CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 12   Feb 11, 2013

Review For Test 1
Pointers and Recursion

Upcoming Test

Pointers, Recursion, Big Oh, Oh My

The test is on (almost) everything we've done so far. I'll ask about **debugging** and **design** (all my tests will), and cover topics from the homeworks. You'll need to know the syntax for **pointers** in C++, how to make them, pass them, dereference them. Understand how and why **recursion** works, and be able to write/debug recursive functions. Be able to identify the **computational complexity** of short algorithms. Know how and why **linked lists**, **binary search trees**, and simple **sorting algorithms** work.

Announcements

1. Student-led **review session** this week. When?
   Tue at 3?
   Wed at 1?
   Thurs at 2?

2. Different **review session** possible, more helpful info later.

Announcements

3. **RetroGrade Accounts** for 24 people (this class and the other) had to be merged because there were two. If you can't log in, email me *with your student id and email addresses you might have used*.

Test 1 Contents

1. Debugging
2. Design
3. Pointers
4. Linked Lists
5. Recursion
6. Binary Search Trees
7. Computational Complexity (Big Oh)
8. Sorting Algorithms

Test Format

Open note, closed gizmo. Cheaters will be thrown into The Great Pit of Carkoon.

Will take 50 minute class period, extension only if you have a doctor's note. If you totally rock you should finish in 15 minutes.

Questions will be mostly be short answer (e.g. less than a sentence) or short code (less than 10 lines).

Test 1 is 40 points.

Debugging

```
int main() {
  fibonacci(0, 1, 100);
}

void fibonacci(int a, int b,
               int max_val) {
  int c = a + b;
  cout << c << endl;
  if (c != max_val) {
    fibonacci(b, c, max_val);
  }
}
```

Spot the bug, but
don't say anything

Debugging

```
int main() {
  fibonacci(0, 1, 100);
}

void fibonacci(int a, int b,
               int max_val) {
  int c = a + b;
  cout << c << endl;
  if (c != max_val) {
    fibonacci(b, c, max_val);
  }
}
```

Console output:

```
1
2
3
5
8
13
21
34
55
89
144
233
```

... ends with giant numbers and a segfault.

Design

Design a special binary search tree where duplicates are permissible but are not allowed to take up extra space. What would you need to change? Which algorithms would be affected?

Operations:

init_node	report	insert
remove	size	contains
to_array		

Design

Maybe something like this to start it off?

```
struct bt_node {
    bt_node* left;
    bt_node* right;
    int value;
    int num_values;
};
```

Operations:

init_node	report	insert
remove	size	contains
to_array		

Pointers

Know thy syntax for values, pointers, how to get an address, how to dereference a variable.

```
int betty = 42;
int* betty_ptr = &betty;
cout << "Betty's number is: " << ???? << " "
     << "and her address is " << ????;
```

Pointers

```
int betty = 42;
int* betty_ptr = &betty;
cout << "Betty's number is: " << betty << " "
    << "and her address is " << betty_ptr;


int betty = 42;
int* betty_ptr = &betty;
cout << "Betty's number is: " << *betty_ptr << " "
    << "and her address is " << betty_ptr;
```

They both print out something like:

Betty's number is: 42 and her address is 0x7fff503dd994

Linked Lists

We never implemented a random access function to get a value by index. Let's change that. Write the linked list 'get(int idx)' function. Be sure to catch bad input!

```
int get(node* top, int idx) {
  // implement me w/ while loop
}
```

Recursion

Recursive functions *call themselves.*

Remember:

recursion: see *recursion.*

Recursion

```
int sum(int num) {
  if (num > 0) {
    // add num and the result of
    // recursing with num-1.
    return num + sum(num-1);
  } else {
    // stopping condition
    return 0;
  }
}
```

If we call sum(4), here's what happens.

```
int sum(int num) {
   if (num > 0) {
      // add num and the result of
      // recursing with num-1.
      return num + sum(num-1);
   } else {
      // stopping condition
      return 0;
   }
}
```

num is 4.

```
int sum(int num) {   so we enter this
  if (num > 0) {←── if statement
    // add num and the result of
    // recursing with num-1.
    return num + sum(num-1);
  } else {
    // stopping condition
    return 0;
  }
}
```

**we'll return 4 plus
whatever sum(3)
returns.**

now num is 3

```
int sum(int num) {
    if (num > 0) {          ← enter 'if'...
        // add num and the result of
        // recursing with num-1.
        return num + sum(num-1);
    } else {
        // stopping condition
        return 0;
    }
}
```

return 3 plus whatever sum(2) returns.

now num is 2

```
int sum(int num) {
   if (num > 0) {          ← enter 'if'...
      // add num and the result of
      // recursing with num-1.
      return num + sum(num-1);
   } else {
      // stopping condition
      return 0;
   }
}
```

return 2 plus whatever sum(1) returns.

now num is 1

```
int sum(int num) {
    if (num > 0) {        ← enter 'if'...
        // add num and the result of
        // recursing with num-1.
        return num + sum(num-1);
    } else {
        // stopping condition
        return 0;
    }
}
```

return 1 plus whatever sum(0) returns.
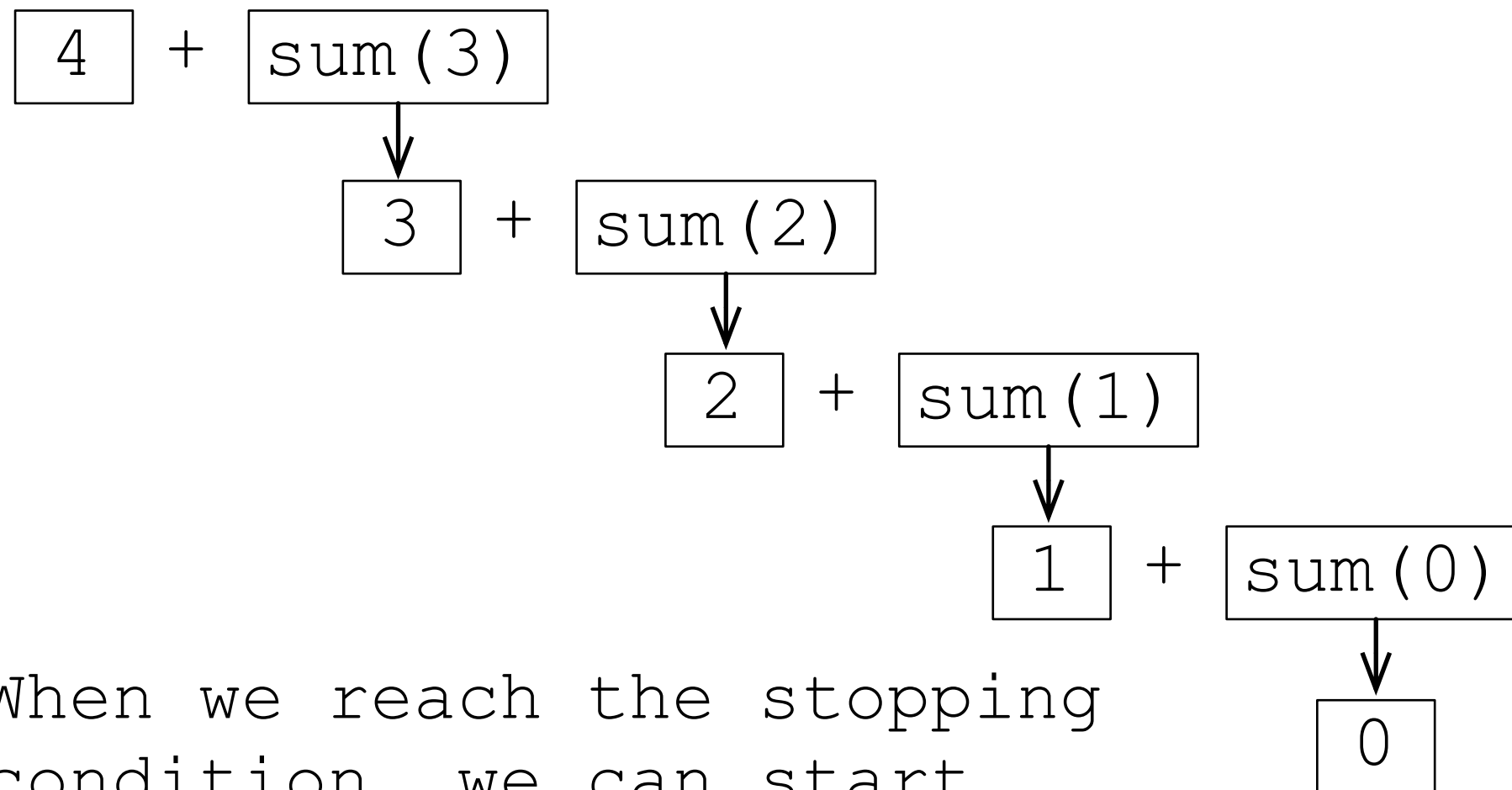
now num is 0

```
int sum(int num) {              DO NOT
    if (num > 0) {              enter 'if'...
        // add num and the result of
        // recursing with num-1.
        return num + sum(num-1);
    } else {
        // stopping condition
        return 0;
    }
}
```
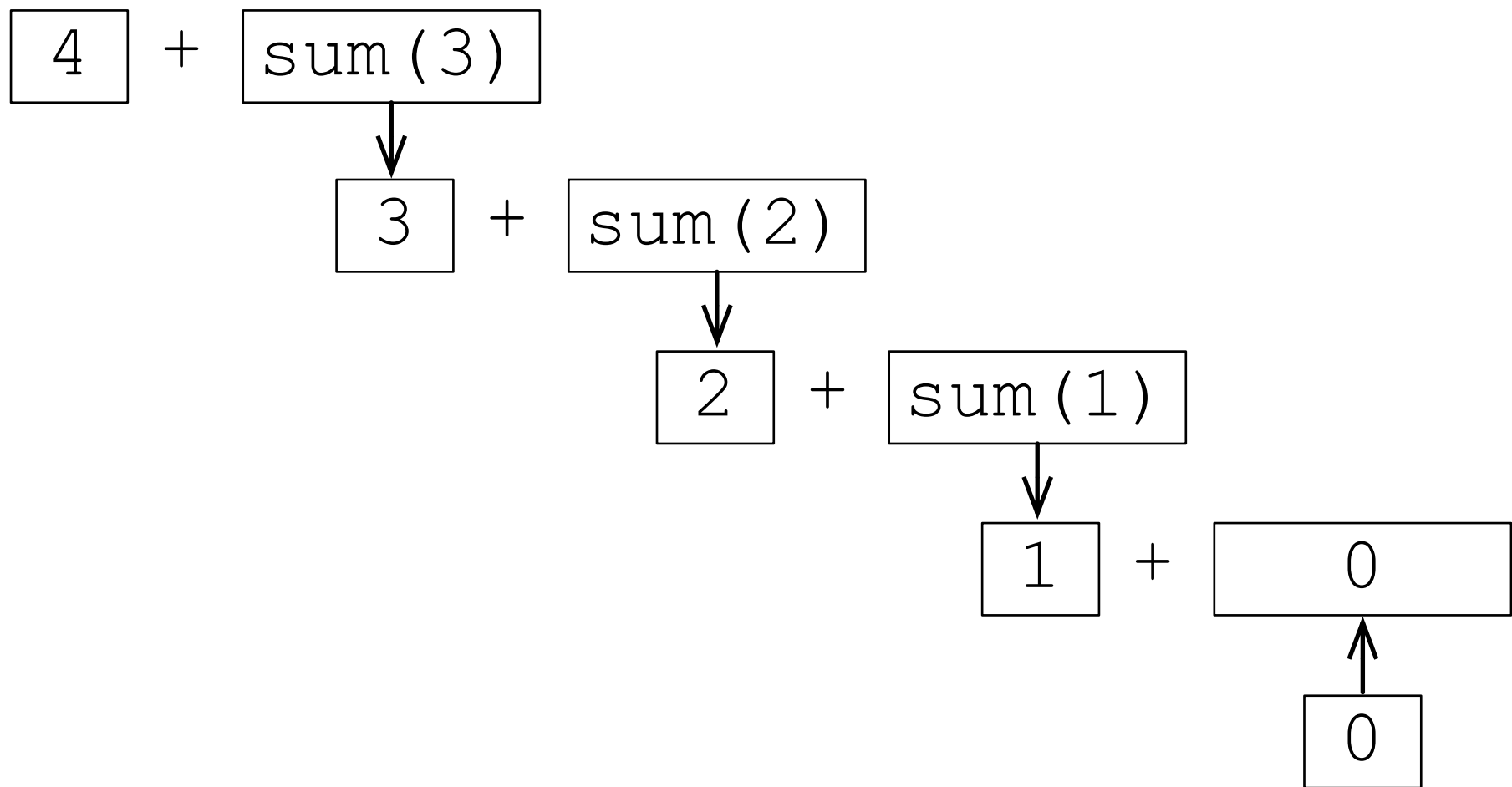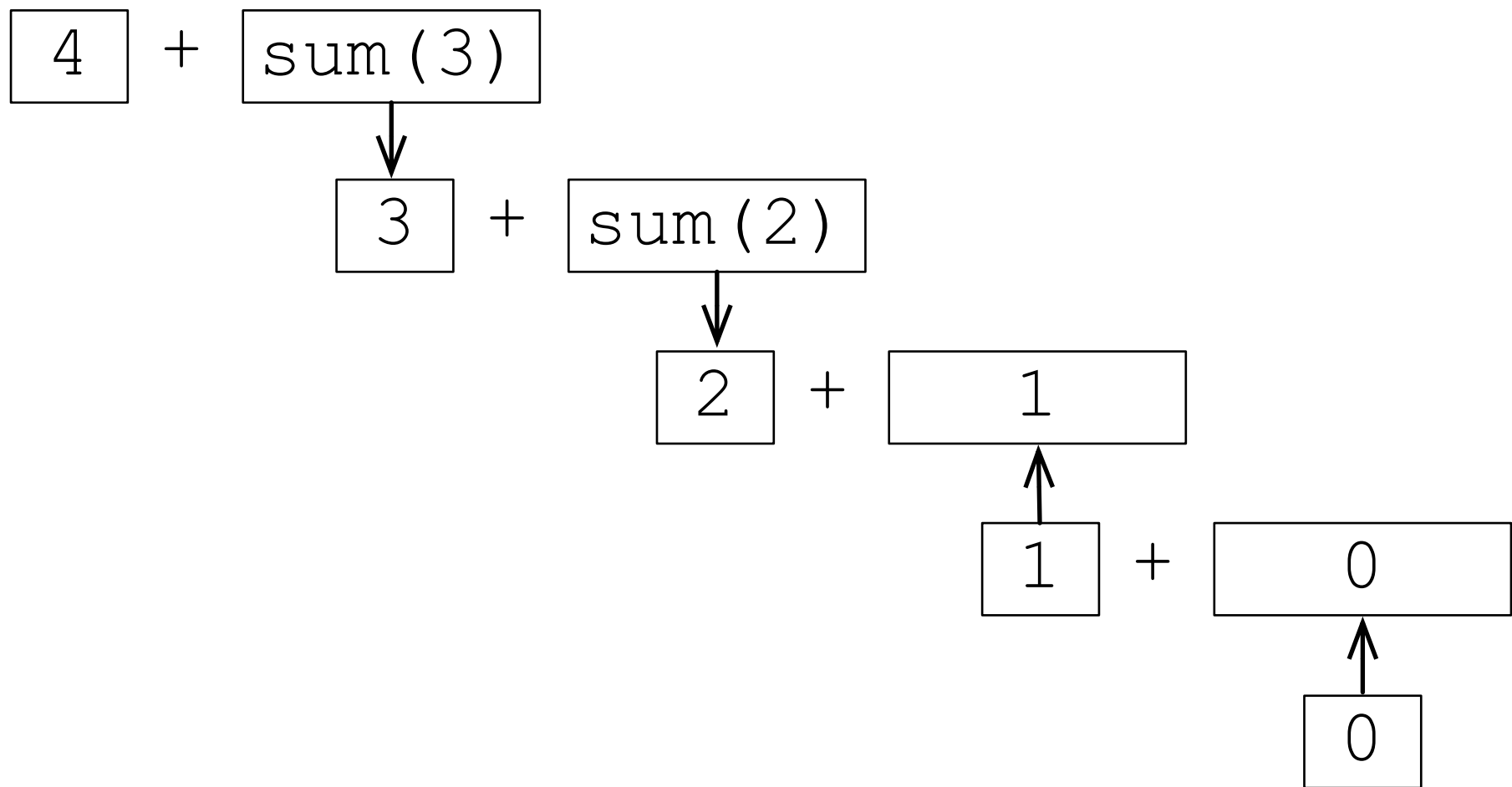
**We return 0 and we
do not recurse.**
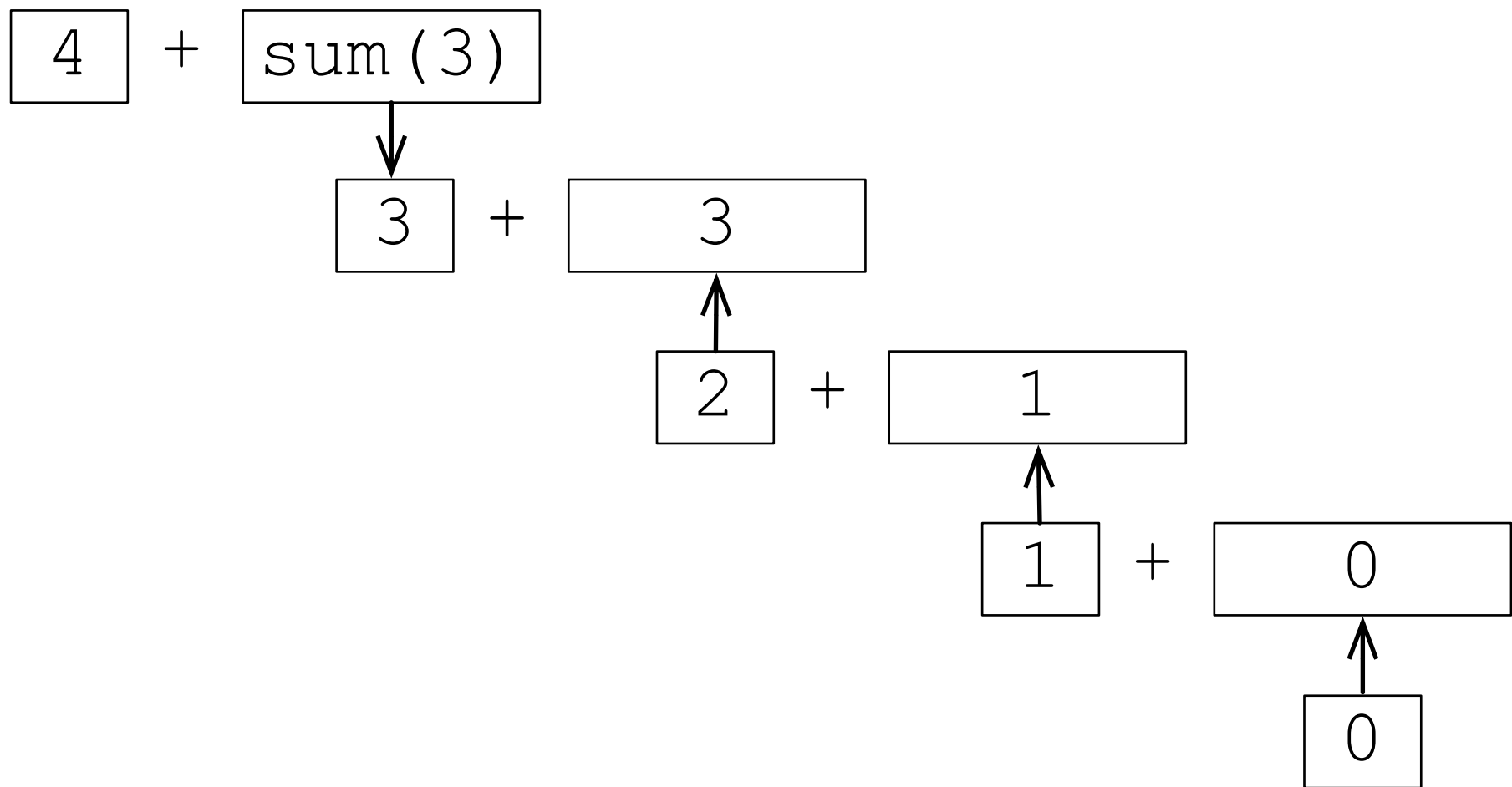
That recursive call stack looks
like this:

| 4 | + | sum(3) |
      ↓
      | 3 | + | sum(2) |
            ↓
            | 2 | + | sum(1) |
                  ↓
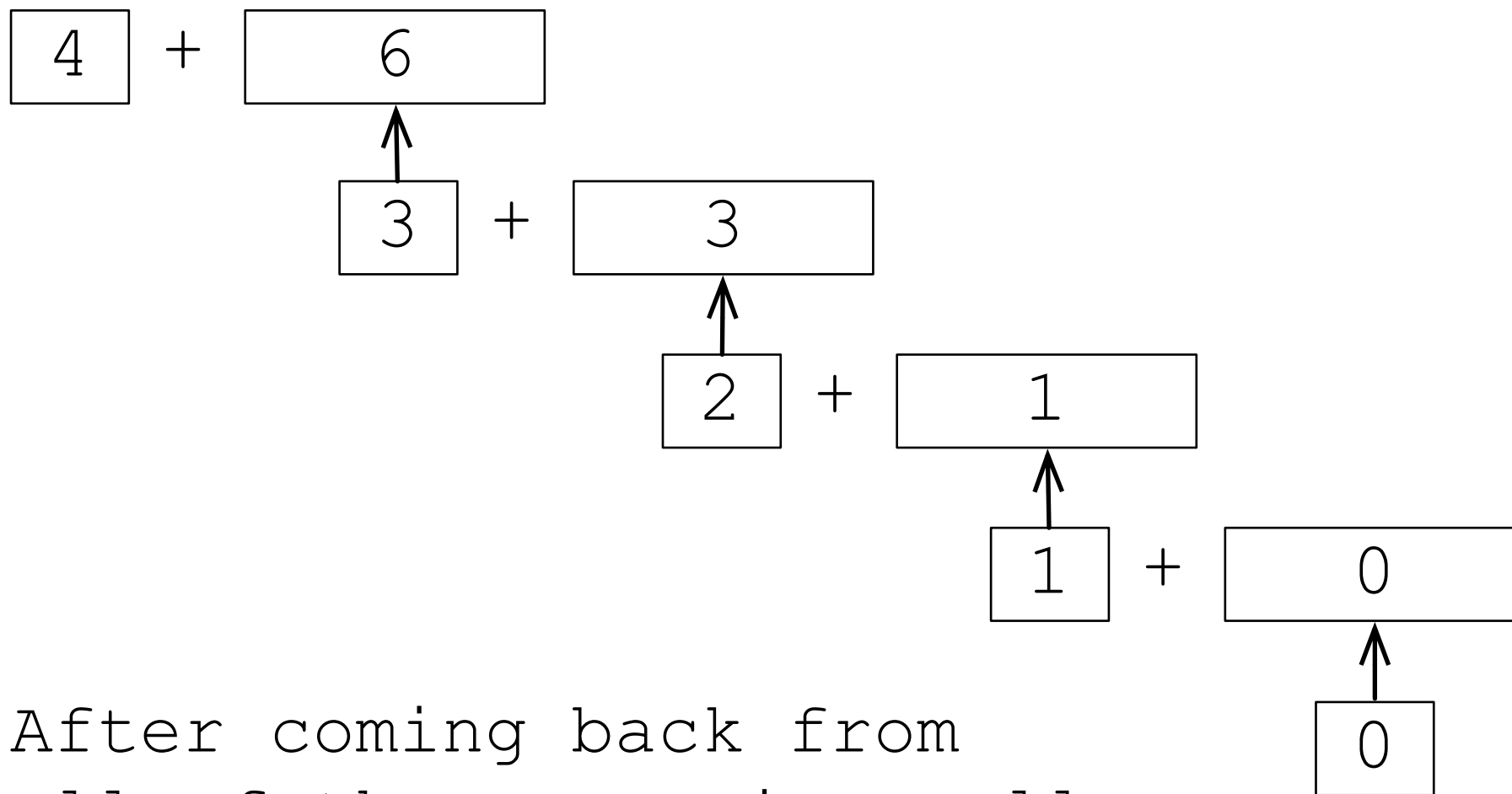 | 1 | + | sum(0) |
                        ↓
                        | 0 |

When we reach the stopping
condition, we can start
filling in the sum() calls
with the proper return value.

4 + sum(3)

3 + sum(2)

2 + 1

1 + 0

0

4 + sum(3)

↓

3 + 3

↑

2 + 1

↑

1 + 0

↑

0

4 + 6

3 + 3

2 + 1

1 + 0

0

After coming back from
all of the recursive calls,
the result is 4 + 6 = **10.**

Computational Complexity

Computational Complexity analysis is one of the most important intellectual tools you'll learn. Don't just be able to get the right answer. Understand the intuition behind why this is interesting, and how it can save you a *lot of mental duress*. If you know in advance that an algorithm is fast, slow, intractable, then you can plan your life around it.

Consider the length of time it takes to perform basic operations using real hardware...

Typical Operation Times

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Source: Peter Norvig. http://norvig.com/21-days.html

Complexity Example

```
for (int i=0; i < N; i++) {
  for (int j=0; j < N; j++) {
   product[i][j] = i * j;
  }
}
```

What's the computational complexity of this algorithm? How many times will we run the inner loop?

We say this is **O(n²)** because the runtime of the algorithm *is proportional to the square of the input size.*

Complexity: best, avg, worst

We're often concerned about the best, the average (or typical), and worst-case complexity of an algorithm. We can optimize for each, based on whatever situation we have. We can roll our own algorithm or we can use something well-known.

When to optimize

Best case: optimize for situations where we have reasonable understanding of the data, and know the best case will be common.

Average case: optimize for the typical case when input is not particularly weird, but we don't know what is coming. This is what most algos do.

Worst case: optimize worst-case scenario to prevent catastrophes: missile defense, air traffic control, real-time medical hardware, etc.

Sorting Algorithms

We used pointers (by way of C++'s vector template class) and recursion in the sorting homework.

Algorithms differ in complexity in their best/avg/worst case scenarios.

To sort **n** numbers (using best variant of each):

Algo	Worst	Avg	Best
bubble sort	$O(n^2)$	$O(n^2)$	$O(n)$
quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$

Test Sample Questions

I will put sample questions up as soon as I have time to write them.

The test will be slightly easier than the sample questions, and I will base the test questions on the ones posed in the sample questions.

I'll ask about design, debugging, pointers, recursion, linked lists, binary search trees, computational complexity, and sorting algorithms.

Next Time: B-Tree intro

I will do more review next time if anybody asks, but I will prepare a thing on B-Trees to get that ball rolling in case nobody asks.