CS 600.226: Data Structures

Michael Schatz

Sept 19 2018

Lecture 9. Stacks

Agenda

1. ***Review HW2***
2. ***Recap on Sorting***
3. ***Stacks***

Assignment 2: Due Friday Sept 21 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment02/README.md

Assignment 2: Arrays of Doom!

Out on: September 14, 2018
Due by: September 21, 2018 before 10:00 pm
Collaboration: None
Grading:

Functionality 65%
ADT Solution 20%
Solution Design and README 5%
Style 10%

Overview

The second assignment is mostly about arrays, notably our own array specifications and implementations, not just the built-in Java arrays. Of course we also once again snuck a small ADT problem in there...

Note: The grading criteria now include **10% for programming style**. Make sure you use Checkstyle with the correct configuration file from Github!

Assignment 2: Due Friday Sept 21 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment02/README.md

Problem 1: Revenge of Unique (30%)

You wrote a small Java program called Unique for Assignment 1. The program accepted any number of command line arguments (each of which was supposed to be an integer) and printed each unique integer it received back out once, eliminating duplicates in the process.

For this problem, you will implement a new version of Unique called ***UniqueRevenge*** with two major changes:

- First, you are no longer allowed to use Java arrays (nor any other advanced data structure), but you can use our Array interface and our SimpleArray implementation from lecture (also available on github)
- Second, you're going to modify the program to read the integers from standard input instead of processing the command line.

Assignment 2: Due Friday Sept 21 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment02/README.md

Problem 2: Flexible Arrays (20%)

Develop an algebraic specification for the abstract data type FlexibleArray which works like the existing Array ADT for the most part **except** that both its **lower** and its **upper** index bound are set when the array is created. The lower as well as upper bound can be **any** integer, provided the lower bound is **less than or equal** the upper bound.

Write up the specification for FlexibleArray in the format we used in lecture and **comment** on the design decisions you had to make. Also, tell us what kind of array **you** prefer and why.

Hints

- A FlexibleArray for which the lower bound equals the upper bound has exactly one slot.
- Your FlexibleArray is **not** the Array ADT we did in lecture; it doesn't have to support the exact same set of operations.

Assignment 2: Due Friday Sept 21 @ 10pm

https://github.com/schatzlab/datastructures2018/blob/master/assignments/assignment02/README.md

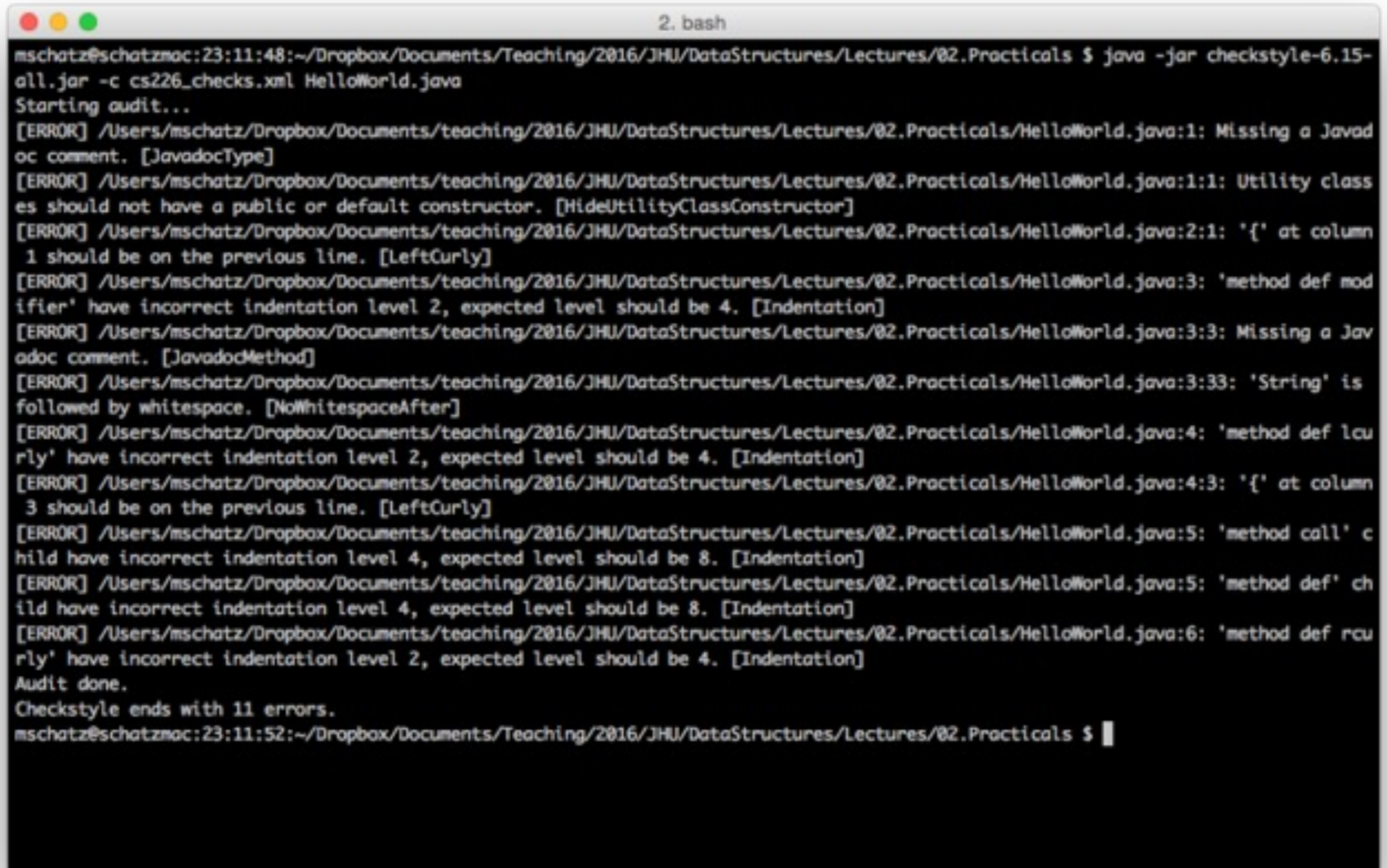
Problem 3: Sparse Arrays (35%)

A **sparse** array is an array in which **relatively few** positions have values that differ from the initial value set when the array was created. For sparse arrays, it is wasteful to store the value of **all** positions explicitly since **most of them never change** and take the default value of the array. Instead, we want to store positions that **have actually been changed**.

For this problem, write a class SparseArray that implements the Array interface we developed in lecture (the same interface you used for Problem 1 above). **Do not modify the Array interface in any way!** Instead of using a plain Java array like we did for SimpleArray, your SparseArray should use a **linked list** of Node objects to store values, similar to the ListArray from lecture (and available in github). However, your nodes no longer store just the **data** at a certain position, they also store **the position itself**!

Introduction to Checkstyle

http://checkstyle.sourceforge.net/



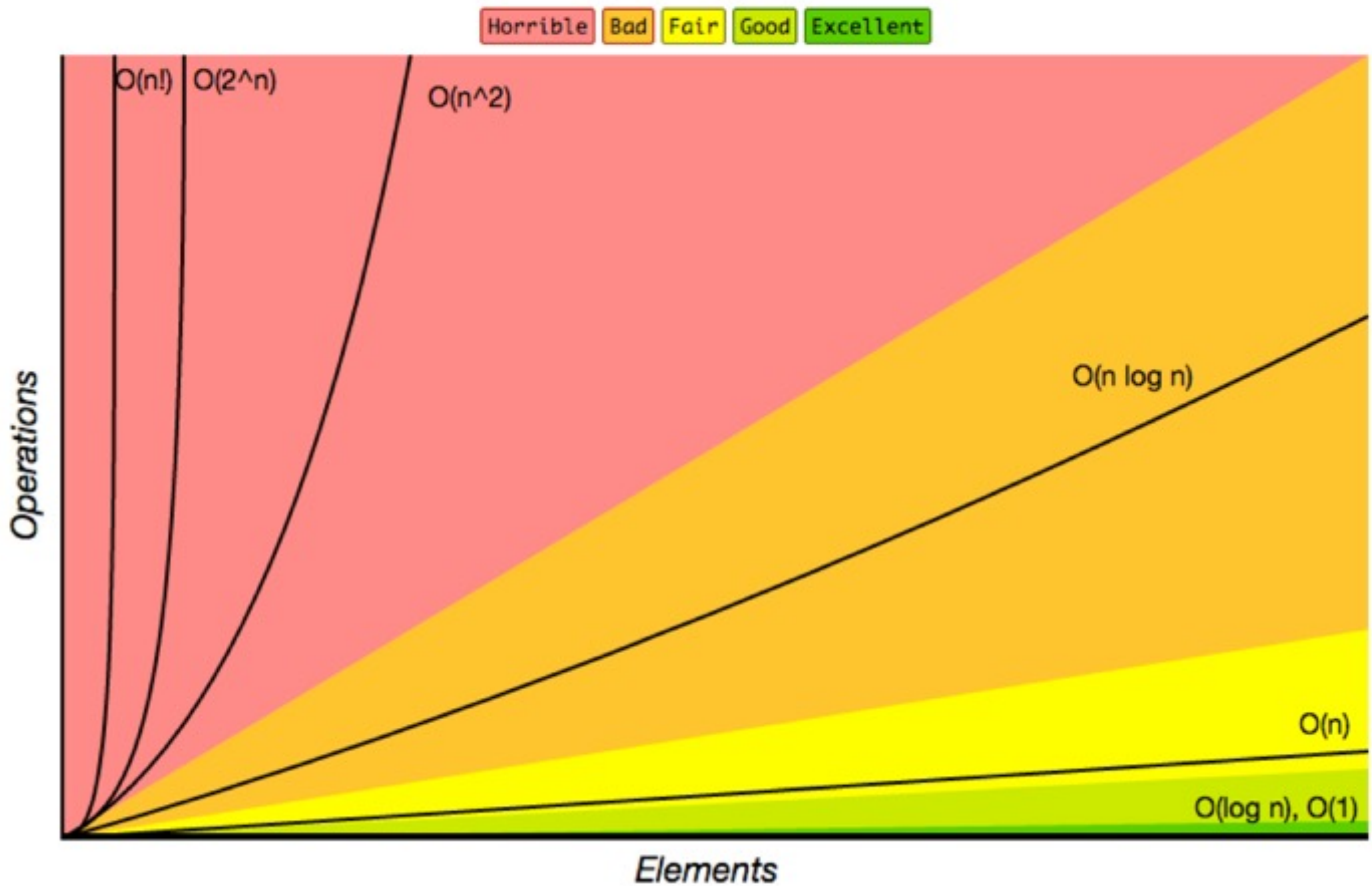
```
2. bash
mschatz@schatzmac:23:11:48:~/Dropbox/Documents/Teaching/2016/JHU/DataStructures/Lectures/02.Practicals $ java -jar checkstyle-6.15-all.jar -c cs226_checks.xml HelloWorld.java
Starting audit...
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:1: Missing a Javadoc comment. [JavadocType]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:1:1: Utility classes should not have a public or default constructor. [HideUtilityClassConstructor]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:2:1: '{' at column 1 should be on the previous line. [LeftCurly]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:3: 'method def modifier' have incorrect indentation level 2, expected level should be 4. [Indentation]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:3:3: Missing a Javadoc comment. [JavadocMethod]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:3:33: 'String' is followed by whitespace. [NoWhitespaceAfter]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:4: 'method def lcurly' have incorrect indentation level 2, expected level should be 4. [Indentation]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:4:3: '{' at column 3 should be on the previous line. [LeftCurly]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:5: 'method call' child have incorrect indentation level 4, expected level should be 8. [Indentation]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:5: 'method def' child have incorrect indentation level 4, expected level should be 8. [Indentation]
[ERROR] /Users/mschatz/Dropbox/Documents/teaching/2016/JHU/DataStructures/Lectures/02.Practicals/HelloWorld.java:6: 'method def rcurly' have incorrect indentation level 2, expected level should be 4. [Indentation]
Audit done.
Checkstyle ends with 11 errors.
mschatz@schatzmac:23:11:52:~/Dropbox/Documents/Teaching/2016/JHU/DataStructures/Lectures/02.Practicals $
```

```
$ java -jar datastructures2018/resources/checkstyle-8.12-all.jar \
      -c datastructures2018/resources/cs226_checks.xml HelloWorld.java
```

Agenda

1. ***Review HW2***
2. ***Recap on Sorting***
3. ***Stacks***

Growth of functions



http://bigocheatsheet.com/

Trying every permutation

```
  public static void main(String [] args) {
    if (args.length == 0) {
      System.out.println("Permute num");
      return;
    }

    int len = Integer.parseInt(args[0]);
    int [] keys = new int[len];
    for (int i = 0; i < len; i++) { keys[i] = i+1; }
    permute(keys, 0, len-1);
    System.err.println("There are " + numtries
                       + " permutations of " + len + " items.");
  }
}
```

```
$ for i in `seq 1 20`;
  do echo $i; java Permute $i > $i.log ; done
1
There are 1 permutations of 1 items.
2
There are 2 permutations of 2 items.
3
There are 6 permutations of 3 items.
4
There are 24 permutations of 4 items.
5
There are 120 permutations of 5 items.
```

Why Sort?

Sorting is very powerful to organize large amounts of data
Becomes a core routine in many data structures

When data are sorted you can do ***binary search***!
- I'm thinking of a number between 1 and 1,000,000
- ***How many hi/lo guesses will it take to figure it out?***

lg(1,000,000) = 20

How many hi/lo guesses to find my special number?
26 05 38 28 93 81 71 15 96 33 99 13 58 96 09

Same Data, Sorted Order
05 09 13 15 26 28 33 38 58 71 81 93 96 96 99

Why Sort?

Sorting is very powerful to organize large amounts of data
Becomes a core routine in many data structures

When data are sorted you can do ***binary search***!
- I'm thinking of a number between 1 and 1,000,000
- ***How many hi/lo guesses will it take to figure it out?***

How many hi/lo guesses to find my special number?

`26 05 38 28 93 81 71 15 96 33 99 13 58 96 09`

Same Data, Sorted Order

`05 09 13 15 26 28 33 38 58 71 81 93 96 96 99`

Why Sort?

Sorting is very powerful to organize large amounts of data
Becomes a core routine in many data structures

When data are sorted you can do ***binary search***!
• I'm thinking of a number between 1 and 1,000,000
• ***How many hi/lo guesses will it take to figure it out?***

How many hi/lo guesses to find my special number?
`26 05 38 28 93 81 71 15 96 33 99 13 58 96 09`

Same Data, Sorted Order
`05 09 13 15 26 28 33 38 58 71 81 93 96 96 99`

Why Sort?

Sorting is very powerful to organize large amounts of data
Becomes a core routine in many data structures

When data are sorted you can do ***binary search***!
- I'm thinking of a number between 1 and 1,000,000
- ***How many hi/lo guesses will it take to figure it out?***

How many hi/lo guesses to find my special number?

```
26 05 38 28 93 81 71 15 96 33 99 13 58 96 09
```

Same Data, Sorted Order

```
05 09 13 15 26 28 33 38 58 71 81 93 96 96 99
```

Why Sort?

Sorting is very powerful to organize large amounts of data
Becomes a core routine in many data structures

When data are sorted you can do ***binary search***!
• I'm thinking of a number between 1 and 1,000,000
• ***How many hi/lo guesses will it take to figure it out?***

How many hi/lo guesses to find my special number?
26 05 38 28 93 81 71 15 96 33 99 13 58 96 09

Same Data, Sorted Order
05 09 13 15 26 28 33 38 58 71 81 93 96 96 99

Selection Sort

Quickly sort these numbers into ascending order:

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

Find the minimum — O(N)

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

Flip it into the right place — O(1)

6, 29, 14, 31, 39, 64, 78, 50, 13, 63, 61, 19

Find the next smallest — O(N-1)

6, 29, 14, 31, 39, 64, 78, 50, 13, 63, 61, 19

Flip it into the right place — O(1)

6, 13, 14, 31, 39, 64, 78, 50, 29, 63, 61, 19

Find the next smallest — O(N-2)

6, 13, 14, 31, 39, 64, 78, 50, 29, 63, 61, 19

Flip it into the right place — O(1)

6, 13, 14, 31, 39, 64, 78, 50, 29, 63, 61, 19

…

Selection Sort Analysis

```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        int t = a[i]; a[i] = a[min]; a[min] = t;
    }
}
```

Analysis

- Outer loop: i = 0 to n
- Inner loop: j = i to n
- Total Running time: Outer * Inner = O(n^2)

$$T = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Selection Sort Analysis

```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        int t = a[i]; a[i] = a[min]; a[m
    }
}
```

Formally you would prove the recurrence using induction: Discrete Math class!

Requires almost no extra space: In-place algorithm

“Feels slow” since inner loop only seeks out one number (the next biggest)

Analysis

- Outer loop: i = 0 to n
- Inner loop: j = i to n
- Total Running time: Outer * Inner = $O(n^2)$

$$T = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Bubble sort

Sort these values by bubbling up the next largest value

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

[14, 29], 6, 31, 39, 64, 78, 50, 13, 63, 61, 19
[14, 29], 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

14, [29, 6], 31, 39, 64, 78, 50, 13, 63, 61, 19
14, [6, 29], 31, 39, 64, 78, 50, 13, 63, 61, 19

14, 6, [29, 31], 39, 64, 78, 50, 13, 63, 61, 19
14, 6, [29, 31], 39, 64, 78, 50, 13, 63, 61, 19

14, 6, 29, [31, 39], 64, 78, 50, 13, 63, 61, 19
14, 6, 29, [31, 39], 64, 78, 50, 13, 63, 61, 19

14, 6, 29, 31, [39, 64], 78, 50, 13, 63, 61, 19
14, 6, 29, 31, [39, 64], 78, 50, 13, 63, 61, 19

14, 6, 29, 31, 39, [64, 78], 50, 13, 63, 61, 19
14, 6, 29, 31, 39, [64, 78], 50, 13, 63, 61, 19

14, 6, 29, 31, 39, 64, [78, 50], 13, 63, 61, 19
14, 6, 29, 31, 39, 64, [50, 78], 13, 63, 61, 19

14, 6, 29, 31, 39, 64, 50, [78, 13], 63, 61, 19
14, 6, 29, 31, 39, 64, 50, [13, 78], 63, 61, 19

14, 6, 29, 31, 39, 64, 50, 13, [78, 63], 61, 19
14, 6, 29, 31, 39, 64, 50, 13, [63, 78], 61, 19

14, 6, 29, 31, 39, 64, 50, 13, 63, [78, 61], 19
14, 6, 29, 31, 39, 64, 50, 13, 63, [61, 78], 19

14, 6, 29, 31, 39, 64, 50, 13, 63, 61, [78, 19]
14, 6, 29, 31, 39, 64, 50, 13, 63, 61, [19, 78]

14, 6, 29, 31, 39, 64, 50, 13, 63, 61, 19, 78

On the first pass, sweep list
to bubble up the largest element
(also move smaller items down)

Bubble sort

Sort these values by bubbling up the next largest value

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

[14, 6], 29, 31, 39, 64, 50, 13, 63, 61, 19, 78
[6, 14], 29, 31, 39, 64, 50, 13, 63, 61, 19, 78

6, [14, 29], 31, 39, 64, 50, 13, 63, 61, 19, 78
6, [14, 29], 31, 39, 64, 50, 13, 63, 61, 19, 78

6, 14, [29, 31], 39, 64, 50, 13, 63, 61, 19, 78
6, 14, [29, 31], 39, 64, 50, 13, 63, 61, 19, 78

6, 14, 29, [31, 39], 64, 50, 13, 63, 61, 19, 78
6, 14, 29, [31, 39], 64, 50, 13, 63, 61, 19, 78

6, 14, 29, 31, [39, 64], 50, 13, 63, 61, 19, 78
6, 14, 29, 31, [39, 64], 50, 13, 63, 61, 19, 78

6, 14, 29, 31, 39, [64, 50], 13, 63, 61, 19, 78
6, 14, 29, 31, 39, [50, 64], 13, 63, 61, 19, 78

6, 14, 29, 31, 39, 50, [64, 13], 63, 61, 19, 78
6, 14, 29, 31, 39, 50, [13, 64], 63, 61, 19, 78

6, 14, 29, 31, 39, 50, 13, [64, 63], 61, 19, 78
6, 14, 29, 31, 39, 50, 13, [63, 64], 61, 19, 78

6, 14, 29, 31, 39, 50, 13, 63, [64, 61], 19, 78
6, 14, 29, 31, 39, 50, 13, 63, [61, 64], 19, 78

6, 14, 29, 31, 39, 50, 13, 63, 61, [64, 19], 78
6, 14, 29, 31, 39, 50, 13, 63, 61, [19, 64], 78

6, 14, 29, 31, 39, 50, 13, 63, 61, 19, 64, 78

On the second pass, sweep list
to bubble up the second largest element
(also move smaller items down)

Bubble sort

Sort these values by bubbling up the next largest value

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

[6, 14], 29, 31, 39, 50, 13, 63, 61, 19, 64, 78
6, [14, 29], 31, 39, 50, 13, 63, 61, 19, 64, 78
6, 14, [29, 31], 39, 50, 13, 63, 61, 19, 64, 78
6, 14, 29, [31, 39], 50, 13, 63, 61, 19, 64, 78
6, 14, 29, 31, [39, 50], 13, 63, 61, 19, 64, 78
6, 14, 29, 31, 39, [50, 13], 63, 61, 19, 64, 78
6, 14, 29, 31, 39, [13, 50], 63, 61, 19, 64, 78
6, 14, 29, 31, 39, 13, [50, 63], 61, 19, 64, 78
6, 14, 29, 31, 39, 13, 50, [63, 61], 19, 64, 78
6, 14, 29, 31, 39, 13, 50, [61, 63], 19, 64, 78
6, 14, 29, 31, 39, 13, 50, 61, [63, 19], 64, 78
6, 14, 29, 31, 39, 13, 50, 61, [19, 63], 64, 78

6, 14, 29, 31, 39, 13, 50, 61, 19, 63, 64, 78

On the third pass, sweep list
to bubble up the third largest element

How many passes will we need to do?

O(n)

How much work does each pass take?

O(n)

What is the total amount of work?

n passes, each requiring O(n) => O(n^2)

Note, you might get lucky and finish much sooner than this

"Feels faster": multiple swaps on the inner loop, but…

"Feels slow" because inner loop sweeps entire list to move one number into sorted position

Insertion Sort

Quickly sort these numbers into ascending order:

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19

14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19
14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 50, 64, 78, 13, 63, 61, 19
6, 13, 14, 29, 31, 39, 50, 64, 78, 63, 61, 19
6, 13, 14, 29, 31, 39, 50, 63, 64, 78, 61, 19
6, 13, 14, 29, 31, 39, 50, 61, 63, 64, 78, 19
6, 13, 14, 19, 29, 31, 39, 50, 61, 63, 64, 78

To be sorted

Sorted elements

Base Case: Declare the first element as a correctly sorted array

Repeat: Iteratively add the next unsorted element to the partially sorted array at the correct position

Slide the unsorted element into the correct position:

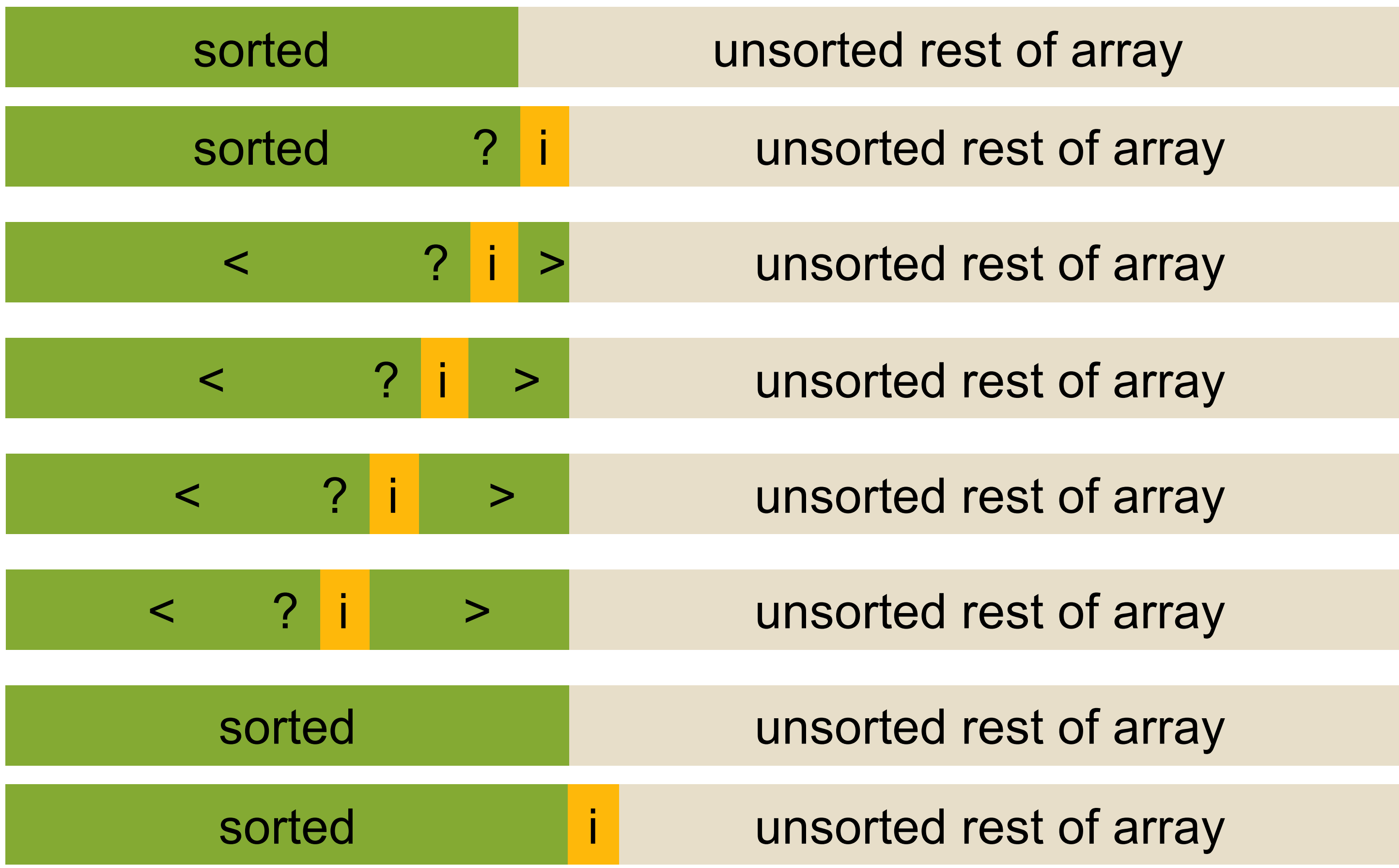
14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19
14, 6, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19
6, 14, 29, 31, 39, 64, 78, 50, 13, 63, 61, 19

"Feels fast" because you always have a partially sorted list, but some insertions will be expensive

Insertion Sort

Quickly sort these numbers into ascending order:

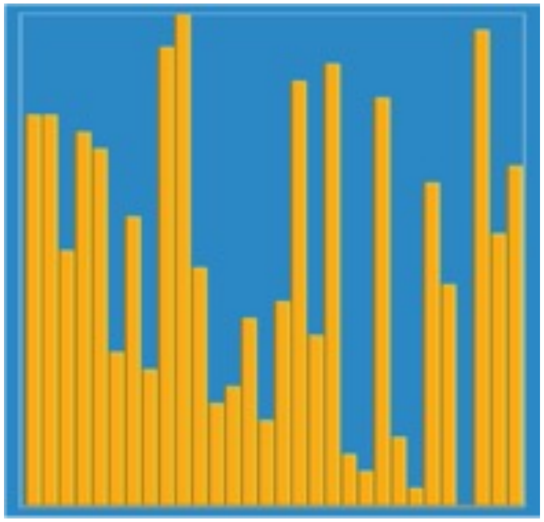
14, 29, 6, 31, 39, 64, 78, 50, 13, 63, 61, 19



Outer loop: n elements to move into correct position
Inner loop: $O(n)$ work to move element into correct position

Total Work:
$O(n^2)$

Quadratic Sorting Algorithms



Selection Sort
Move next smallest into position

Bubble Sort
Swap up bigger values over smaller

Insertion Sort
Slide next value into correct position

Asymptotically all three have the same performance
… but can perform significantly different on some datasets

Sorting Race!

Problem 2: All Sorts of Sorts (50%)

Your second task for this assignment is to explore some of the basic sorting algorithms and their analysis. All of these algorithms are quadratic in terms of their asymptotic performance, but they nevertheless differ in their actual performance. We'll focus on the following three algorithms:

- Bubble Sort (with the "stop early if no swaps" extension)
- Selection Sort
- Insertion Sort

The archive for this assignment contains a basic framework for evaluating sorting algorithms. You'll need a working `StatableArray` class from Problem 1, and you'll need to understand the following interface as well (again compressed, be sure to to use and read the full interface available on Piazza!):

```
public interface SortingAlgorithm<T extends Comparable<T>> {
    void sort(Array<T> array);
    String name();
}
```

Let's look at the simple stuff first: An object is considered an algorithm suitable for sorting in this framework if (a) we can ask it to sort a given `Array` and (b) we can ask it for its name (e.g. "Insertion Sort").

The more complicated stuff is at the top: The use of `extends` inside the angle brackets means that any type T we want to sort must implement the interface `Comparable` as well. It obviously can't just be any old type, it must be a type for which the expression "a is less than b" actually makes sense. Using `Comparable` in this form is Java's way of saying that we can order the objects; you should definitely read up on the details here!

As an example for all this, we have provided an implementation of `SelectionSort` on Piazza already. (Actually, there are also two other algorithms, `NullSort` and `GnomeSort`, just so you start out with a few to run right away.)

You need to write classes implementing `BubbleSort` and `InsertionSort` for this problem. Just like our example algorithms, your classes have to `implement` the `SortingAlgorithm` interface.

All of this should be fairly straightforward once you get used to the framework. Speaking of the framework, the way you actually "run" the various algorithms is by using the `PolySort.java` program we've provided as well. You should be able to compile and run it without yet having written any sorting code yourself. Here's how:

```
$ java PolySort 4000 <random.data
Algorithm           Sorted?  Size       Reads        Writes       Seconds

Null Sort           false    4,000      0            0            0.000007
Gnome Sort          true     4,000      32,195,307   8,045,828    0.243852
Selection Sort      true     4,000      24,009,991   7,992        0.252085
```

This will read the first 4000 strings from the file `random.data` and sort them using all available algorithms. As you can see, the program checks if the algorithm actually worked (Sorted?) and reports how many operations of the underlying `StatableArray` were used in order to perform the sort (Reads, Writes). Finally, the program also prints out long it took to sort the array (Seconds) but that number will vary widely across machines so you can really only use it for relative comparisons on the machine actually running the experiment.

However, the main point of all this is *not* the coding work. Instead, the main point is to evaluate and compare the sorting algorithms on different sets of data. We've provided three sets of useful test data on Piazza and you can use the command line argument to vary how much of it is used (thereby changing the size of the problem). You should try to quantify how the various algorithms differ and explain why they differ as well (i.e. what about a given algorithm makes it better or worse than another one for a given data set). In your `README` file you should describe the series of experiments you ran, what data you collected, and what your conclusions about the performance of these algorithms are. Some ideas for what to address:

- Does the actual running time correspond to the asymptotic complexity as you would expect?
- What explains the practical differences between these algorithms?
- Does it matter what kind of data (random, already sorted in ascending order, sorted in descending order) you are sorting?

Just to be clear: Yes, we'll need the code, and it should be up to the usual standards. But the "report" you put in your `README` is just as important as the code!

Sorting Race!

Problem 2: All Sorts of Sorts (50%)

Your second task for this assignment is to explore some of the basic sorting algorithms and their analysis. All of these algorithms are quadratic in terms of their asymptotic performance, but they nevertheless differ in their actual performance. We'll focus on the following three algorithms:

- Bubble Sort (with the "stop early if no swaps" extension)
- Selection Sort
- Insertion Sort

The archive for this assignment contains a basic framework for evaluating sorting algorithms. You'll need a working `StatableArray` class from Problem 1, and you'll need to understand the following interface as well (again compressed, be sure to to use and read the full interface available on Piazza!):

```
public interface SortingAlgorithm<T extends Comparable<T>> {
    void sort(Array<T> array);
    String name();
}
```

Do you think quadratic sort is the best you can do?

;-)

Let's look at the simple stuff first: An object is considered an algorithm suitable for sorting in this framework if (a) we can ask it to [illegible] `Array` and (b) we can ask it for its name (e.g. "Insertion Sort").

The more complicated stuff is at the top: The use of `extends` inside the angle brackets means that any type T we want to sort m[illegible] the interface `Comparable` as well. It obviously can't just be any old type, it must be a type for which the expression "a is less than b" actually makes sense. Using `Comparable` in this form is Java's way of saying that we can order the objects; you should de[illegible]p on the details here!

As an example for all this, we have provided an implementation of `SelectionSort` on Piazza already. (Actually, there are also two other algorithms, `NullSort` and `GnomeSort`, just so you start out with a few to run right away.)

You need to write classes implementing `BubbleSort` and `InsertionSort` for this problem. Just like our example algorithms, your classes have to `implement` the `SortingAlgorithm` interface.

All of this should be fairly straightforward once you get used to the framework. Speaking of the framework, the way you actually "run" the various algorithms is by using the `PolySort.java` program we've provided as well. You should be able to compile and run it without yet having written any sorting code yourself. Here's how:

```
$ java PolySort 4000 <random.data
Algorithm           Sorted?  Size        Reads       Writes      Seconds

Null Sort           false    4,000       0           0           0.000007
Gnome Sort          true     4,000       32,195,307  8,045,828   0.243852
Selection Sort      true     4,000       24,009,991  7,992       0.252085
```

This will read the first 4000 strings from the file `random.data` and sort them using all available algorithms. As you can see, the program checks if the algorithm actually worked (Sorted?) and reports how many operations of the underlying `StatableArray` were used in order to perform the sort (Reads, Writes). Finally, the program also prints out long it took to sort the array (Seconds) but that number will vary widely across machines so you can really only use it for relative comparisons on the machine actually running the experiment.

However, the main point of all this is *not* the coding work. Instead, the main point is to evaluate and compare the sorting algorithms on different sets of data. We've provided three sets of useful test data on Piazza and you can use the command line argument to vary how much of it is used (thereby changing the size of the problem). You should try to quantify how the various algorithms differ and explain why they differ as well (i.e. what about a given algorithm makes it better or worse than another one for a given data set). In your `README` file you should describe the series of experiments you ran, what data you collected, and what your conclusions about the performance of these algorithms are. Some ideas for what to address:

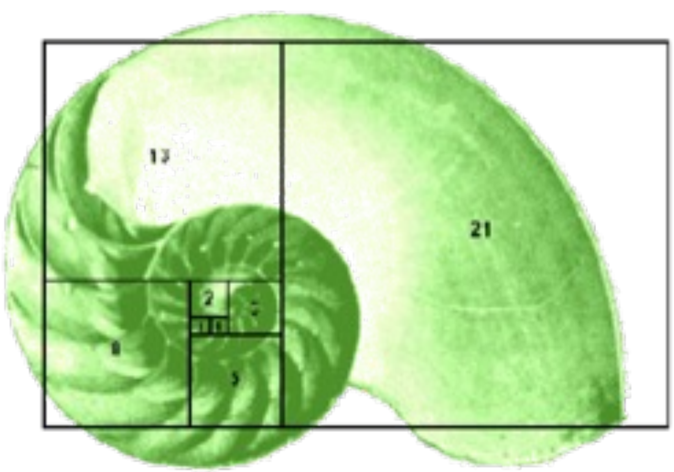
- Does the actual running time correspond to the asymptotic complexity as you would expect?
- What explains the practical differences between these algorithms?
- Does it matter what kind of data (random, already sorted in ascending order, sorted in descending order) you are sorting?

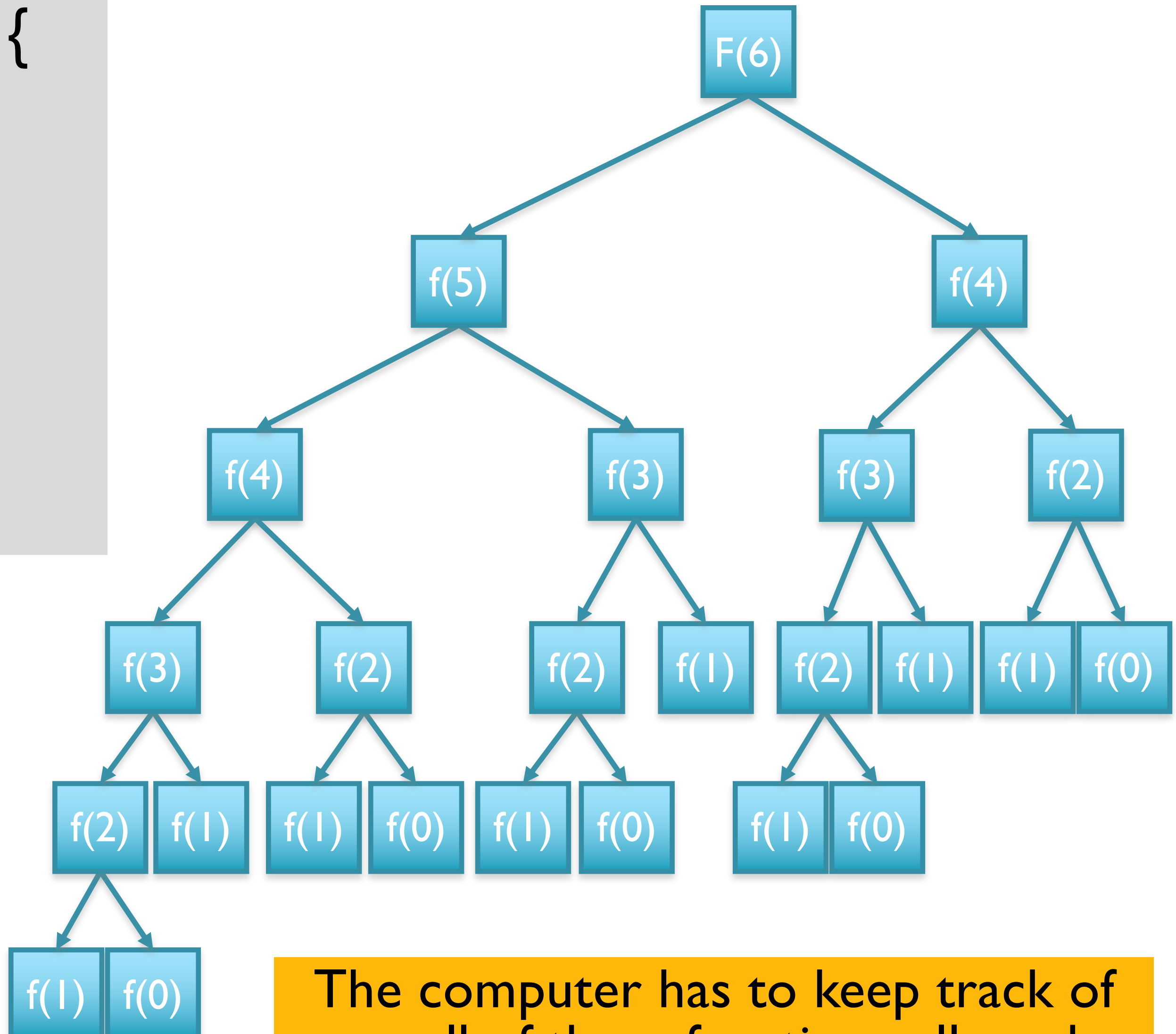
Just to be clear: Yes, we'll need the code, and it should be up to the usual standards. But the "report" you put in your `README` is just as important as the code!

Part 3: Stacks

Fibonacci Sequence

```
public static int fib(int n) {
   if (n <= 1) {
     return 1;
   }
   return fib(n-1) +
             fib(n-2);
}
```





The computer has to keep track of all of these function calls and keep everything in order!

Introducing the call stack

```
100:   int f(int a) {
101:       int y = 42;
102:       int z = 13;
           …
400:       int x = g(13 + a);
           …
497:       return x + y + z;
498:   }

500:   int g(double a) {
501:    return h(a*2);
502:   }

504:   int h(double x) {
505:        int k = (int) x;
506:        return k+1;
507:   }

509:   static public void main() {
510:       int val = f(15);
510:       System.out.println(val);
511:   }
```

h
Return val to g line 501
x=56
k=56
val = 57

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

Introducing the call stack

The ***call stack*** keeps track of the local variables and return location for the current function. This makes it possible for program execution to jump from function to function without loosing track of where the program should resume

A ***stack frame*** records the information for each function call, with local variables and the address of where to resume processing after this function is complete.

=> Take a computer architecture course for more info

Importantly the computer only needs to ***add or remove items from the very top of the stack***, making it easy for the computer to keep track of where to go next!

More generally, stacks are a very useful data structure for ***Last-In-First-Out (LIFO)*** processing

h
Return val to g line 501
x=56
k=56
val = 57

g
Return val to f line 400
a=28
val = <h(56)>

f
Return val to main line 510
a = 15
y = 42
z = 13
x = <g(28)>

main
val = <f(15)>

Stacks

Stacks are very simple but surprisingly useful data structures for storing a collection of information

• Any number of items can be stored, but you can only manipulate the top of the stack:

- ***Push***: adds a new element to the top
- ***Pop***: takes off the top element
- ***Top***: Lets you peek at top element's value without removing it from stack

Many Applications

• In hardware call stack

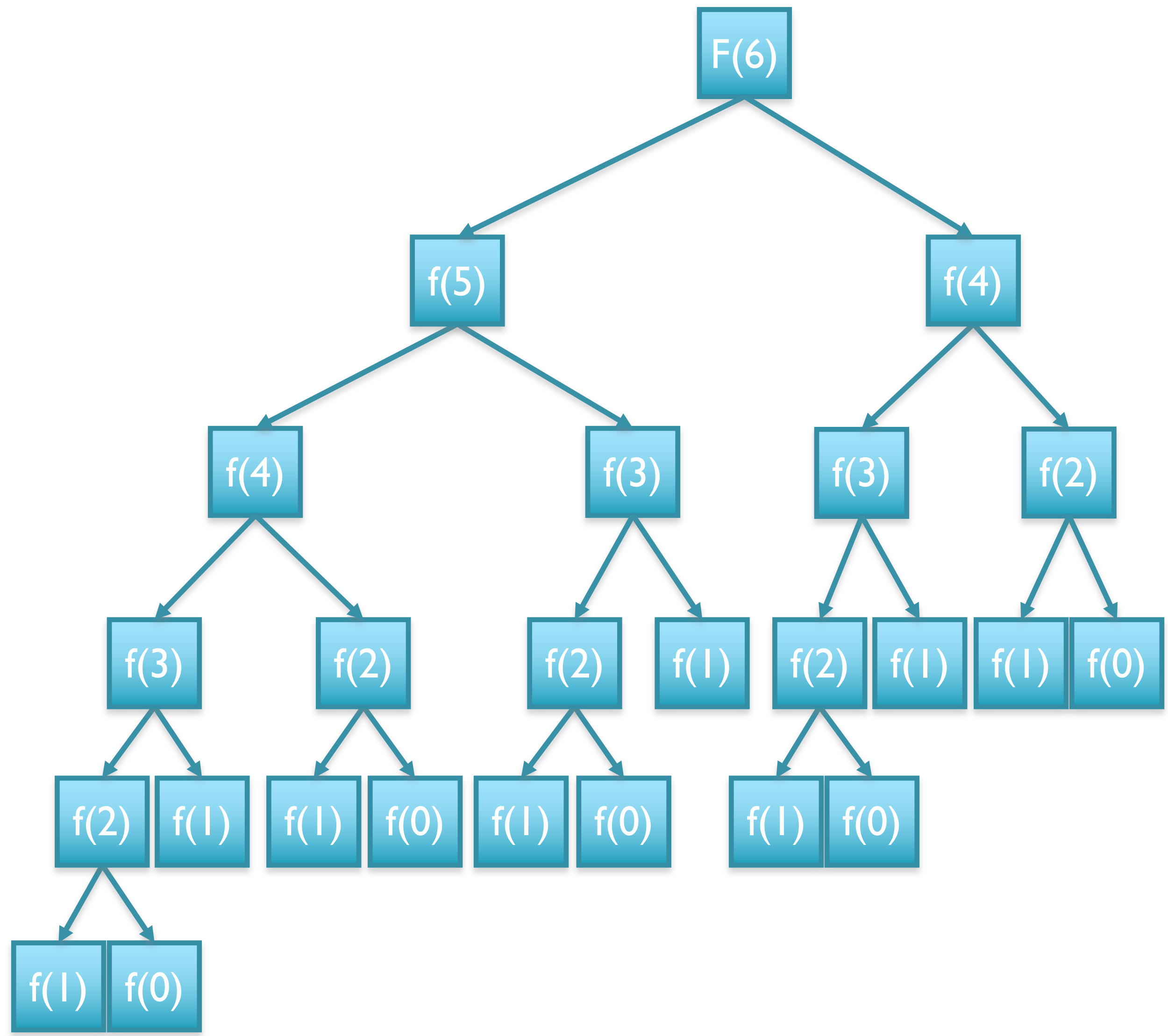
• Memory management systems

• Parsing arithmetic instructions:

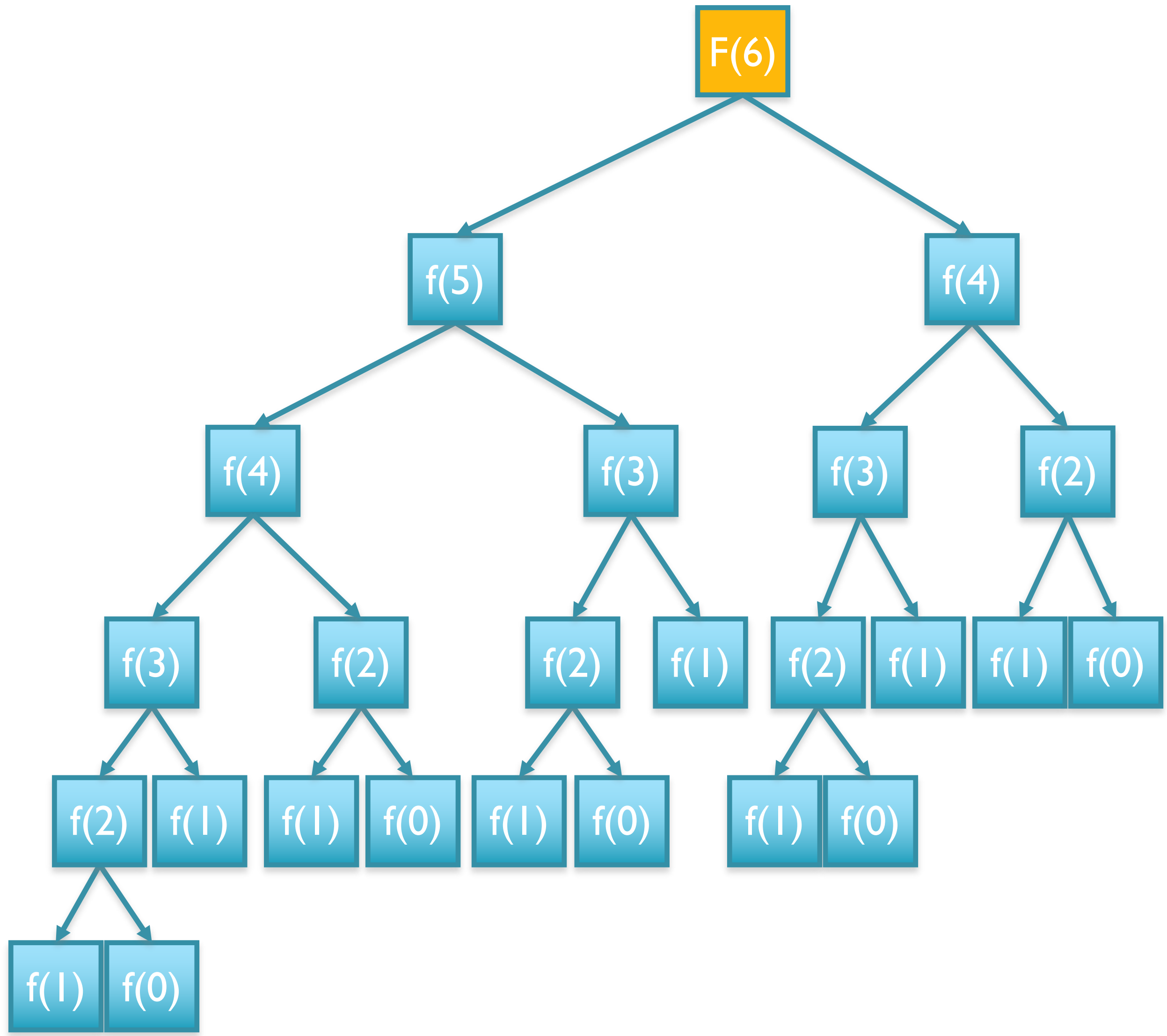
((x+3) / (x+9)) * (42 * sin(x))

• Back-tracing, such as searching within a maze

Fibonacci Sequence

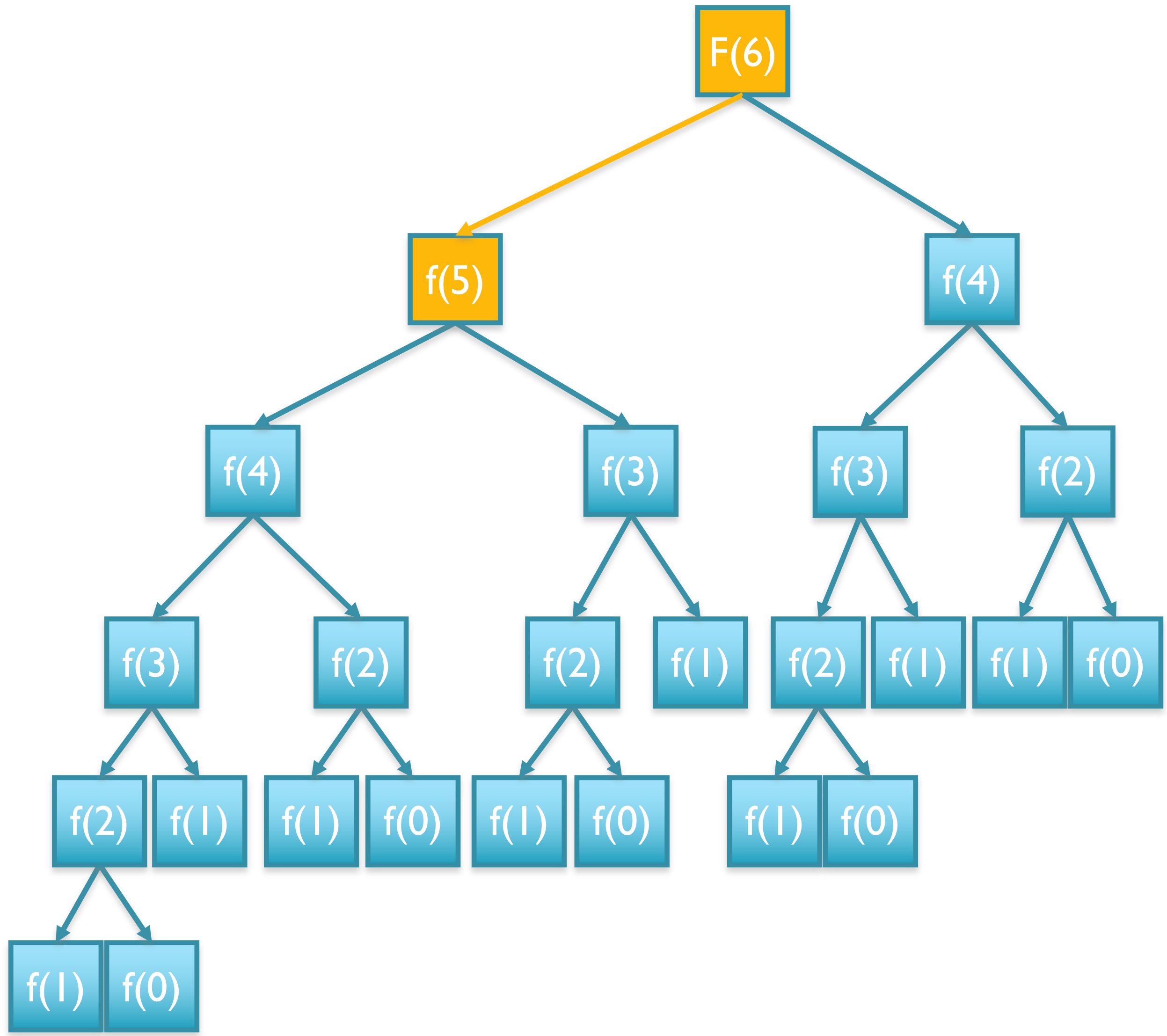


Fibonacci Sequence



F(6)
a = F(5)

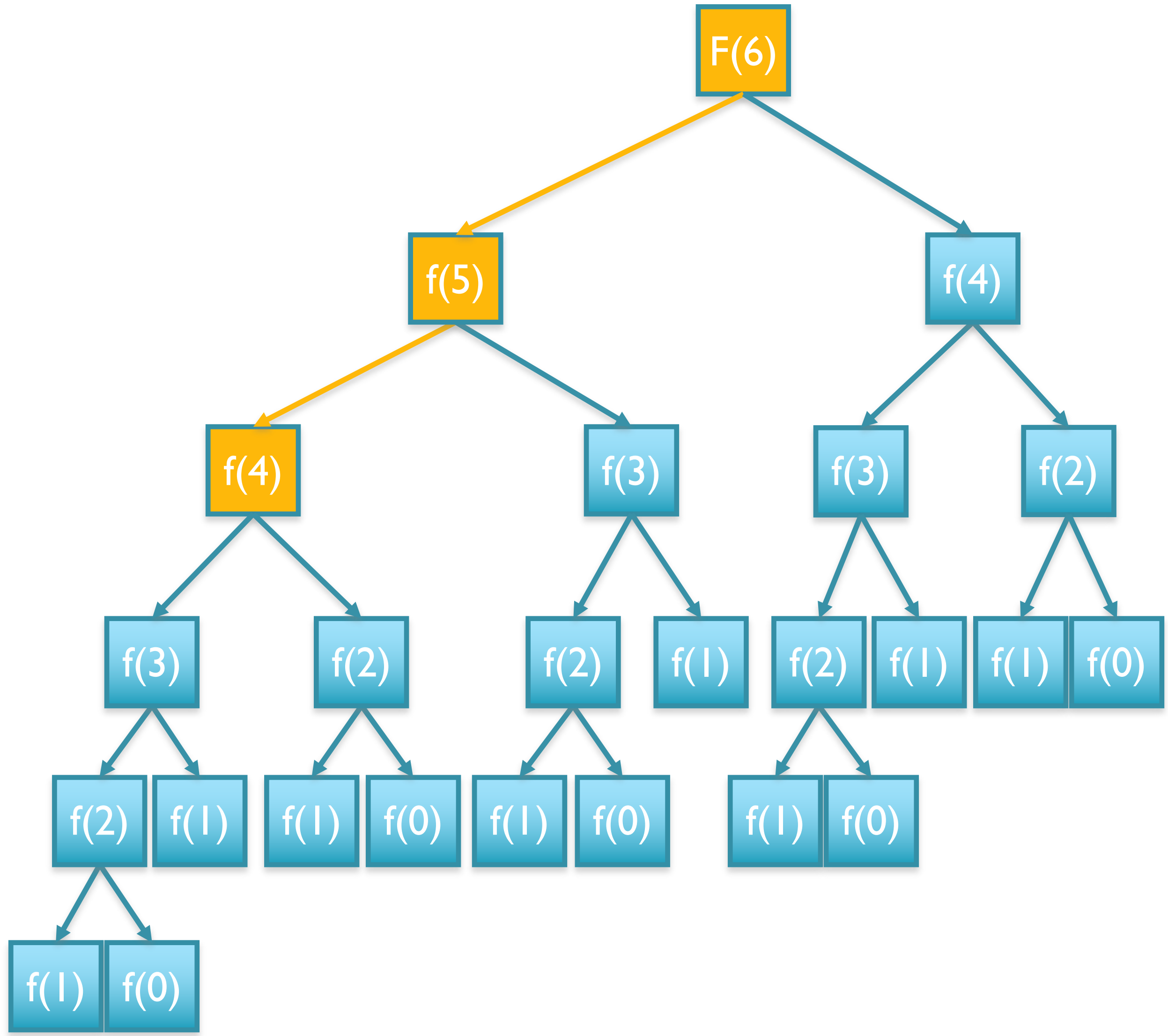
Fibonacci Sequence



F(5)
a = F(4)

F(6)
a = F(5)

Fibonacci Sequence



F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)

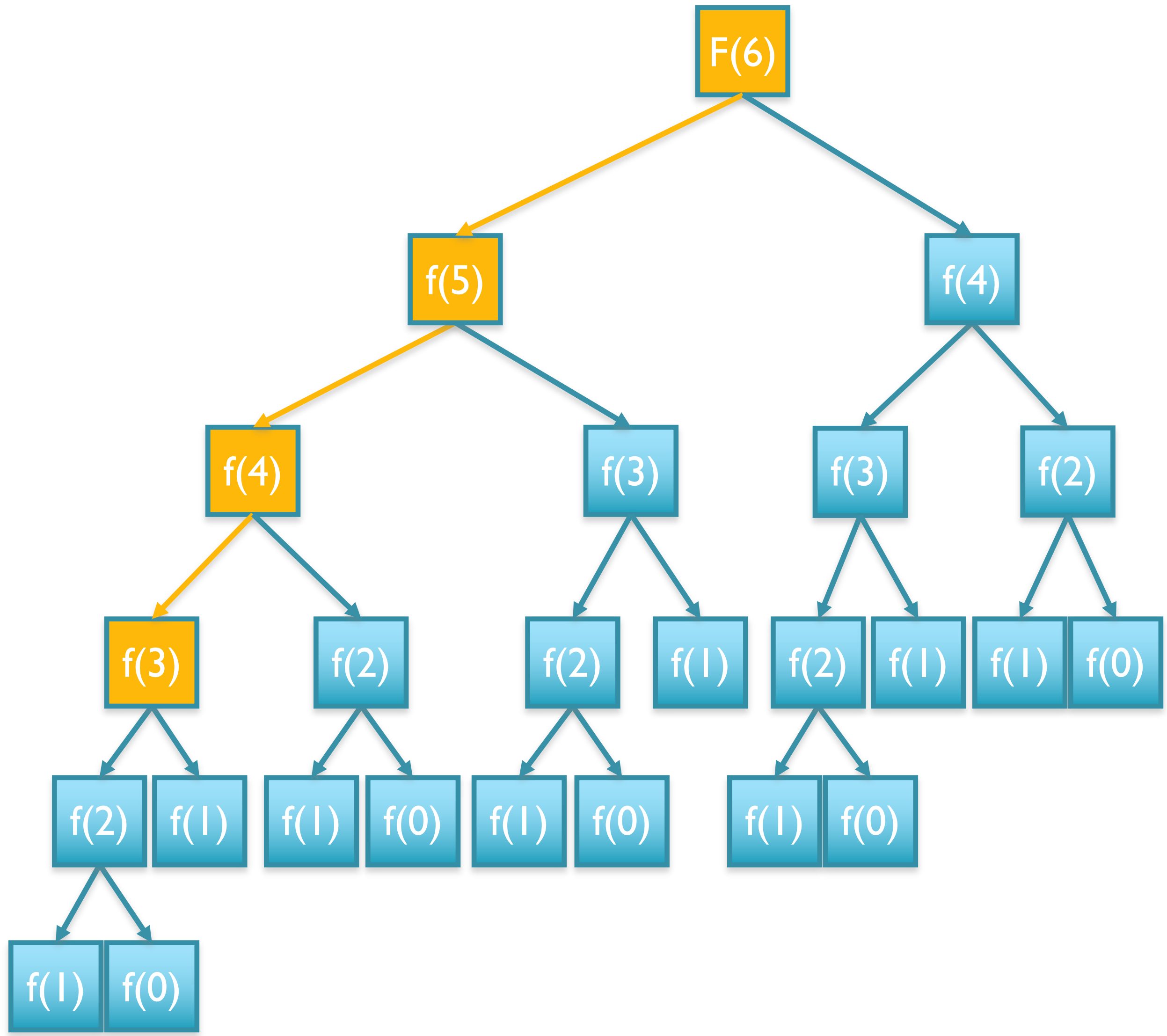
Fibonacci Sequence

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

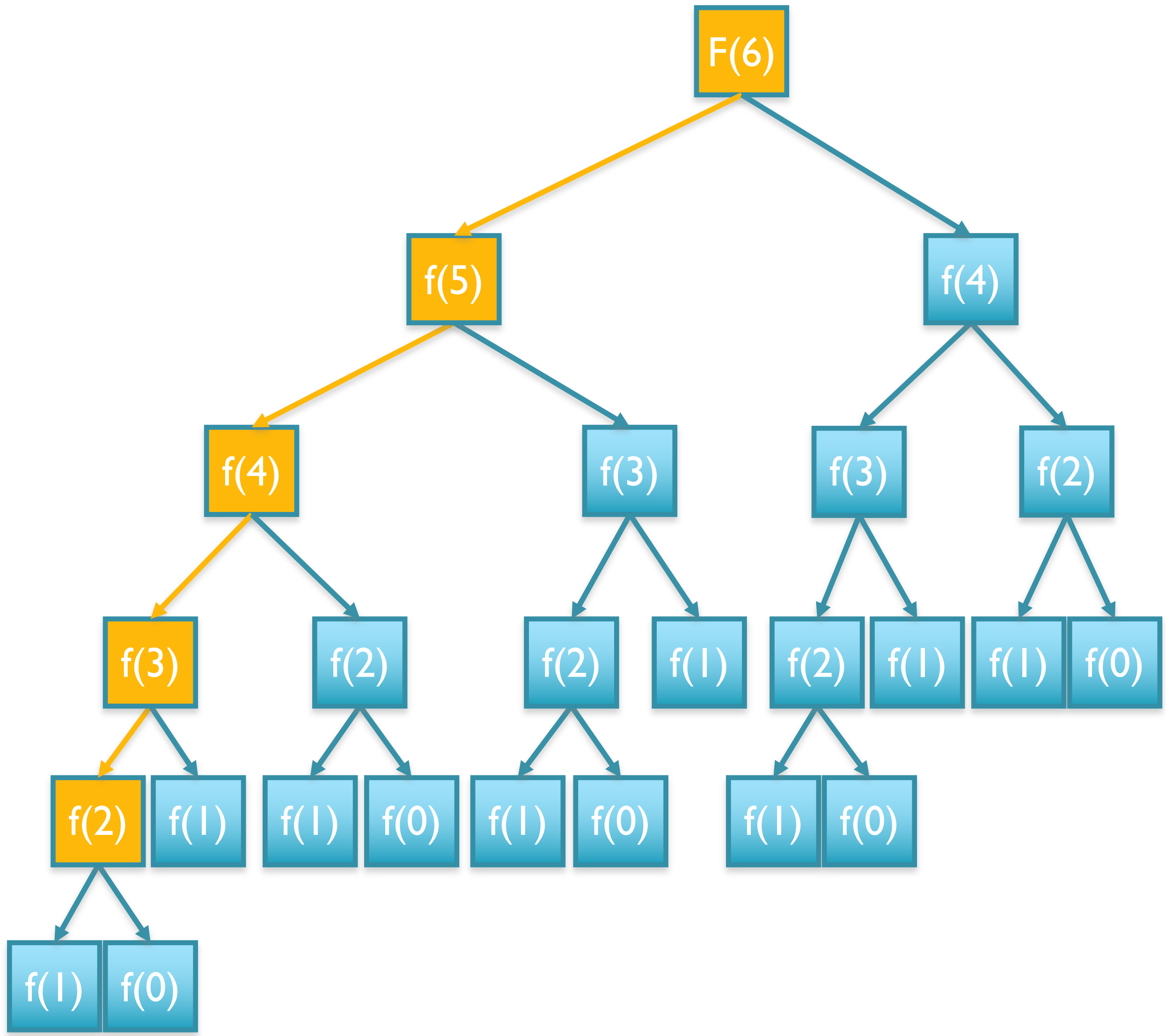
F(2)
a = F(1)

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

F(1)
return 1

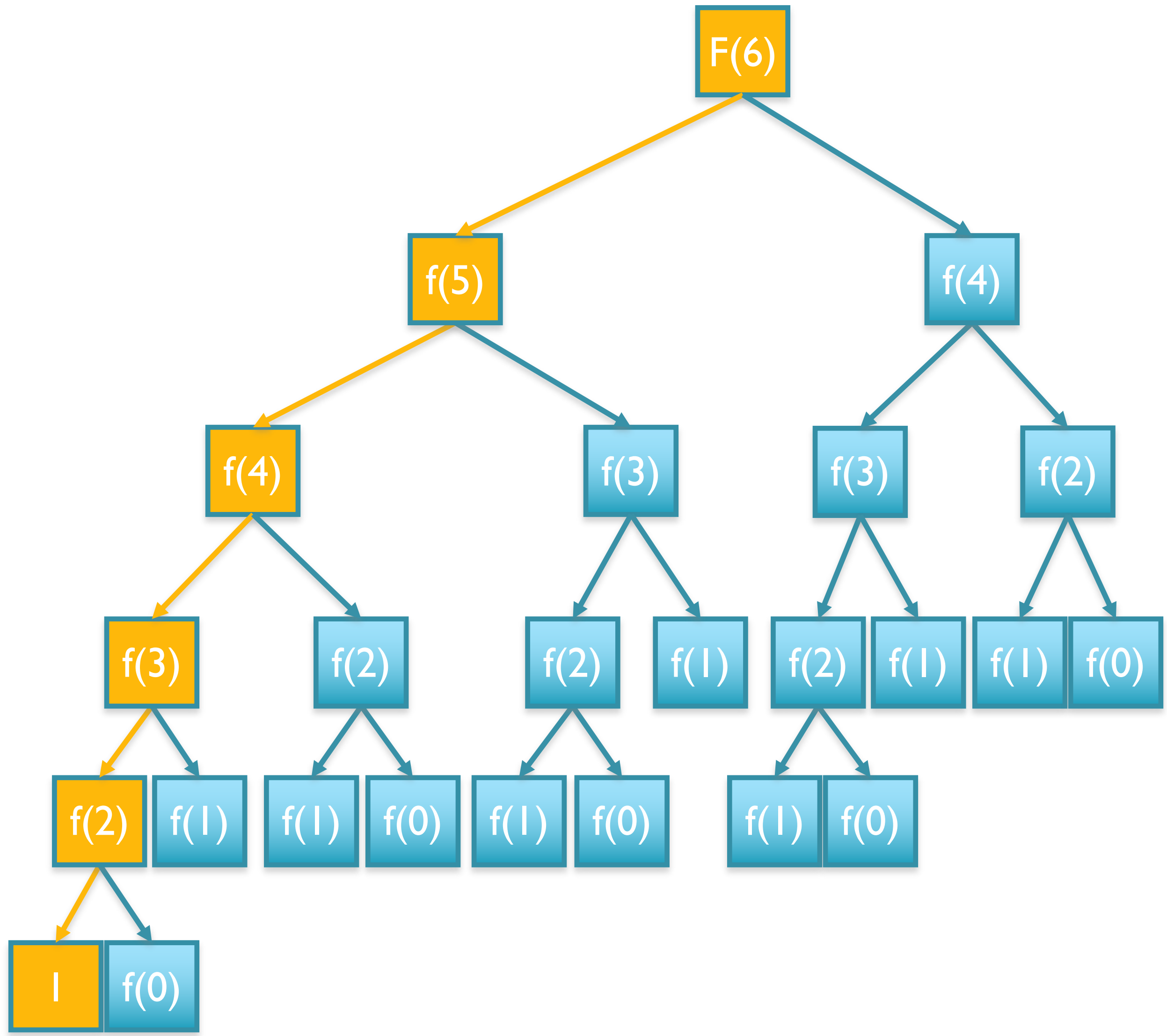
F(2)
a = F(1)

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

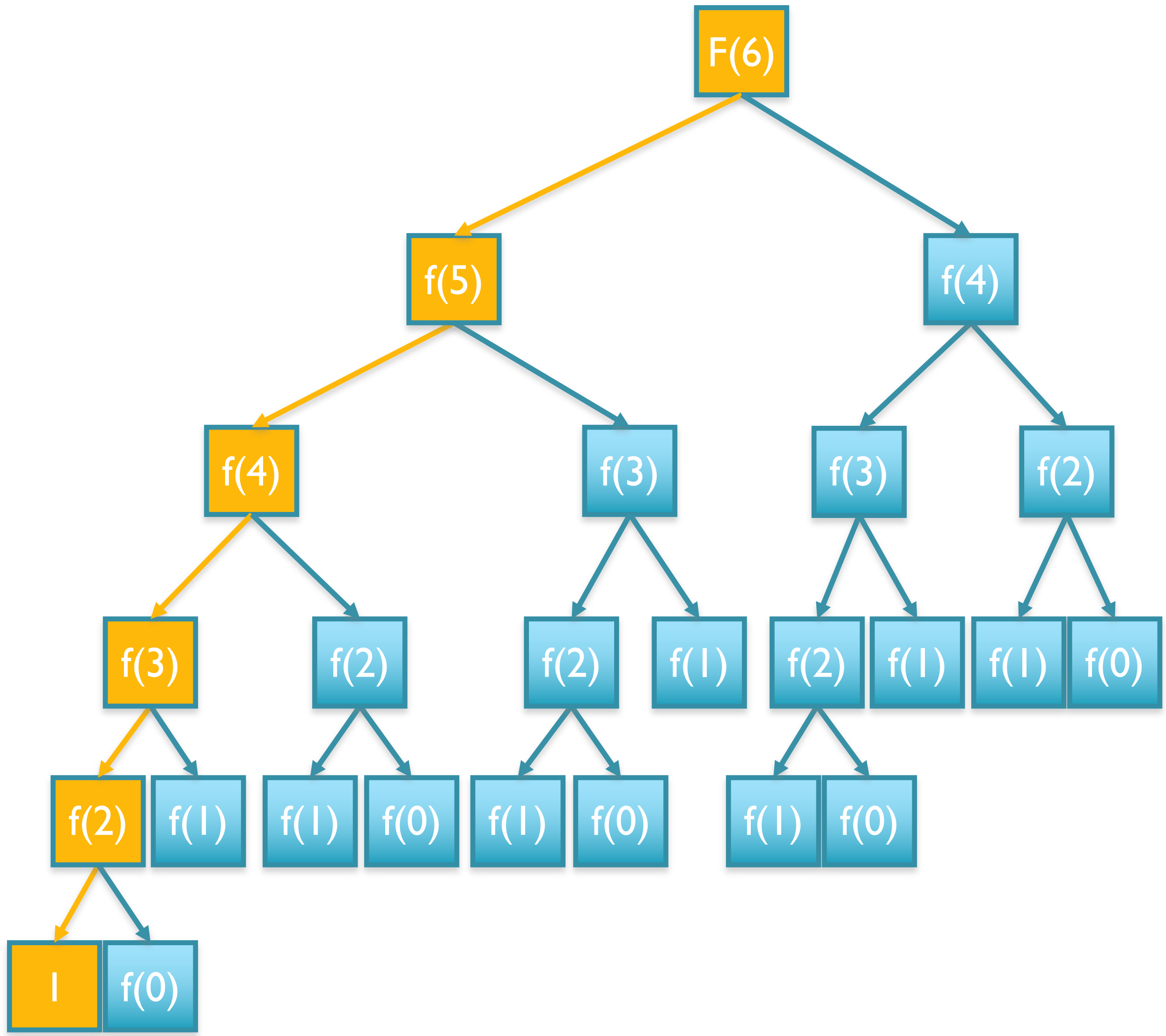
F(2)
b = F(0)

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

F(0)
return 1

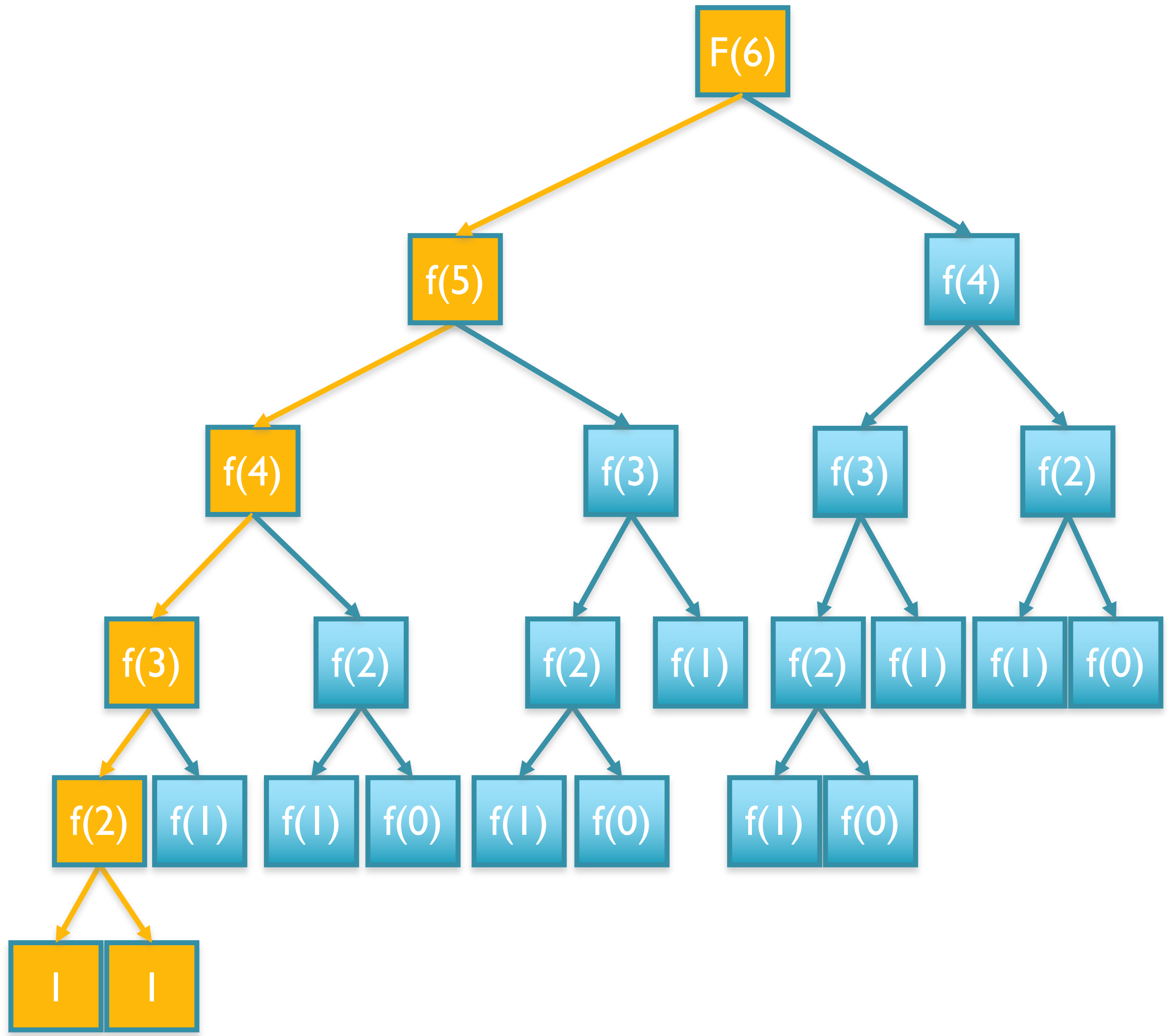
F(2)
b = F(0)

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

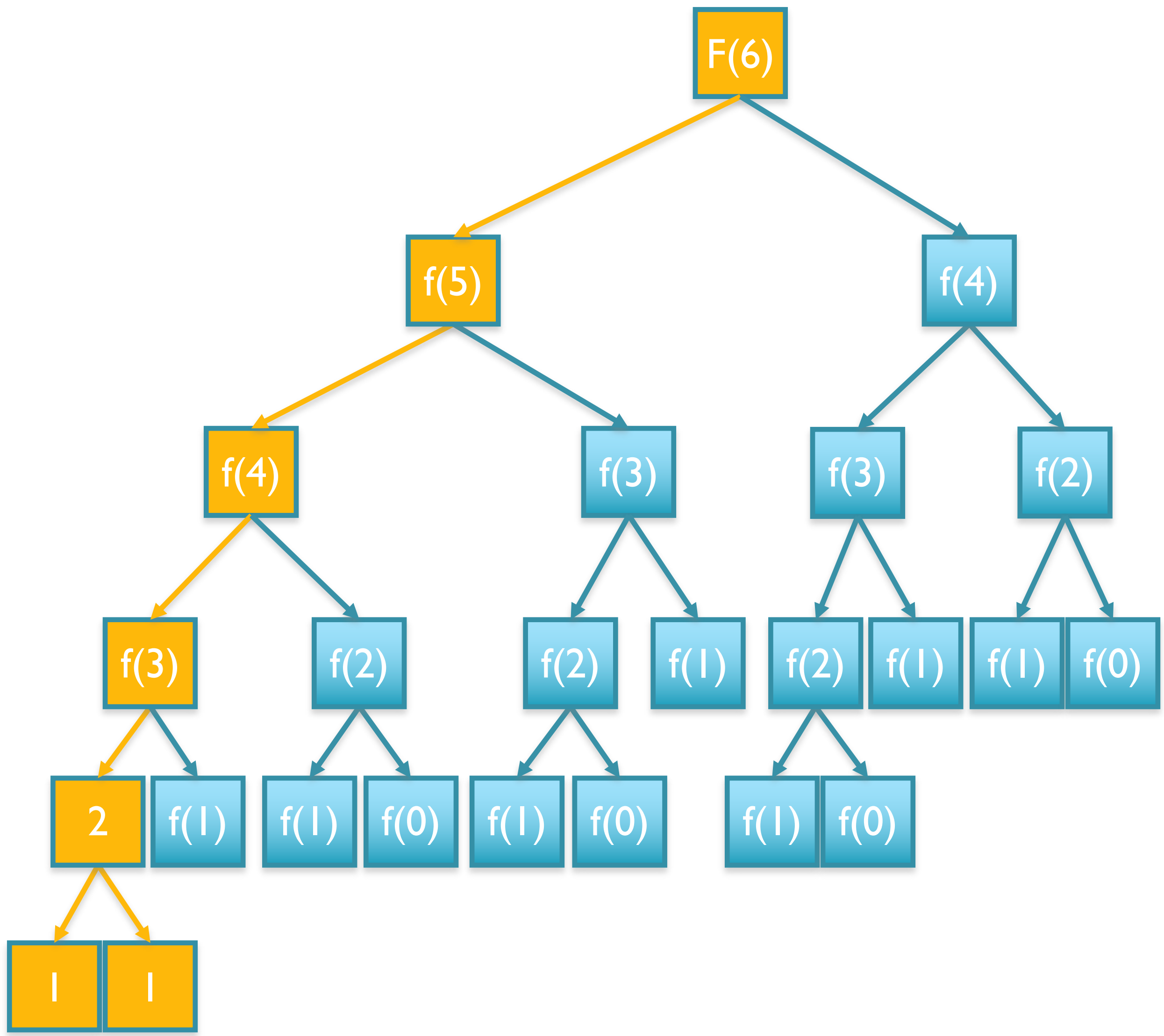
F(2)
return 2

F(3)
a = F(2)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



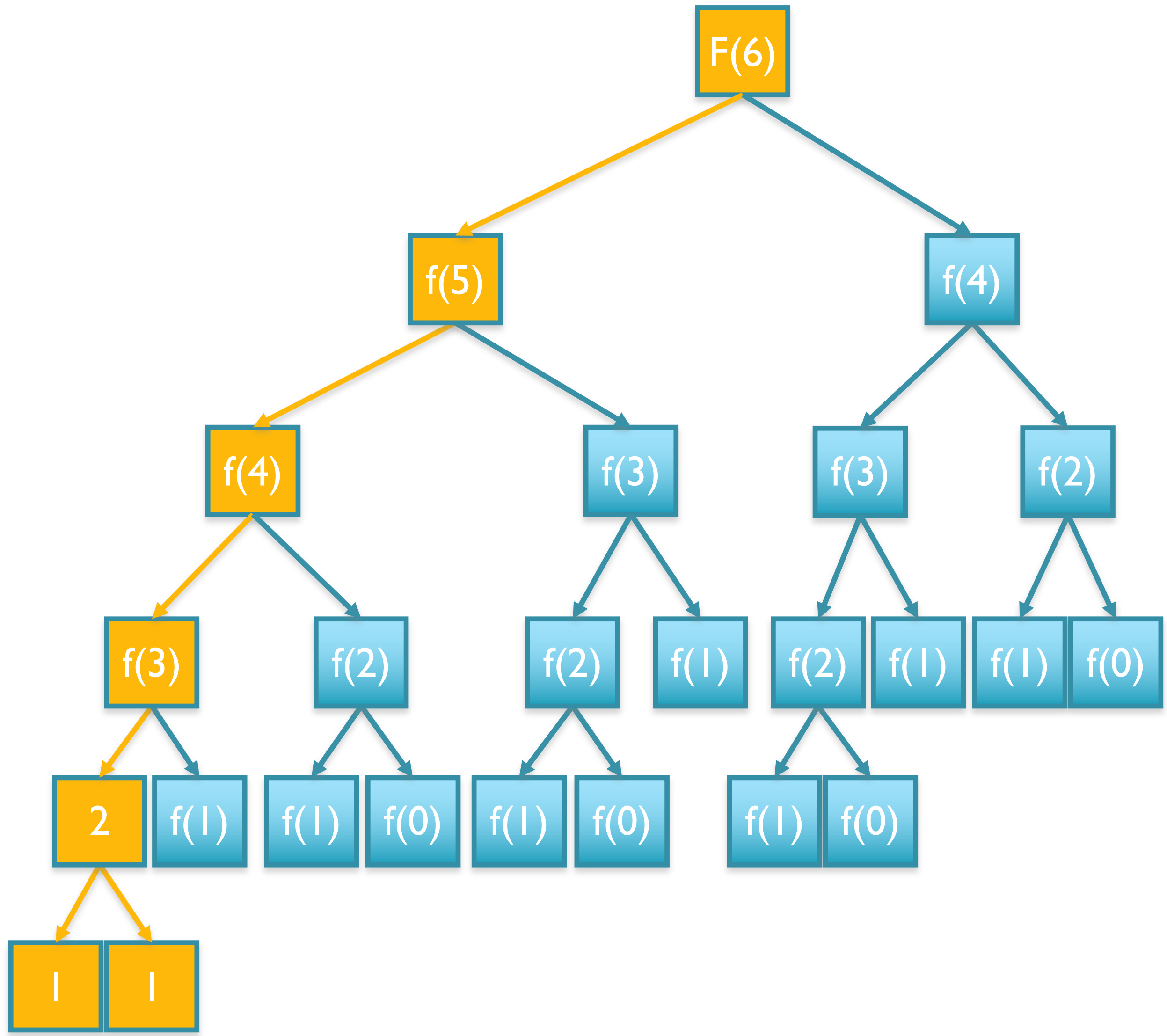
Fibonacci Sequence

F(3)
a = F(2)

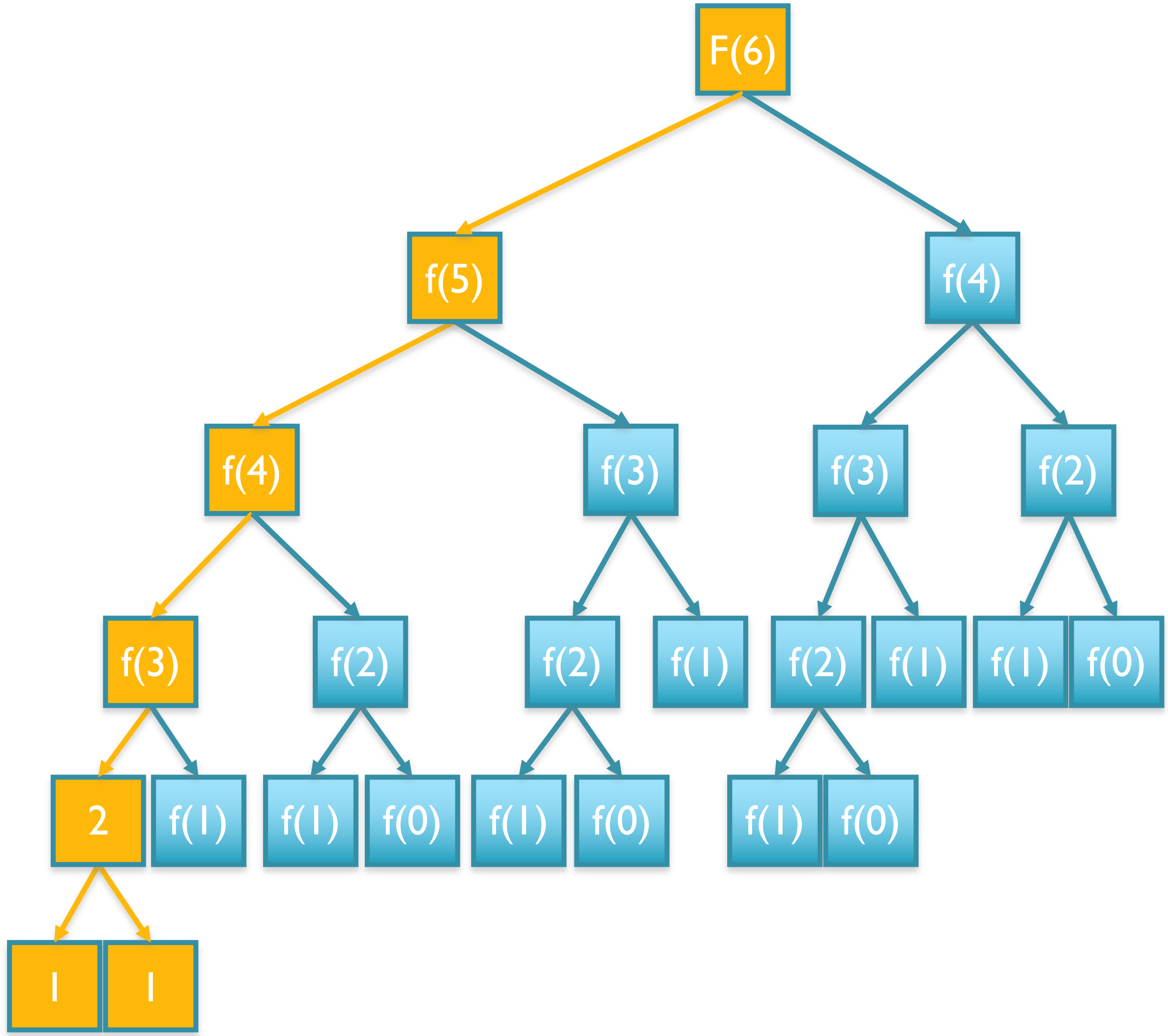
F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence



F(3)
b = F(1)

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)

Fibonacci Sequence

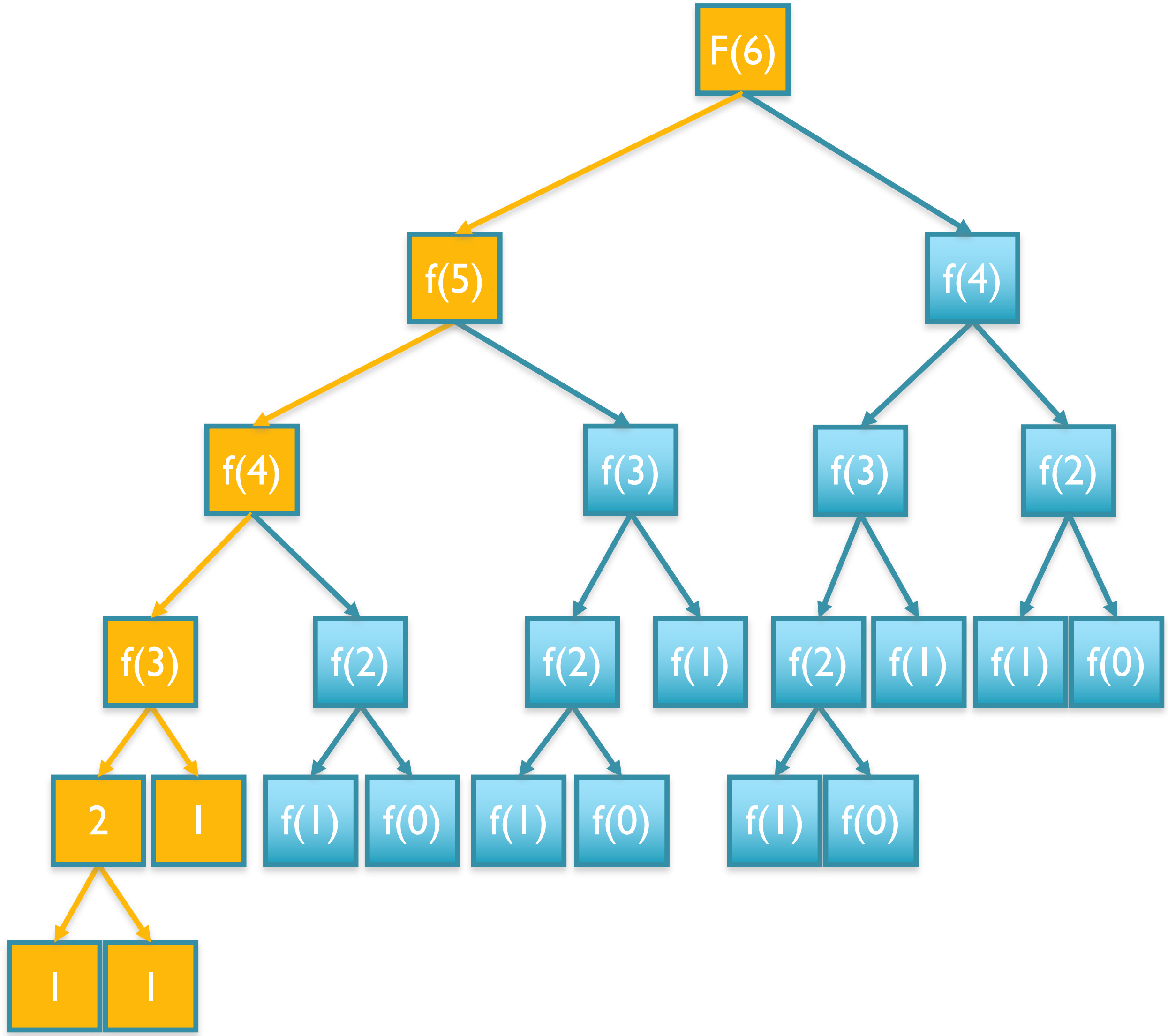
F(1)
return 1

F(3)
b = F(1)

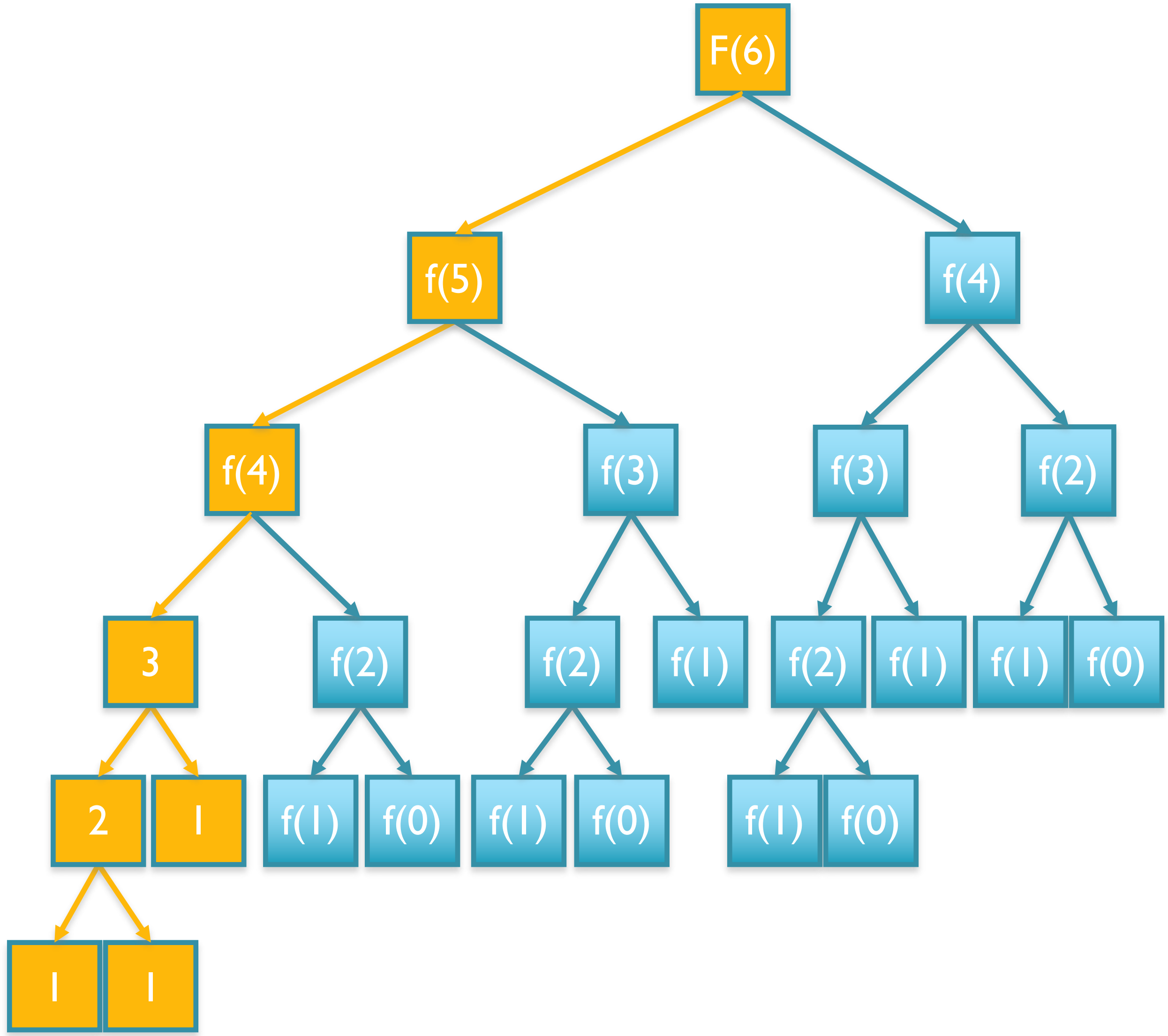
F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence



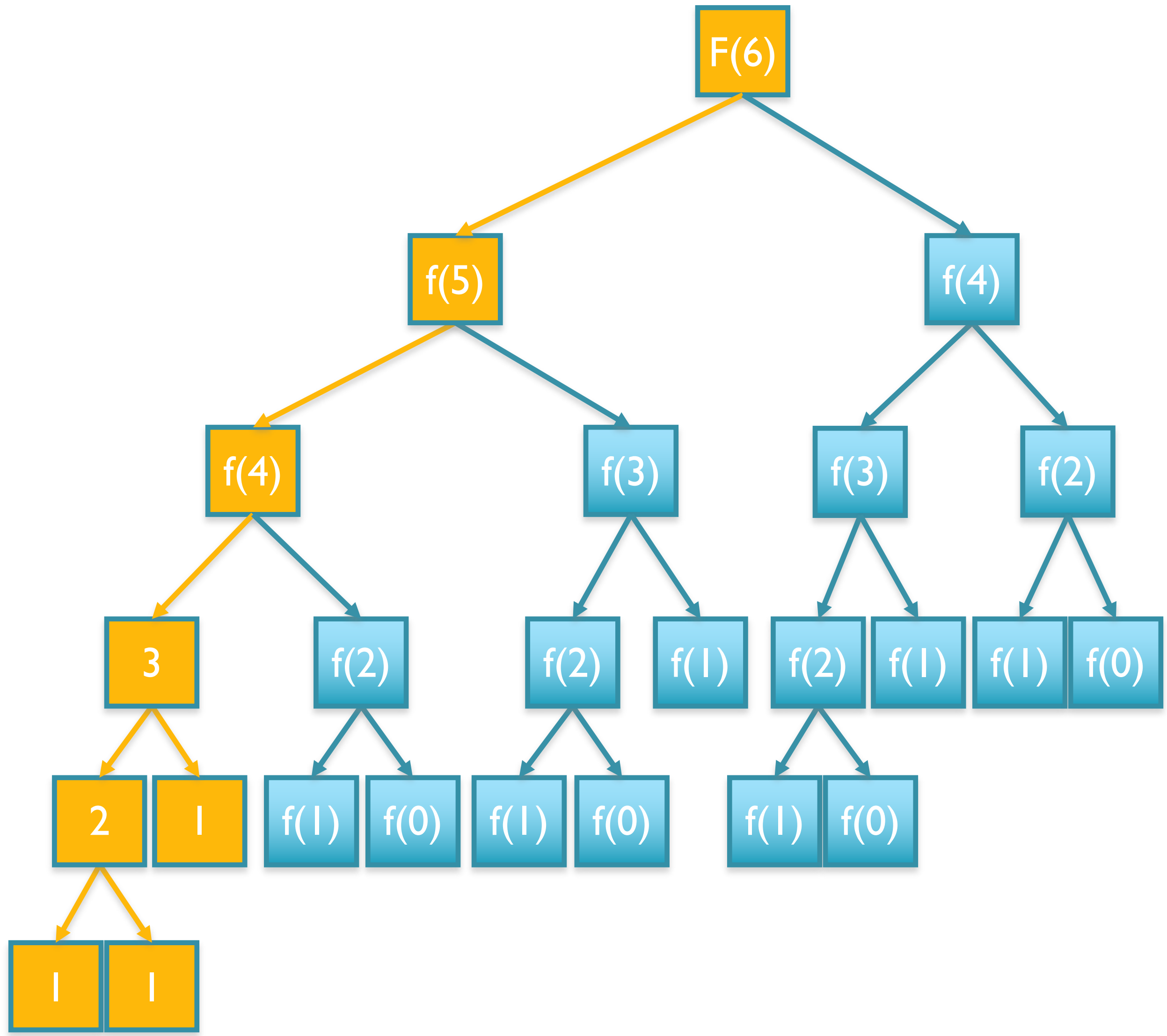
F(3)
return 3

F(4)
a = F(3)

F(5)
a = F(4)

F(6)
a = F(5)

Fibonacci Sequence



F(4)
b = F(2)

F(5)
a = F(4)

F(6)
a = F(5)

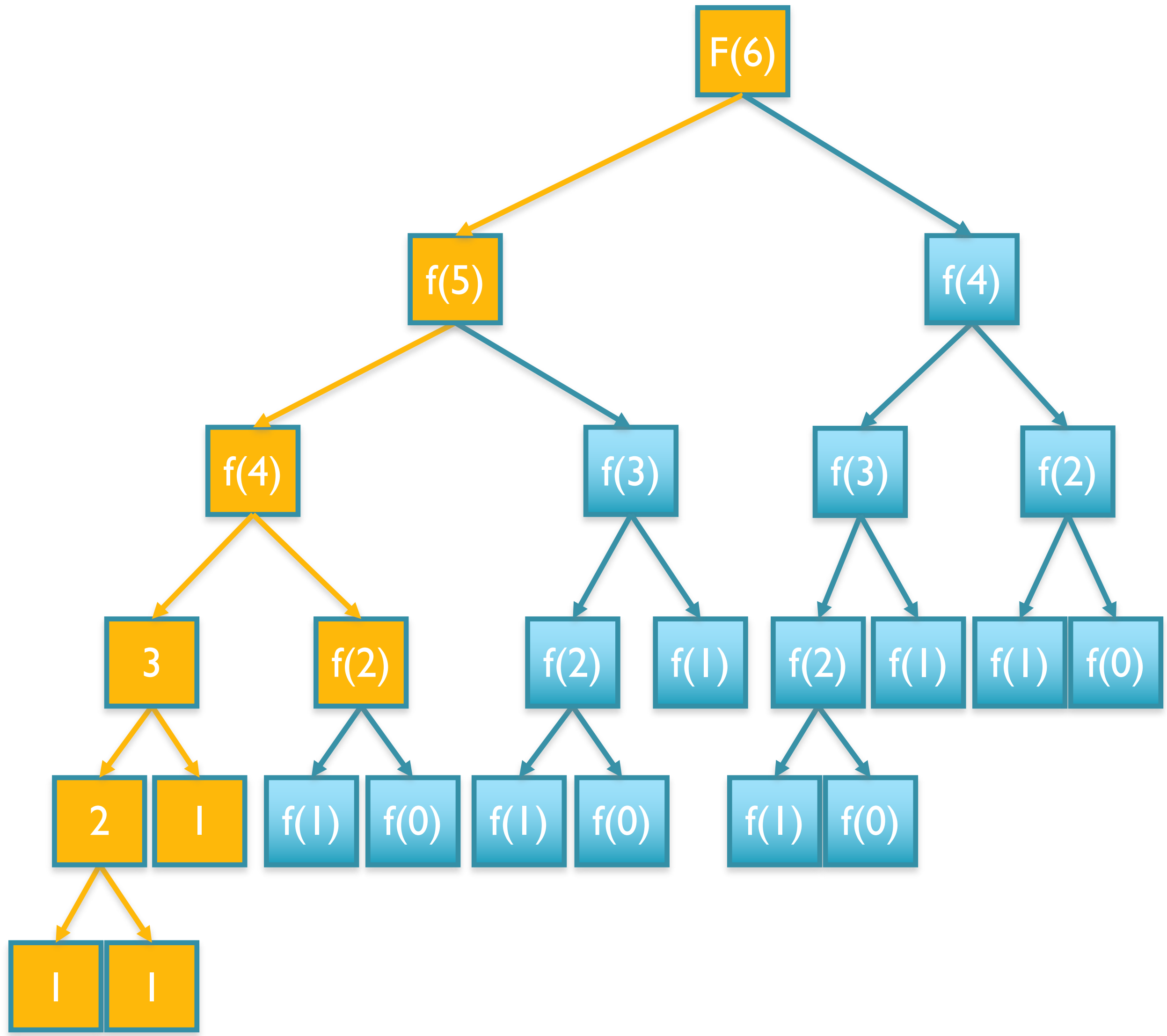
Fibonacci Sequence

F(2)
A = F(1)

F(4)
b = F(2)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence

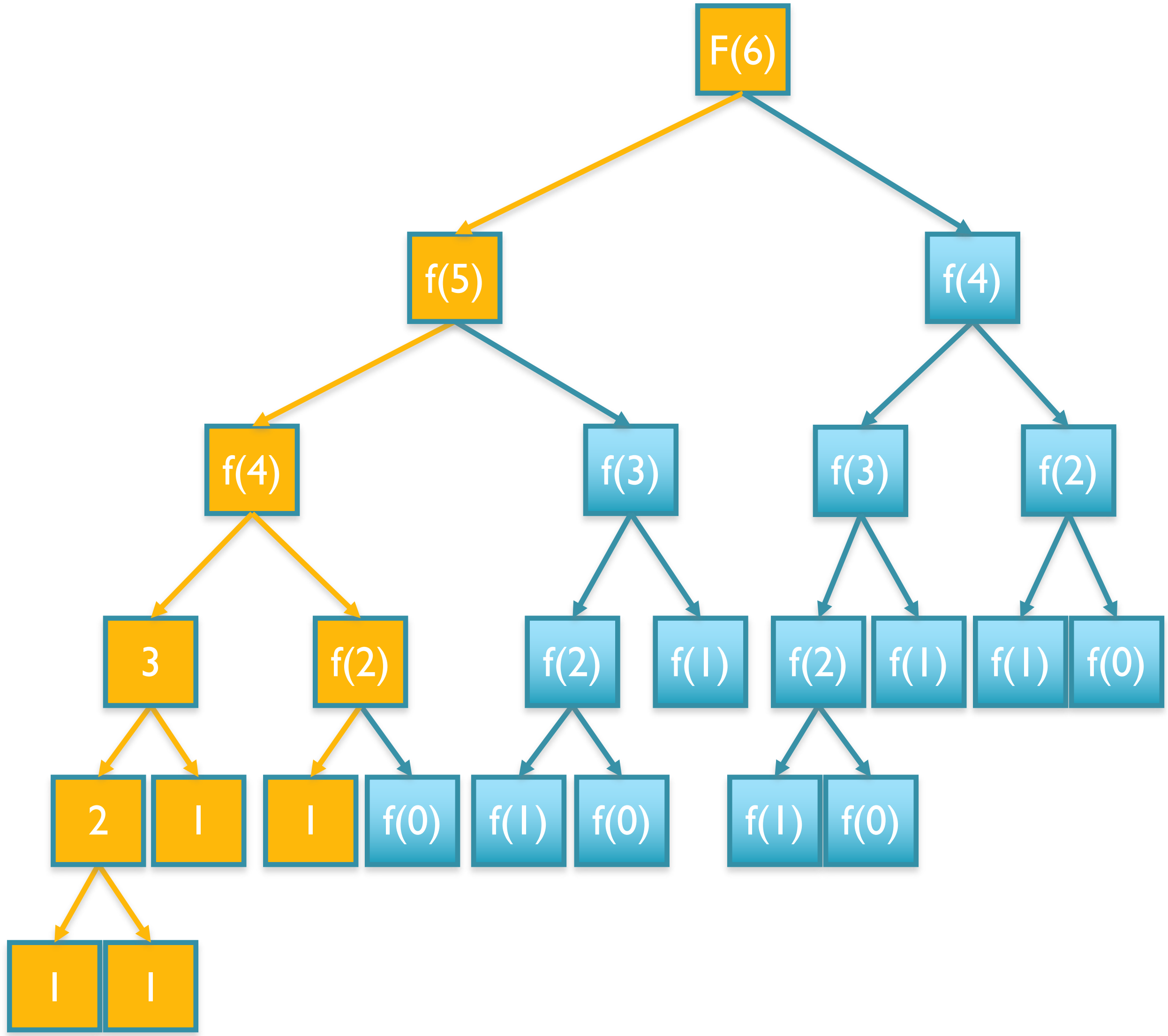
F(1)
return 1

F(2)
A = F(1)

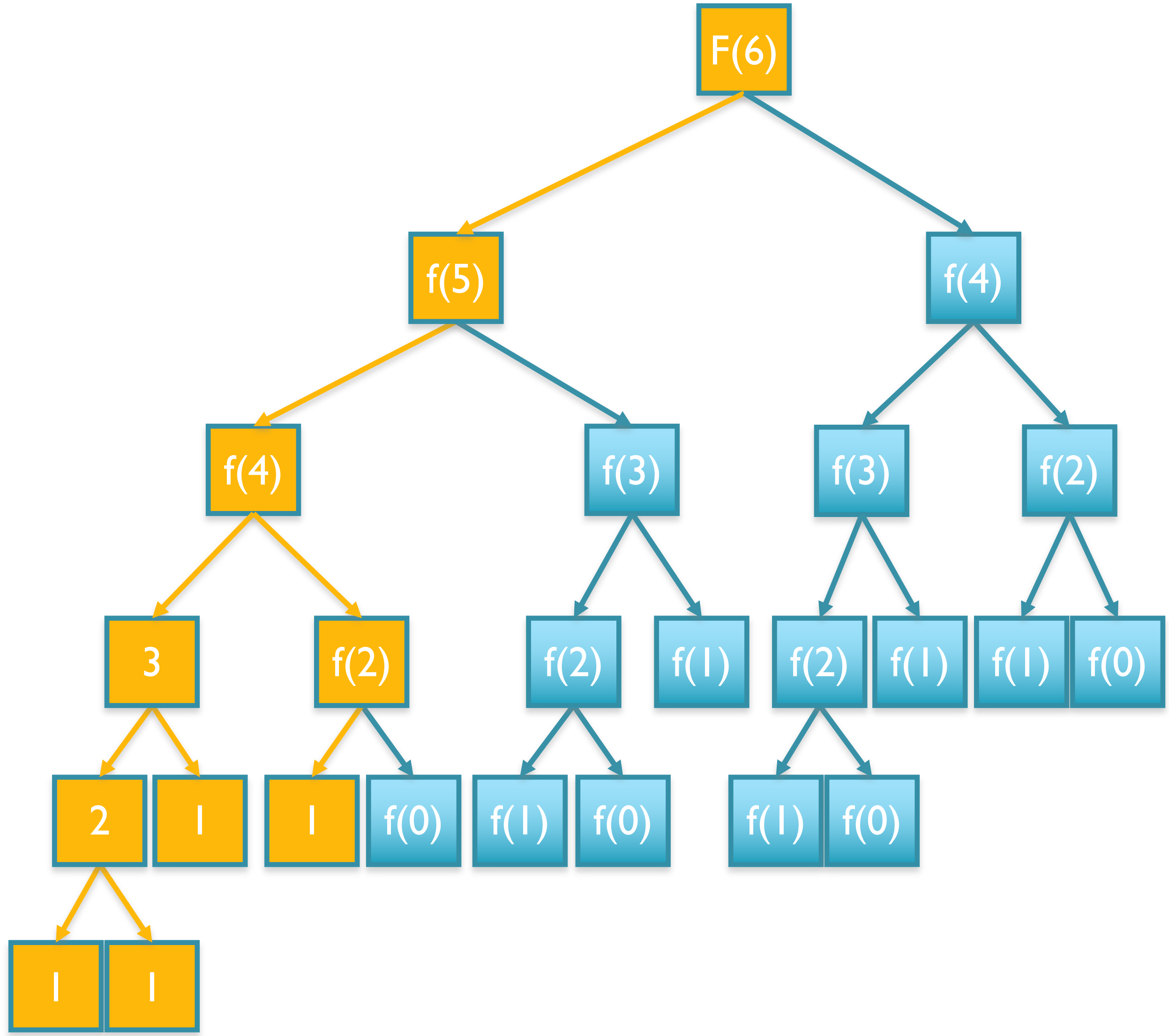
F(4)
b = F(2)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence



F(2)
b = F(0)

F(4)
b = F(2)

F(5)
a = F(4)

F(6)
a = F(5)

Fibonacci Sequence

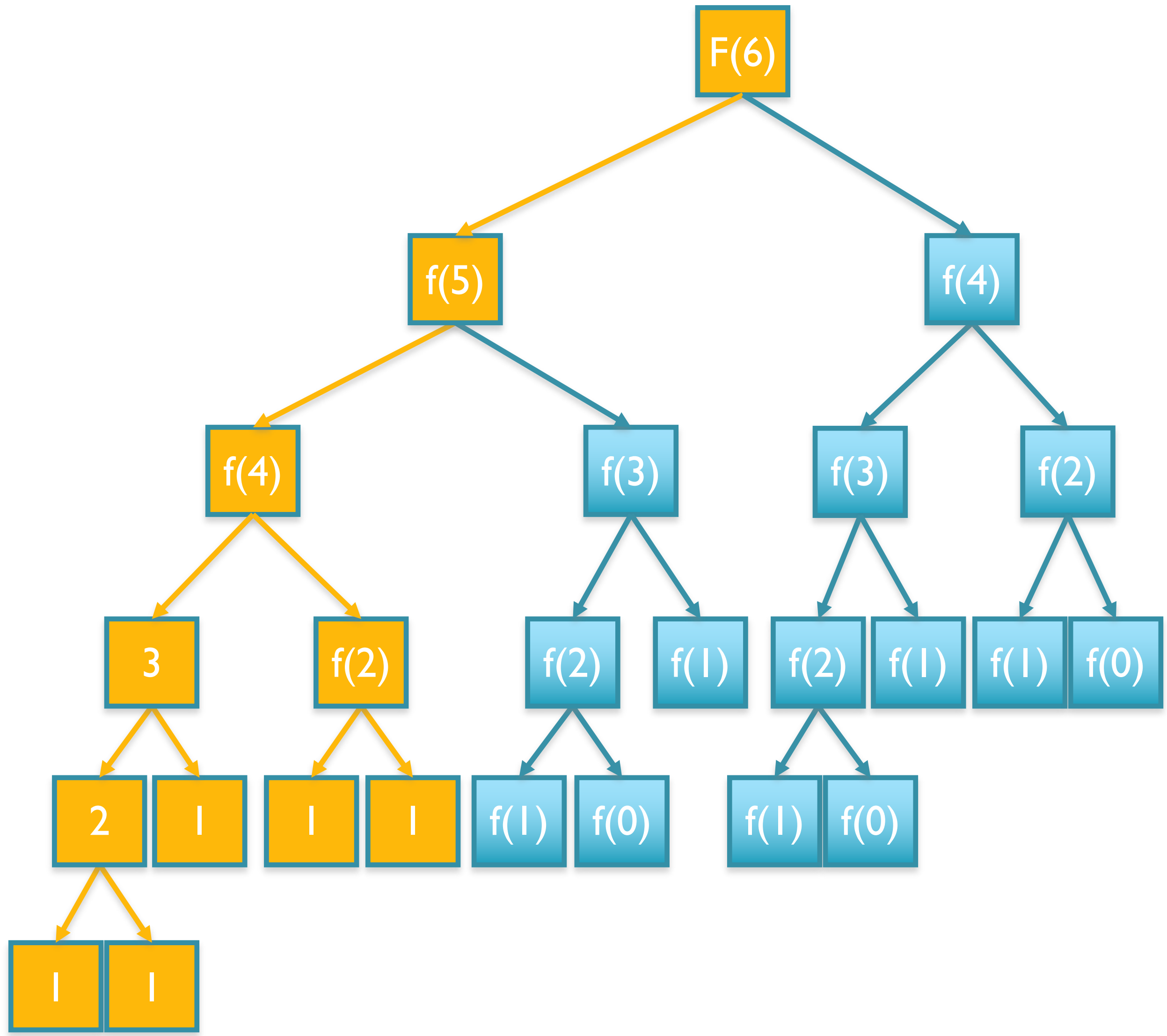
F(0)
return 1

F(2)
b = F(0)

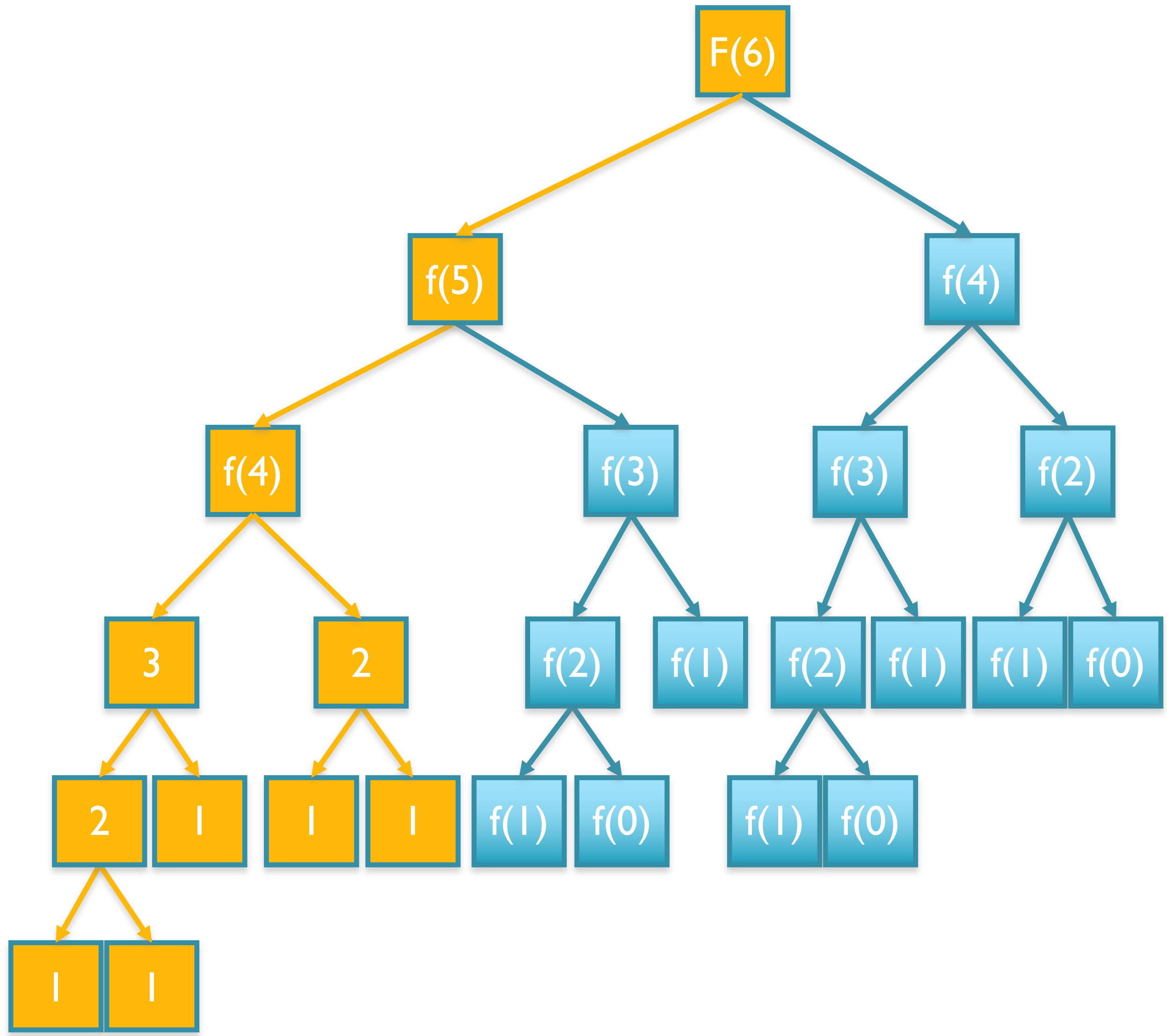
F(4)
b = F(2)

F(5)
a = F(4)

F(6)
a = F(5)



Fibonacci Sequence



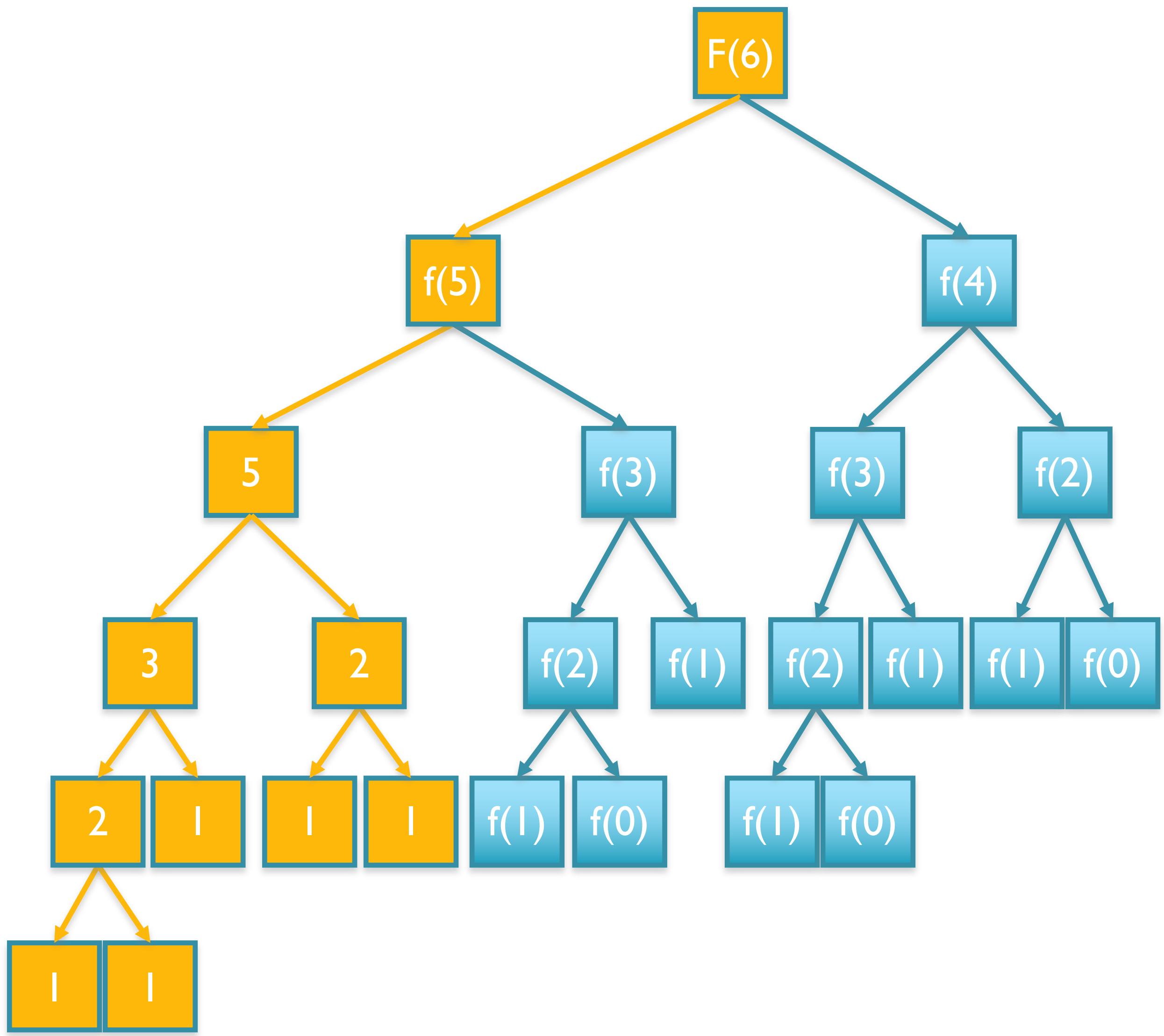
F(2)
return 2

F(4)
b = F(2)

F(5)
a = F(4)

F(6)
a = F(5)

Fibonacci Sequence



F(4)
return 5

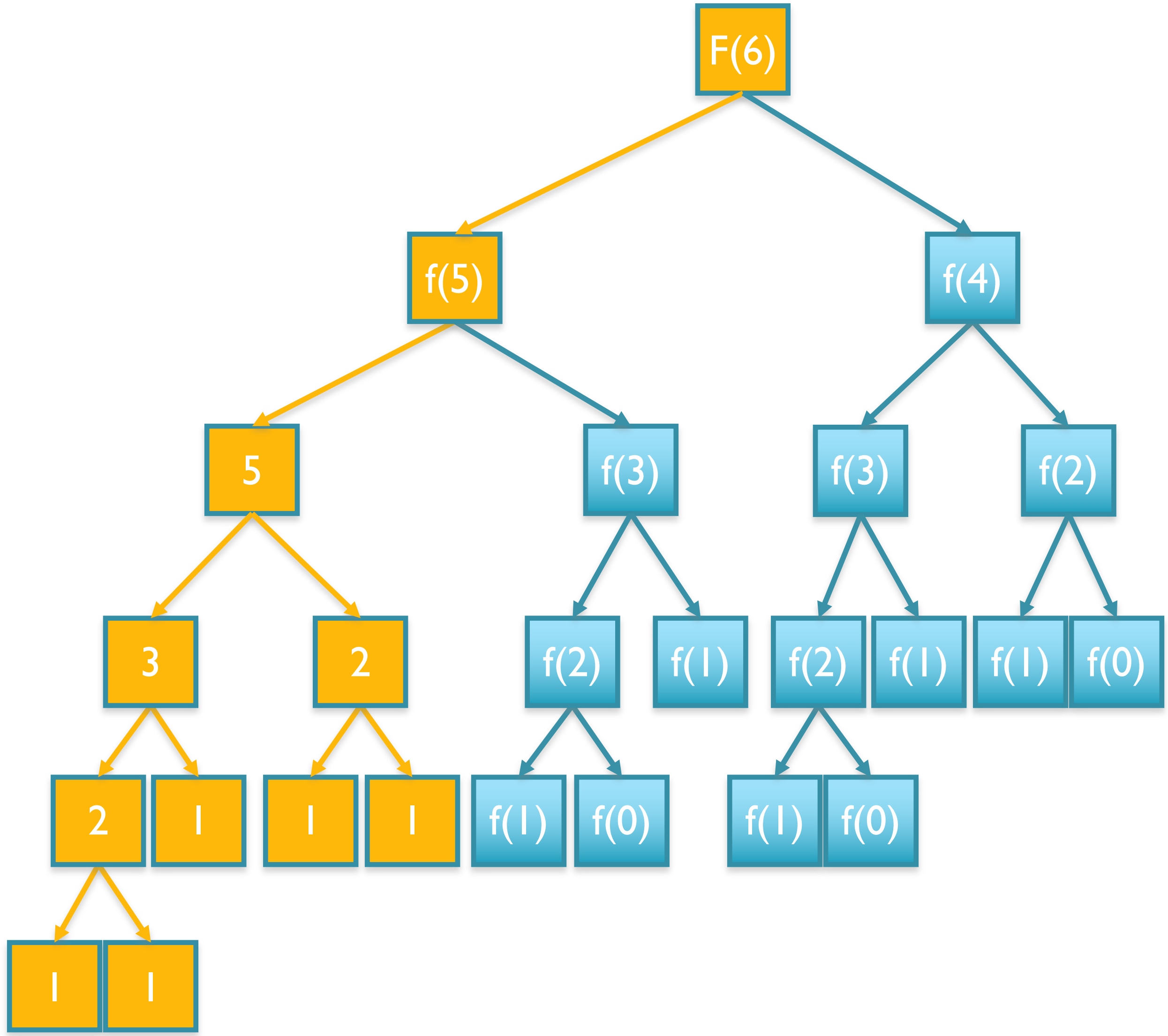
F(5)
a = F(4)

F(6)
a = F(5)

Fibonacci Sequence

Equivalent to running through a maze and always keeping your right hand on the wall

Notice we only look at the top of the stack to keep track of where we have been and were we should go next!



F(5)
a = F(4)

F(6)
a = F(5)

Stack ADT

```
adt Stack
  uses Any, Boolean
  defines Stack<T: Any>
  operations
    new:
    push:
    pop:
    top:
    empty:
  preconditions

  axioms
```

Stack ADT

```
adt Stack
   uses Any, Boolean
   defines Stack<T: Any>
   operations
      new:    ---> Stack<T>
      push:   Stack<T> x T ---> Stack<T>
      pop:    Stack<T> ---> Stack<T>
      top:    Stack<T> ---> T
      empty: Stack<T> ---> Boolean
   preconditions
      pop(s): not empty(s)
      top(s): not empty(s)
   axioms
      empty(new())
      not empty(push(s, t))
      top(push(s, t)) = t
      pop(push(s, t)) = s
```

Note: pop(push(s,t)) = s is comparing the whole stack, but we havent defined what it means for one stack to equal another!

Relax this flaw by considering the axioms as rewrite rules:

top(pop(push(push(new(), 1), 2))) = 1

Can be rewritten as

top(push(new(), 1)) = 1

and finally to

1=1

Stack ADT

```
adt Stack
  uses Any, Boolean
  defines Stack<T: Any>
  operations
    new:    ---> Stack<T>
    push
    pop:
    top:
    empt
  precon
    pop(
    top(
  axioms
    empt
    not
    top(push(s, t)) = t
    pop(push(s, t)) = s
```

You might be tempted to try to derive a correct equality operation, or add additional constraints (like maximum number of elements that the stack can store) but the ADT gets very messy

See lecture notes for details

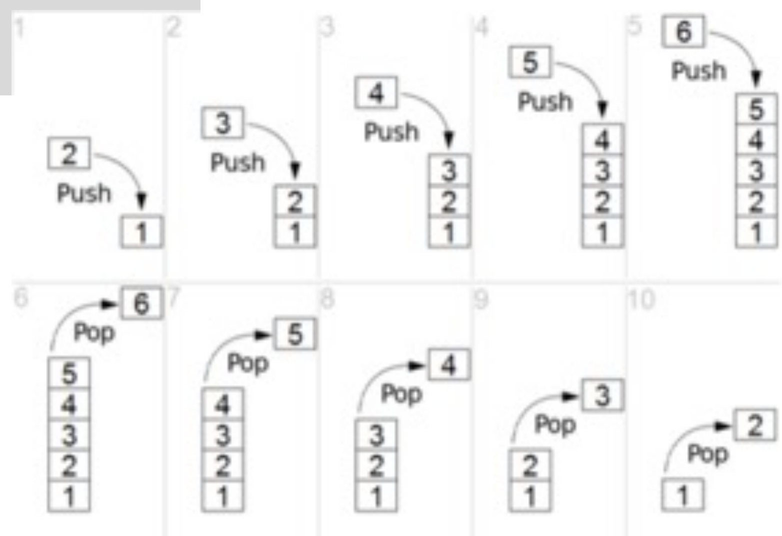
Instead, lets focus on implementation instead

Stack Interface

```java
public interface Stack<T> {
    // checks if empty
    boolean empty();

    // peeks at top value without removing
    T top() throws EmptyException;

    // removes top element
    void pop() throws EmptyException;

    // adds new element to top of stack
    void push(T t);
}
```

How would you implement this interface?

Why?



```java
/**
    Stack implemented using a linked list.

    All operations take O(1) time in the worst case; however
    each push() results in a new object being allocated which
    may be inappropriate for some applications.

    @param <T> Element type.
*/
public class ListStack<T> implements Stack<T> {
    private static class Node<T> {
        Node<T> next;
        T data;
    }

    private Node<T> first;

    /**
        Create an empty stack.
    */
    public ListStack() {
    }

    @Override
    public T top() throws EmptyException {
        try {
            return this.first.data;
        } catch (NullPointerException e) {
            throw new EmptyException();
        }
    }

    @Override
    public void pop() throws EmptyException {
        try {
            this.first = this.first.next;
        } catch (NullPointerException e) {
            throw new EmptyException();
        }
    }

    @Override
    public void push(T t) {
        Node<T> n = new Node<T>();
        n.data = t;
        n.next = this.first;
        this.first = n;
    }

    @Override
    public boolean empty() {
        return this.first == null;
    }

    @Override
    public String toString() {
        String s = "[";
        for (Node<T> n = this.first; n != null; n = n.next) {
            s += n.data.toString();
            if (n.next != null) {
                s += ", ";
            }
        }
        s += "]";
        return s;
    }
}
```

ListStack

vs.

ArrayStack

Which is better?

Why?

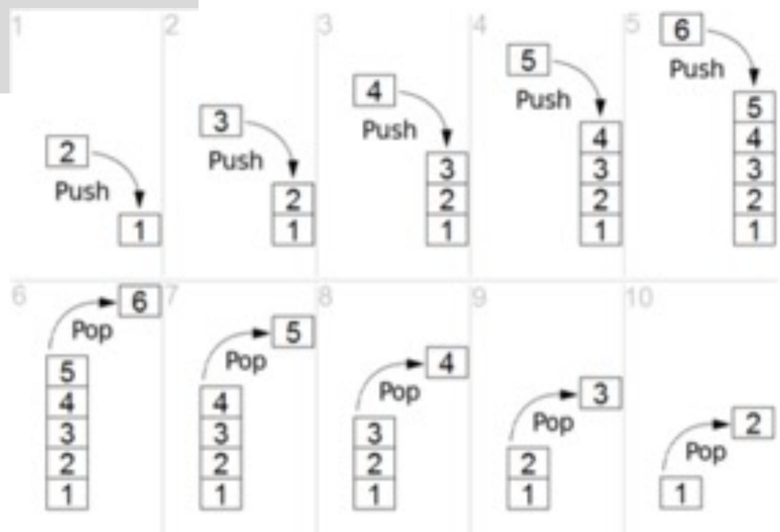
```java
/**
    Stack implemented using a growing array.

    All operations except push() take O(1) time in the worst
    case; push() takes O(1) amortized time because the array
    may need to be resized; however, compared to ListStack,
    fewer push() operations result in objects being allocated.

    @param <T> Element type.
*/
public class ArrayStack<T> implements Stack<T> {
    private T[] data;
    private int used;

    /**
        Create an empty stack.
    */
    public ArrayStack() {
        this.data = (T[]) new Object[1];
    }

    @Override
    public T top() throws EmptyException {
        if (this.empty()) {
            throw new EmptyException();
        }
        return this.data[this.used - 1];
    }

    @Override
    public void pop() throws EmptyException {
        if (this.empty()) {
            throw new EmptyException();
        }
        this.used -= 1;
    }

    private boolean full() {
        return this.data.length == this.used;
    }

    private void grow() {
        T[] bigger = (T[]) new Object[this.data.length * 2];
        for (int i = 0; i < this.used; i++) {
            bigger[i] = this.data[i];
        }
        this.data = bigger;
    }

    @Override
    public void push(T t) {
        if (this.full()) {
            this.grow();
        }
        this.data[this.used] = t;
        this.used += 1;
    }

    @Override
    public boolean empty() {
        return this.used == 0;
    }

    @Override
    public String toString() {
        String s = "[";
        for (int i = this.used - 1; i >= 0; i--) {
            s += this.data[i].toString();
            if (i > 0) {
                s += ", ";
            }
        }
        s += "]";
        return s;
    }
}
```

Stack Interface

```java
public interface Stack<T> {
    // checks if empty
    boolean empty();

    // peeks at top value without removing
    T top() throws EmptyException;

    // removes top element
    void pop() throws EmptyException;

    // adds new element to top of stack
    void push(T t);
}
```

How would you *test* the implementation?

Why?



Introducing JUnit

Lecture 2: SimpleCounter.java

```
public class SimpleCounter implements Counter {

    public static void main(String[] args) {
        Counter c = new SimpleCounter();
        assert c.value() == 0;
        System.out.println("Counter is now: " + c.value());
        c.up();
        assert c.value() == 1;
        System.out.println("Counter is now: " + c.value());
        c.down();
        assert c.value() == 0;
        System.out.println("Counter is now: " + c.value());
        c.down();
        c.up();
        c.up();
        c.up();
        System.out.println("Counter is now: " + c.value());
        assert c.value() == 2;
    }
}
```

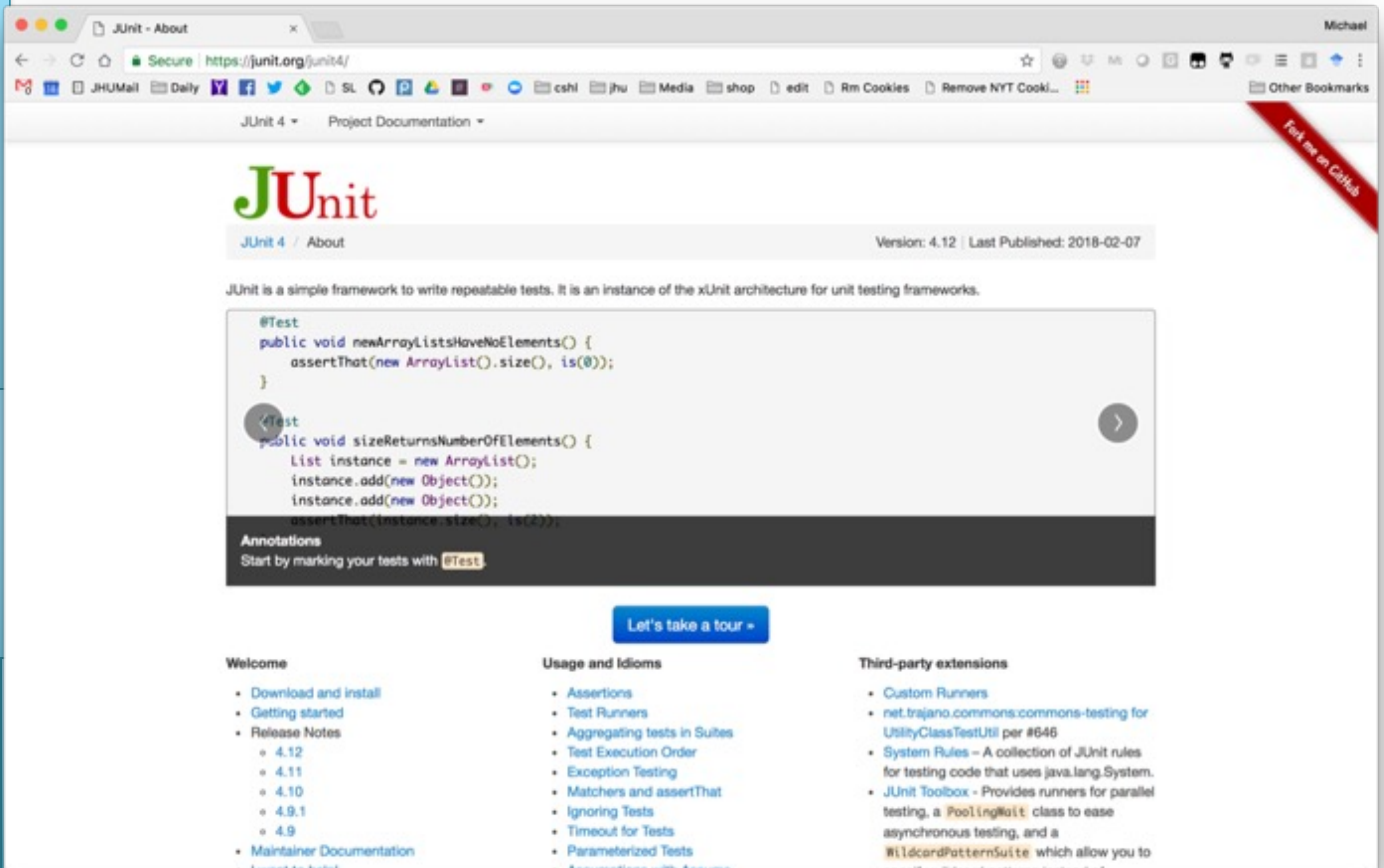
Asserts are very useful for testing, but are very limited especially because first failed assert kills the entire progaram ☹

Lecture 2: SimpleCounter.java

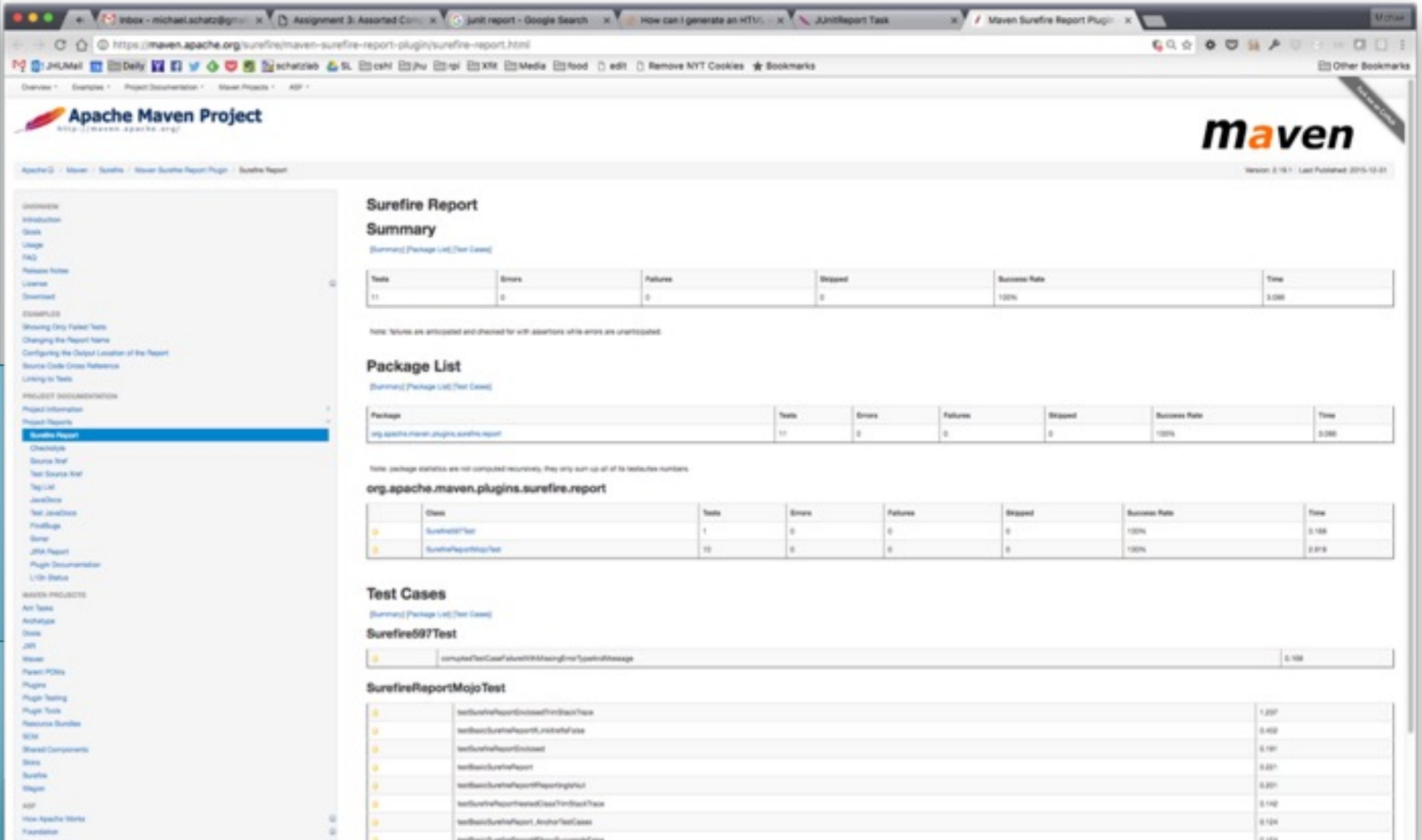
```
public static void main(String[] args) {
  MyArray a = new MyArray(5, "Mike");

  a.put(3, "Peter");
  a.put(2, 1234);

  for (int i = 0; i < a.length(); i++) {
    System.out.println("a[" + i + "]: " +
                       a.get(i) + " " + a.get(i).getClass());
  }

  try {
    System.out.println("a[" + 57 + "]: " + a.get(57));
  } catch (IndexException e) {
    System.out.println("Caught IndexException (as expected)");
  }
}
```

Printing is useful while developing, but becomes unscalable in large programs with lots of methods to test

Introducing JUnit



Sample Report



JUnit According to Peter ☺

So why use a testing framework like JUnit instead of writing the tests like we did so far, using Java's assert instruction and a main method with the test code in it?

- ***For one thing, JUnit allows you to modularize your tests better.*** It's not uncommon for large software projects to have just as much testing code as actual program code, and so the principles you use to make regular code easier to read (splitting things into methods and classes, etc.) should also apply to test code.

- ***Also, JUnit allows you to run all your test cases every time***, it doesn't stop at the first failing test case like assert does. This way you can get feedback about multiple failed tests all at once.

- ***Finally, lots of companies expect graduates to have some experience with testing frameworks***, so why not pick it up now? Note that testing is not just for software developers anymore, increasingly people working with software developers but who are themselves not software developers will be asked to contribute to testing a certain application being developed by their company.

So it's a really useful skill to have on your list.

JUnit According to Peter ☺

So why use a testing framework like JUnit instead of writing the tests like we did so far, using Java’s assert instruction and a main method with the test code in it?

- ***For one thing, JUnit allows you to modularize your tests better.*** It’s not uncommon for large software projects to have just as much testing c … r … c.) s …

From HW3 on out, when we say “write test cases” as part of an assignment, we mean “write JUnit 4 test cases” as described here.

- *A* … ’t s … f …
- ***I*** … ***ce*** … ***with testing frameworks,*** so why not pick it up now? Note that testing is not just for software developers anymore, increasingly people working with software developers but who are themselves not software developers will be asked to contribute to testing a certain application being developed by their company.

So it’s a really useful skill to have on your list.

Next Steps

1. Work on HW2

2. Check on Piazza for tips & corrections!