current C++ program runs in
`getExtension(`***name***`)`	returns file's extension, e.g. `"foo.cpp"` → `".cpp"`
`getHead(`***name***`)`, `getTail(`***name***`)`	separate a file path into the directory and file part; for `"a/b/c/d.txt"`, head is `"a/b/c"`, tail is `"d.txt"`
`isDirectory(`***name***`)`	returns whether this file name represents a directory
`isFile(`***name***`)`	returns whether this file name represents a regular file
`listDirectory(`***name***`, `***v***`)`	fills the given `Vector<string>` with the names of all files contained in the given directory
`readEntireFile(`***name***`, `***v***`)`	reads lines of the given file into a vector of strings
`renameFile(`***old***`, `***new***`)`	changes a file's name

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

Wrapper Functions

We cannot vary the indentation without an extra parameter:

```
void crawl(string filename, string indent) {
```

This doesn't match the function signature we must implement, so we need to use a wrapper function.

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?
- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

crawl solution

```
// Wrapper to print out this file and everything in it.
void crawl(const string& filename) {
    crawl(filename, "");
}

// Prints information about this file recursively,
// and (if it is a directory) any files inside it.
void crawl(const string& filename, const string& indent) {
    cout << indent << getTail(filename) << endl;
    if (isDirectory(filename)) {
        // recursive case; print contained files/dirs
        Vector<string> filelist;
        listDirectory(filename, filelist);
        for (string subfile : filelist) {
            crawl(filename + "/" + subfile,
                  indent + "    ");
        }
    }
}
```

Default Parameters

We could also use a default parameter *in this case:*

```
void crawl(const string& filename, const string& indent = "") {
    cout << indent << getTail(filename) << endl;
    if (isDirectory(filename)) {
        // recursive case; print contained files/dirs
        Vector<string> filelist;
        listDirectory(filename, filelist);
        for (string subfile : filelist) {
            crawl(filename + "/" + subfile,
                  indent + "    ");
        }
    }
}
```

But for the purposes of this example we'll stick with a wrapper function. Sometimes default parameters can do the trick, but not always – e.g. if we need to pass by reference, or initialize a data structure dynamically).

Plan For Today

- **Recap:** Recursion So Far
- More Practice: Reversing a File
- Recursion: Runtime
- Announcements
- Memoization
- Wrapper Functions

Next time: fractals (Recursion you can see!)