CS 106X

Lecture 12: Memoization and Structs

Monday, February 6, 2017

Programming Abstractions (Accelerated)
Winter 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 10

Today's Topics

- Logistics

- Assignment four: backtracking!
- Memoization
- More on Structs

Memoization

> Tell me and I forget. Teach me and I rememoize.*

- Xun Kuang, 300 BCE

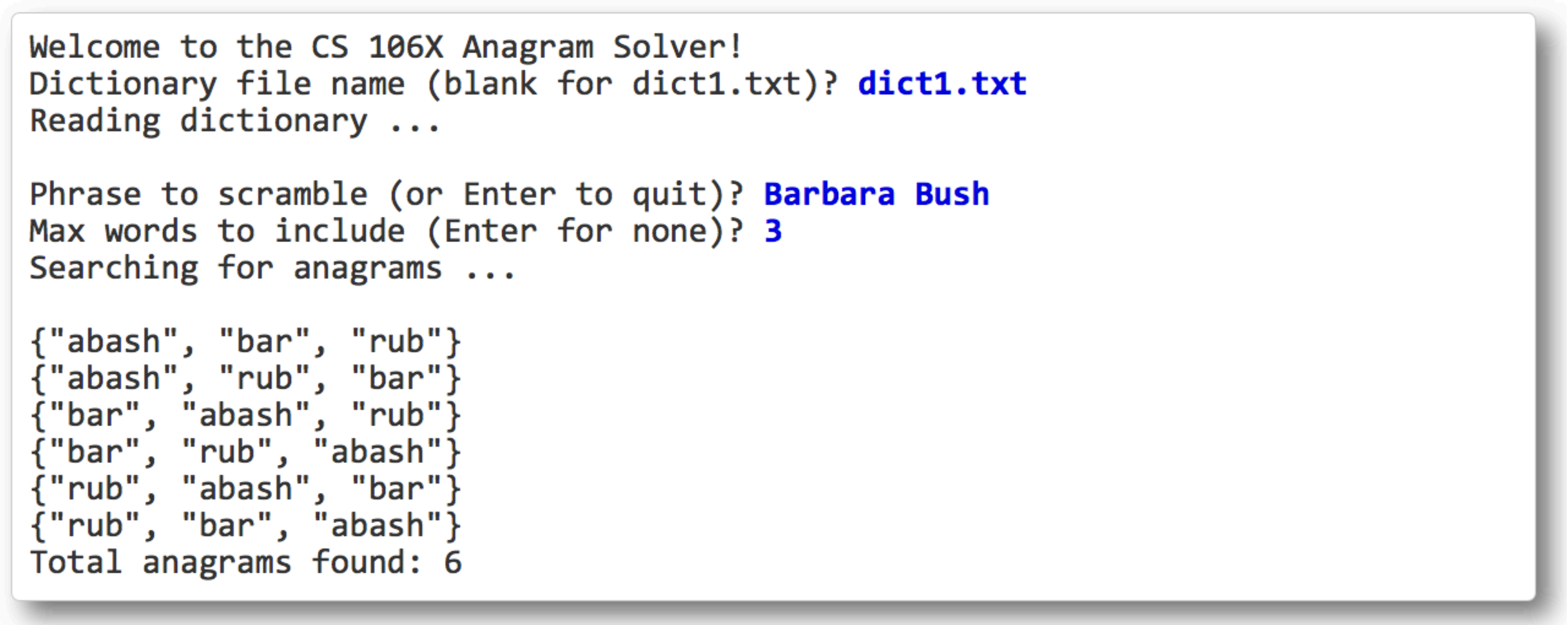
* Some poetic license used when translating quote

Assignment 4: Backtracking

Three Parts:

- Anagrams
- Boggle
- Brute Force Decryption

Assignment 4a: Anagrams



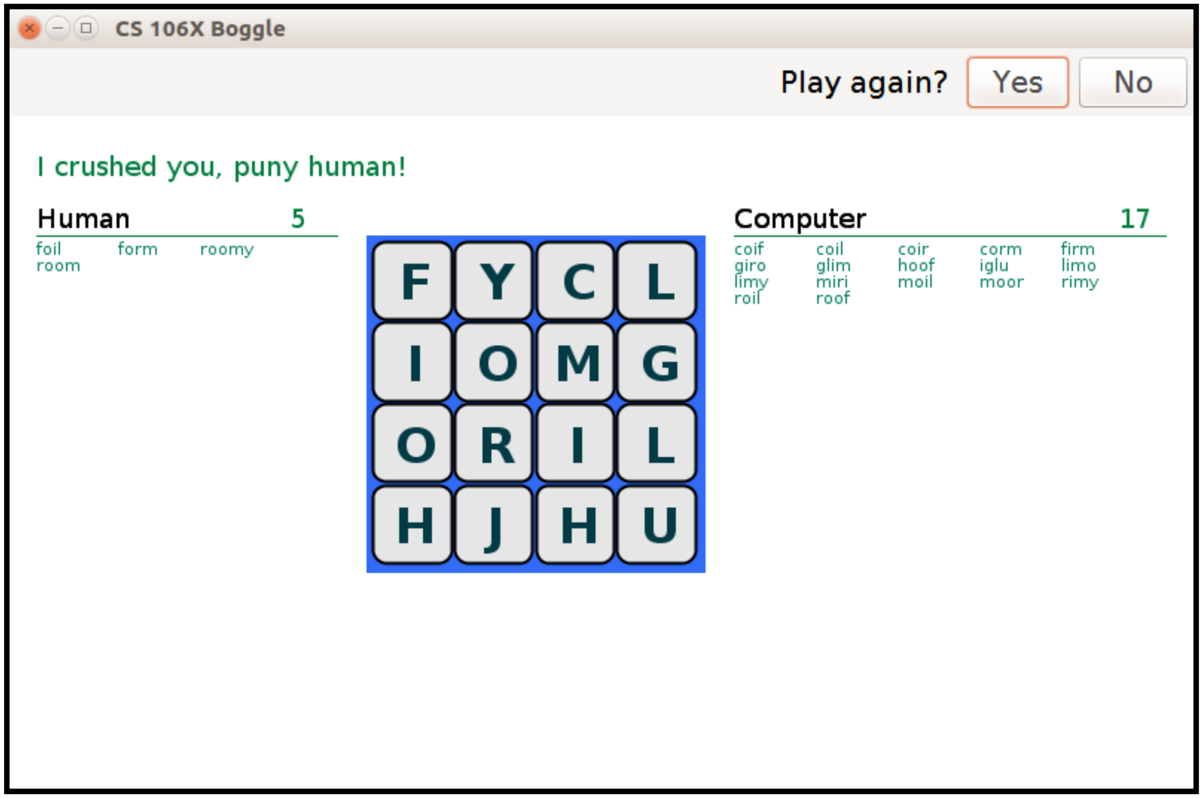
```
Welcome to the CS 106X Anagram Solver!
Dictionary file name (blank for dict1.txt)? dict1.txt
Reading dictionary ...

Phrase to scramble (or Enter to quit)? Barbara Bush
Max words to include (Enter for none)? 3
Searching for anagrams ...

{"abash", "bar", "rub"}
{"abash", "rub", "bar"}
{"bar", "abash", "rub"}
{"bar", "rub", "abash"}
{"rub", "abash", "bar"}
{"rub", "bar", "abash"}
Total anagrams found: 6
```

"...recursively find and print all anagrams that can be formed using all of the letters of the given phrase, and that include at most max words total, in alphabetical order..."

Assignment 4b: Boggle

A classic board game with letter cubes (dice) that is not dog friendly: https://www.youtube.com/watch?v=2shOz1ZLw4c

Assignment 4b: Boggle



I crushed you, puny human!

```
Human                5
foil      form      roomy
room
```

F	Y	C	L
I	O	M	G
O	R	I	L
H	J	H	U

```
Computer                                     17
coif      coil      coir      corm      firm
giro      glim      hoof      iglu      limo
limy      miri      moil      moor      rimy
roil      roof
```

In Boggle, you can make words starting with any letter and going to any adjacent letter (diagonals, too), but you cannot repeat a letter-cube.

Assignment 4c: Brute Force Decryption

Dance like no one is watching. Encrypt like everyone is.

- For this problem, you will extend your Assignment 2 transposition cipher code to attempt a brute-force decryption of a cipher, *without the key*.

Demo

Assignment 4c: Brute Force Decryption

- Transposition ciphers (the way we have implemented them) have an inherent flaw: multiple keys will decrypt a phrase. If the original key was "secret", all of the following keys will decrypt: "SECRET", "pedler", "scales", "teapot", "hedges", "medley", and "521436".

- All of the above keys have the same alphabetic ordering.

- You will leverage this fact to determine all possible permutations for an n-character key, and then you will simply run the decryption algorithm you wrote for assignment 2 on each possible key.

Dance like no one is watching. Encrypt like everyone is.

Assignment 4c: Brute Force Decryption

- You will need to produce a "Top X" (set at five with a constant, although we can change this for testing) list of possible decryptions:

```
Please enter the text to attempt to decrypt: rsuetoye ss tr merxdteopce pHs  t cierete.
Testing keys of length 1
Testing keys of length 2
Testing keys of length 3
Testing keys of length 6
Testing keys of length 7

My best guess for the top 5 decryption possibilities:

Here is some super secret text to decrypt. (100%)

trxy ecrsde re ut te mees tpetosceHrop i.s (62.5%)

re ieHsome supers ecres text to det ypt.rc (50%)

i Hereems sore super sectxt teed to .tcryp (50%)

yrxt ecesdr re ut te seem tpstoeceH opri.s (50%)
```

only check keys that would evenly divide into the ciphertext length! (another flaw in the encryption)

Keep a sorted top X list to return

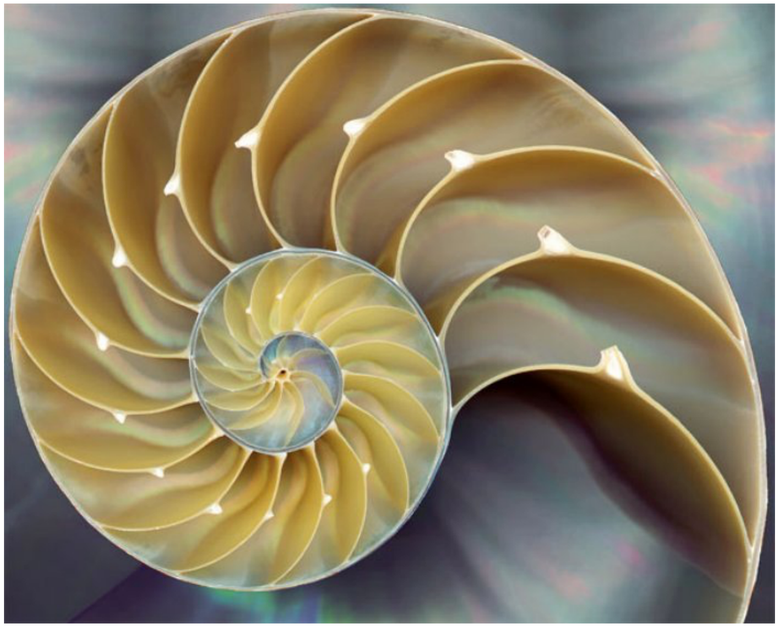
Beautiful Recursion

- Let's look at one of the most beautiful recursive definitions:

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$

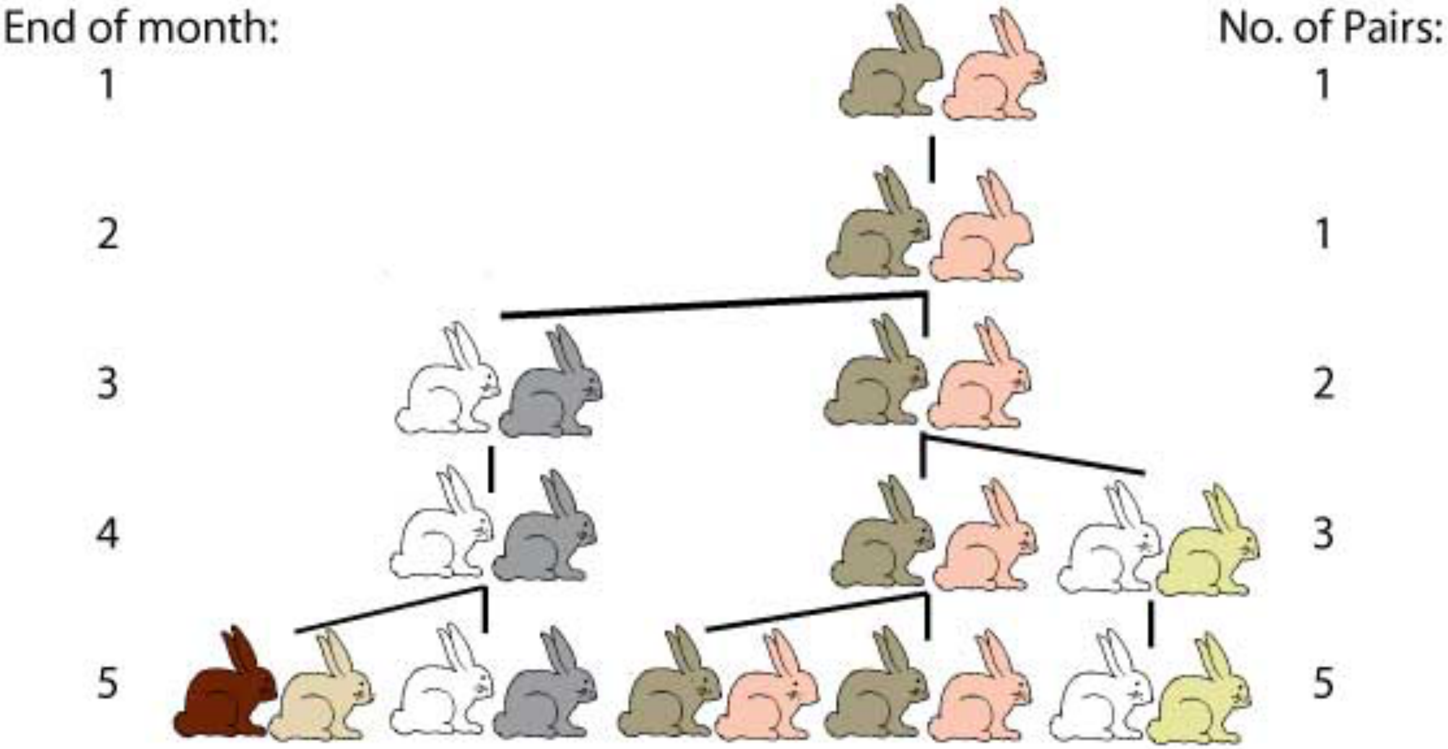
$$\textit{where}\ F_0{=}0,\ F_1{=}1$$

  - This definition leads to this:



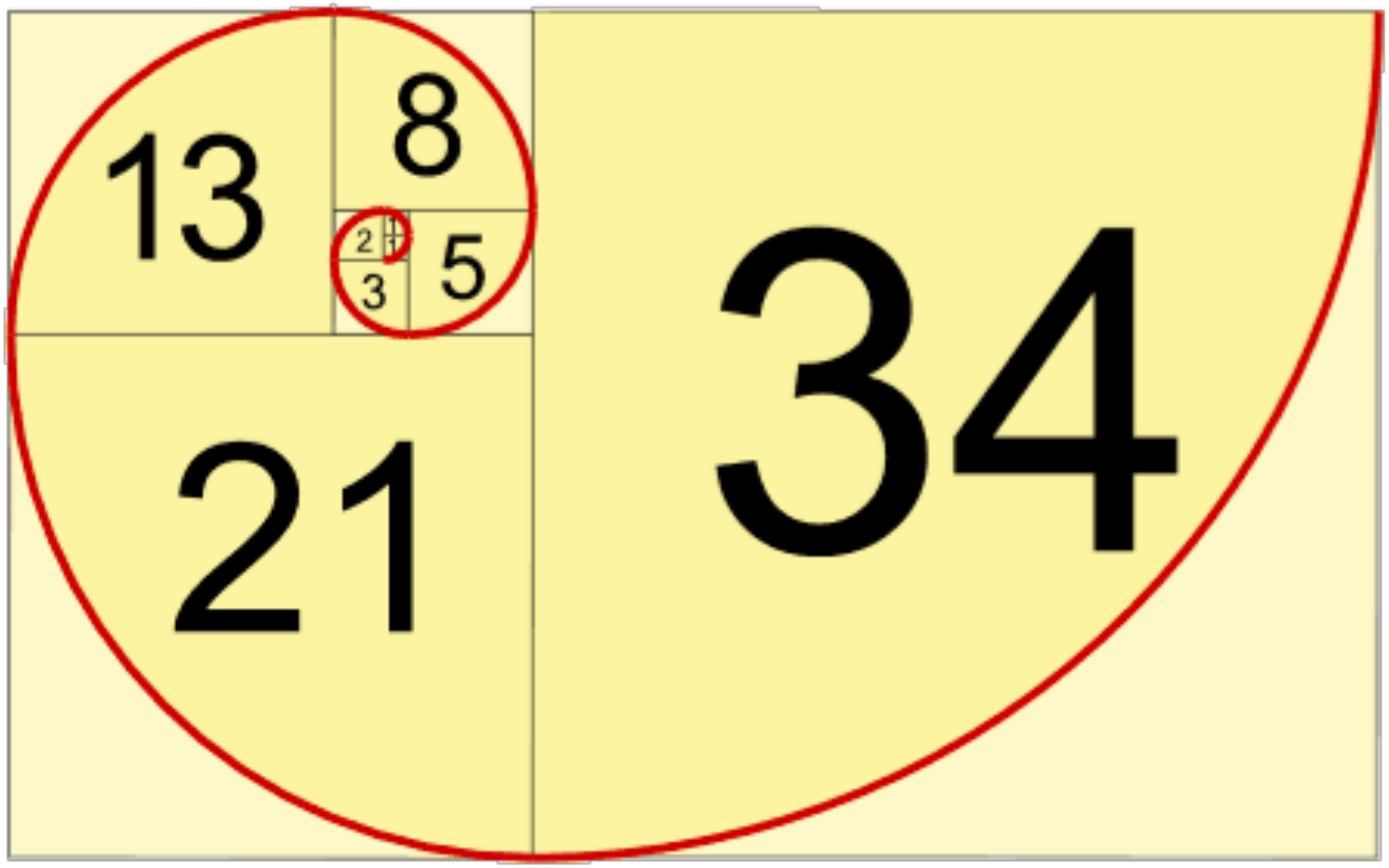
Beautiful Recursion

- And this:



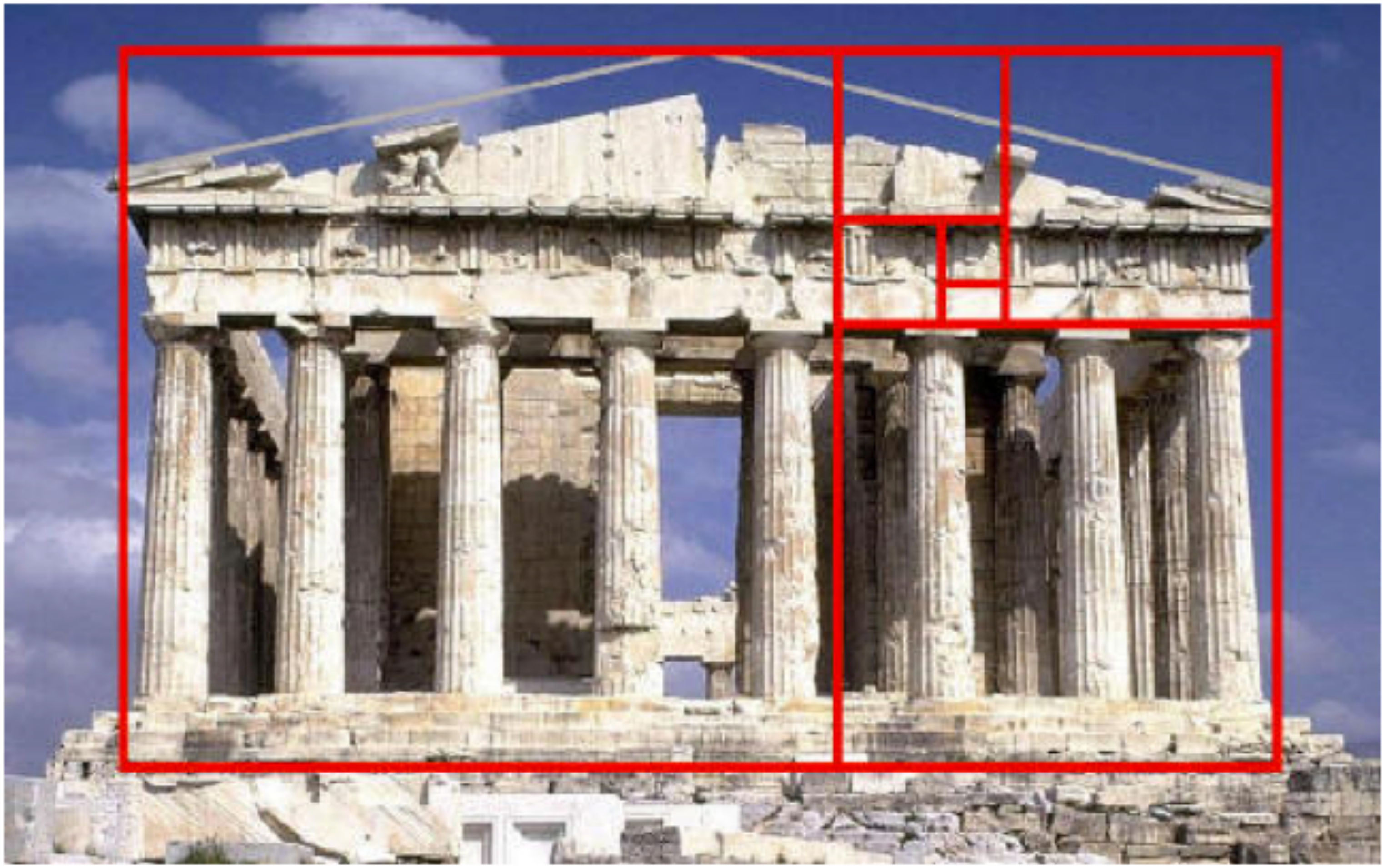
Beautiful Recursion

- And this:



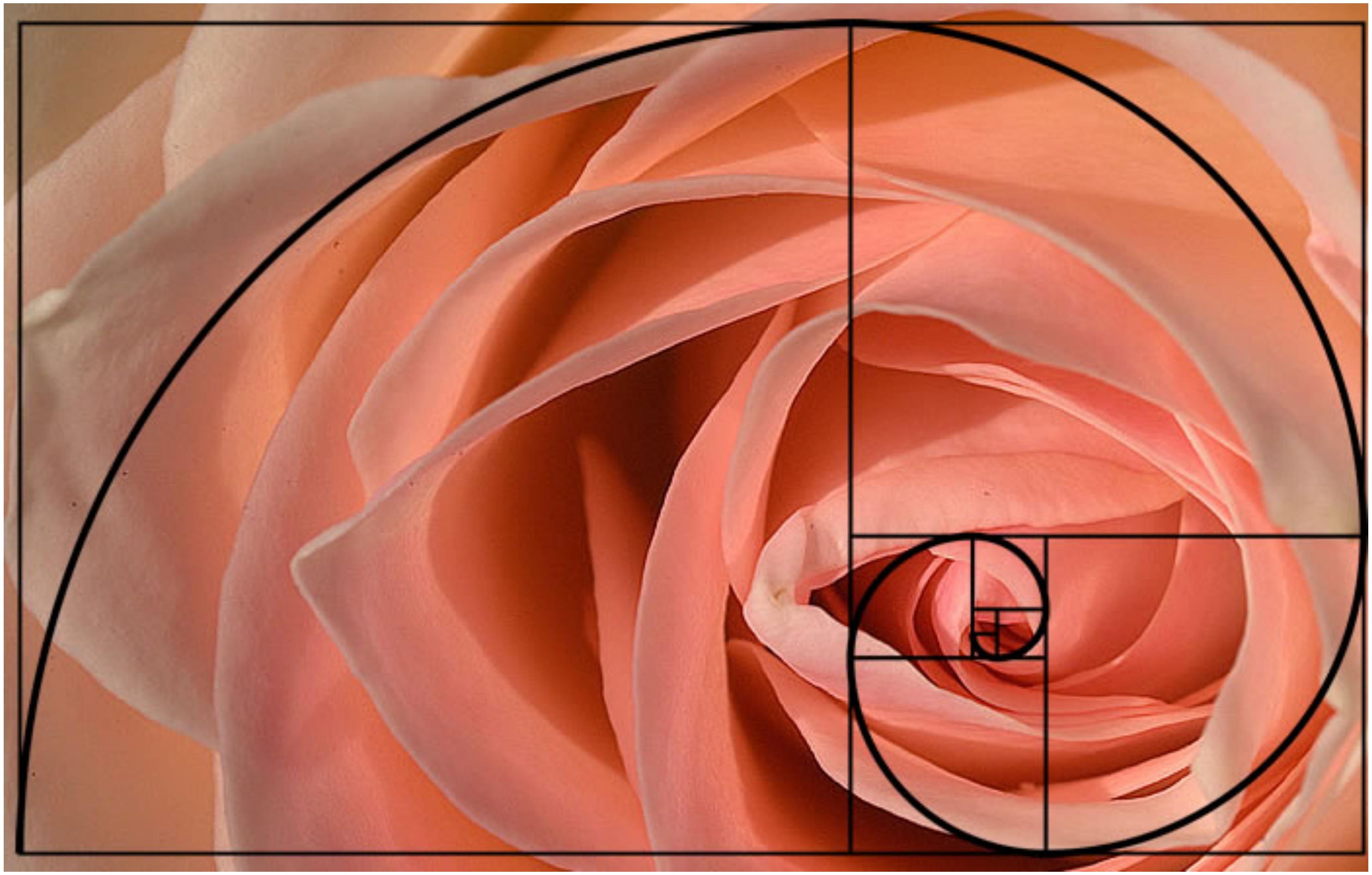
Beautiful Recursion

- And this:



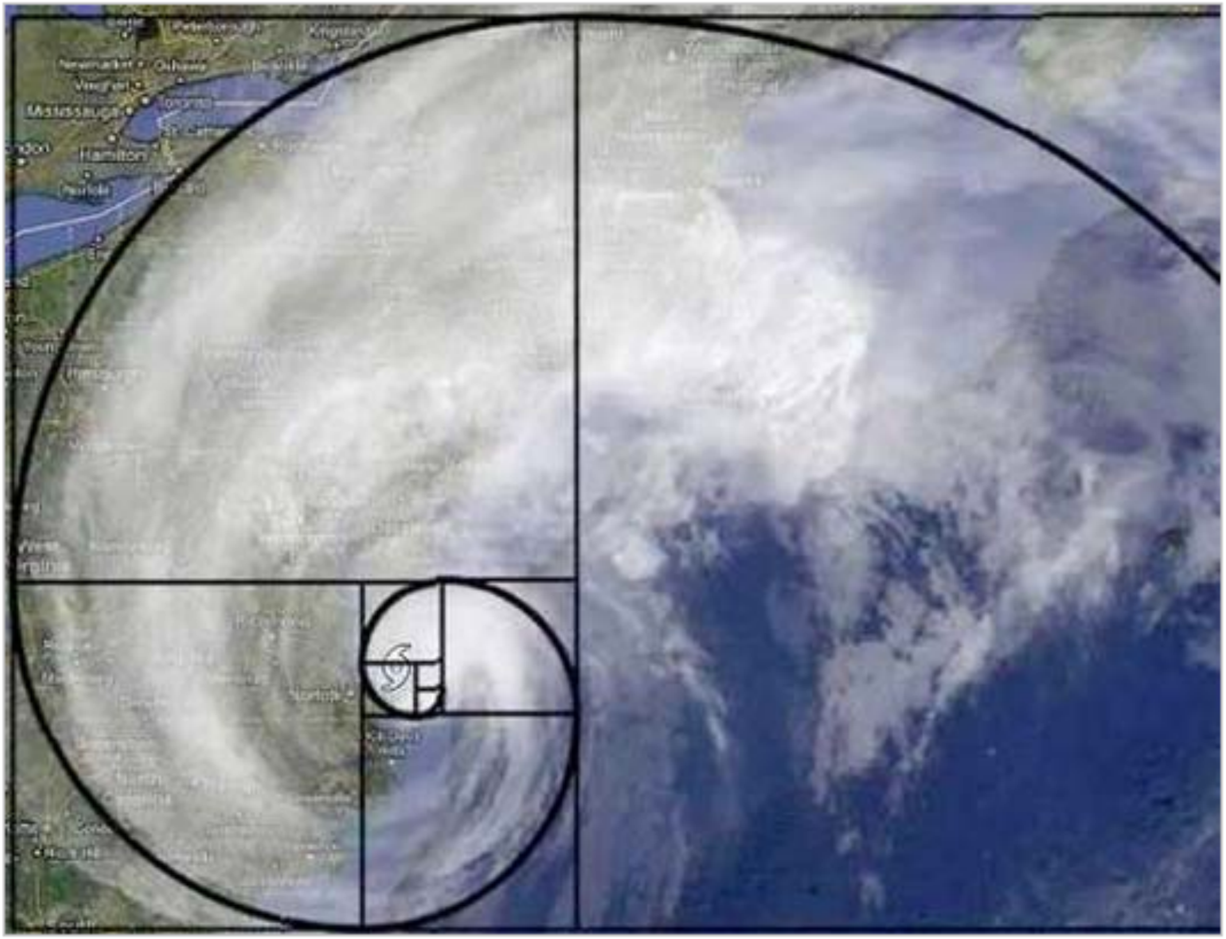
Beautiful Recursion

- And this:



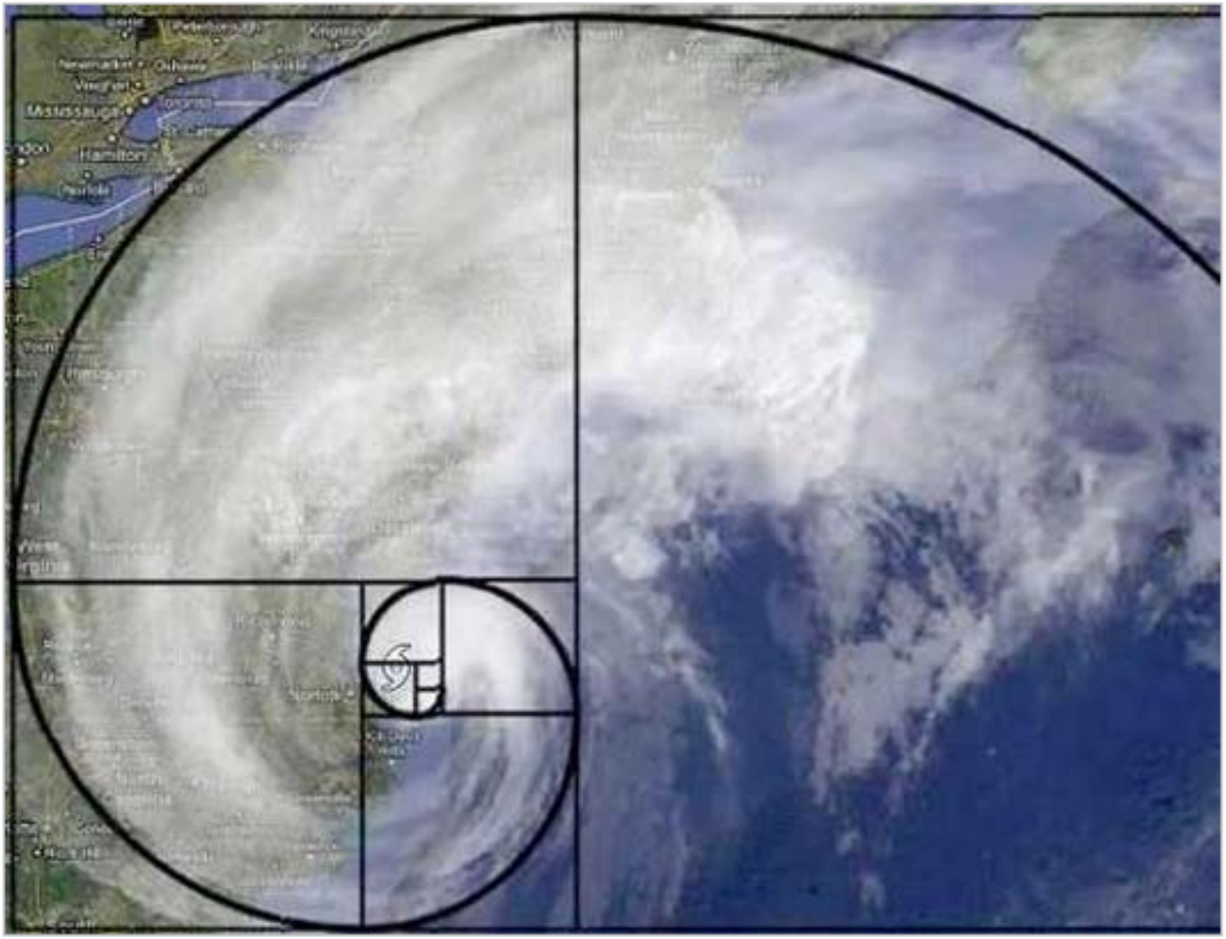
Beautiful Recursion

- And this:



Beautiful Recursion

- And this:



The Fibonacci Sequence

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$
$$\textit{where } F_0=0,\ F_1=1$$

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

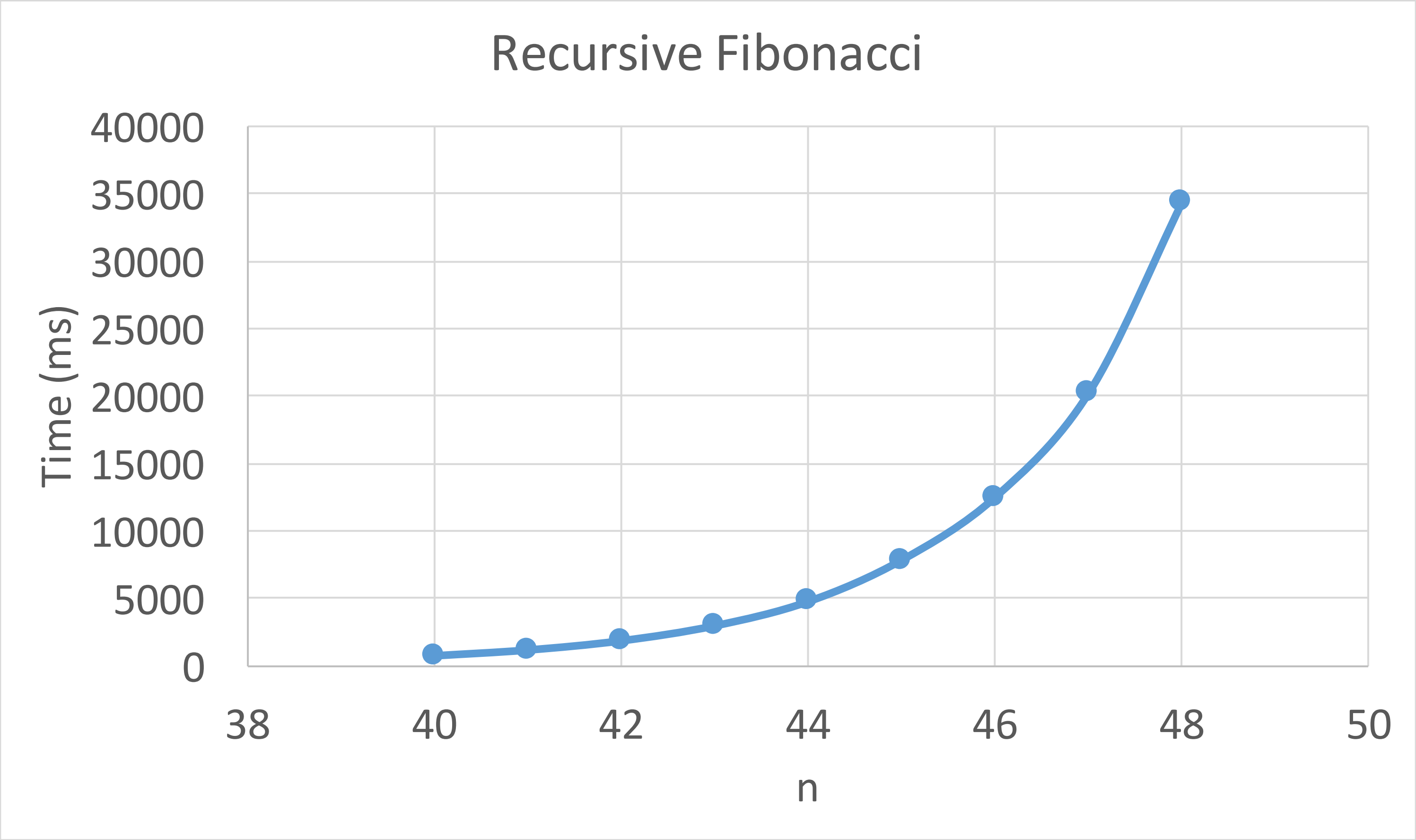
This is particularly easy to code recursively!

```
long plainRecursiveFib(int n) {
    if(n == 0) {
        // base case
        return 0;
    } else if (n == 1) {
        // base case
        return 1;
    } else {
        // recursive case
        return plainRecursiveFib(n - 1) + plainRecursiveFib(n - 2);
    }
}
```

Let's play!

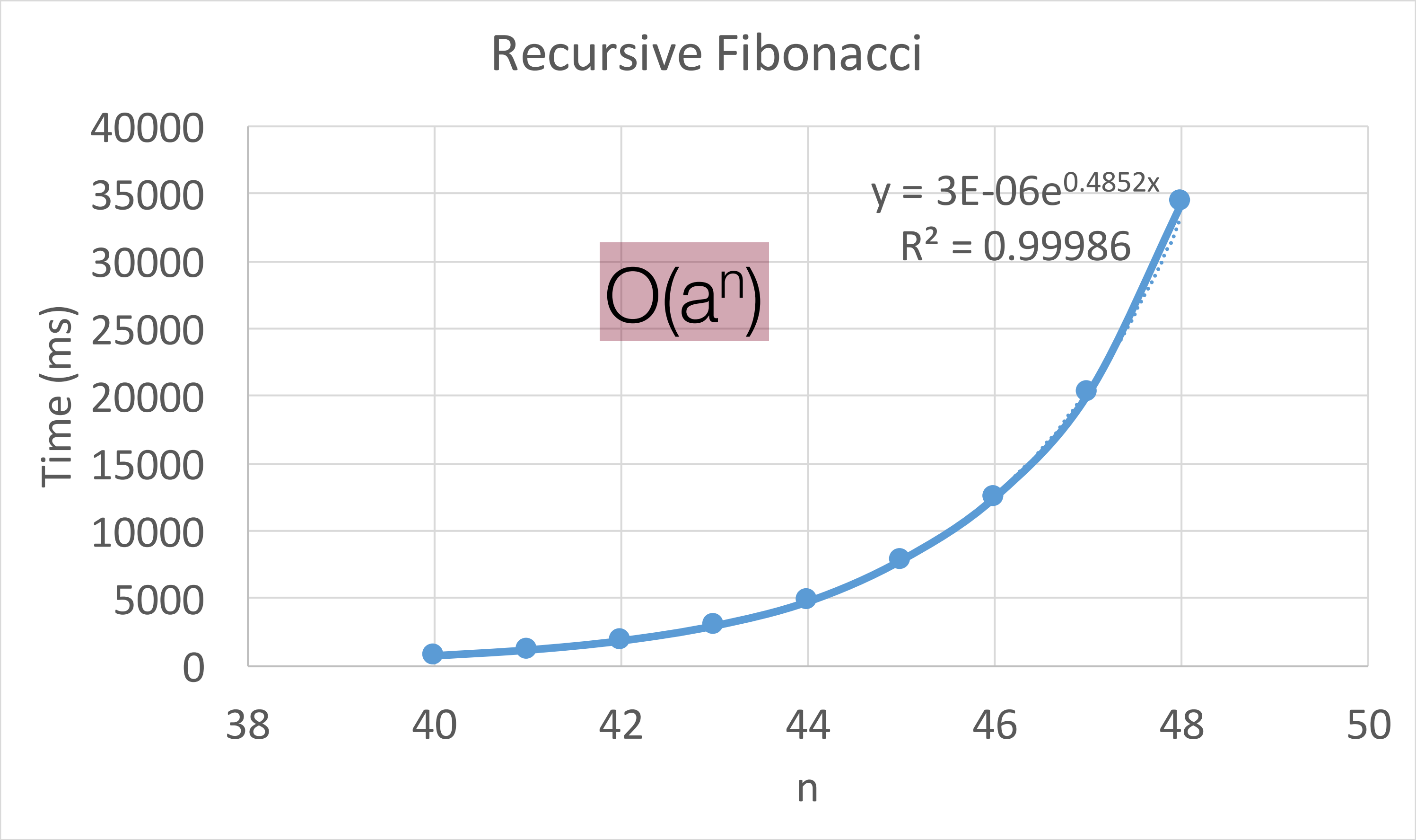
The Fibonacci Sequence

What happened??



The Fibonacci Sequence

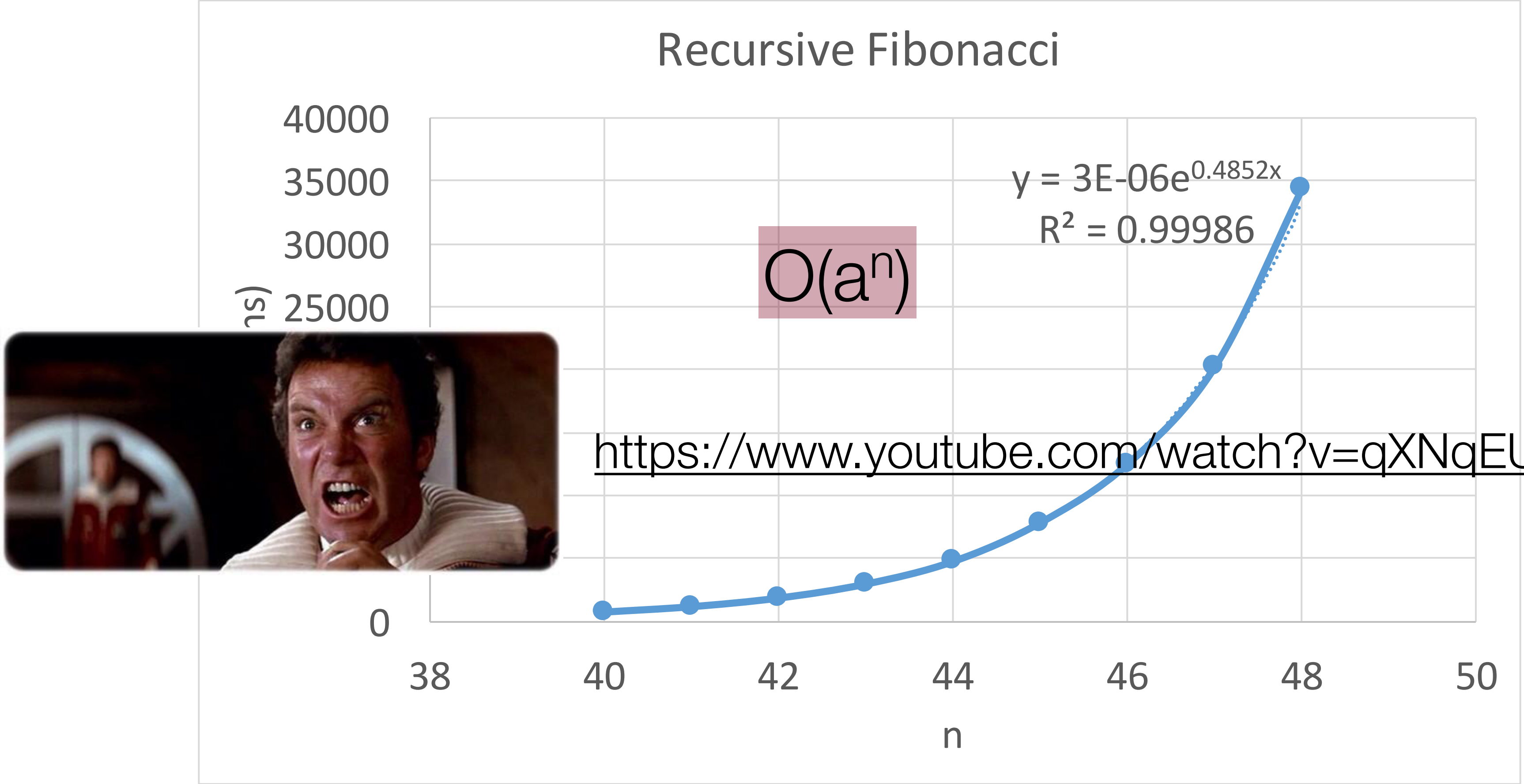
What happened??



$O(a^n)$

The Fibonacci Sequence

What happened??

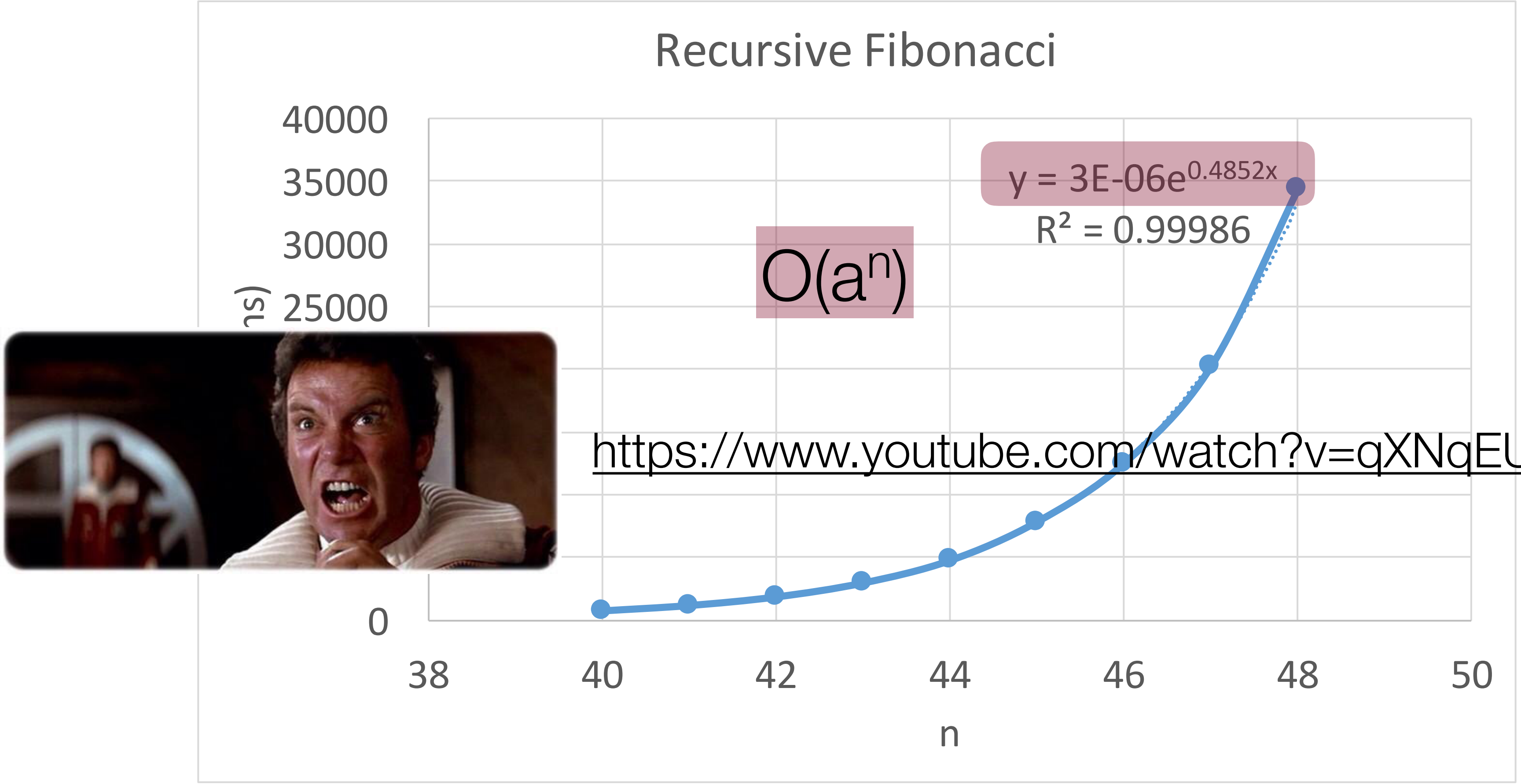


$O(a^n)$

https://www.youtube.com/watch?v=qXNqEURmKtA

The Fibonacci Sequence

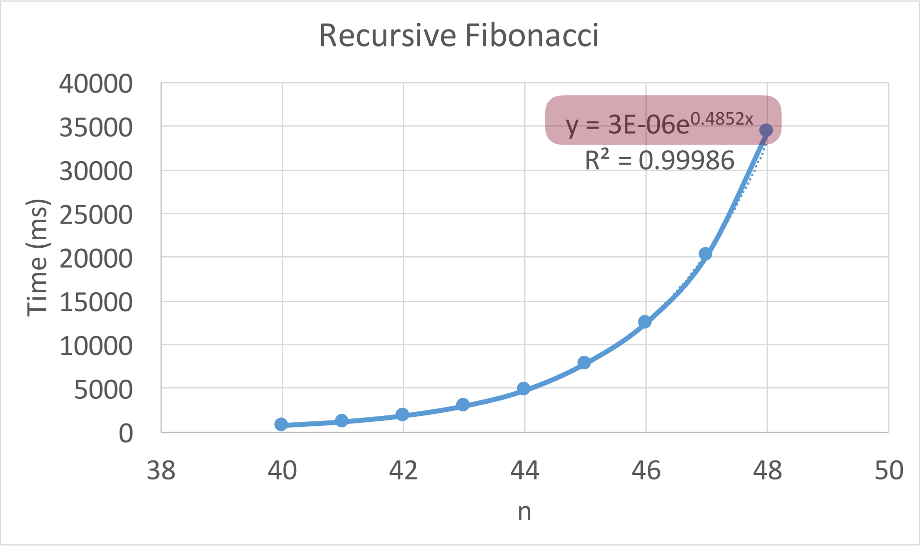
What happened??



$O(a^n)$

https://www.youtube.com/watch?v=qXNqEURmKtA

The Fibonacci Sequence



By the way:

$$3\text{x}10^{-6}e^{0.4852n} \cong O(1.62^n)$$

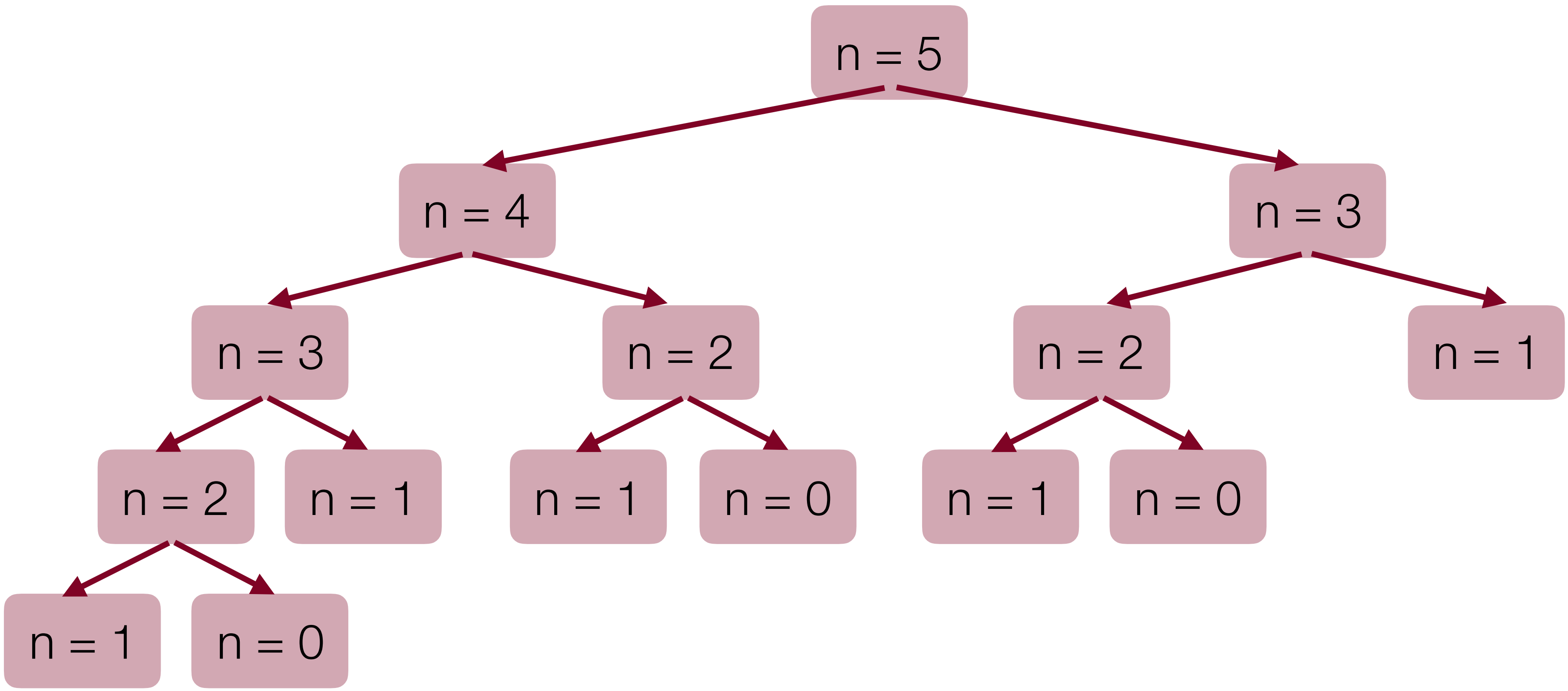
$O(1.62^n)$ is technically $O(2^n)$

because

$$O(1.62^n) < O(2^n)$$

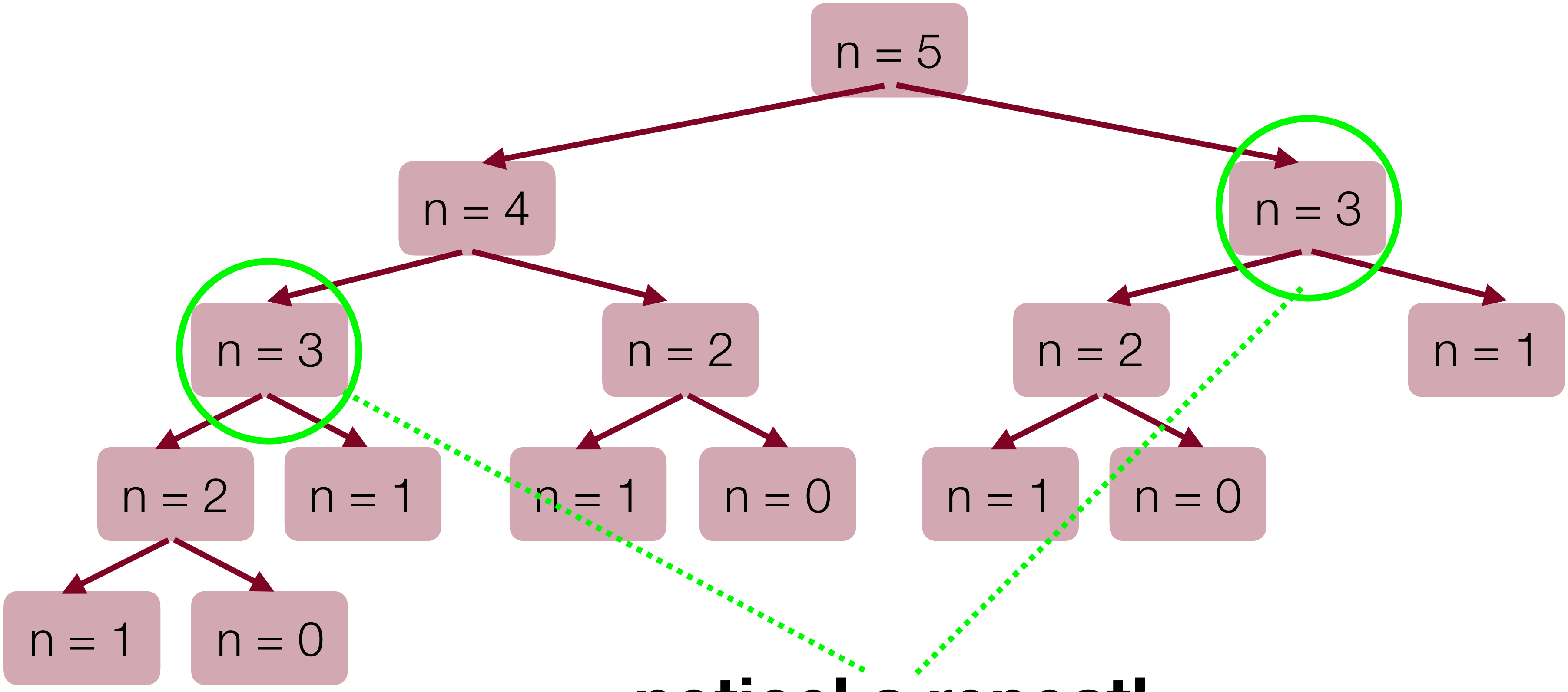
We call this a "tighter bound," and we like round numbers, especially ones that are powers of two. :)

Fibonacci: Recursive Call Tree



This is basically the reverse of binary search: we are splitting into two marginally smaller cases, not splitting into half of the problem size!

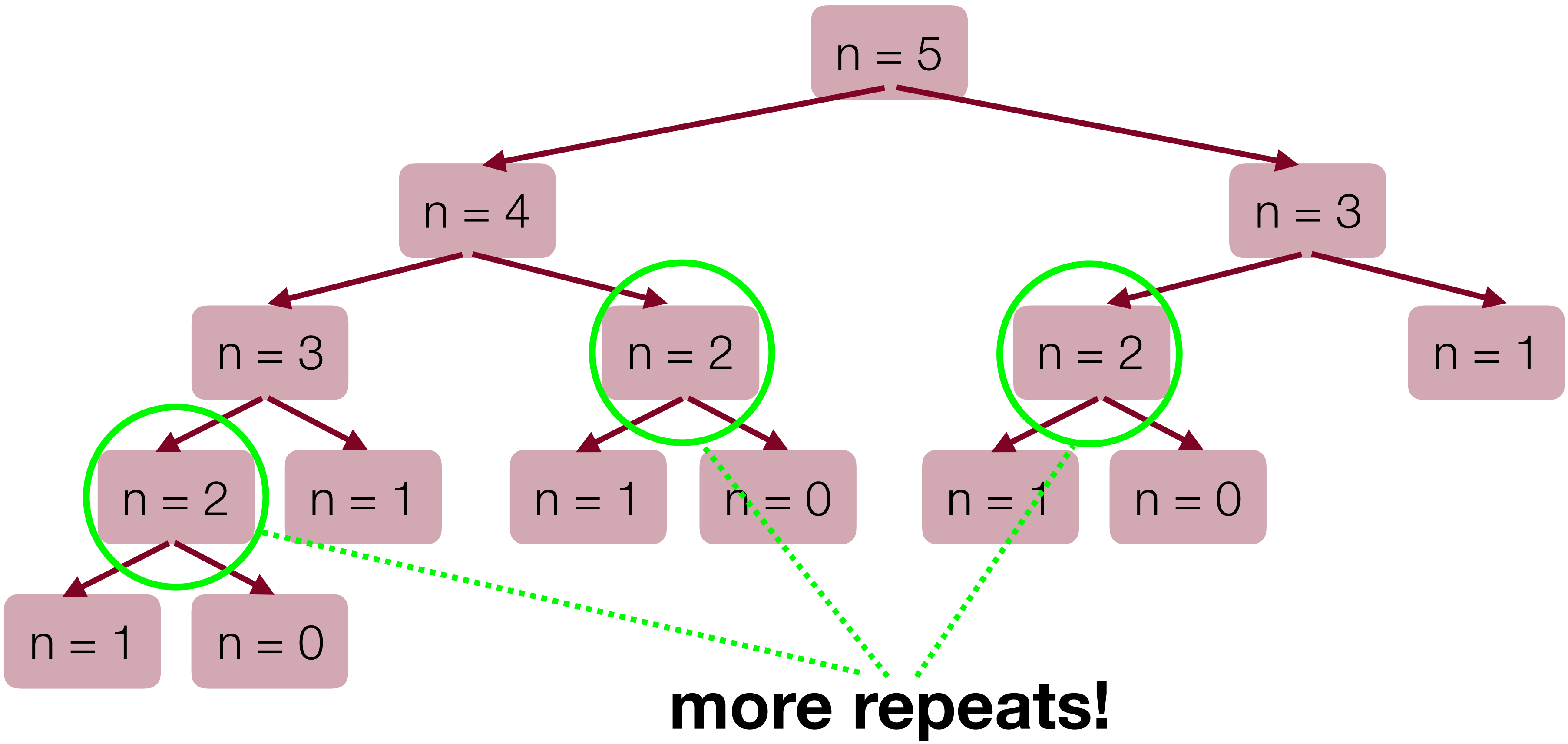
Fibonacci: There is hope!



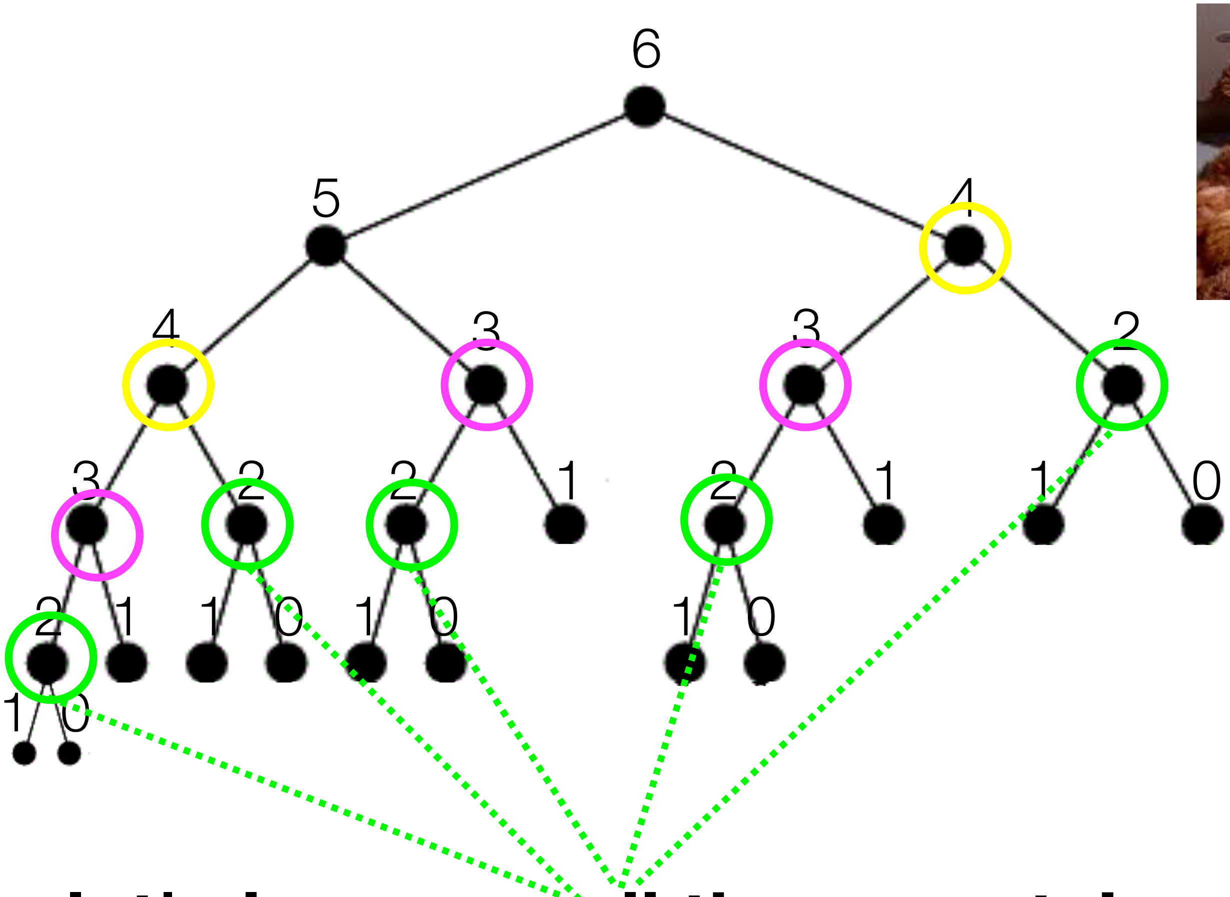
notice! a repeat!

fib(3) is completely calculated twice

Fibonacci: There is hope!

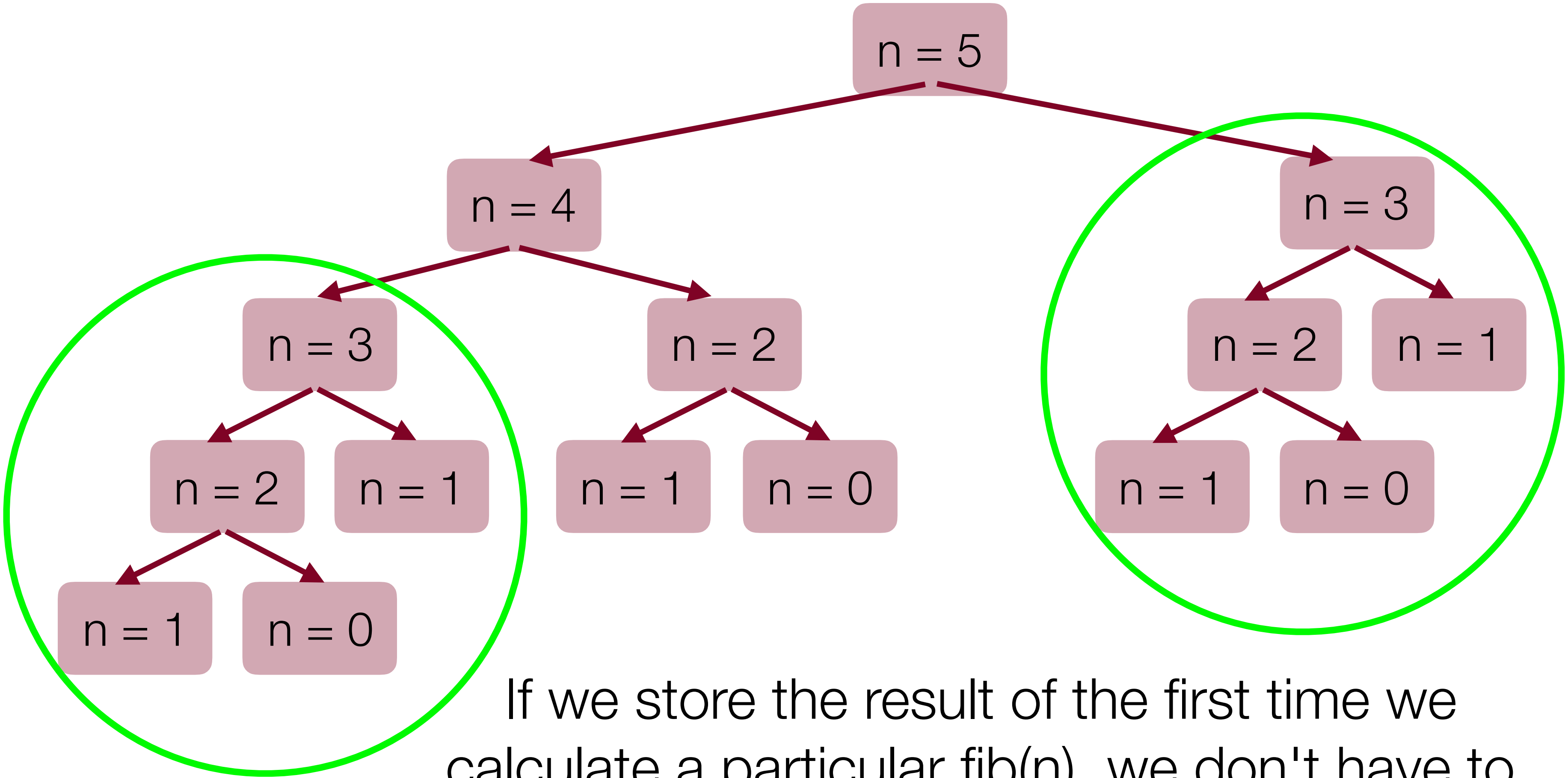


Fibonacci: There is hope!



let's leverage all the repeats!

Fibonacci: There is hope!



If we store the result of the first time we calculate a particular fib(n), we don't have to re-do it!

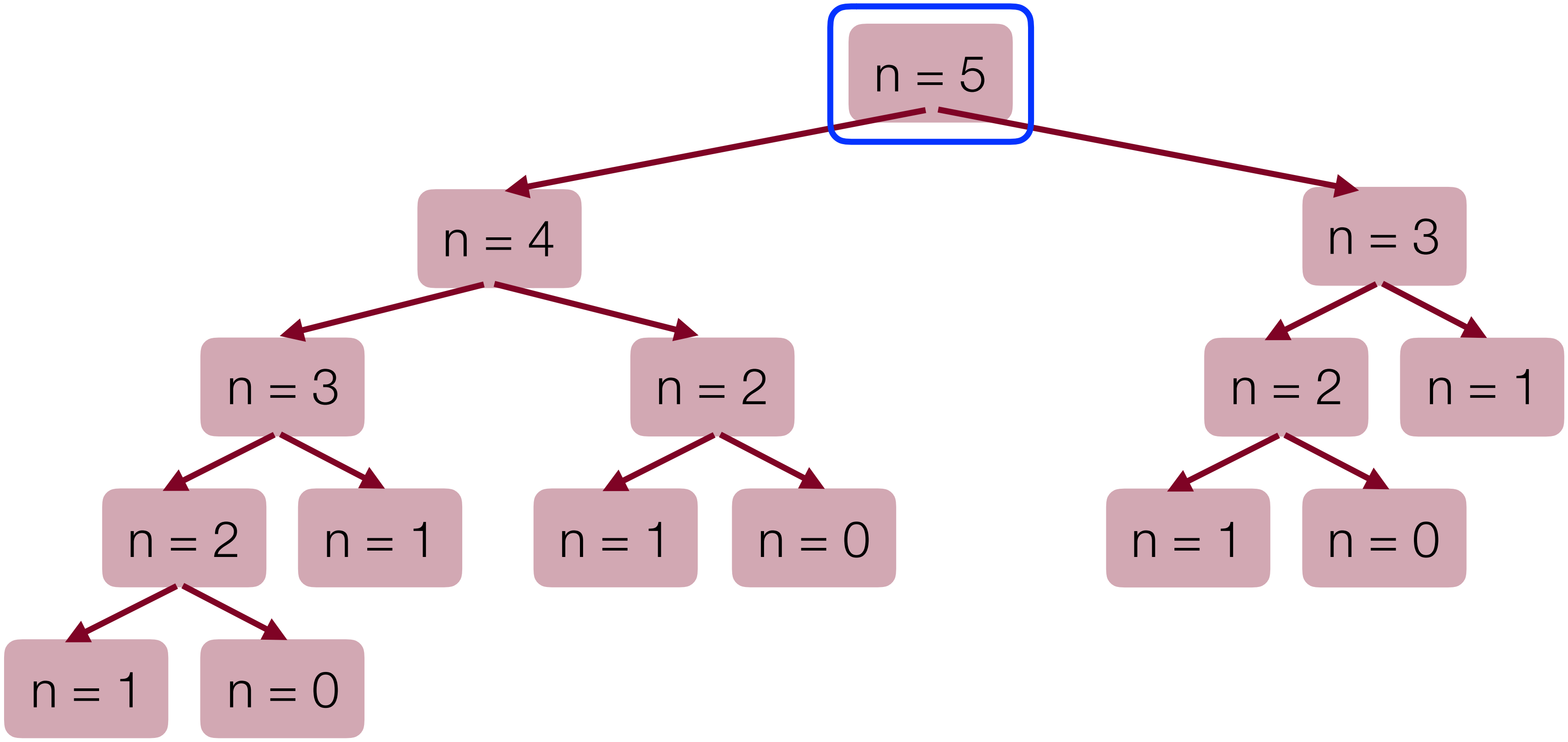
Memoization: Don't re-do unnecessary work!

Memoization: Store previous results so that in future executions, you don’t have to recalculate them.

aka

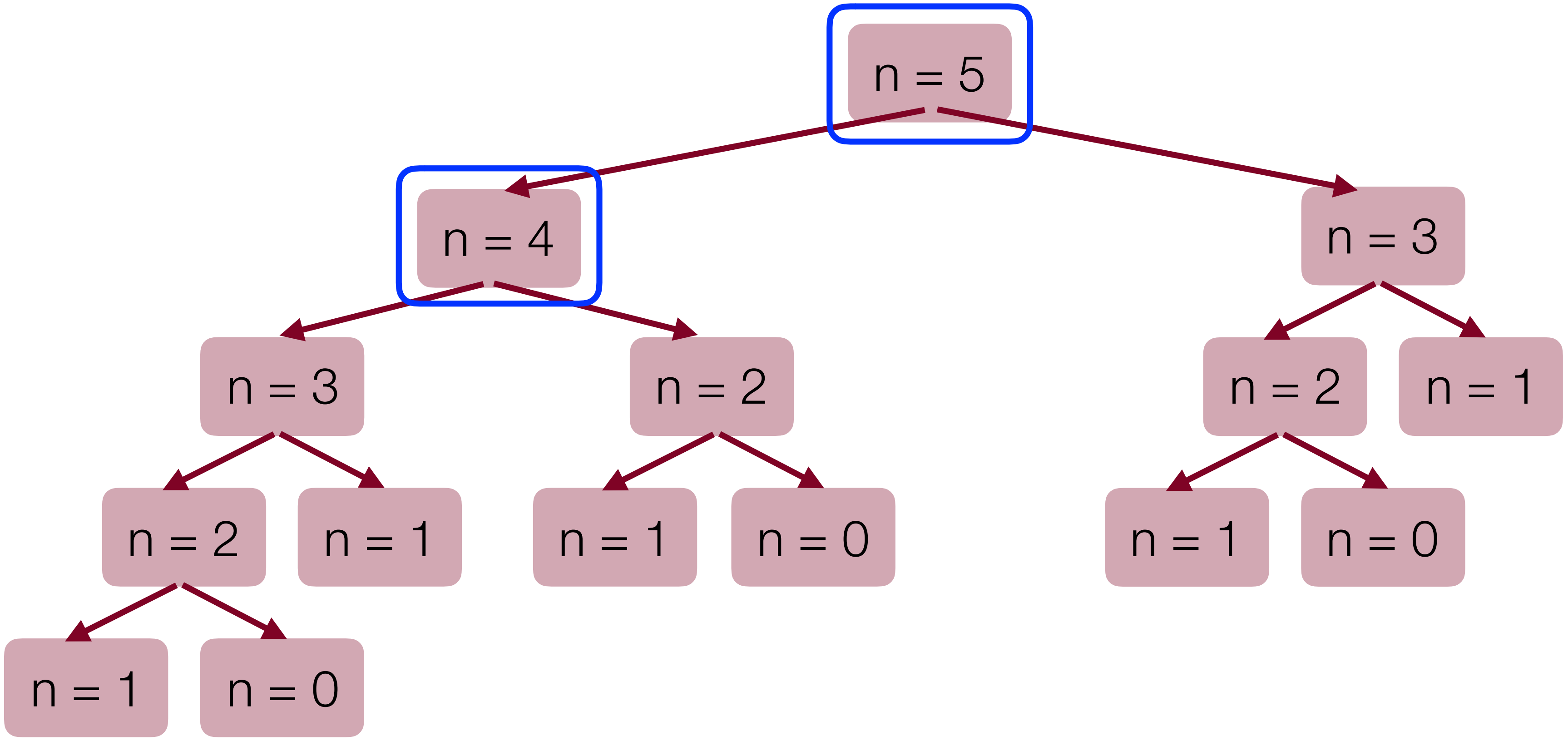
Remember what you have already done!

Memoization: Don't re-do unnecessary work!



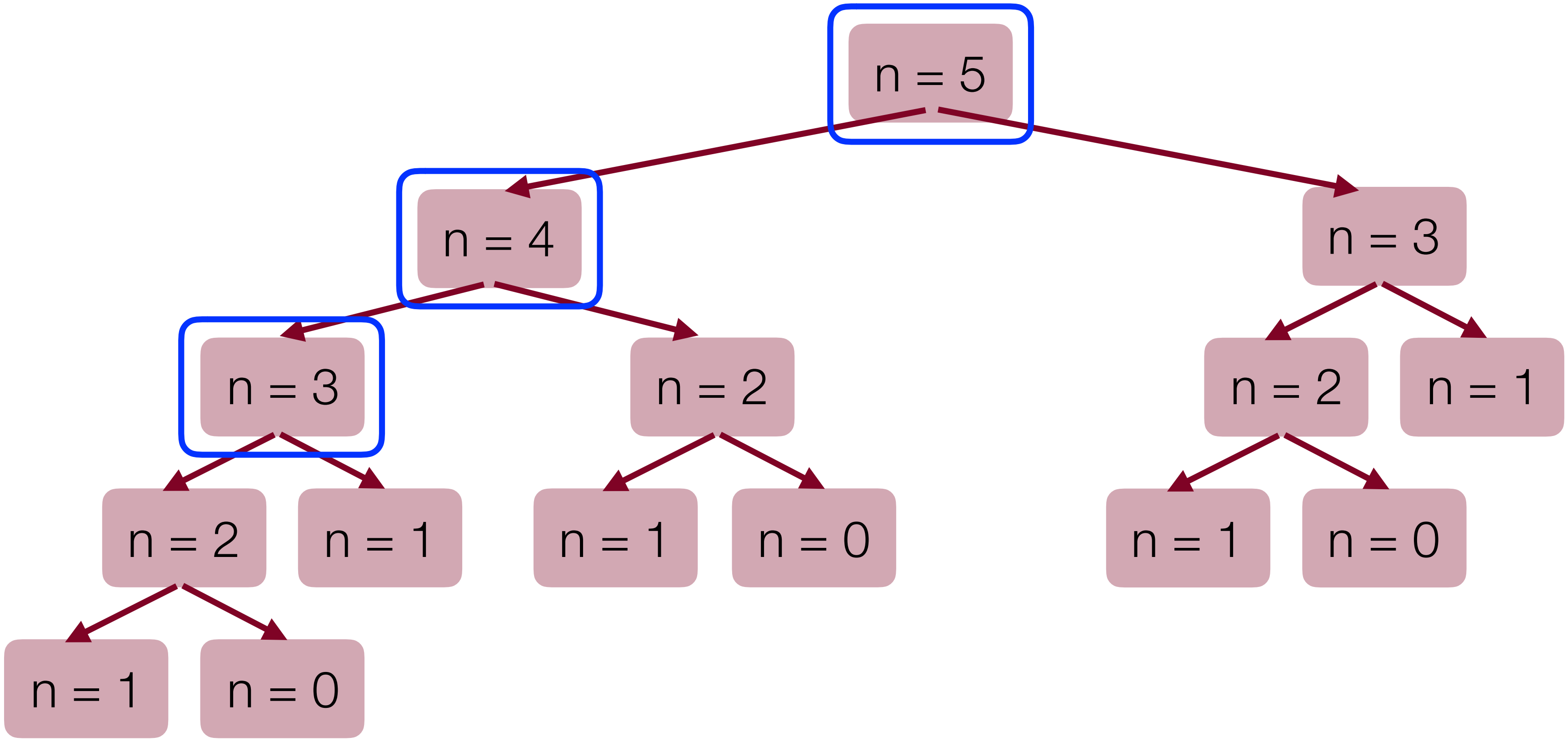
Cache: <empty>

Memoization: Don't re-do unnecessary work!



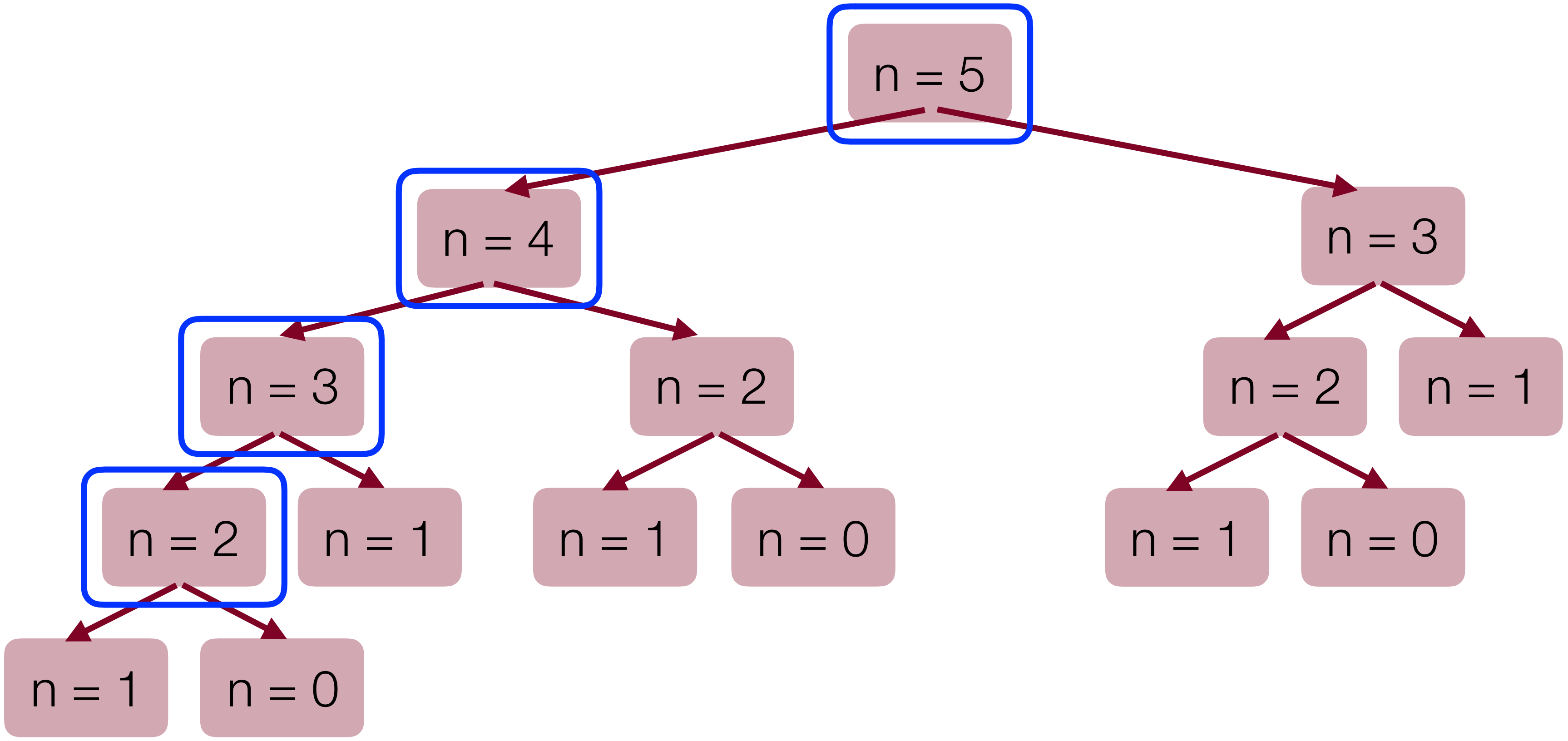
Cache: <empty>

Memoization: Don't re-do unnecessary work!



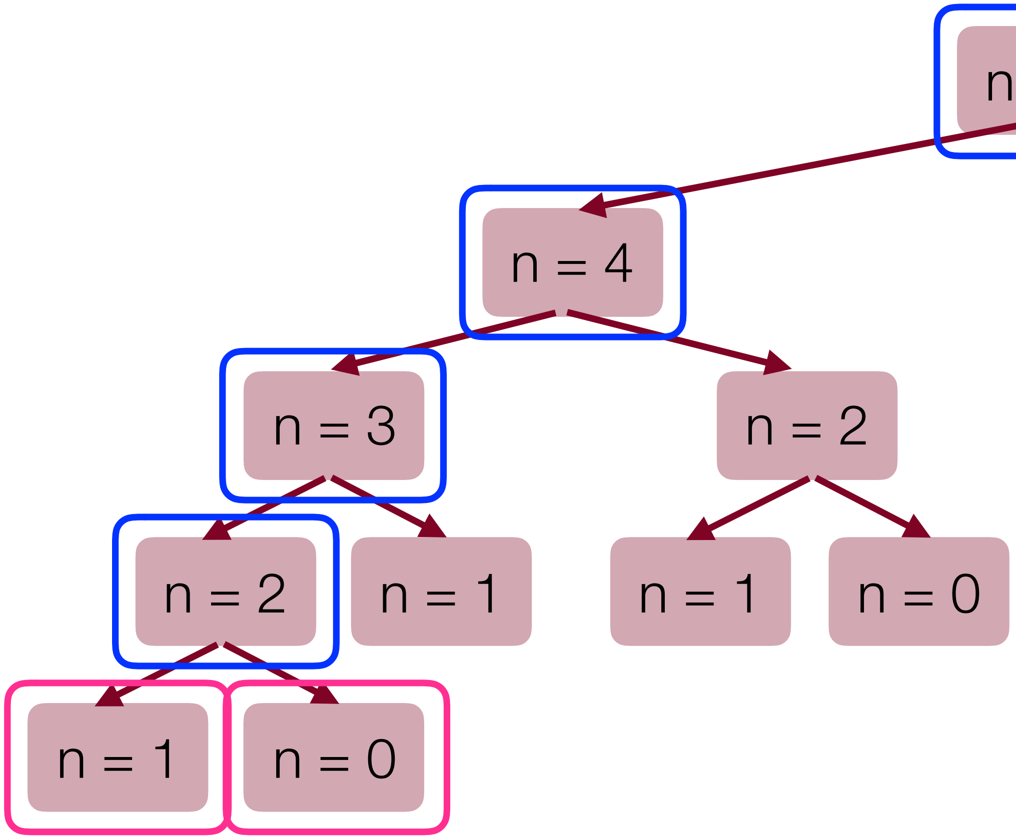
Cache: <empty>

Memoization: Don't re-do unnecessary work!



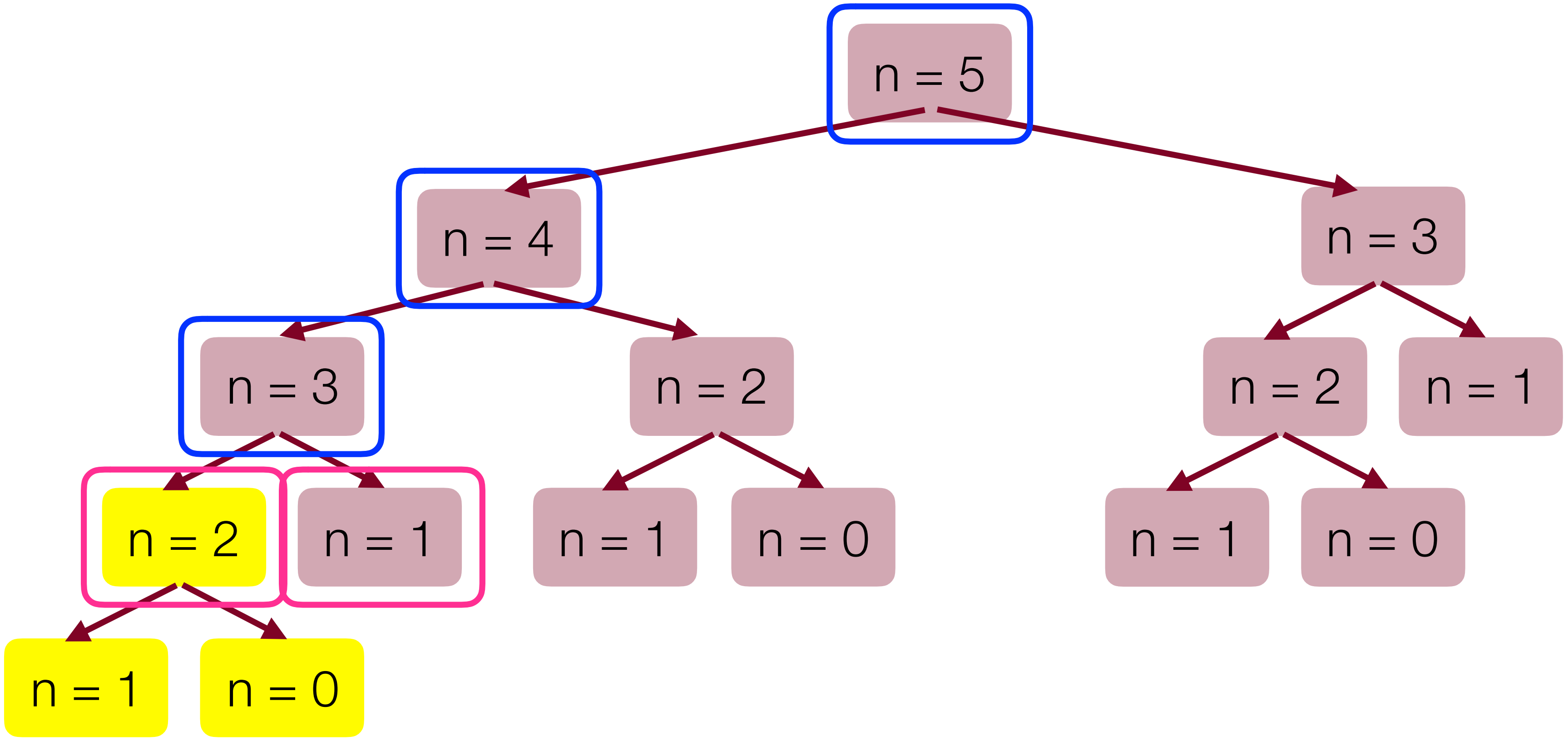
Cache: <empty>

Memoization: Don't re-do unnecessary work!



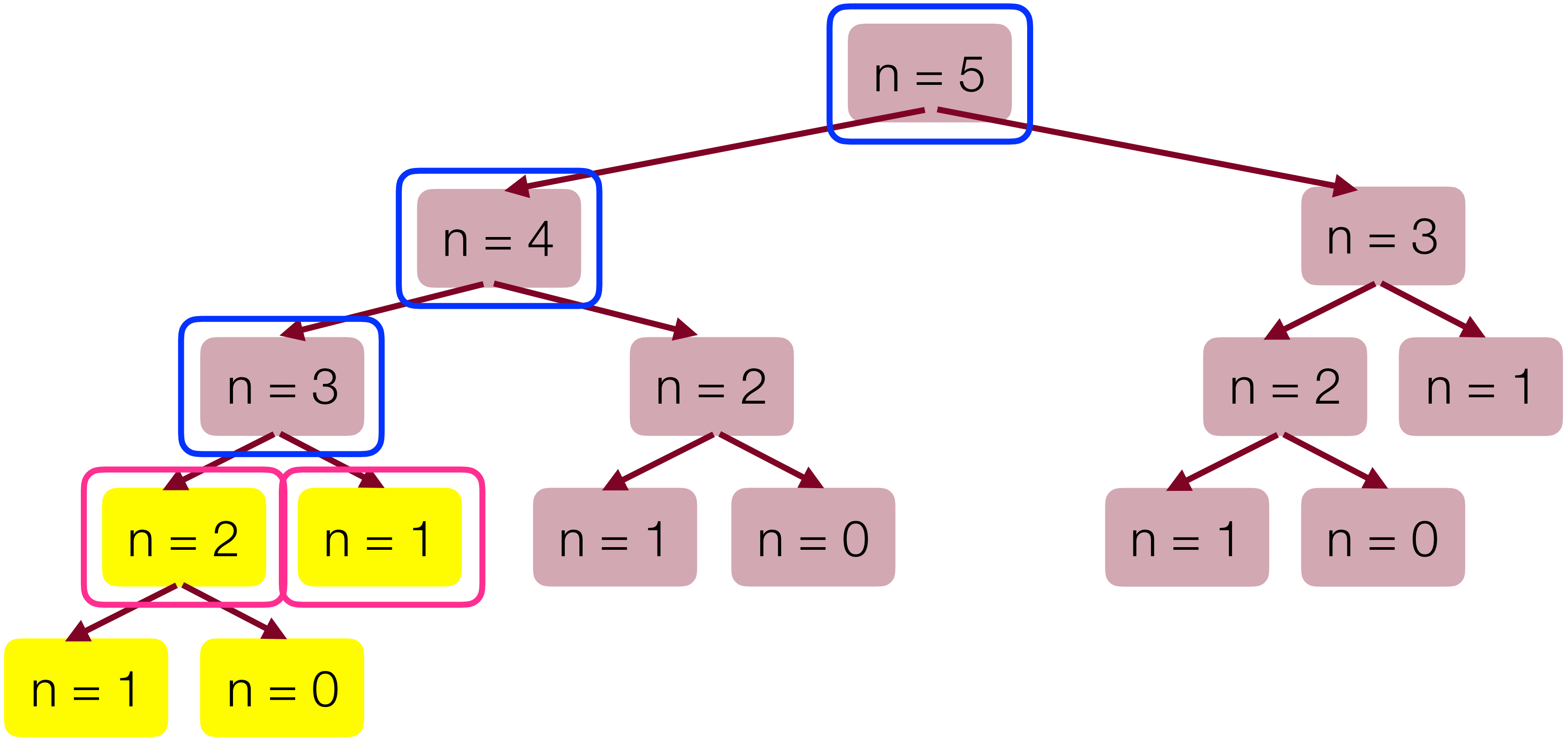
Cache: <empty>

Memoization: Don't re-do unnecessary work!



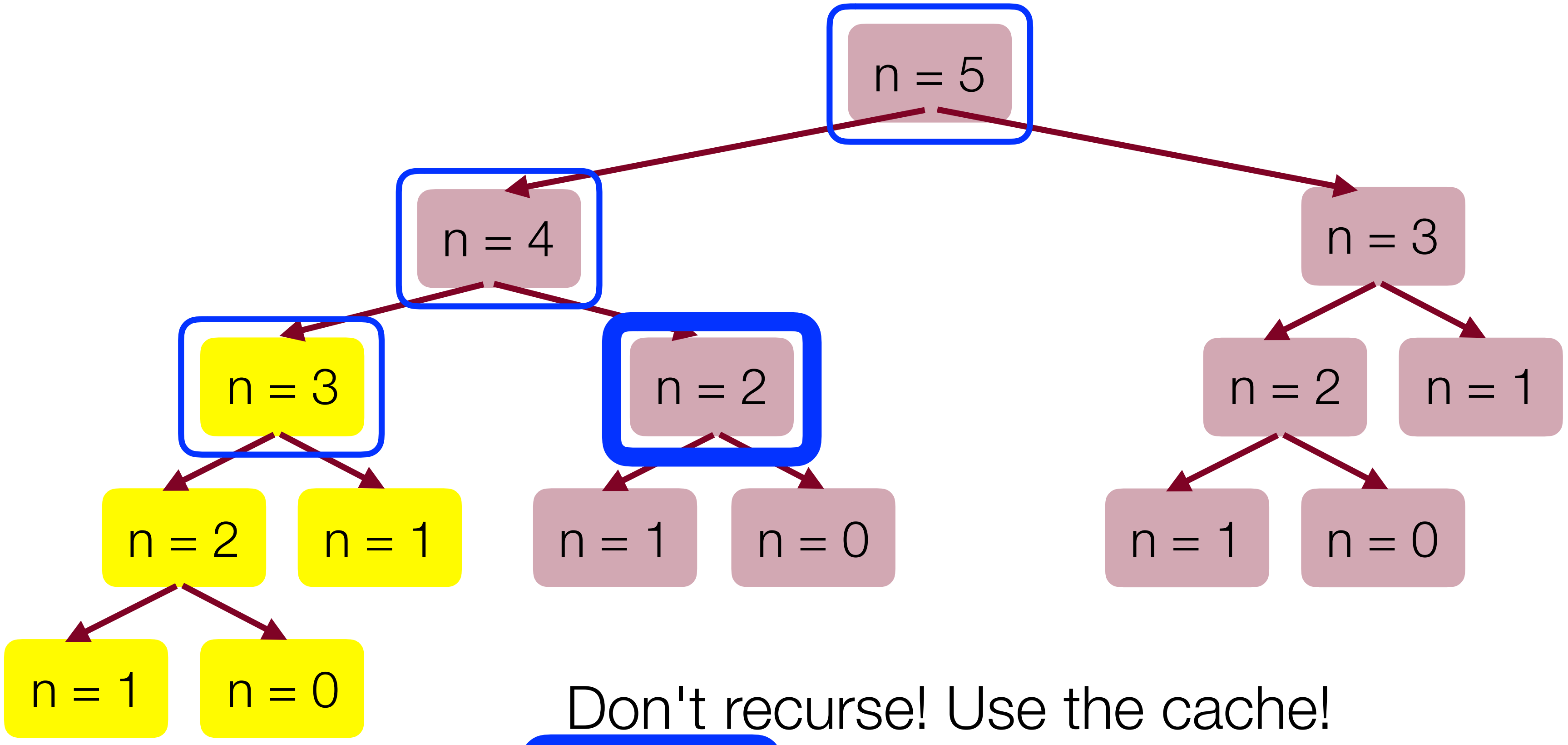
Cache: fib(2) = 1

Memoization: Don't re-do unnecessary work!



Cache: fib(2) = 1, fib(3) = 2

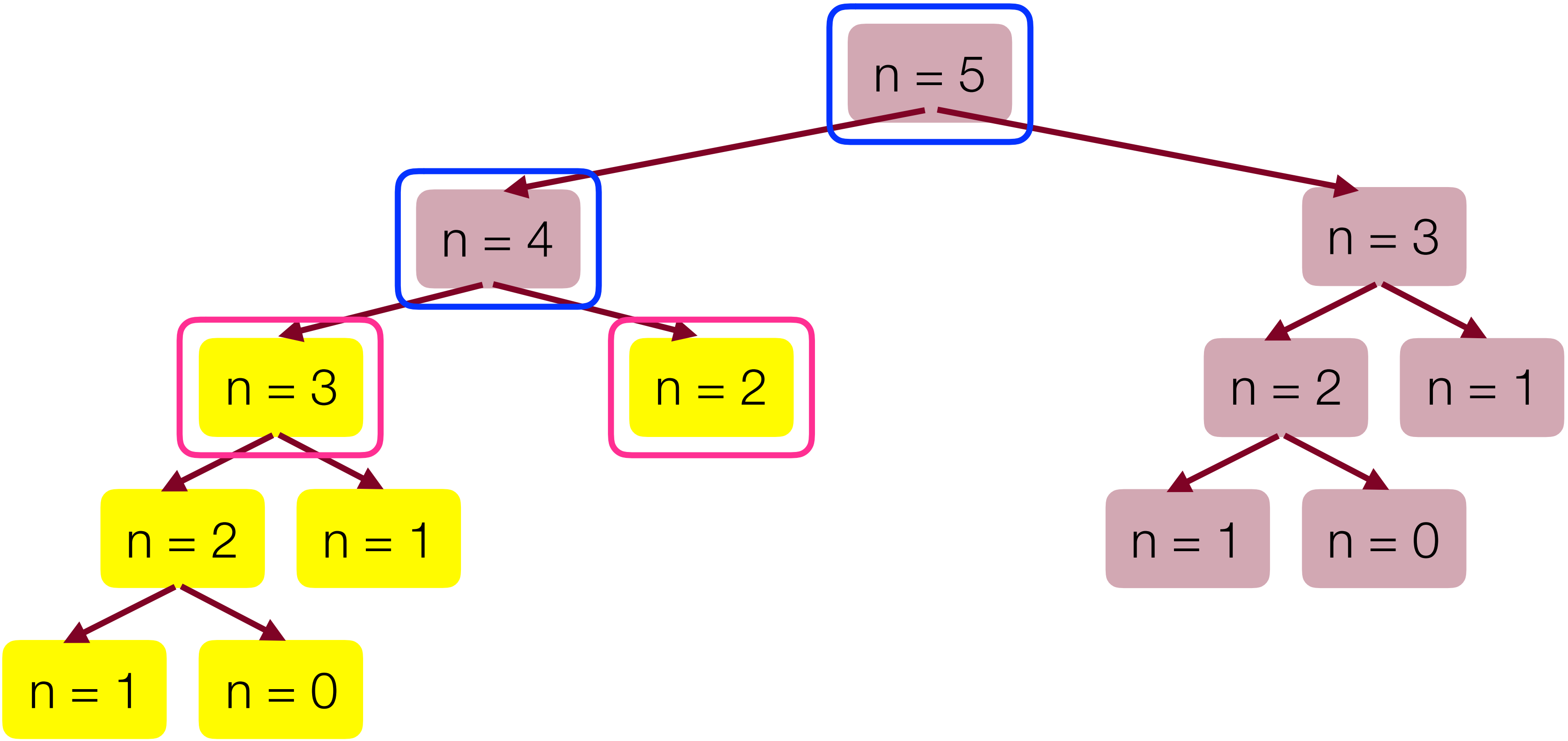
Memoization: Don't re-do unnecessary work!



Don't recurse! Use the cache!

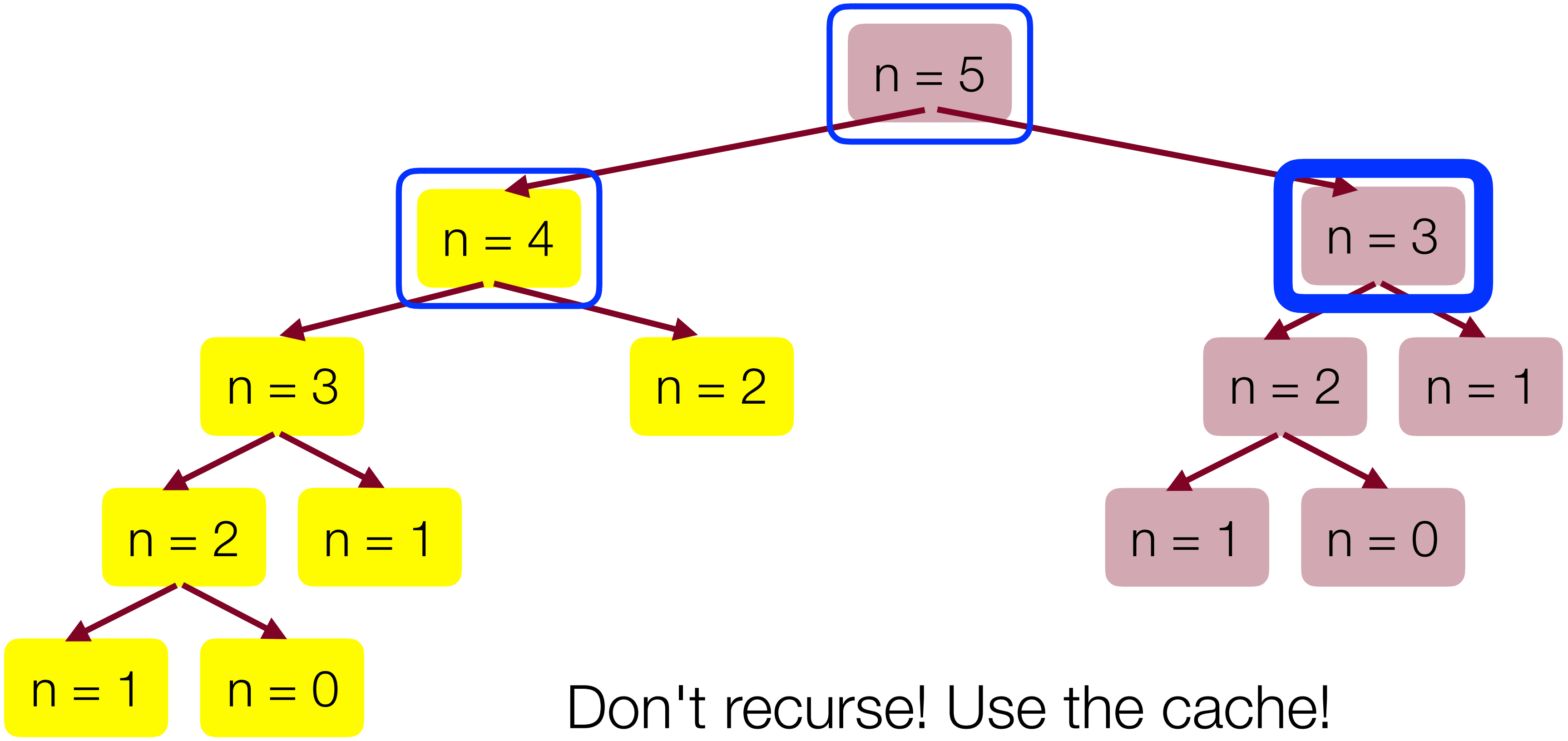
Cache: fib(2) = 1 fib(3) = 2

Memoization: Don't re-do unnecessary work!



Cache: fib(2) = 1, fib(3) = 2

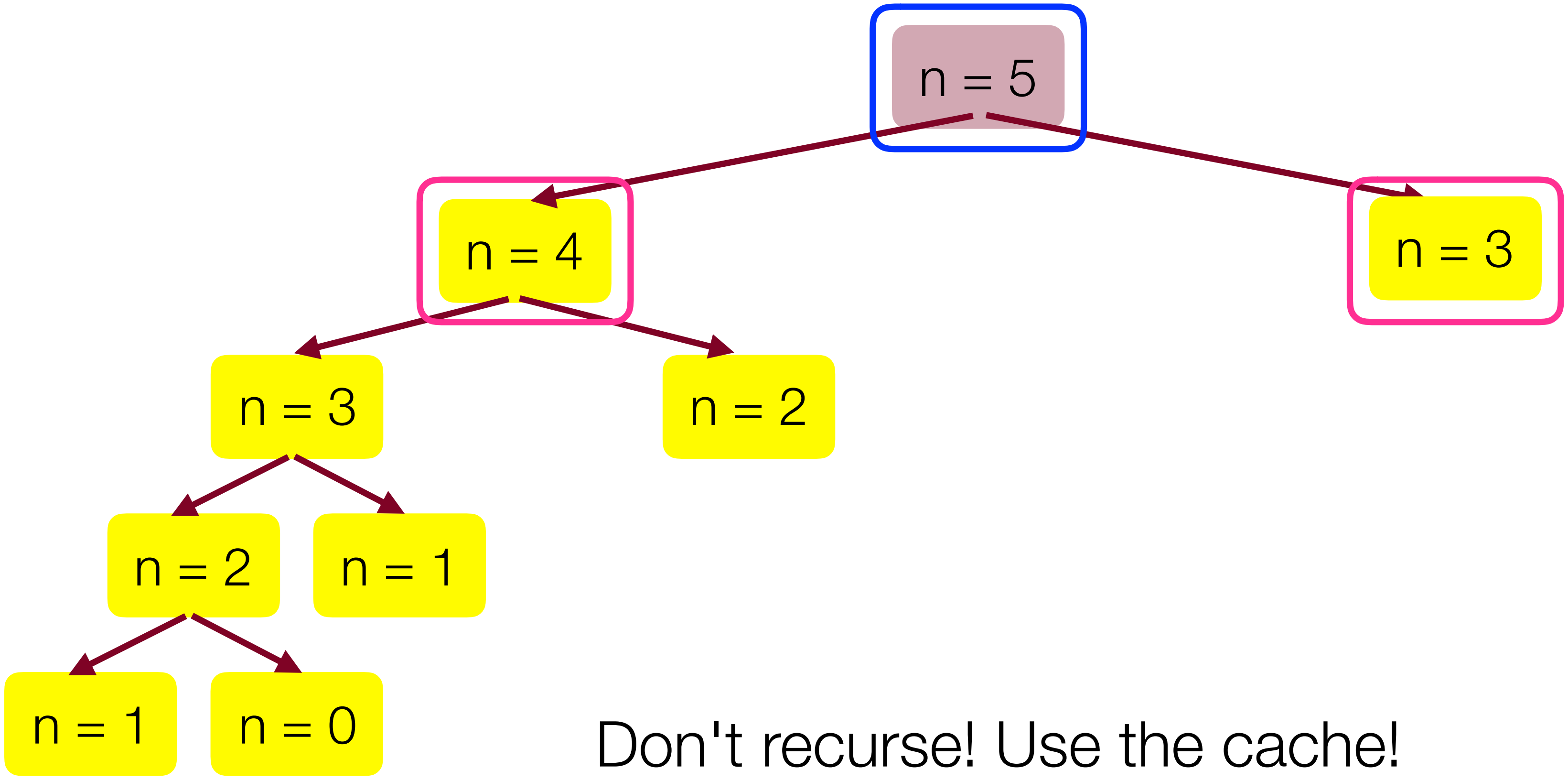
Memoization: Don't re-do unnecessary work!



Don't recurse! Use the cache!

Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3

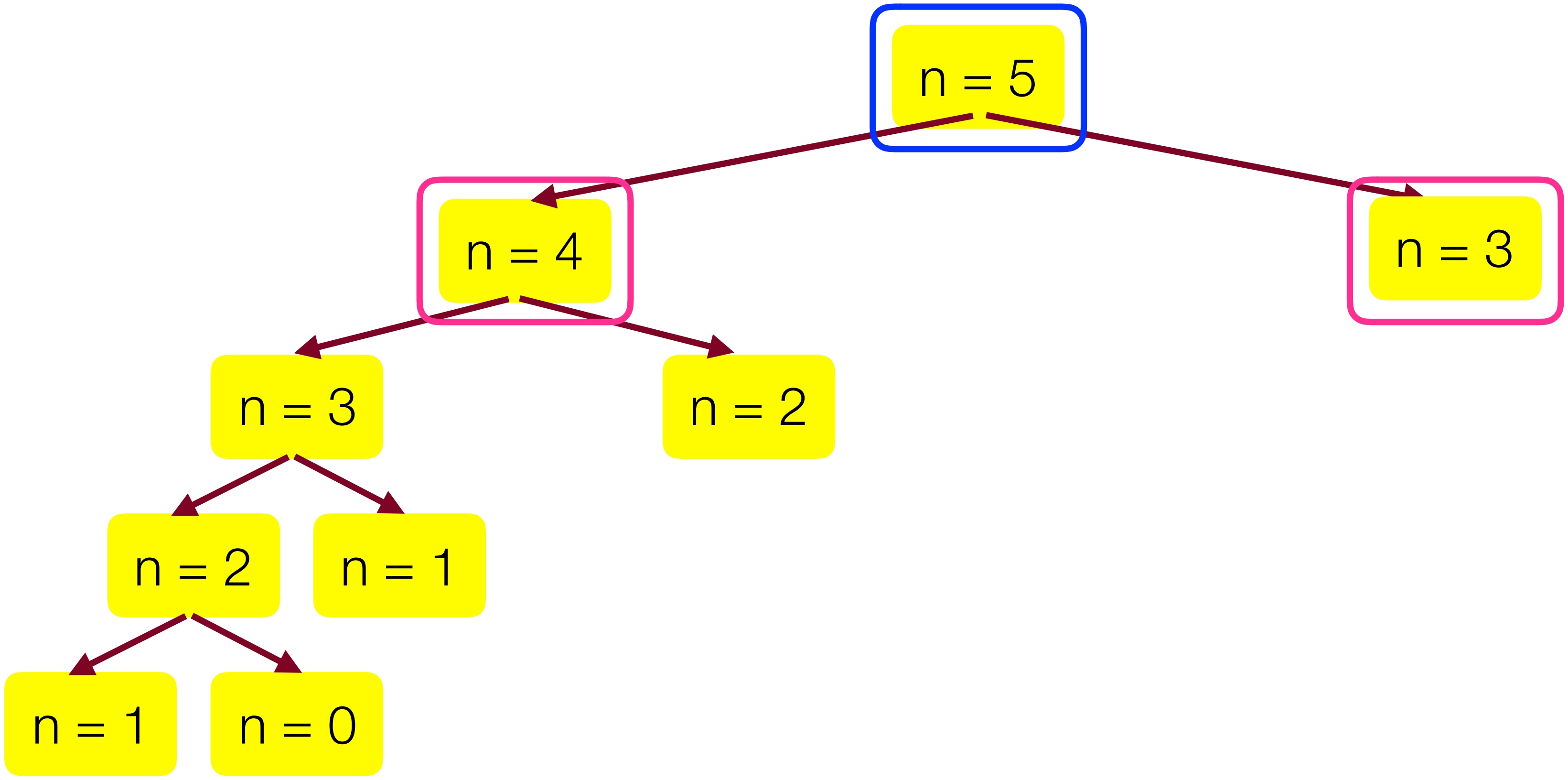
Memoization: Don't re-do unnecessary work!



Don't recurse! Use the cache!

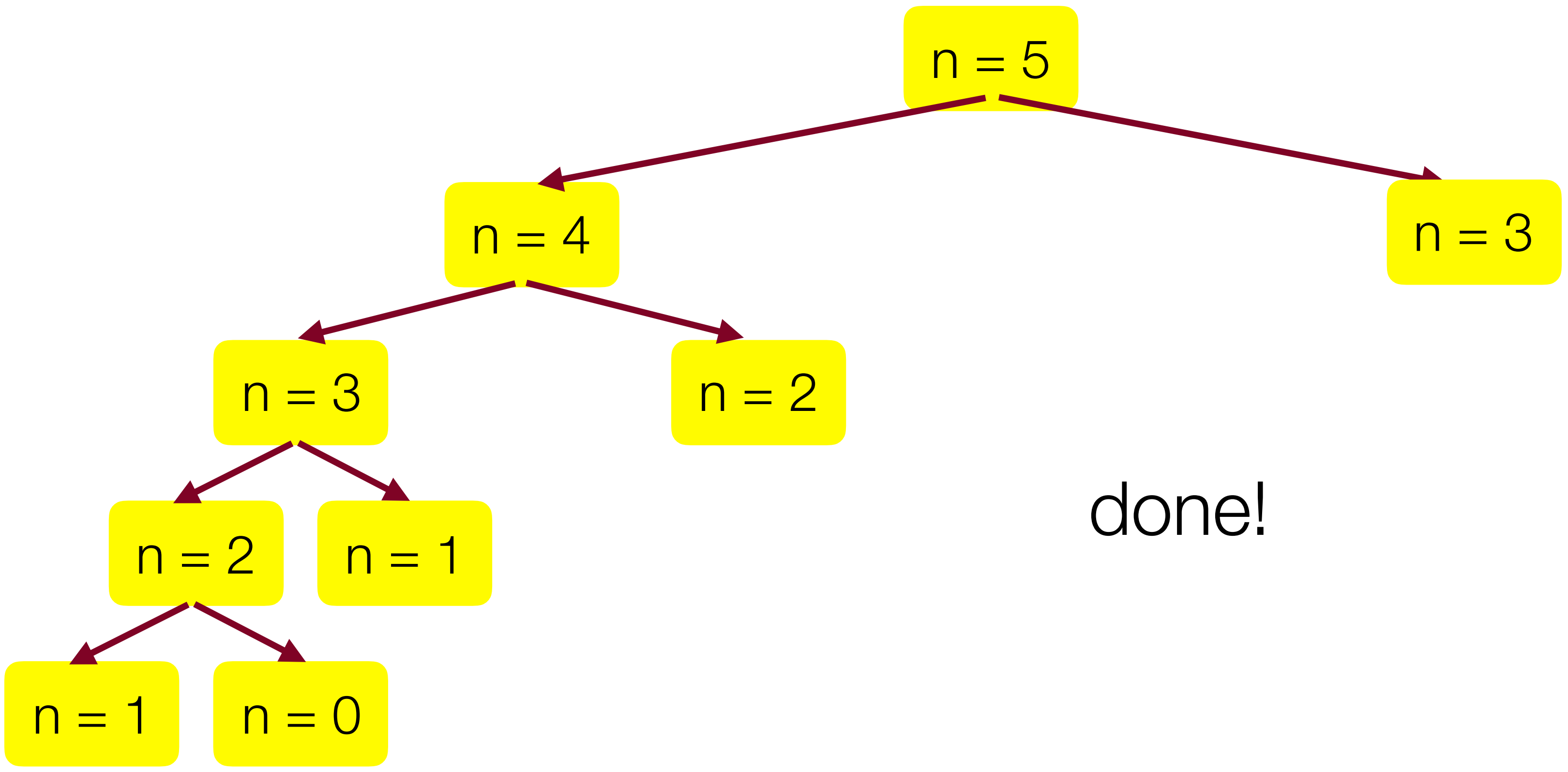
Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3

Memoization: Don't re-do unnecessary work!



Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3, fib(5) = 5

Memoization: Don't re-do unnecessary work!



done!

Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3, fib(5) = 5

Memoization: Don't re-do unnecessary work!

```
long memoizationFib(int n) {
    Map<int, long> cache;
    return memoizationFib(cache, n);
}
```

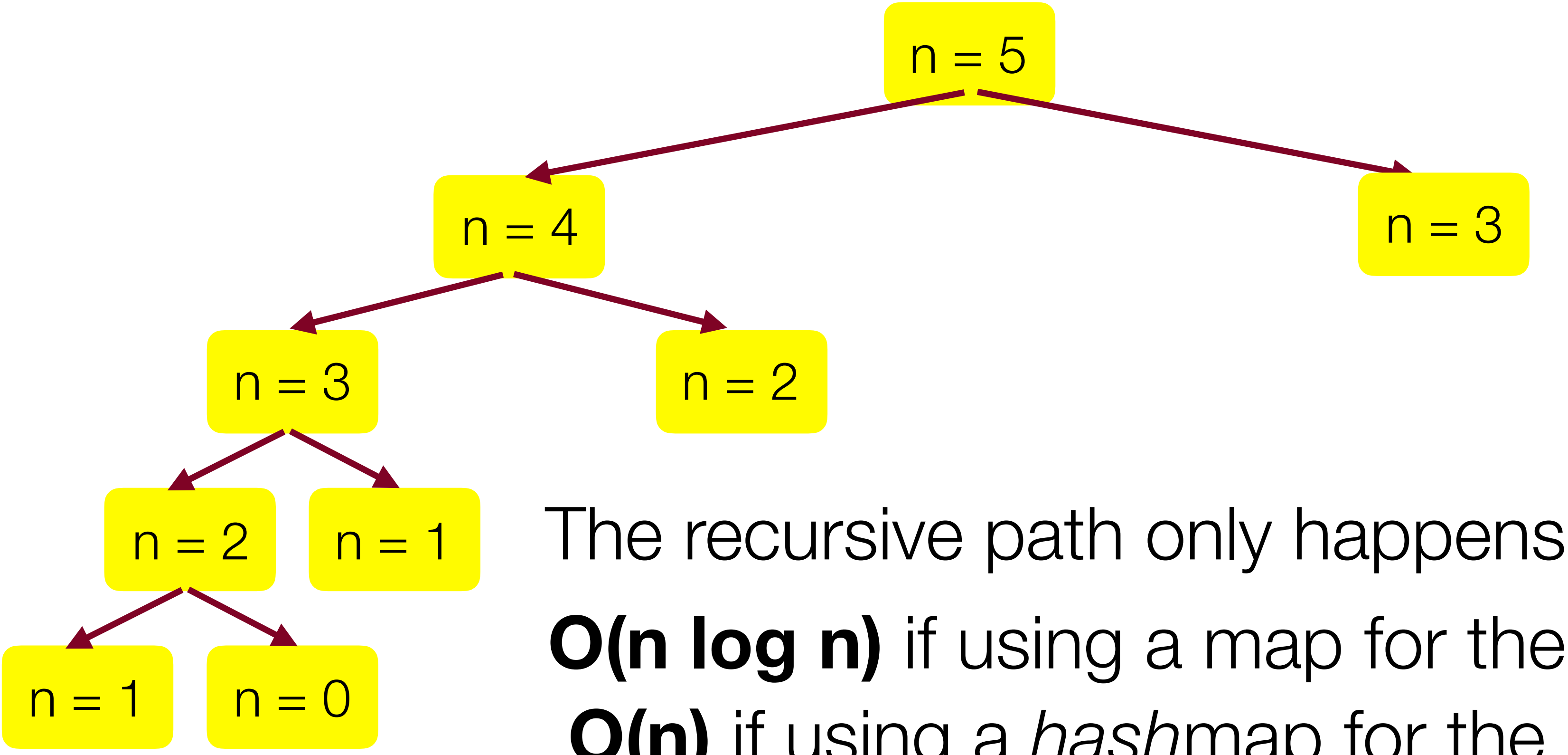
setup for helper function

Memoization: Don't re-do unnecessary work!

```
long memoizationFib(int n) {
    Map<int, long> cache;
    return memoizationFib(cache, n);
}

long memoizationFib(Map<int, long>&cache, int n) {
    if(n == 0) {
        // base case #1
        return 0;
    } else if (n == 1) {
        // base case #2
        return 1;
    } else if(cache.containsKey(n)) {
        // base case #3
        return cache[n];
    }
    // recursive case
    long result = memoizationFib(cache, n-1) + memoizationFib(cache, n-2);
    cache[n] = result;
    return result;
}
```

Memoization: Don't re-do unnecessary work!

Complexity?



The recursive path only happens on the left...

O(n log n) if using a map for the cache

O(n) if using a *hash*map for the cache

Fibonacci: the bigger picture

There are actually many ways to write a fibonacci function.

This is a case where the plain old iterative function works fine:

```
long iterativeFib(int n) {
    if(n == 0) {
       return 0;
    }
    long prev0 = 0;
    long prev1 = 1;
    for (int i=n; i >= 2; i--) {
        long temp = prev0 + prev1;
        prev0 = prev1;
        prev1 = temp;
    }
    return prev1;
}
```

Recursion is used often,
but not *always*.

Fibonacci: Okay, one more...

Another way to keep track of previously-computed values in fibonacci is through the use of a different helper function that simply passes along the previous values:

```
long passValuesRecursiveFib(int n) {
    if (n == 0) {
        return 0;
    }
    return passValuesRecursiveFib(n, 0, 1);
}

long passValuesRecursiveFib(int n, long p0, long p1) {
    if (n == 1) {
        // base case
        return p1;
    }
    return passValuesRecursiveFib(n-1, p1, p0 + p1);
}
```

More on Structs

We have mentioned structs already -- they are useful for keeping track of related data as one type, which can get used like any other type. You can think of a struct as the *Lunchable* of the C++ world.

```
struct Lunchable {
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};

// Vector of Lunchables
Vector<Lunchable> lunchableOrder;
```

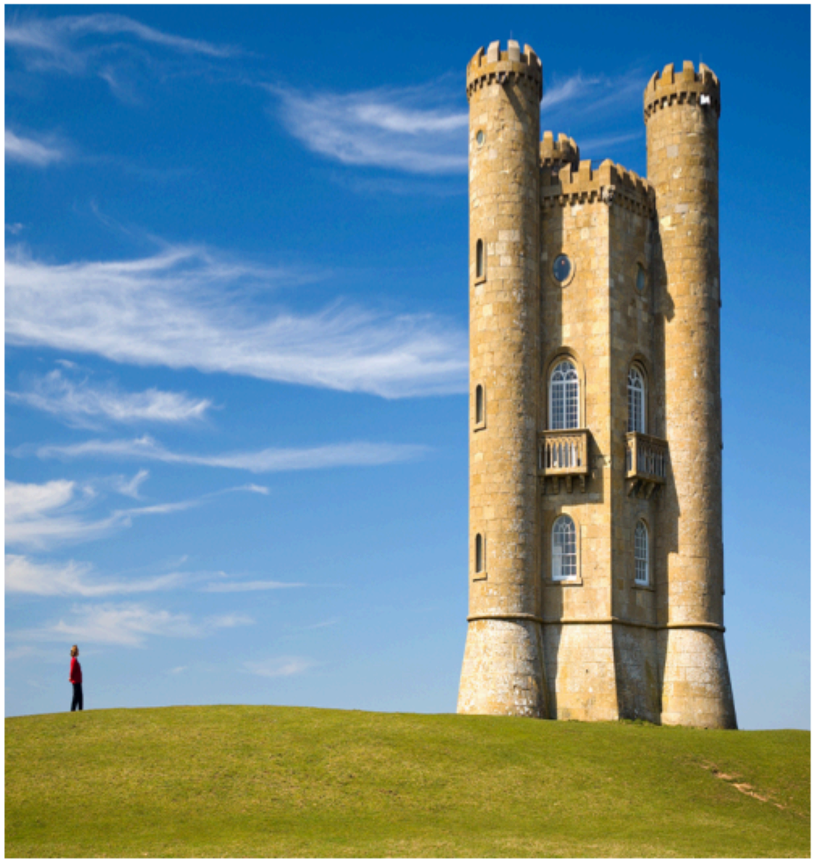
A Real Problem

Your cool picture from that trip to Europe doesn't fit on Instagram!

Bad Option #1: Crop

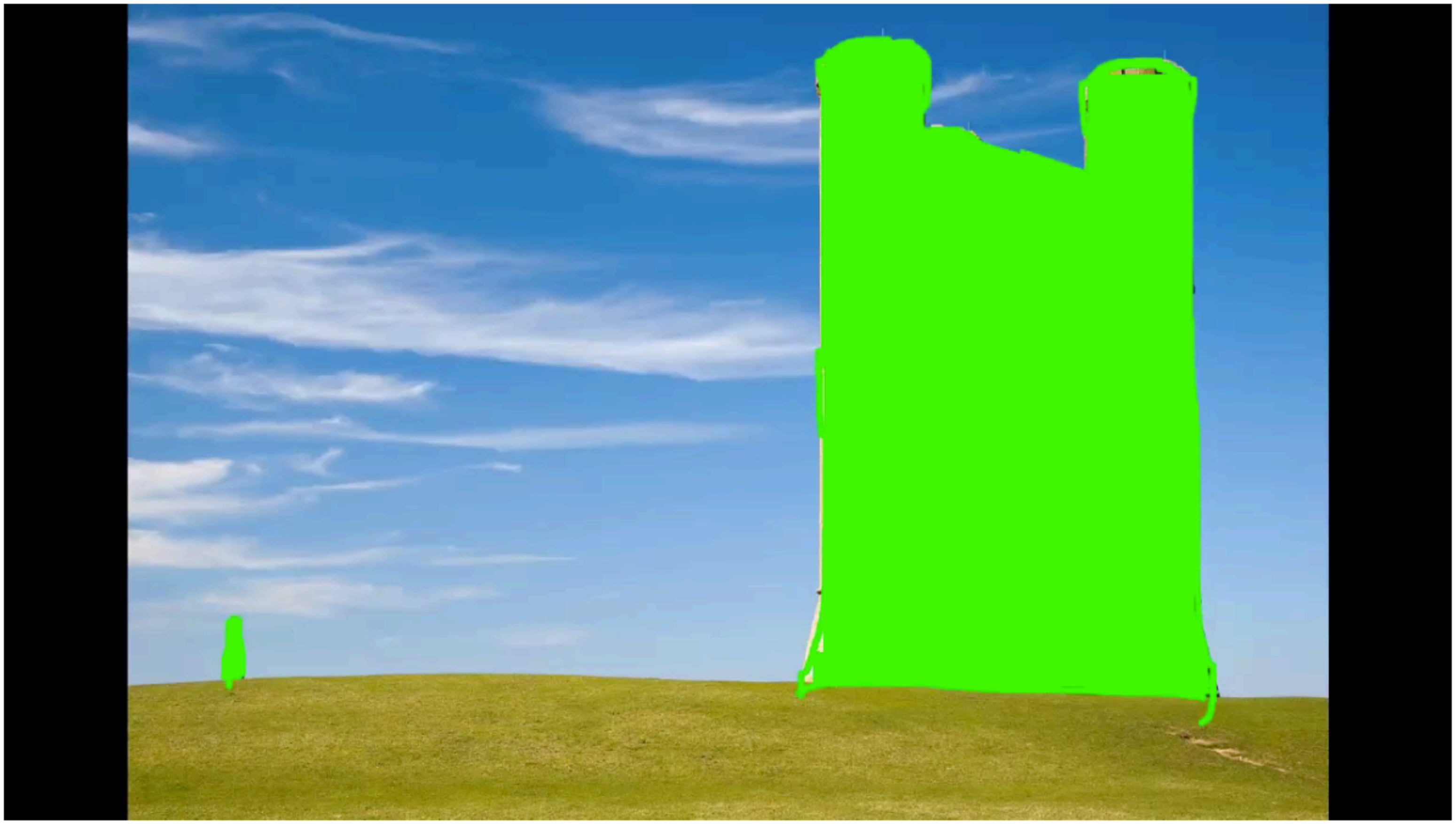
You got cropped out!

Bad Option #2: Resize

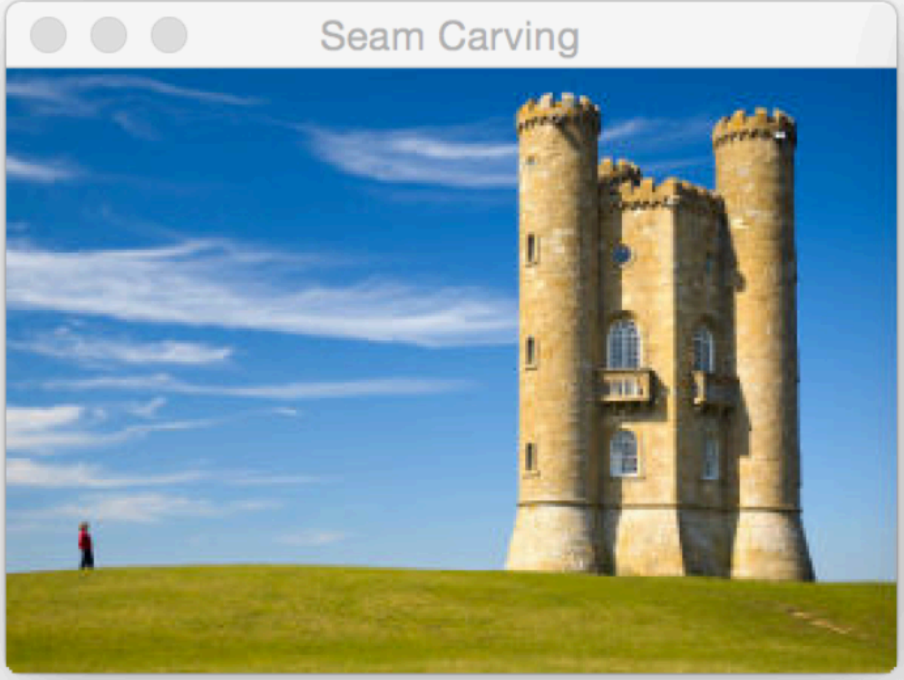


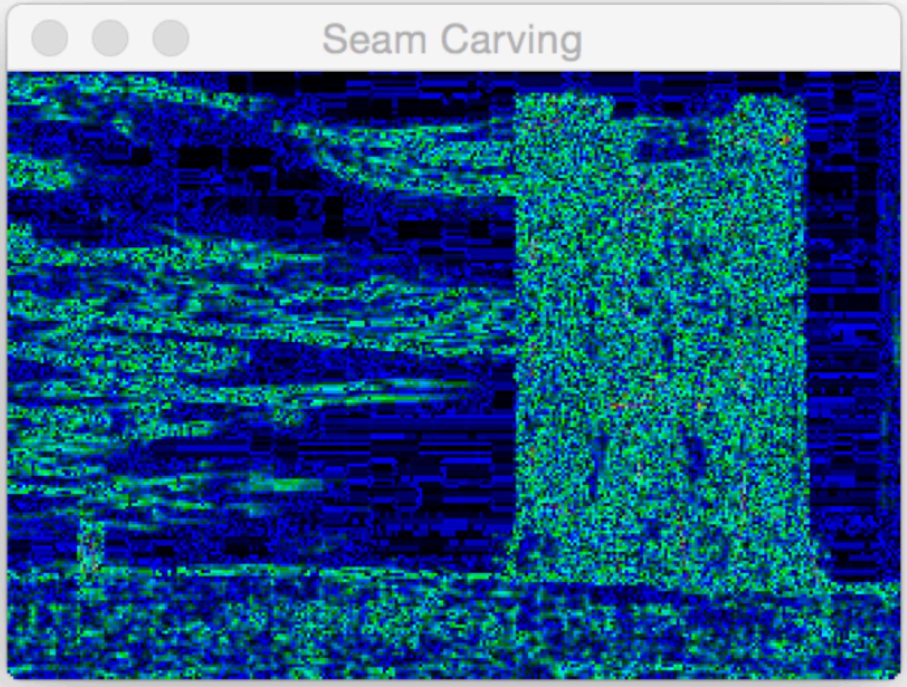
Stretchy castles look weird...

New Algorithm: Seam Carving!



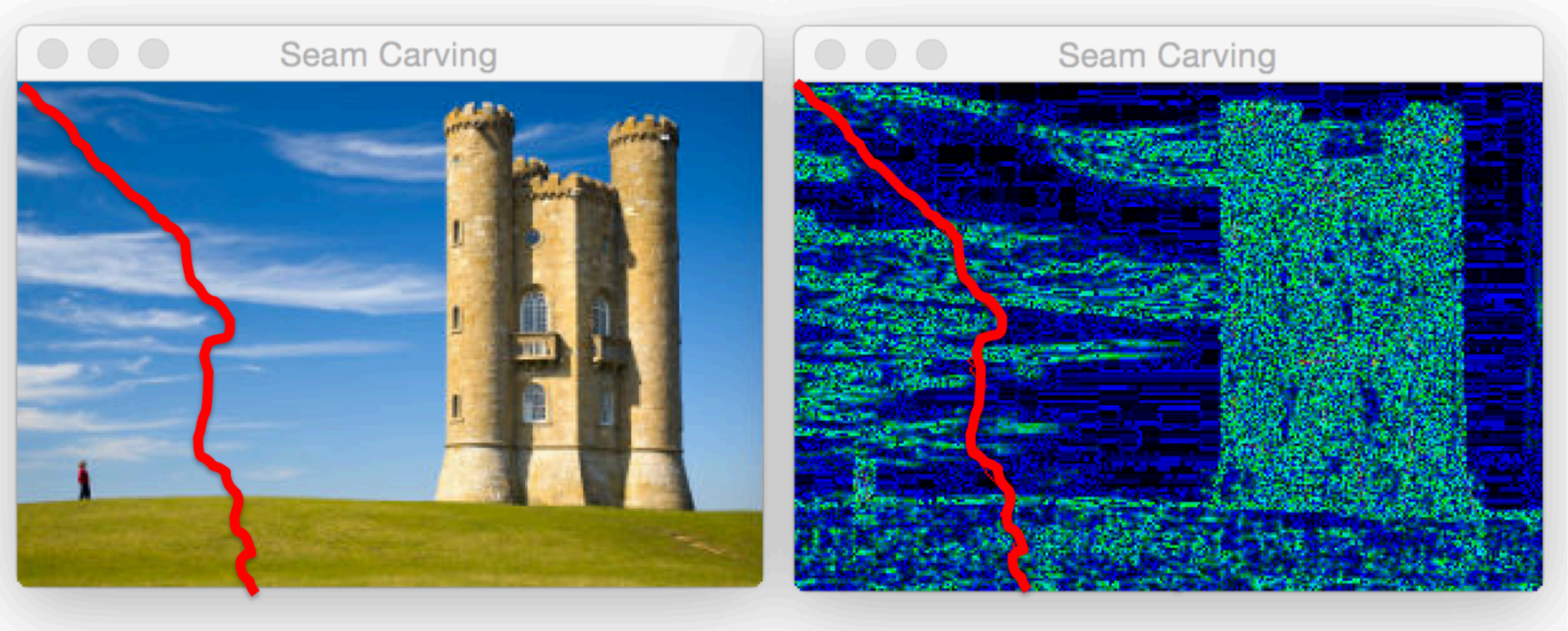
New Algorithm: Seam Carving!





How can you change an image without changing its aspect ration,
but while retaining the important information?

New Algorithm: Seam Carving!



We could delete an entire column of pixels, but we could also weave our way through a path of 1-pixel wide image that removes the least amount of stuff.

How to represent the path

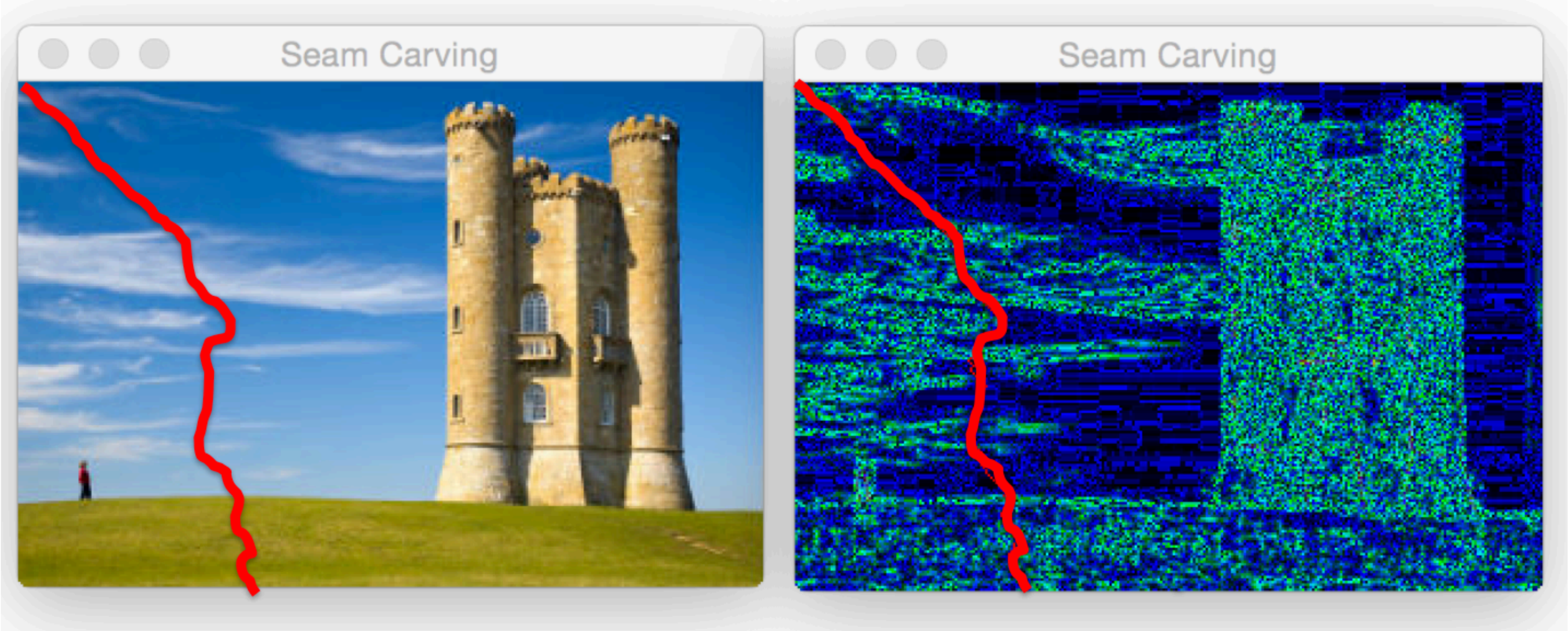
A struct!

```
struct Coord {
    int row;
    int col;
};
```

A path is just a Vector of coordinates:

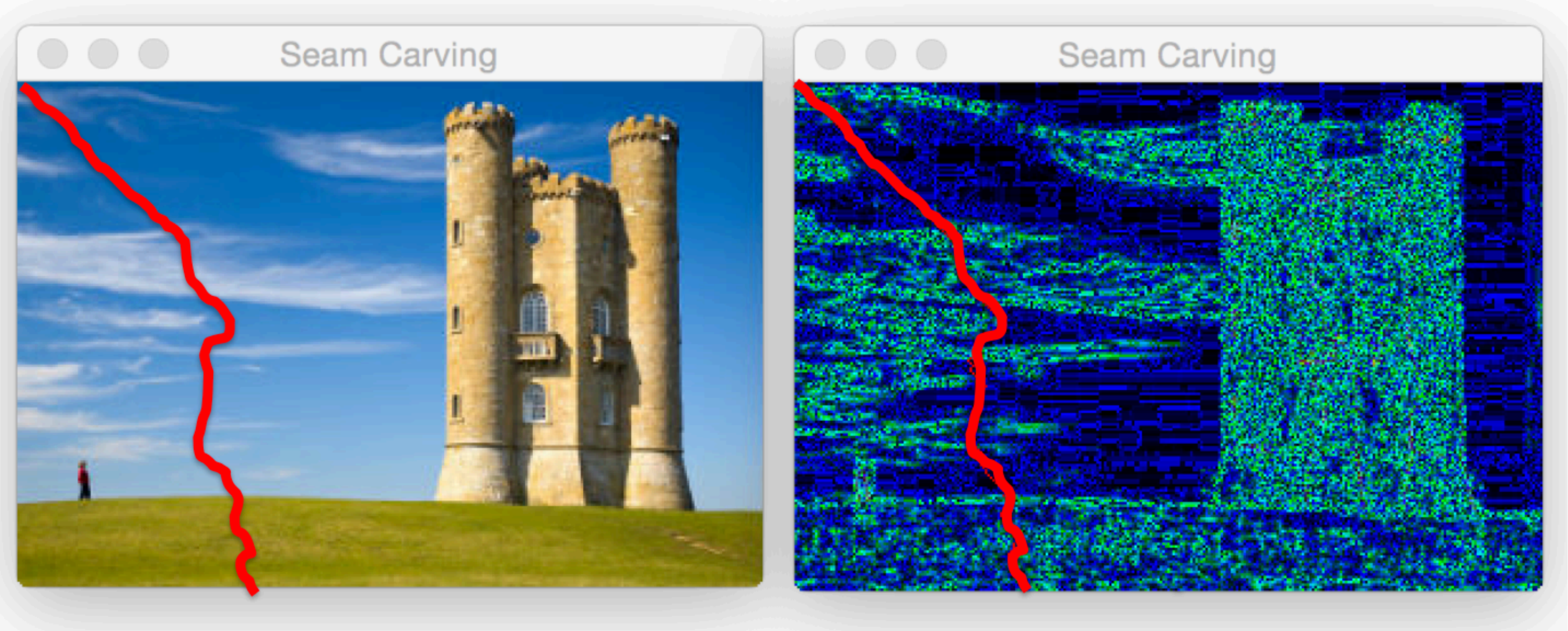
```
int main() {
    Coord myCord;
    myCoord.row = 5;
    myCoord.col = 7;
    cout << myCord.row << endl;
    Vector<Coord> path;
    return 0;
}
```

New Algorithm: Seam Carving!



Important pixels are ones that are considerably different from their neighbors.

New Algorithm: Seam Carving!



Let's write a recursive algorithm that can find the seam that minimizes the sum of all the importances of the pixels.

New Algorithm: Seam Carving!

```
Vector<Coord> getSeam(Grid<double> &weight, Coord curr);
```

References and Advanced Reading

- **References:**
  - https://en.wikipedia.org/wiki/Fibonacci_number
  - https://en.wikipedia.org/wiki/Seam_carving

Extra Slides