Memoization

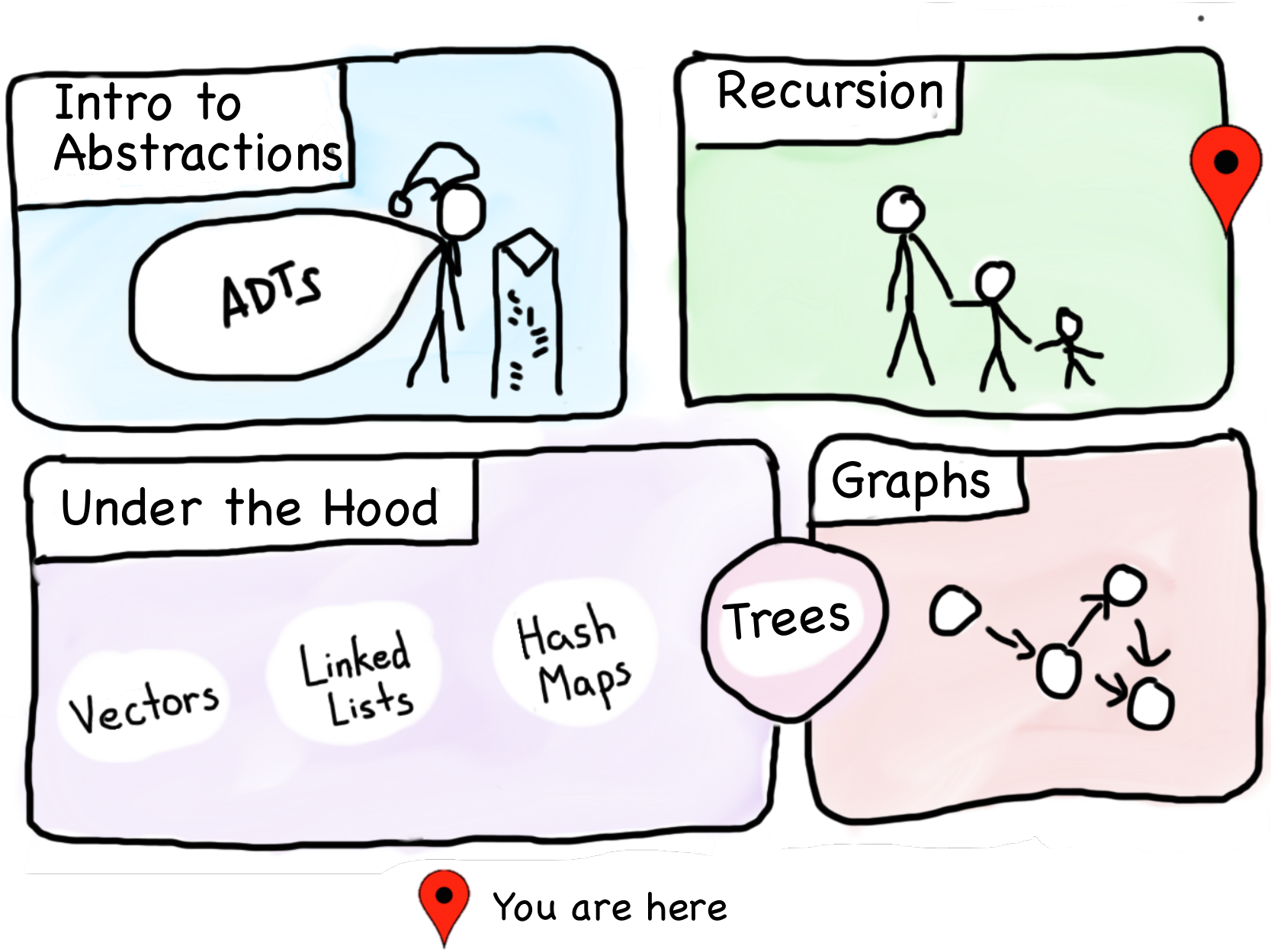
CS 106B

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

> Tell me and I forget. Teach me and I rememoize.*
>
> - Xun Kuang, 300 BCE

* This is almost the correct quote

Course Syllabus

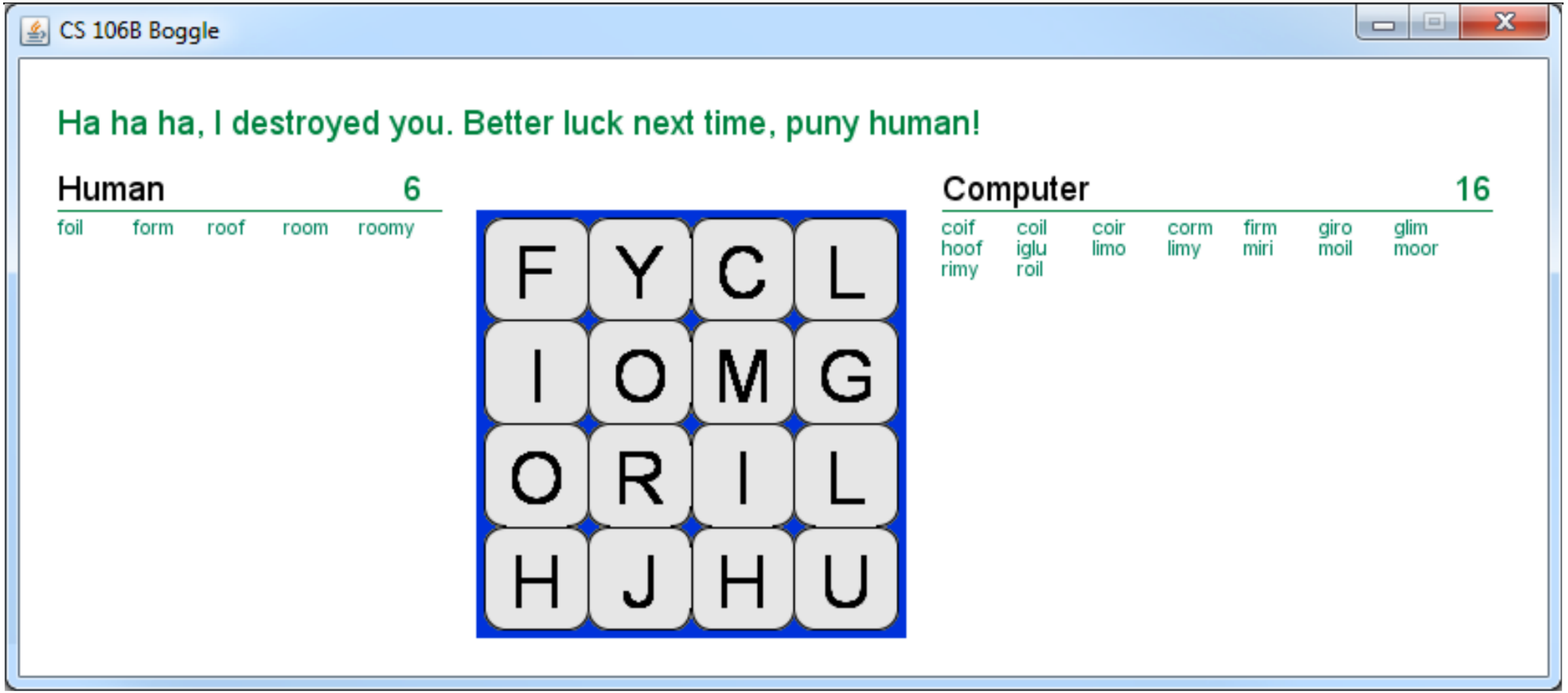


Midterm

Date: Tuesday, Nov 3rd
Time: 7-9pm
Location: Braun Aud + Cemex

- Hand written, four questions.
- Midterm Review: Next Monday (31st of Oct)
- First Practice Midterm on Wednesday
- Material up until Wednesday's class.
- Open book, closed CPU

Boggle

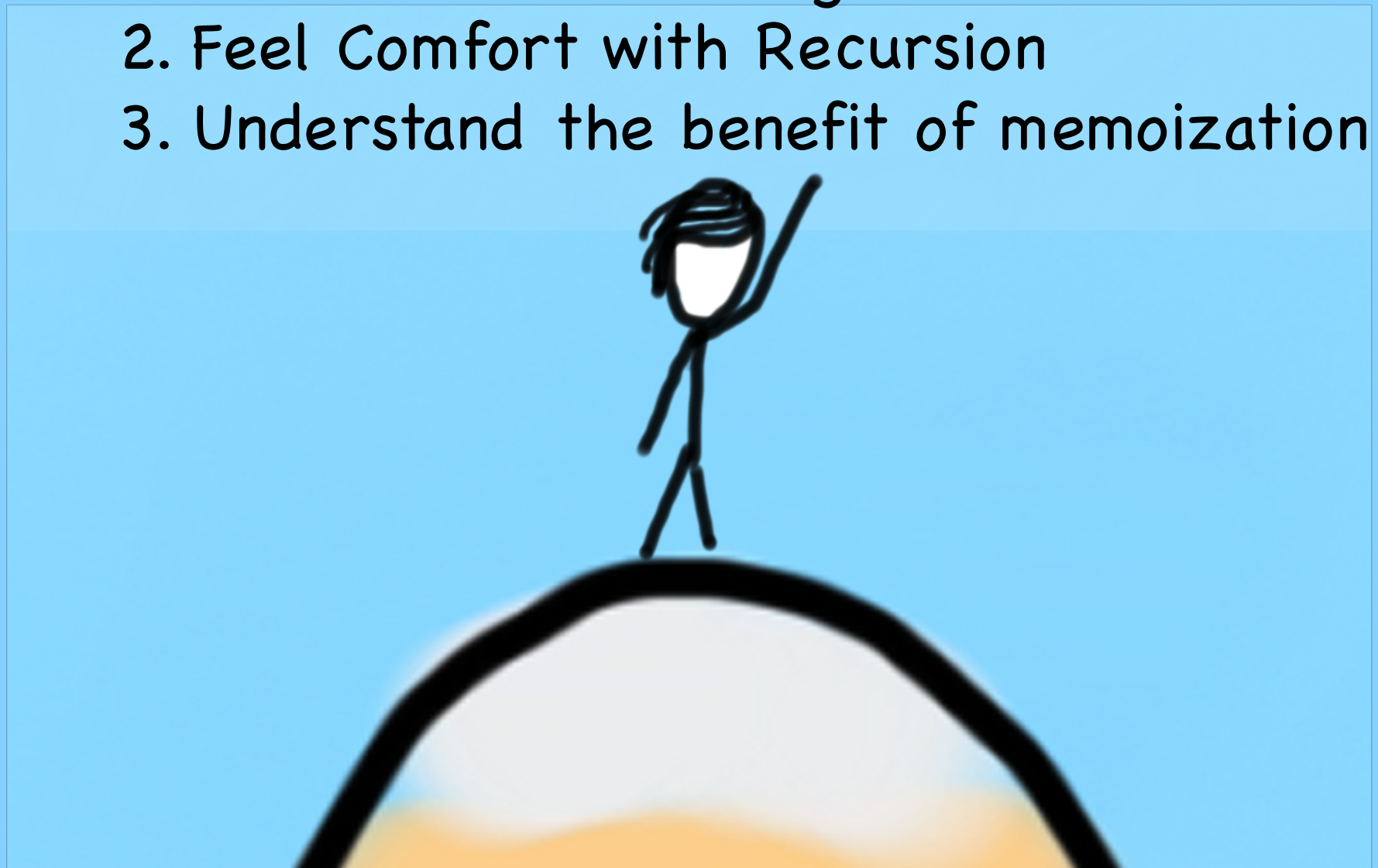

CS 106B Boggle
Ha ha ha, I destroyed you. Better luck next time, puny human!
Human 6
foil form roof room roomy
F Y C L
I O M G
O R I L
H J H U
Computer 16
coif coil coir corm firm giro glim
hoof iglu limo limy miri moil moor
rimy roil

Boggle

- Due in 2 weeks!
- YEAH @ 5pm
- Ready on Wednesday
- Midterm Practice

Today's Goal

1. Feel Comfort with Big O
2. Feel Comfort with Recursion
3. Understand the benefit of memoization



Real Life Problem

This doesn’t fit in instagram ☹

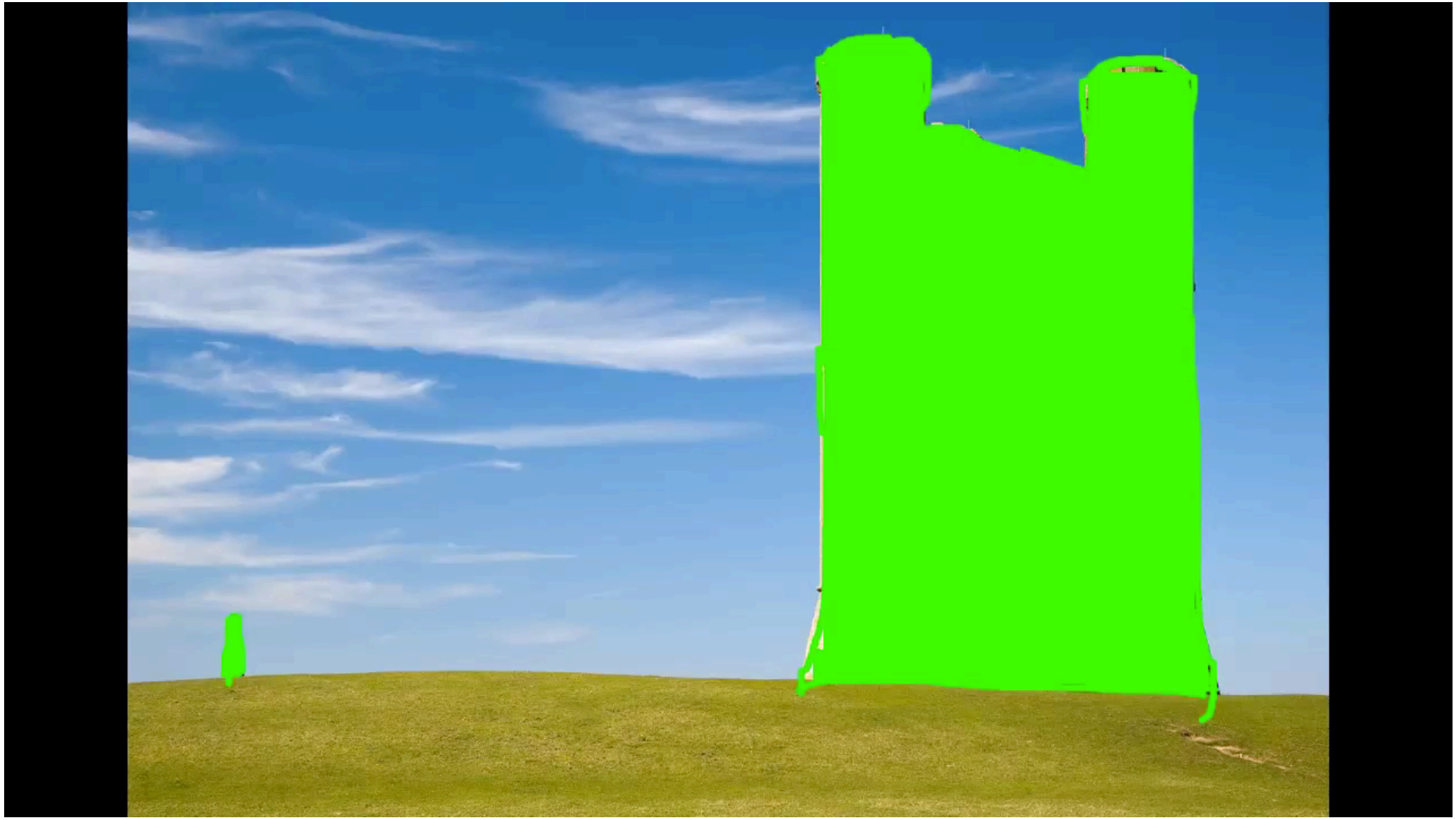
Bad Option 1: Crop

You got cropped out!

Bad Option 2: Resize

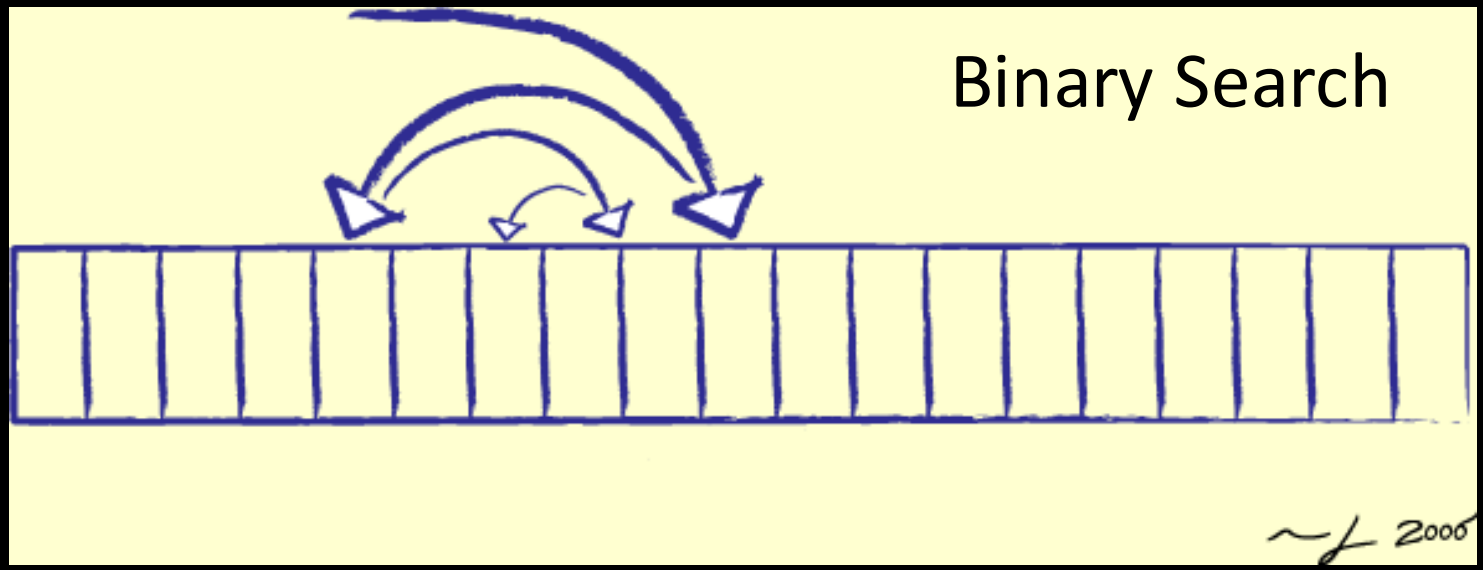
Weird looking castle ☹

New Algorithm: Seam Carving



Review

How do we compare algorithms?



Count # of Opperations

```
int find(Vector<int> & vec, int goal) {
  for(int i = 0; i < vec.size(); i++) {      2n + 1
    if(vec[i] == goal) return i;             2n
  }
  return -1;                                 1
}
```

$$T(n) = 4n + 2$$

Do we really care about the 4?

Do we really care about the +2?

Big O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 2 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$
  - $\log_2 n = \mathbf{O(\log n)}$

Big O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 2 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$
  - $\log_2 n = \mathbf{O(\log n)}$

Keep constants in the base or exponent of a power.

Big O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 2 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$
  - $\log_2 n = \mathbf{O(\log n)}$

Keep constants in the base or exponent of a power.

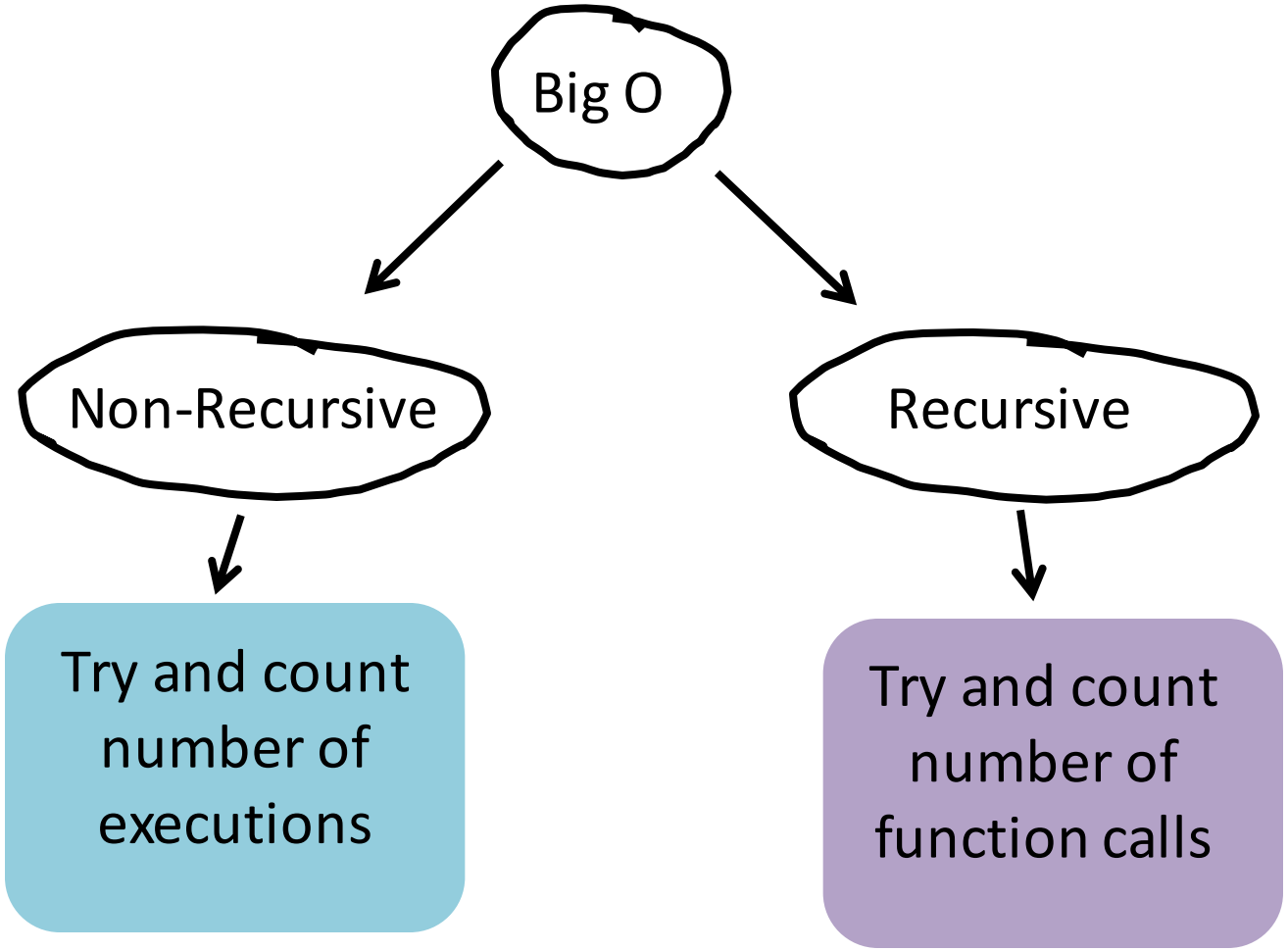
Do not keep constants in logs

$\log n$	n	$n \log n$	n^2	2^n
2	**4**	8	16	16
3	**8**	24	64	256
4	**16**	64	256	65,536
5	**32**	160	1,024	4,294,967,296
6	**64**	384	4,096	1.84×10^{19}
7	**128**	896	16,384	3.40×10^{38}
8	**256**	2,048	65,536	1.16×10^{77}
9	**512**	4,608	262,144	1.34×10^{154}
10	**1,024**	10,240 (.000003s)	1,048,576 (.0003s)	1.80×10^{308}
30	**1,300,000,000** (half a second)	39000000000 (13s)	1690000000000000000 (18 years)	LOL

# of Facebook accounts


THIS IS ME NOT CARING
ABOUT PERFORMANCE TUNING UNLESS IT CHANGES BIG-O
memegenerator.net

Big O Strategy

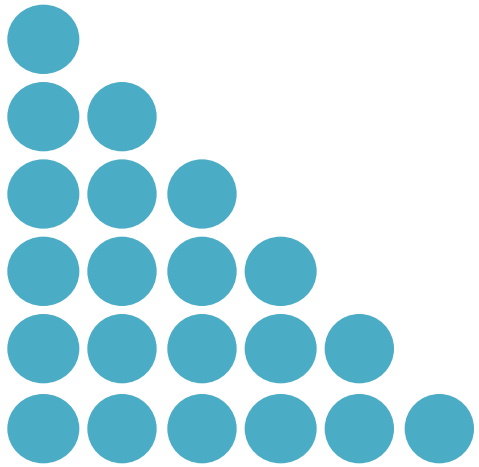


Pro Tip #1: Arithmetic Sum

Arithmetic Sum

- You can convince yourself that

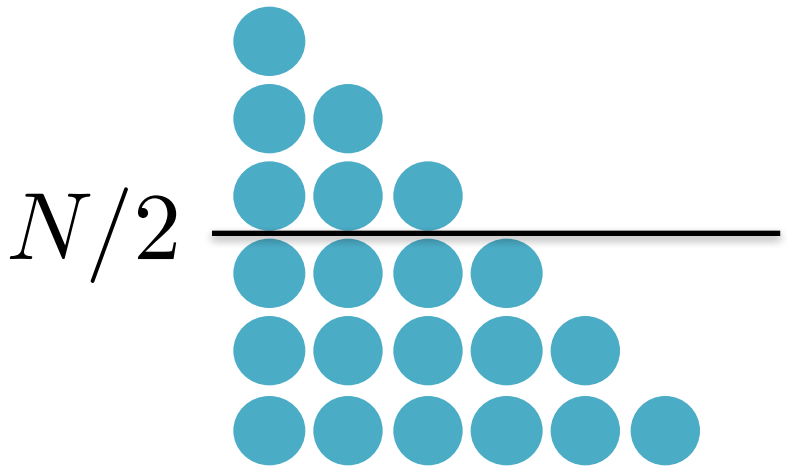
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

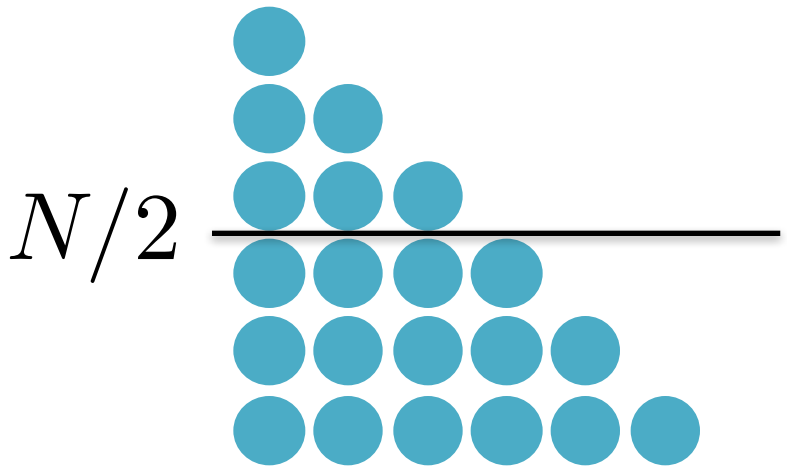
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

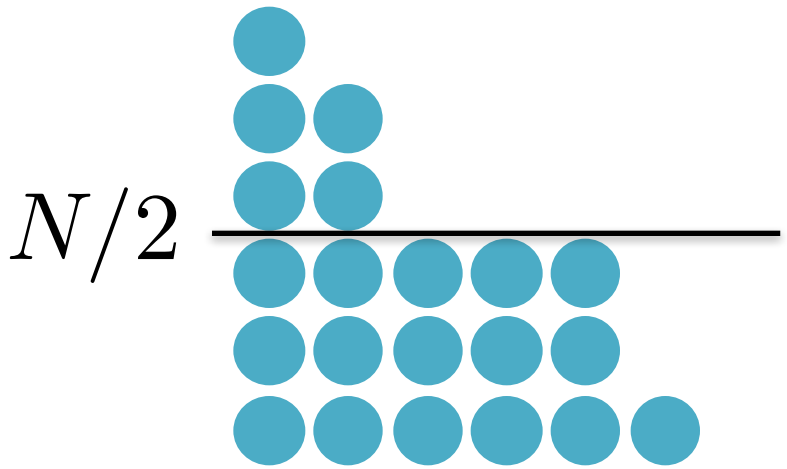
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

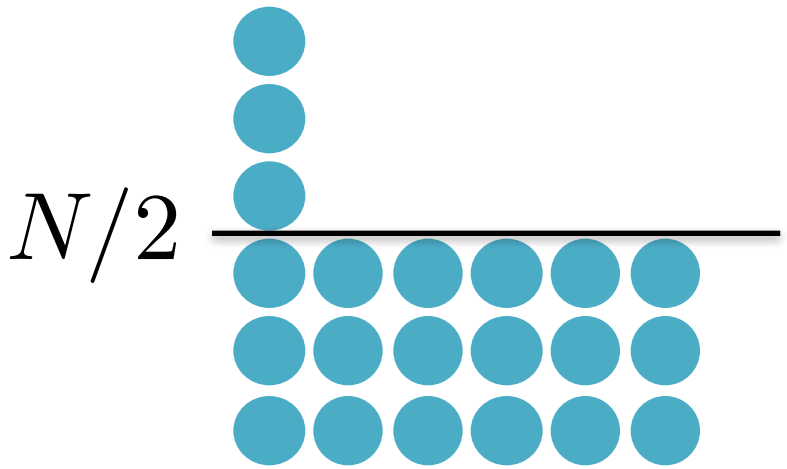
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

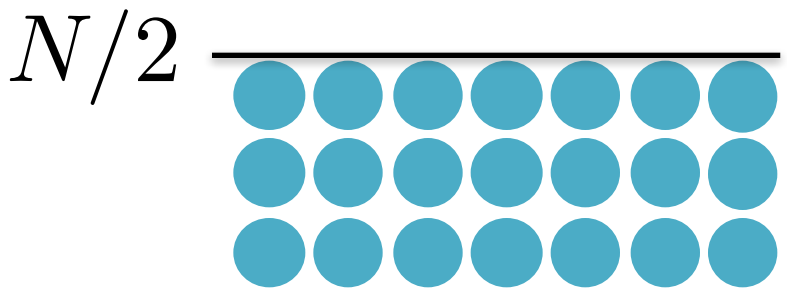
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

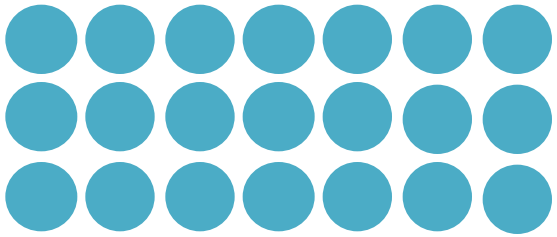
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

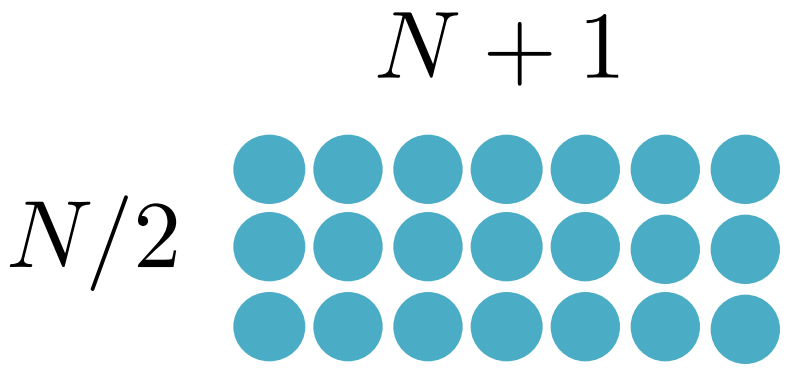
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

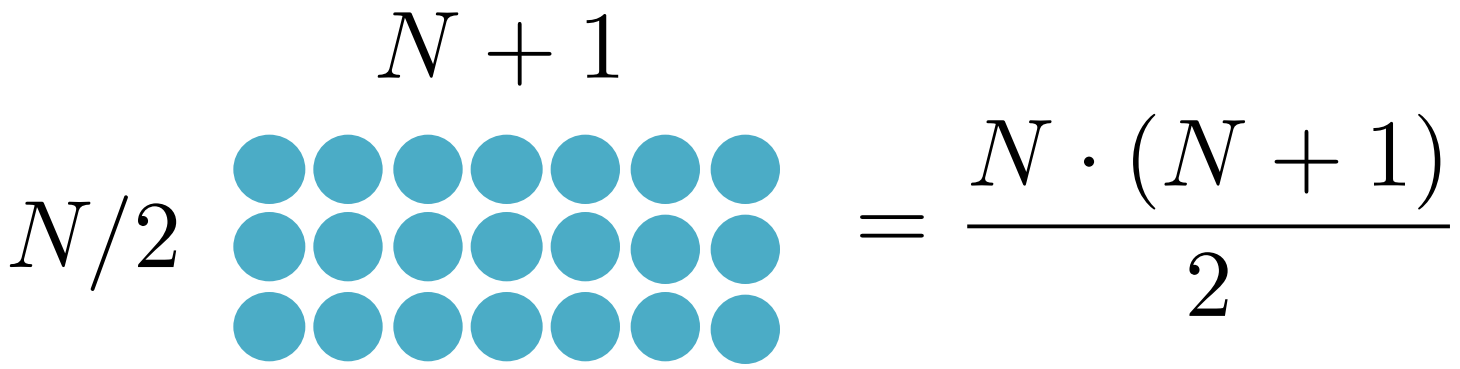
$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$



Arithmetic Sum

- You can convince yourself that

$$1 + 2 + 3 + \cdots + (N-2) + (N-1) + N = \frac{N \times (N+1)}{2}$$

$$= \frac{N \cdot (N+1)}{2}$$

$$= \frac{1}{2}N^2 + \frac{1}{2}N$$

$$\mathcal{O}(N^2)$$

Pro Tip #2: Loops Multiply Big O

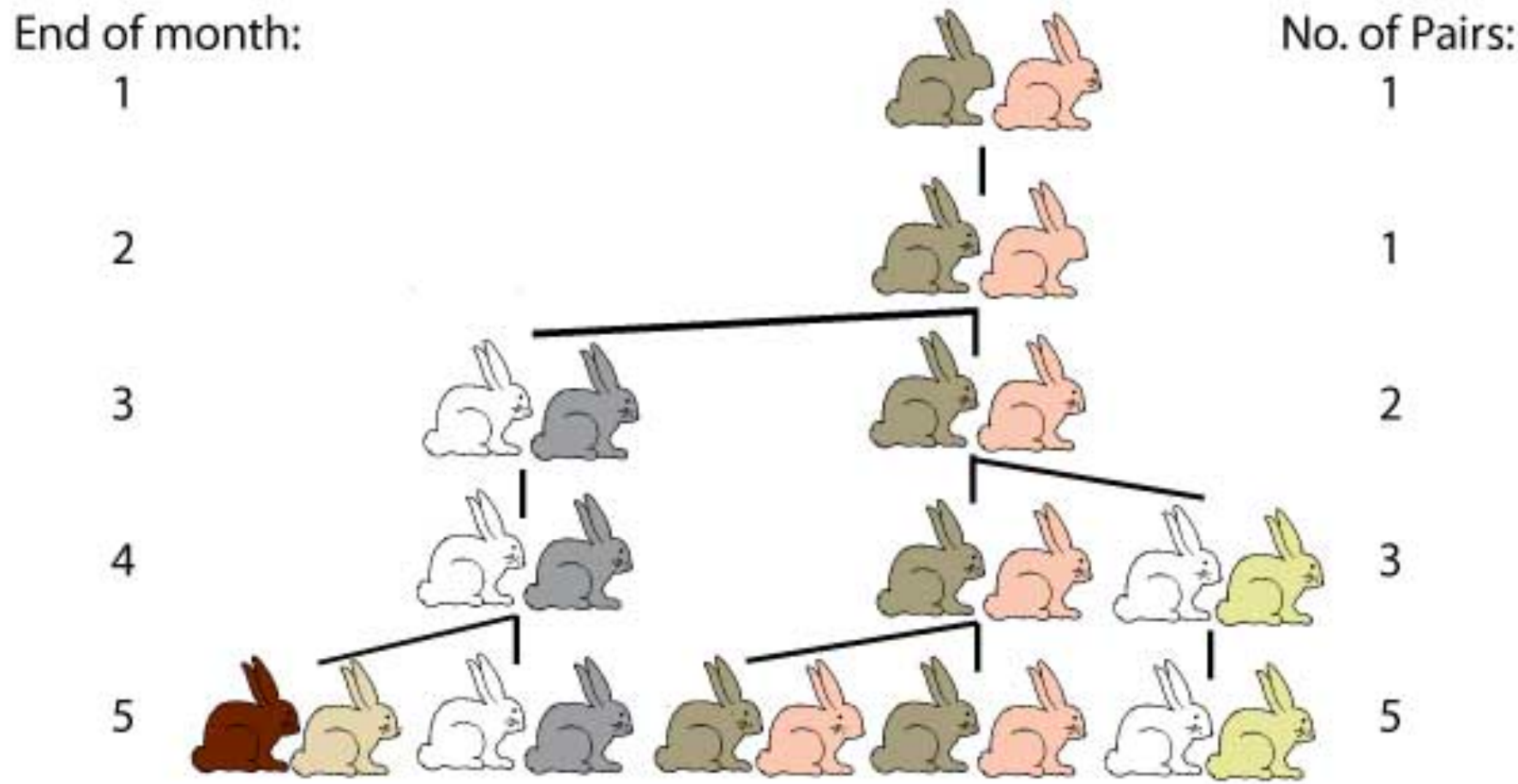
What is the Big O?

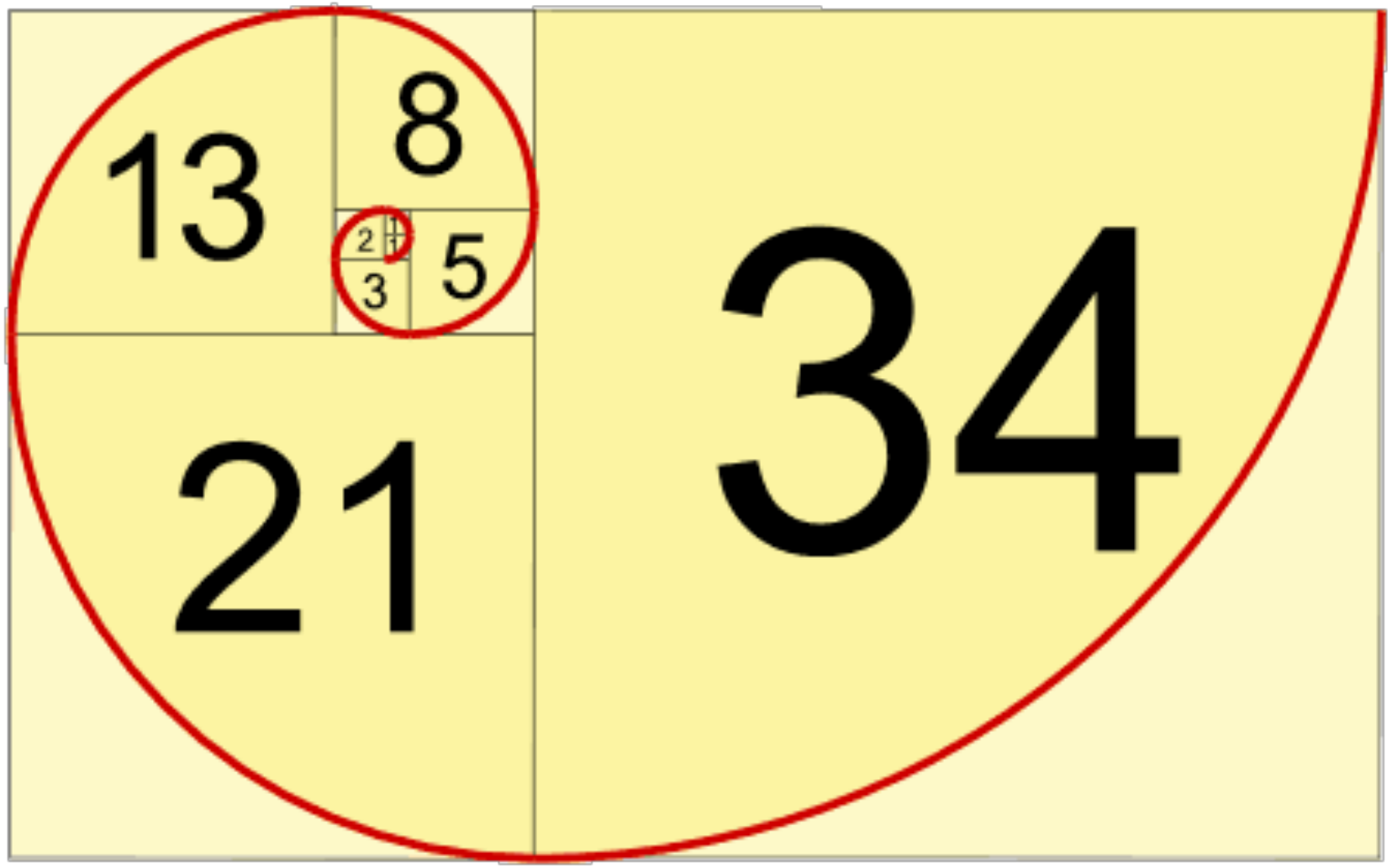
ProTip: Big O is multiplicative in loops

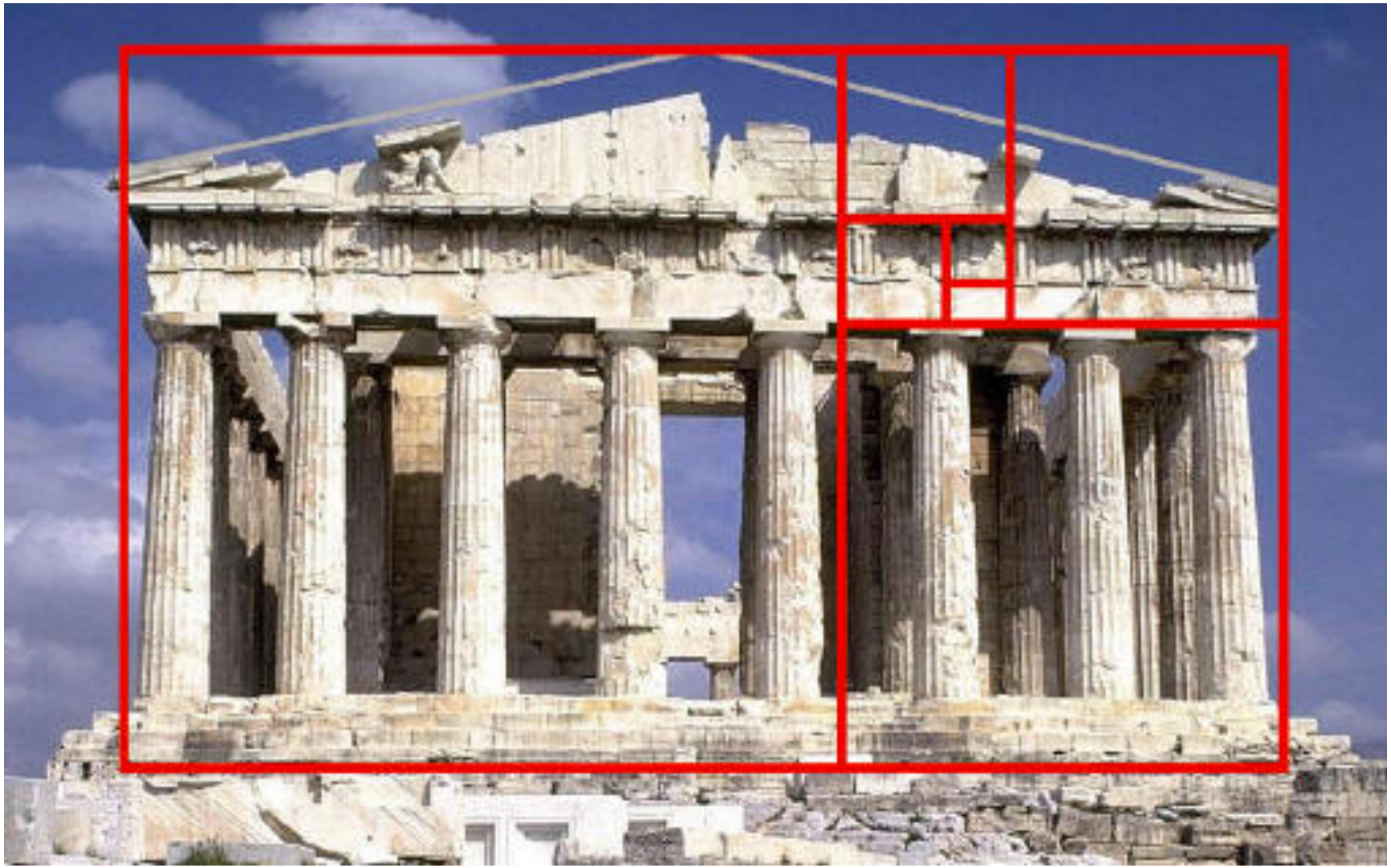
```
void friendly(Set<string>& facebookUsers) {

  int n = facebookUsers.size();
  cout << "n: " << n << endl;

  for(string user : facebookUsers){
    // addFriend is O(log n)
    addFriend(user, "Chris Gregg");
  }
}
```

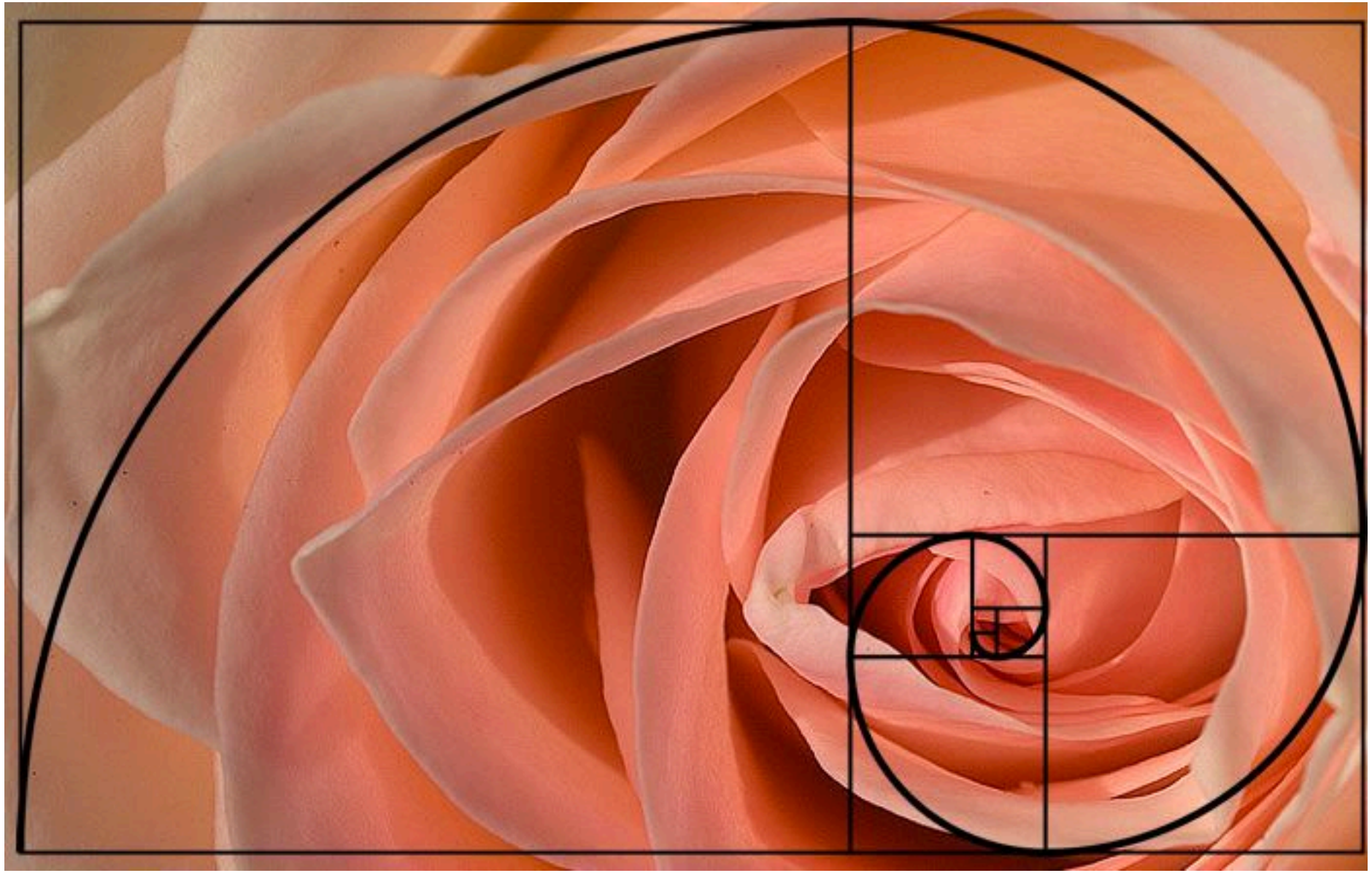
End Aside

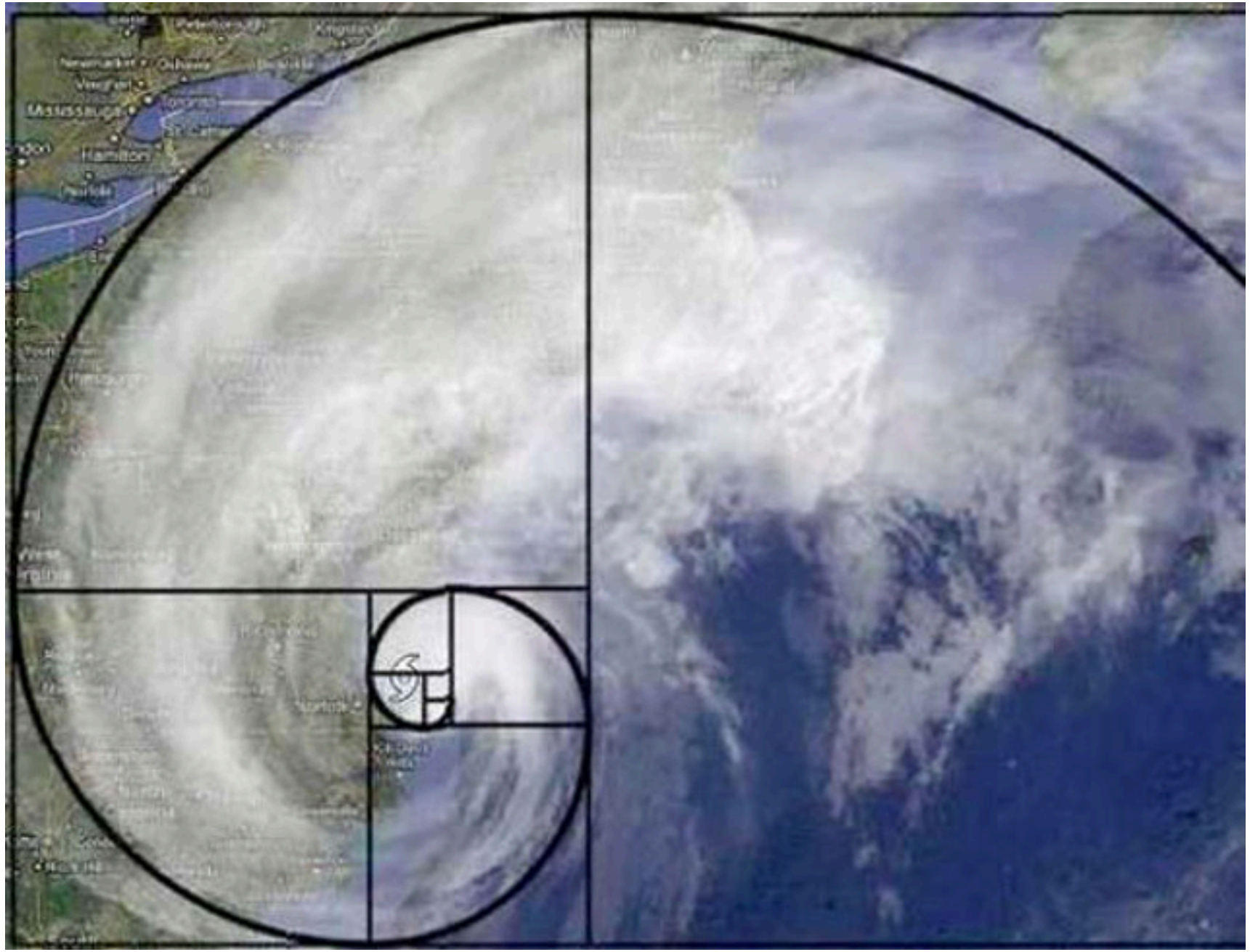
Fibonacci


End of month:
No. of Pairs:
1
1
2
1
3
2
4
3
5
5


13
8
2
1
3
5
21
34






Mississauga
Hamilton

Fibonacci

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

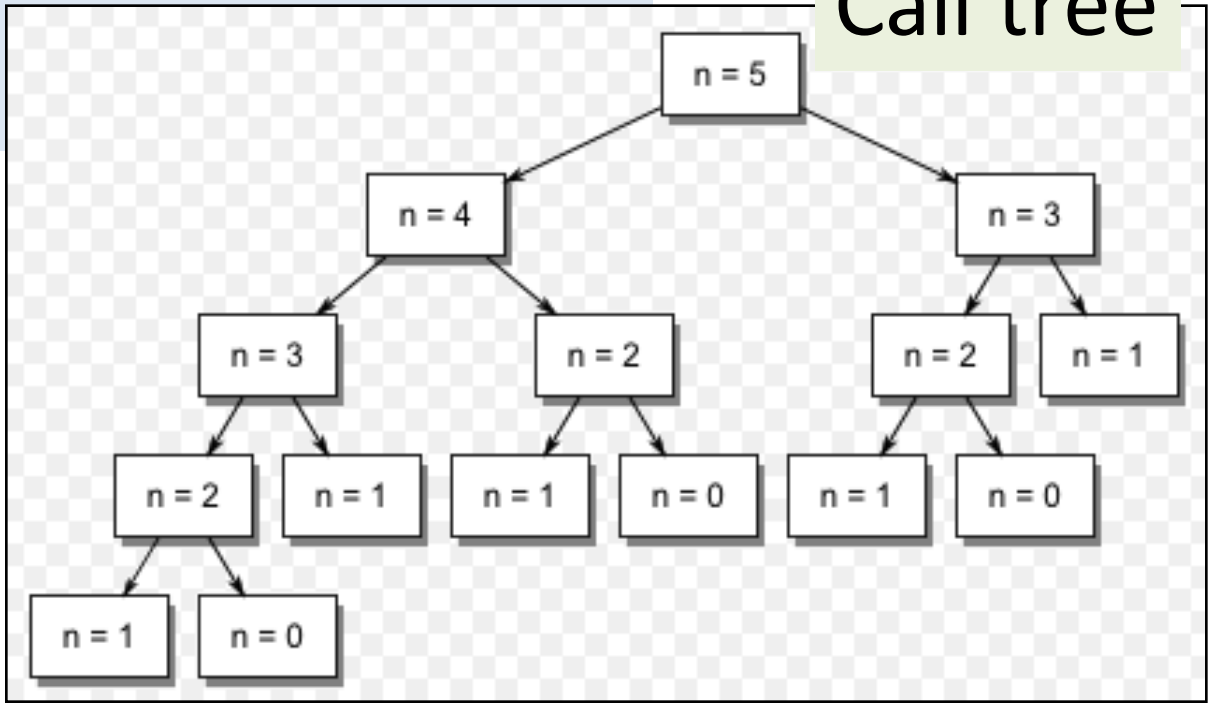
Fibonacci

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

Fibonacci

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

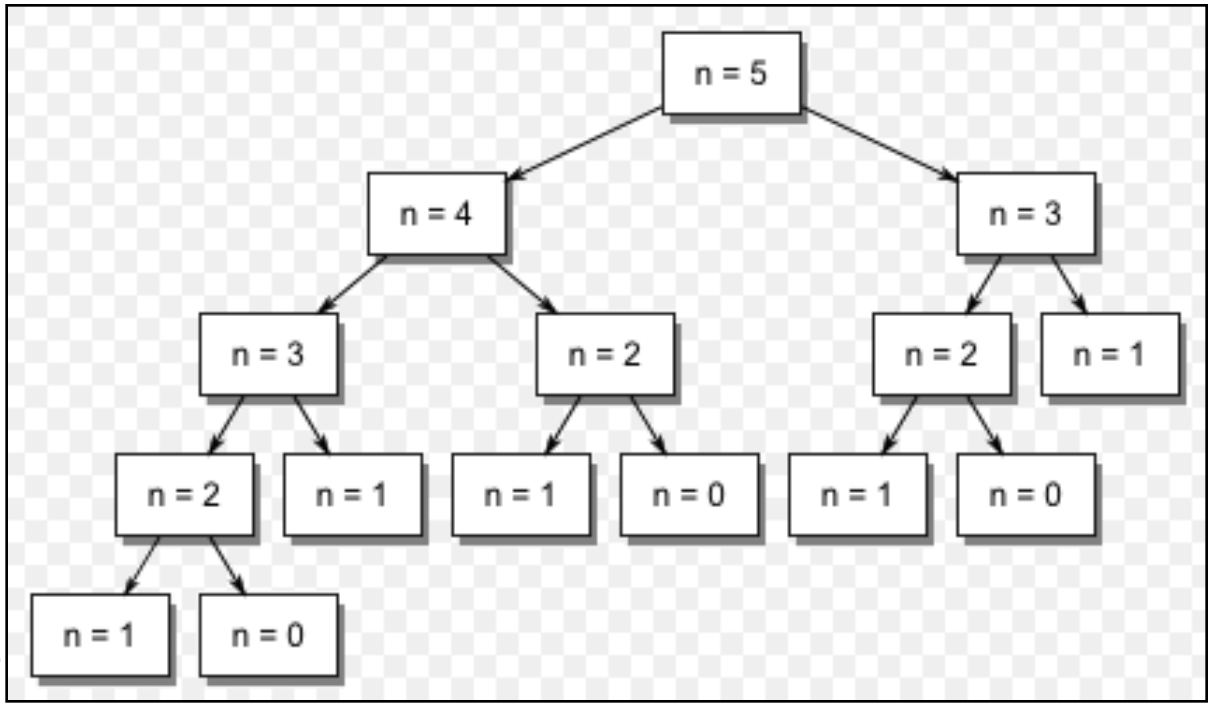
Call tree



Big O?

Fibonacci: Big O

Count the number of recursive calls



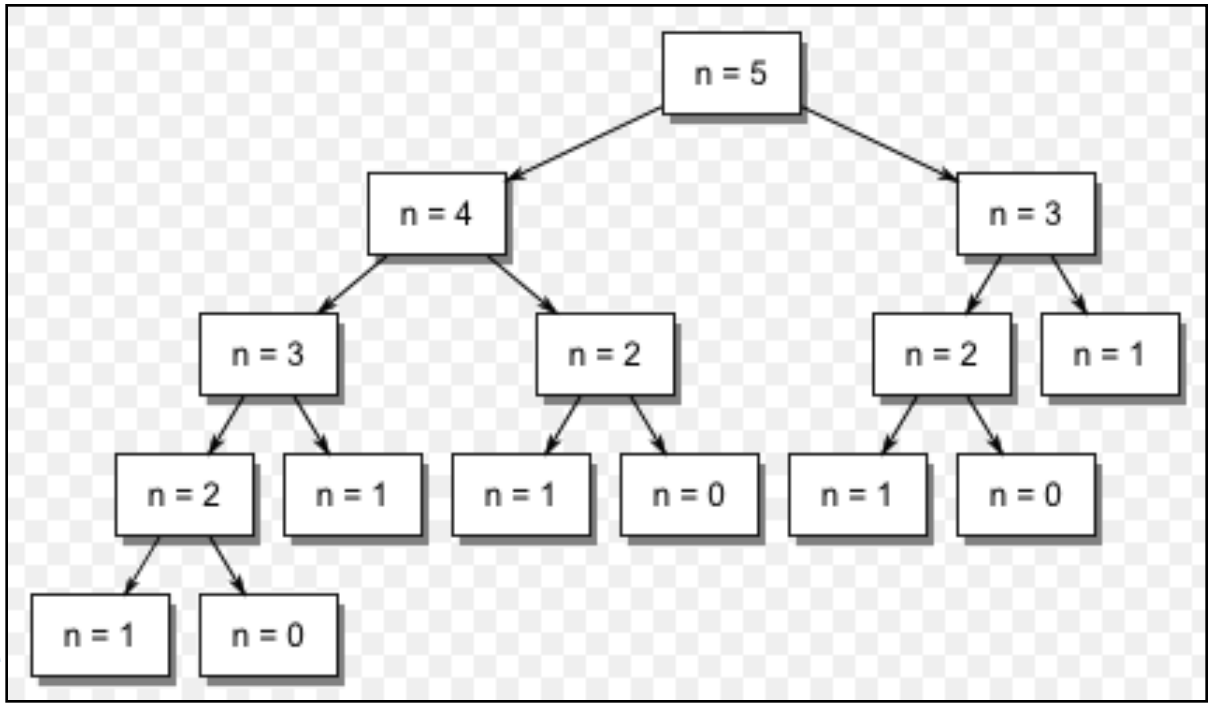
Branching (*b*) = decisions in worse recursive case.
Depth (*d*) = longest chain of recursive calls.

$$\mathcal{O}(b^d) = \mathcal{O}(2^n)$$

Aside

Fibonacci: Big O

Count the number of recursive calls



This is beyond CS106B:

$$\mathcal{O}(1.62^n)$$

Fibonacci: Big O

$\mathcal{O}(1.62^n)$ technically is $\mathcal{O}(2^n)$

since

$$\mathcal{O}(1.62^n) < \mathcal{O}(2^n)$$

We call it a “tighter” bound

End Aside

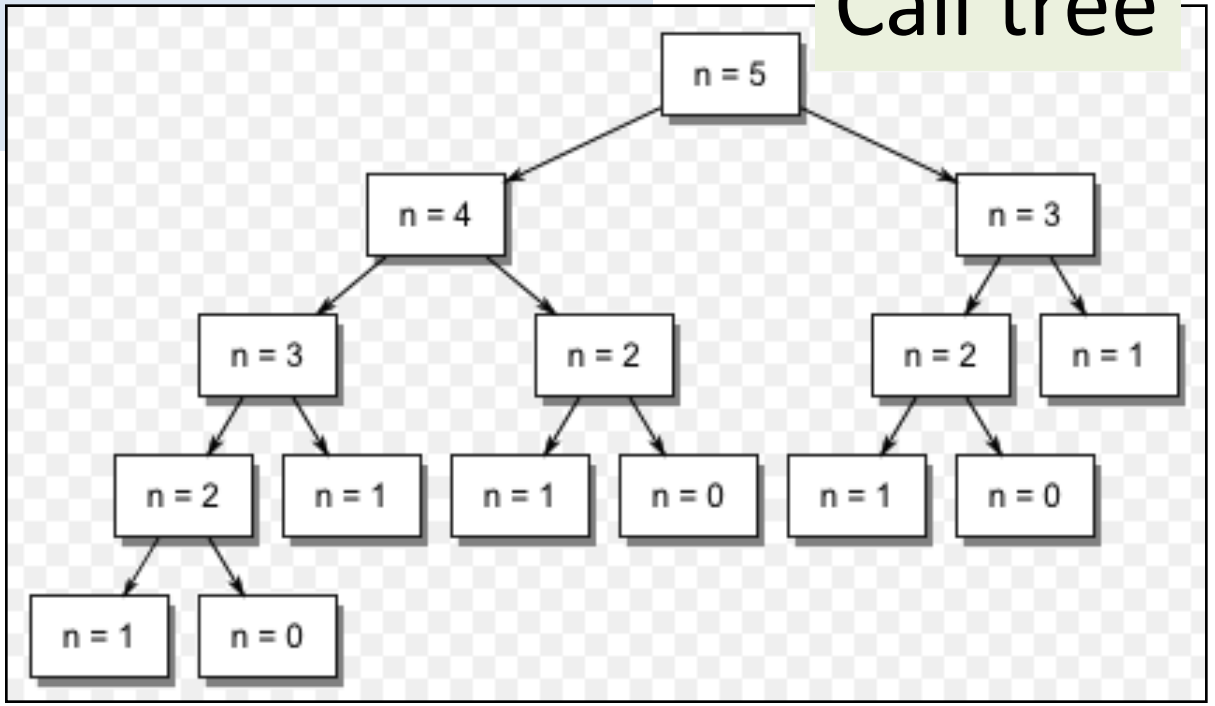
Doesn’t look good.

Call Ghost Busters!

Fibonacci

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

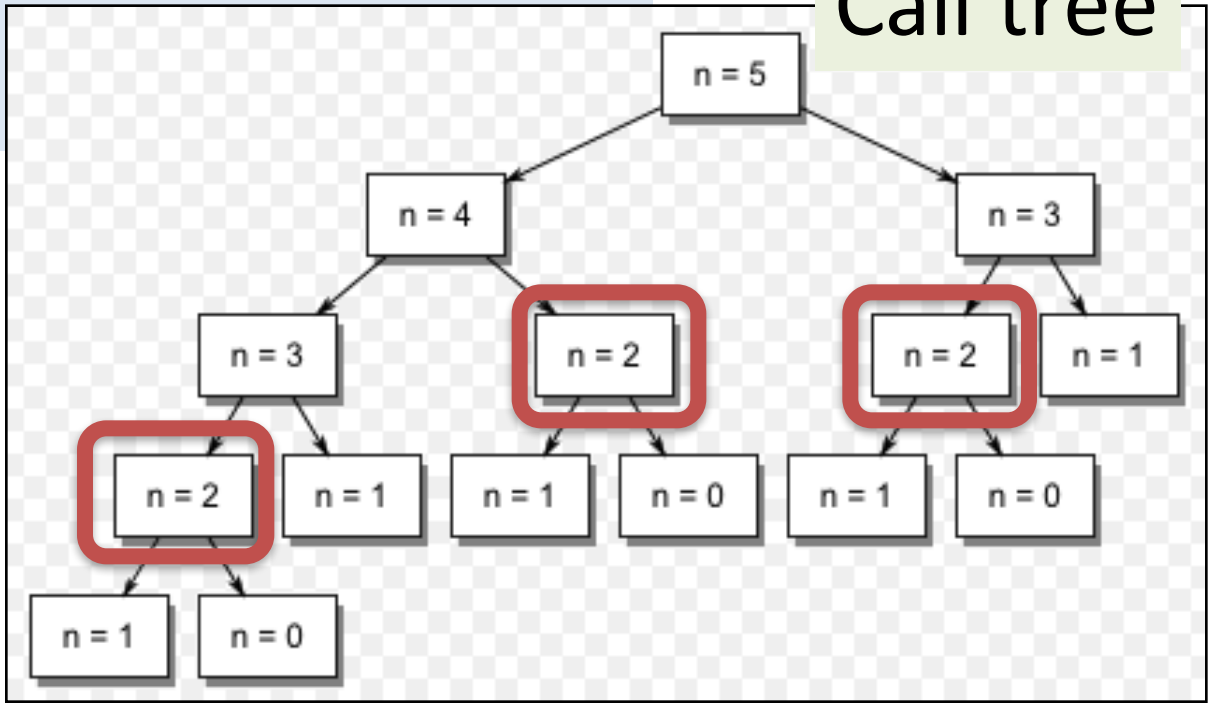
Call tree



Fibonacci

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

Call tree

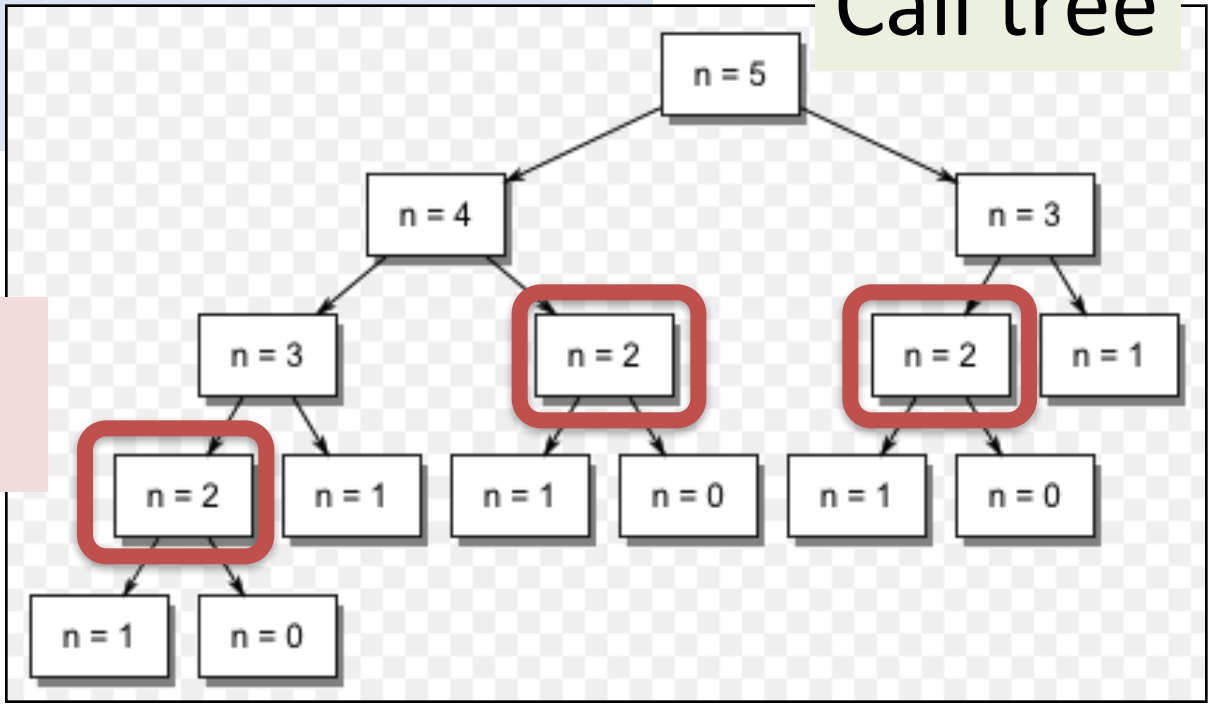


Fibonacci

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

fib(2) is calculated 3 separate times when calculating fib(5)!

Call tree



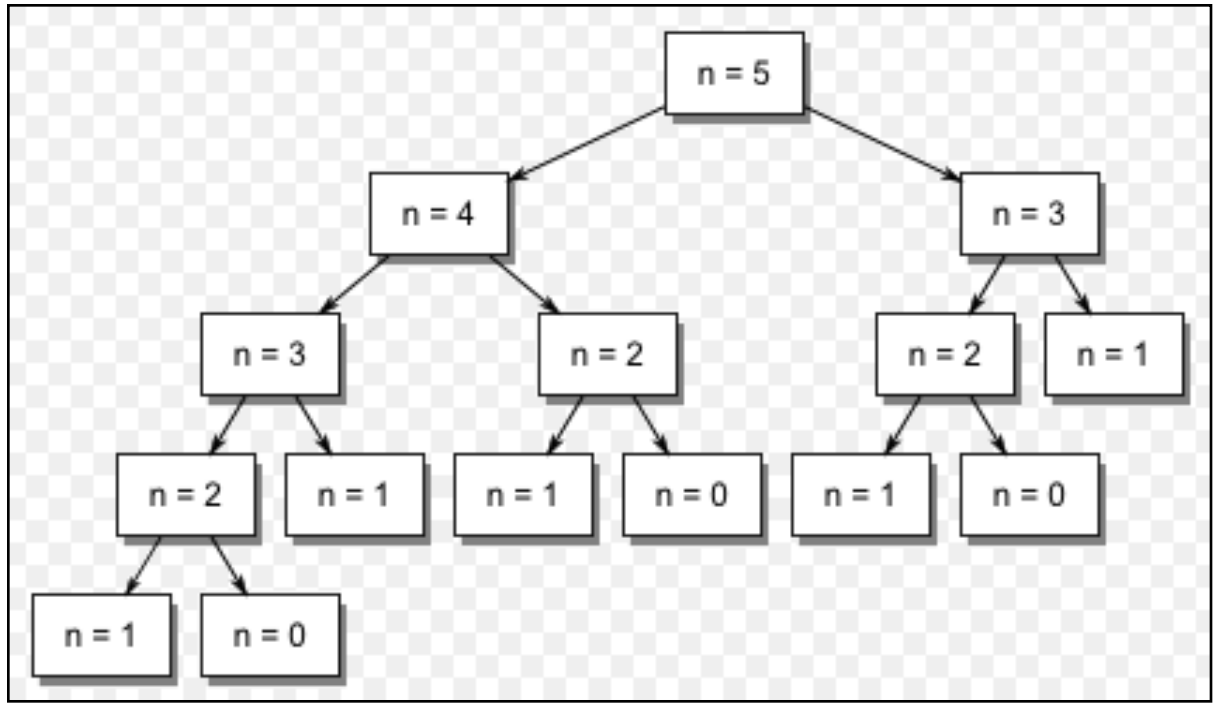
Memoization

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

aka

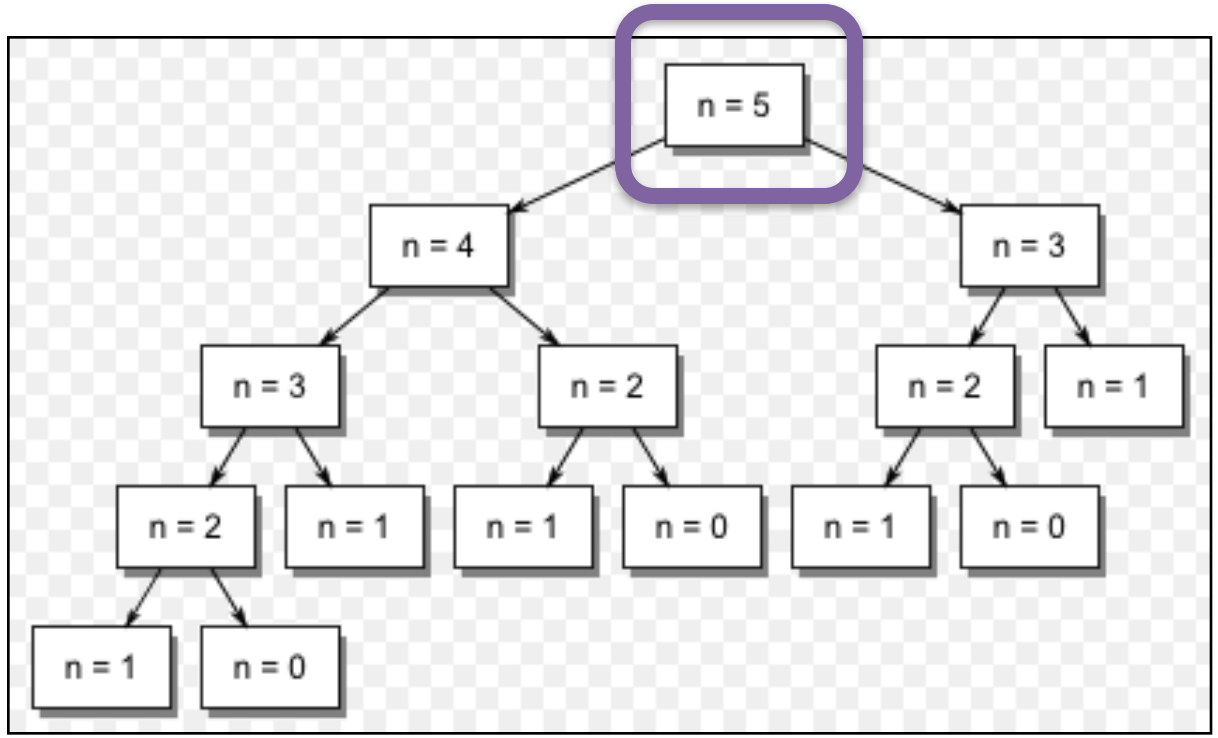
Remember what you have already done!

Memoization



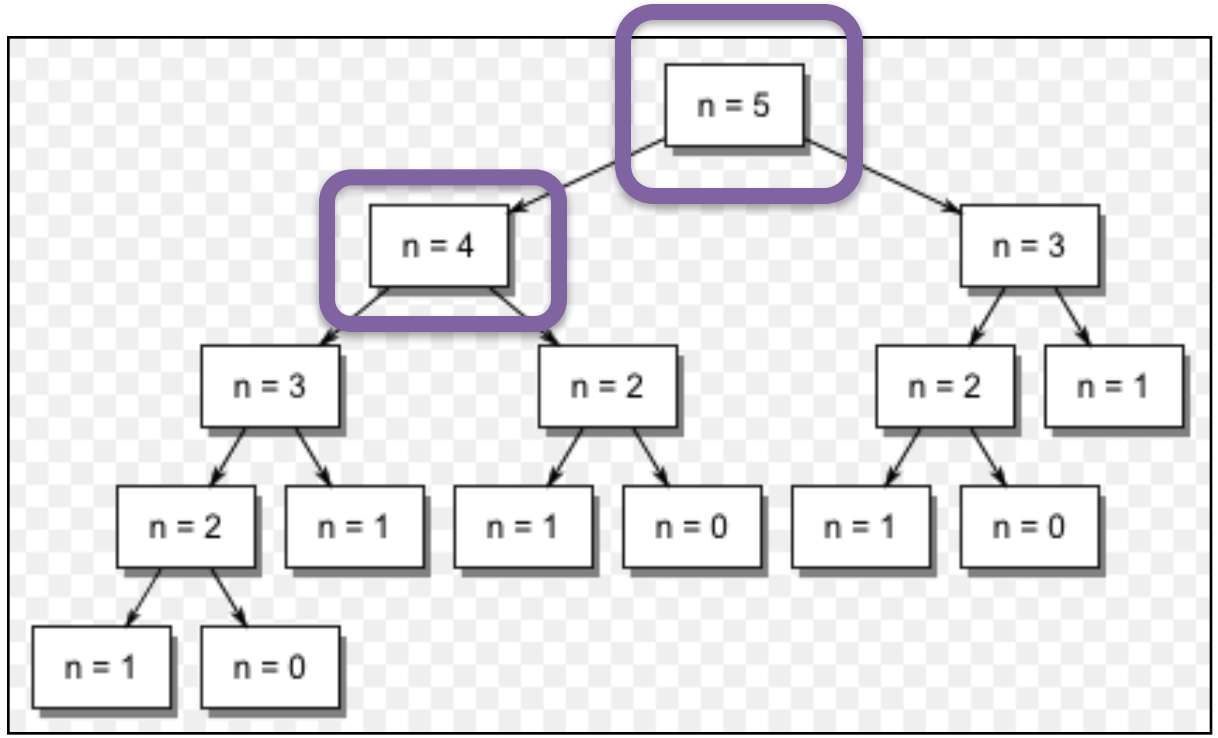
Cache:

Memoization



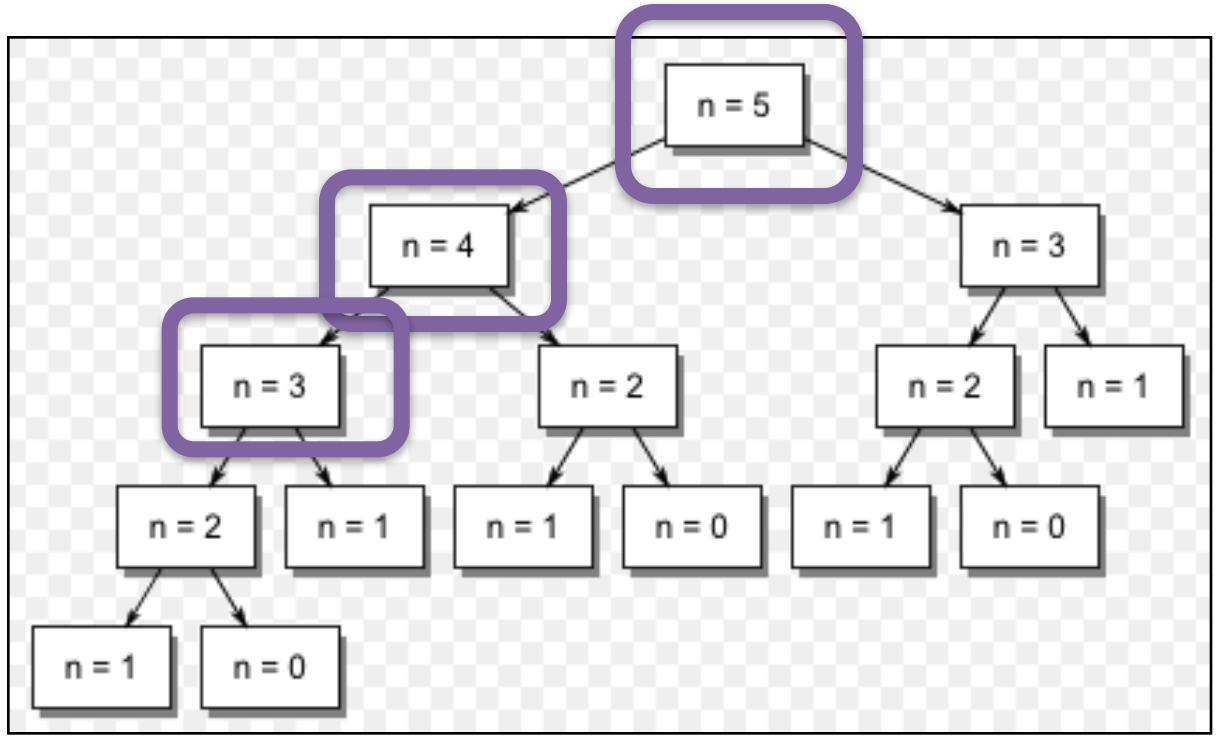
Cache:

Memoization



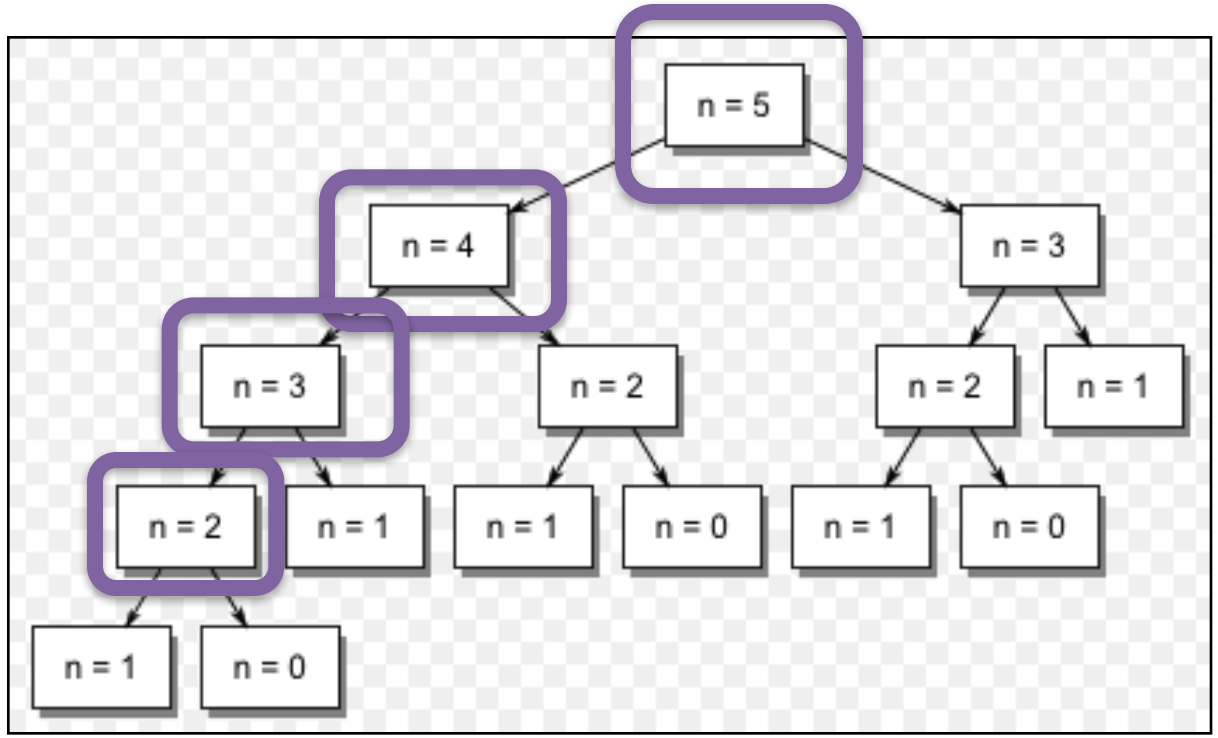
Cache:

Memoization



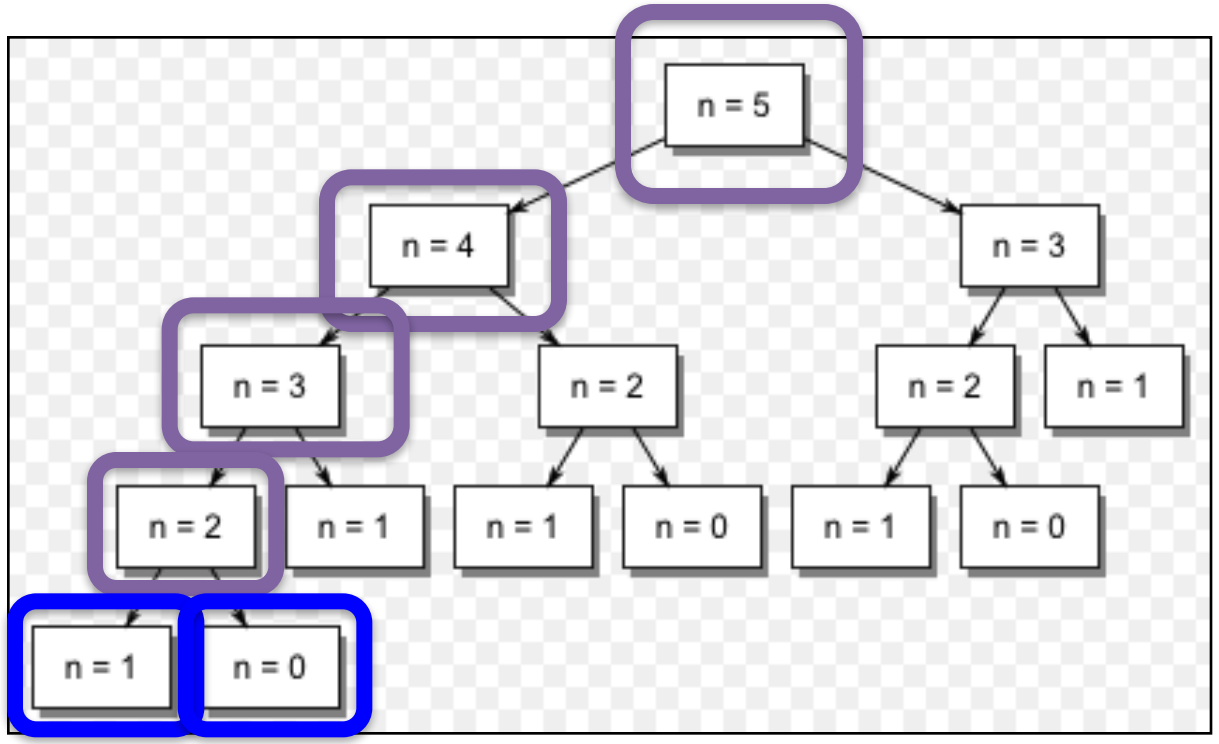
Cache:

Memoization



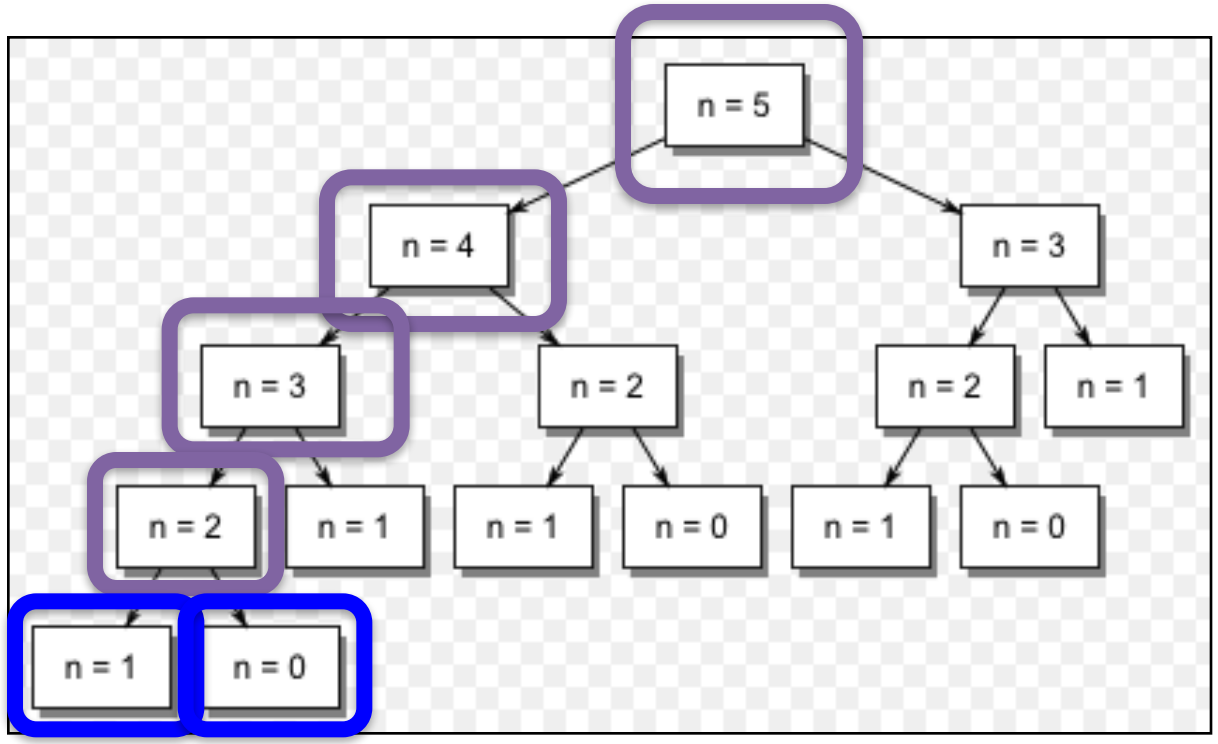
Cache:

Memoization



Cache:

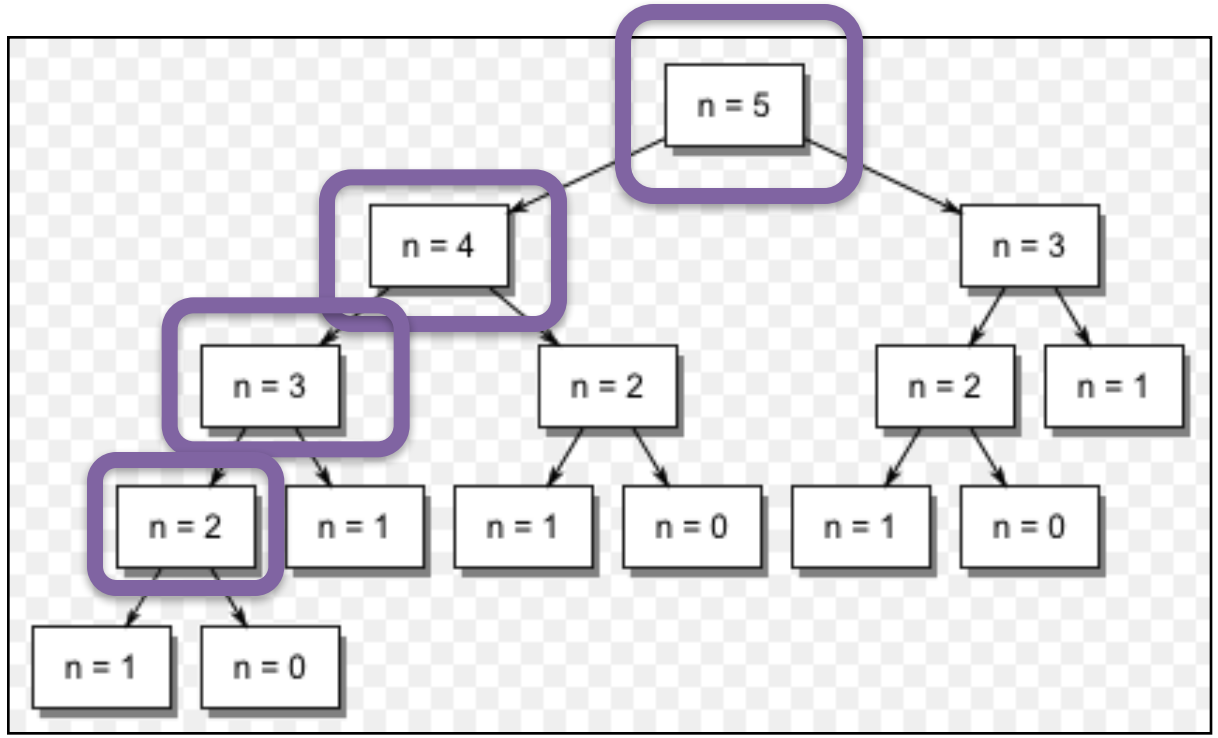
Memoization



Cache:

f(2) = 2

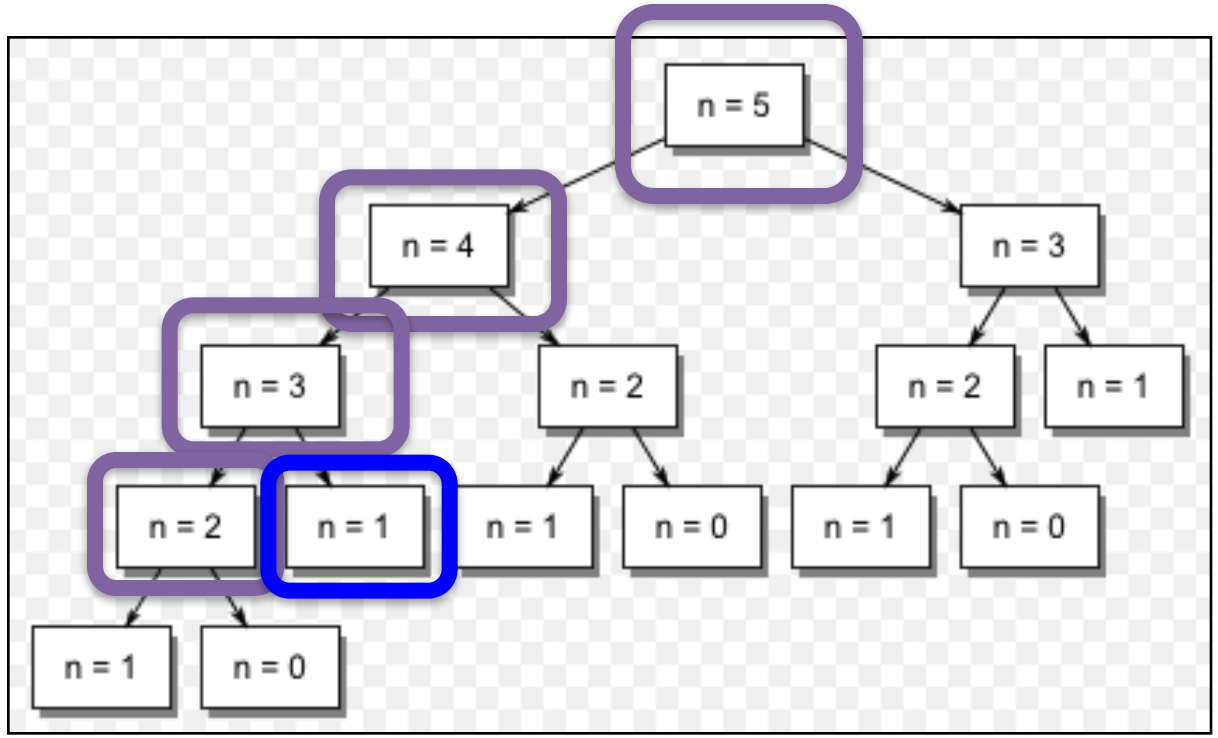
Memoization



Cache:

f(2) = 2

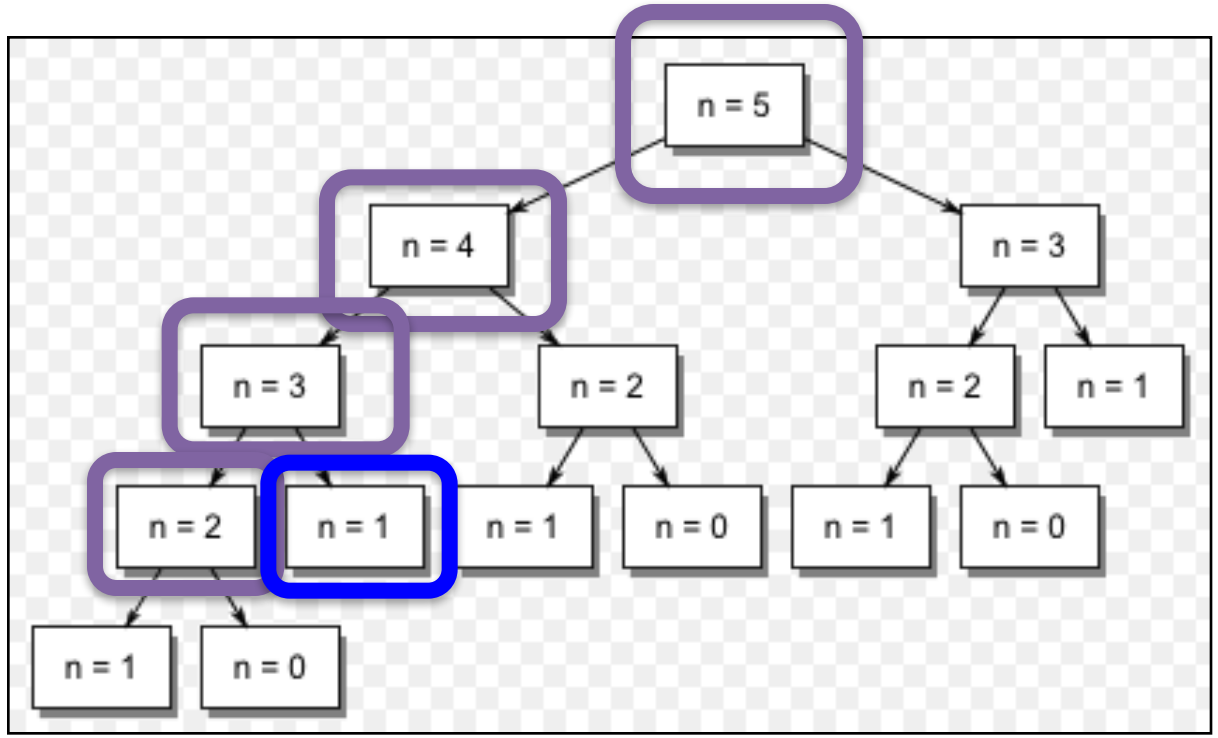
Memoization



Cache:

f(2) = 2

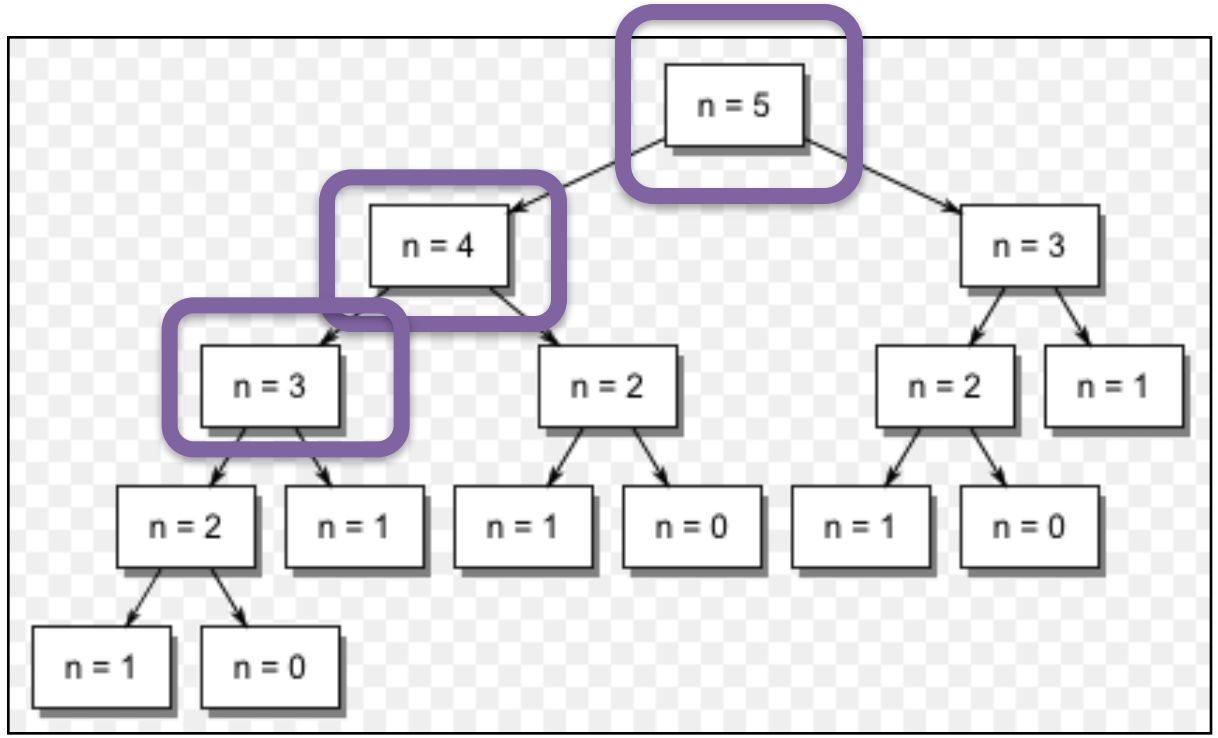
Memoization



Cache:

f(2) = 2, f(3) = 3

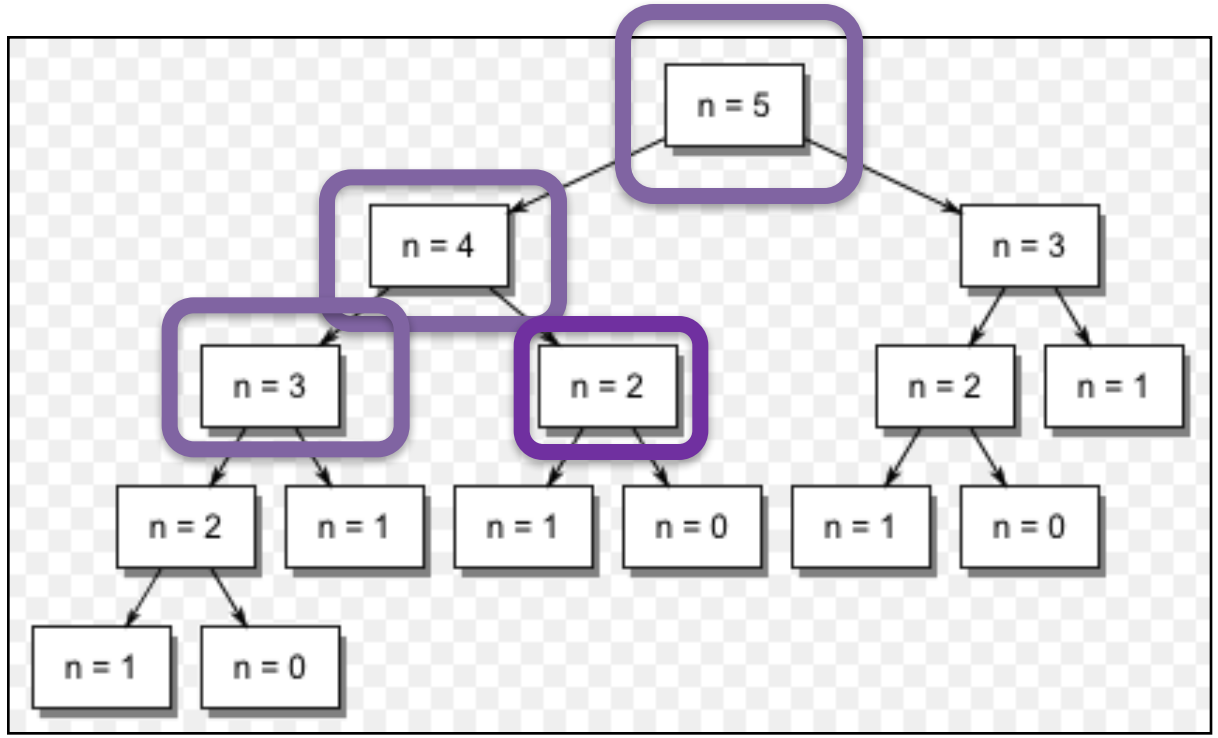
Memoization



Cache:

f(2) = 2, f(3) = 3

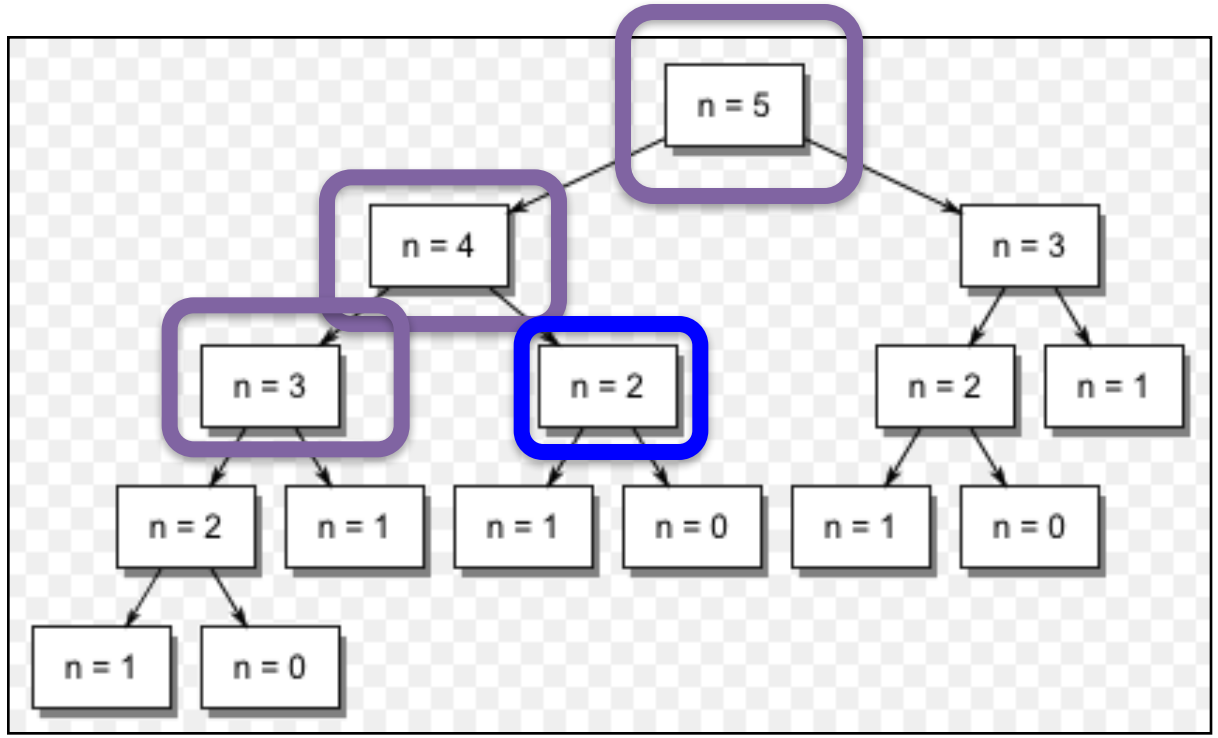
Memoization



Cache:

f(2) = 2, f(3) = 3

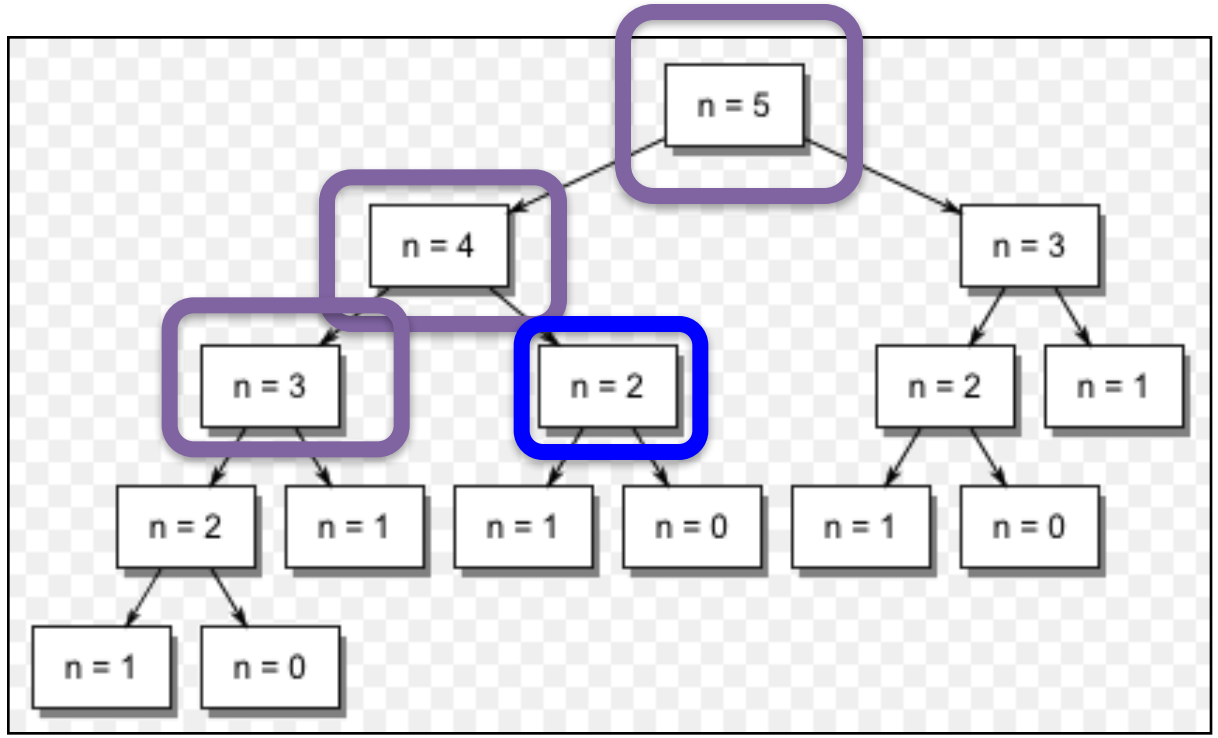
Memoization



Cache:

f(2) = 2, f(3) = 3

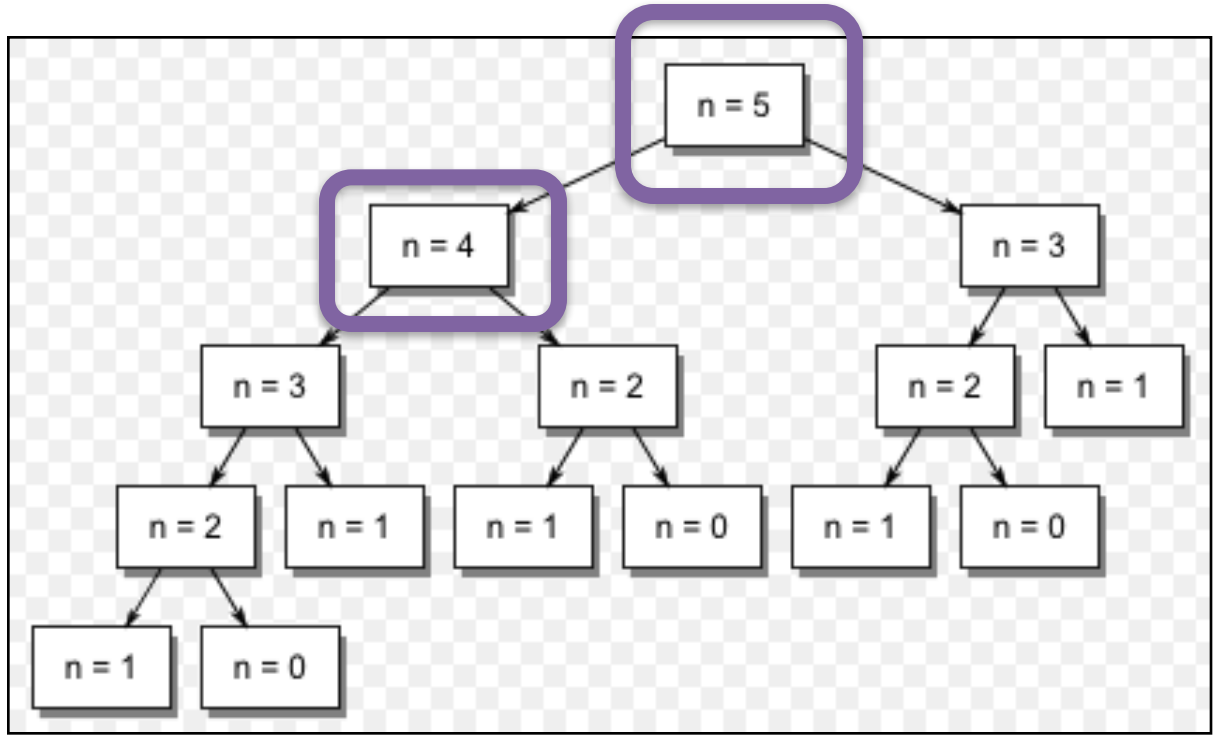
Memoization



Cache:

f(2) = 2, f(3) = 3

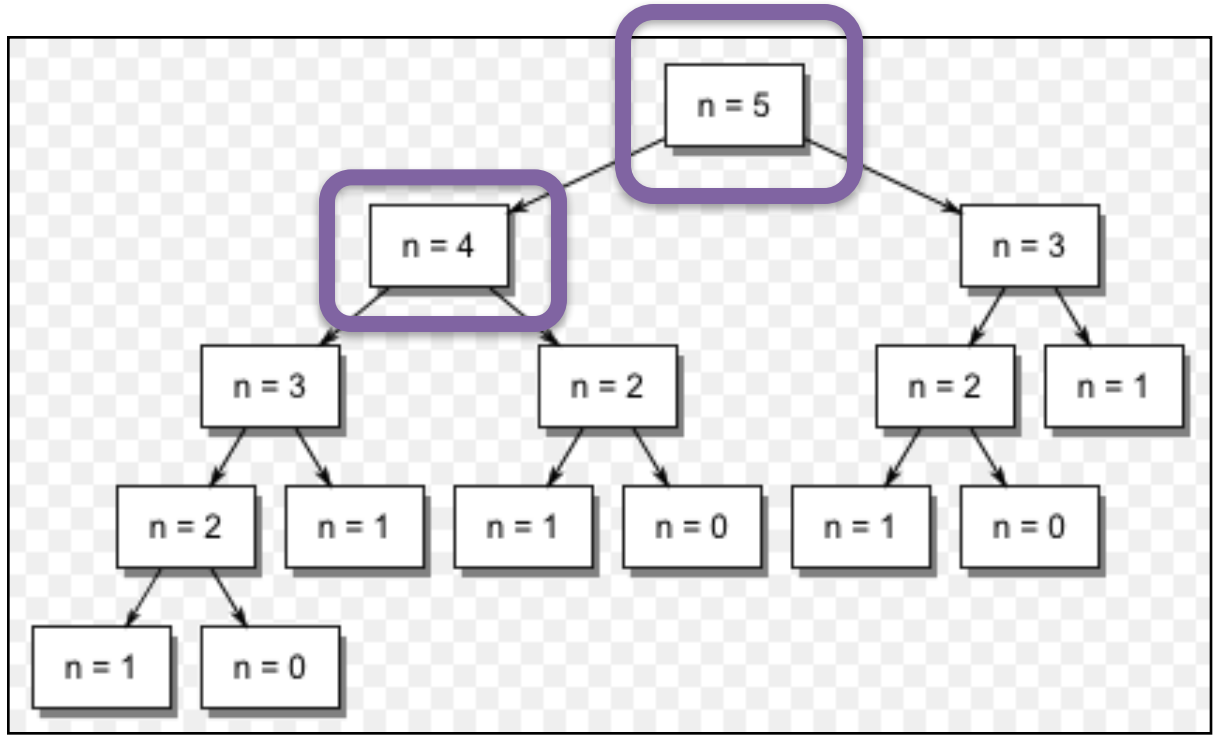
Memoization



Cache:

f(2) = 2, f(3) = 3

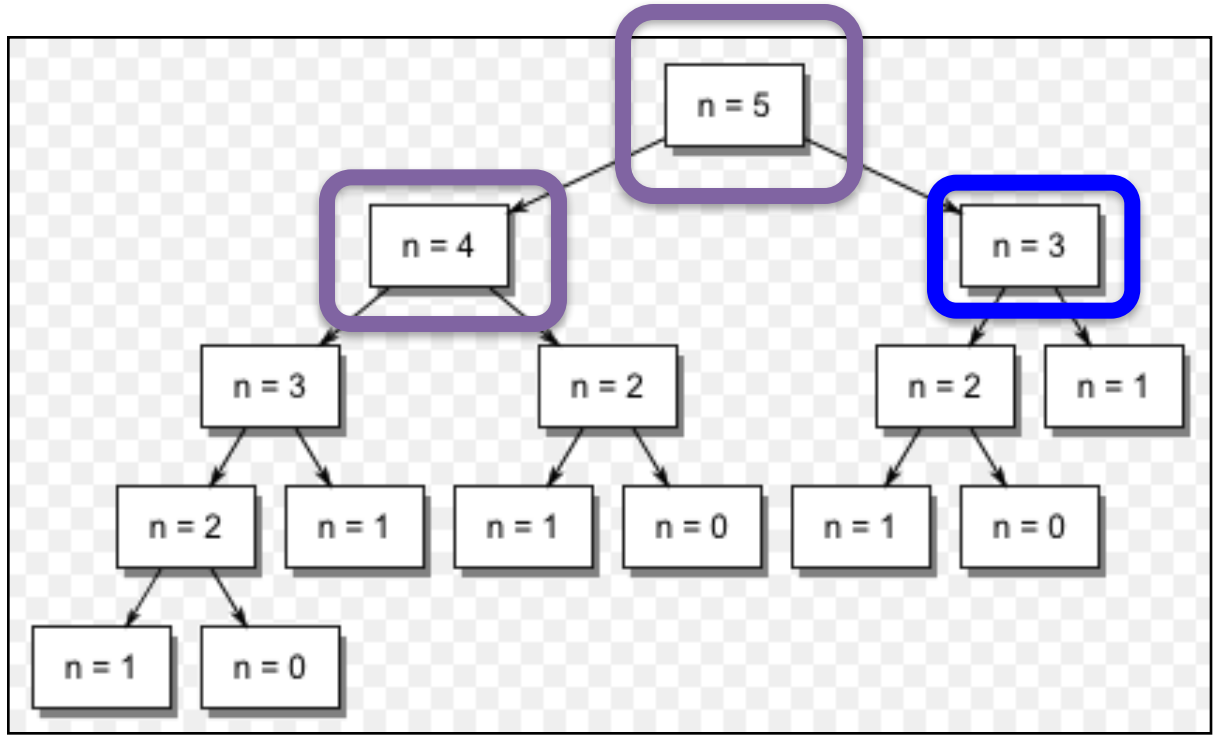
Memoization



Cache:

f(2) = 2, f(3) = 3, f(4) = 5

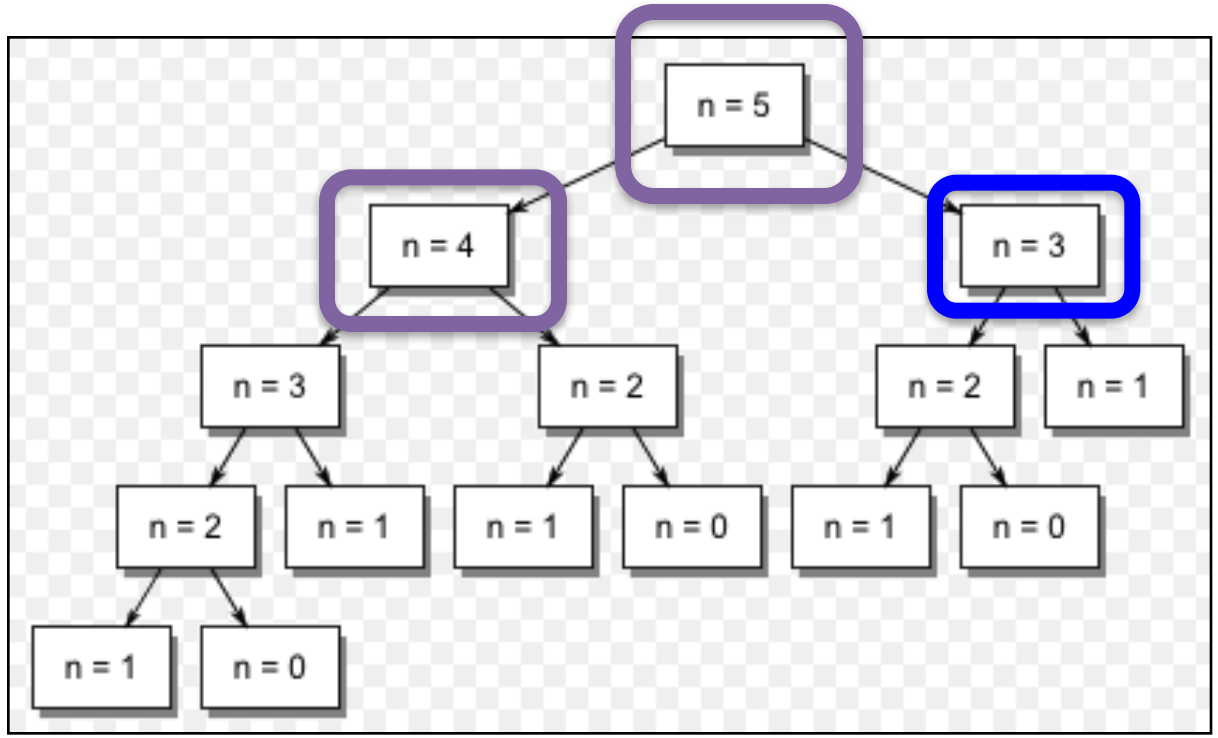
Memoization



Cache:

f(2) = 2, f(3) = 3, f(4) = 5

Memoization



Cache:

f(2) = 2, f(3) = 3, f(4) = 5

Too Fast, Too Furious

Fast Fib

```
int fastFibb(Map<int, int>&cache, int n) {
  // base case
  if(cache.containsKey(n)) return cache[n];
  if(n <= 1) return 1;

  // recursive case
  int result = fastFibb(cache, n-1) + fastFibb(cache, n-2);
  cache[n] = result;
  return result;
}
```

Fast Fib

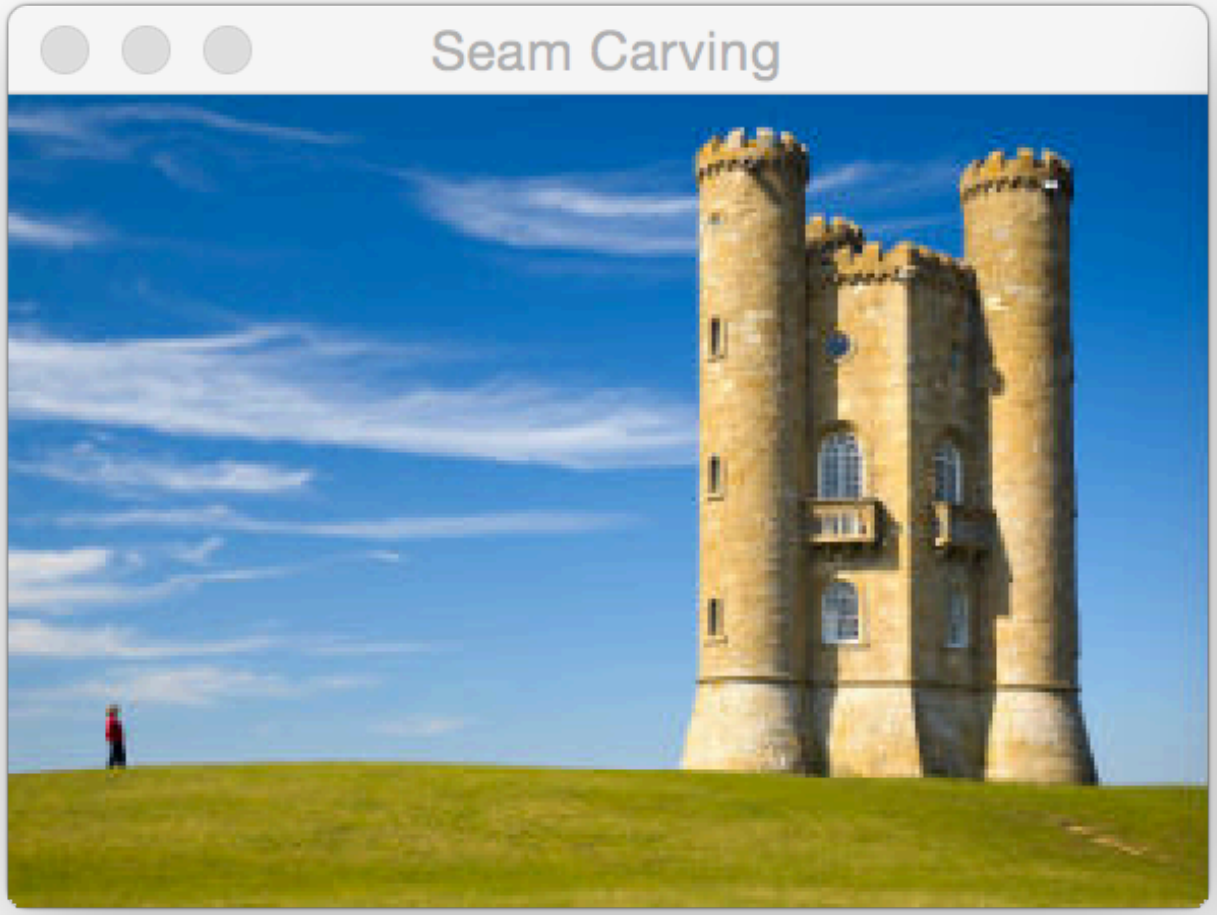
```
int fastFibb(Map<int, int>&cache, int n) {
  // base case
  if(cache.containsKey(n)) return cache[n];
  if(n <= 1) return 1;

  // recursive case
  int result = fastFibb(cache, n-1) + fastFibb(cache, n-2);
  cache[n] = result;
  return result;
}

// This is now a wrapper that calls fastFibb
int fibb(int n) {
  Map<int, int> cache;
  return fastFibb(cache, n);
}
```

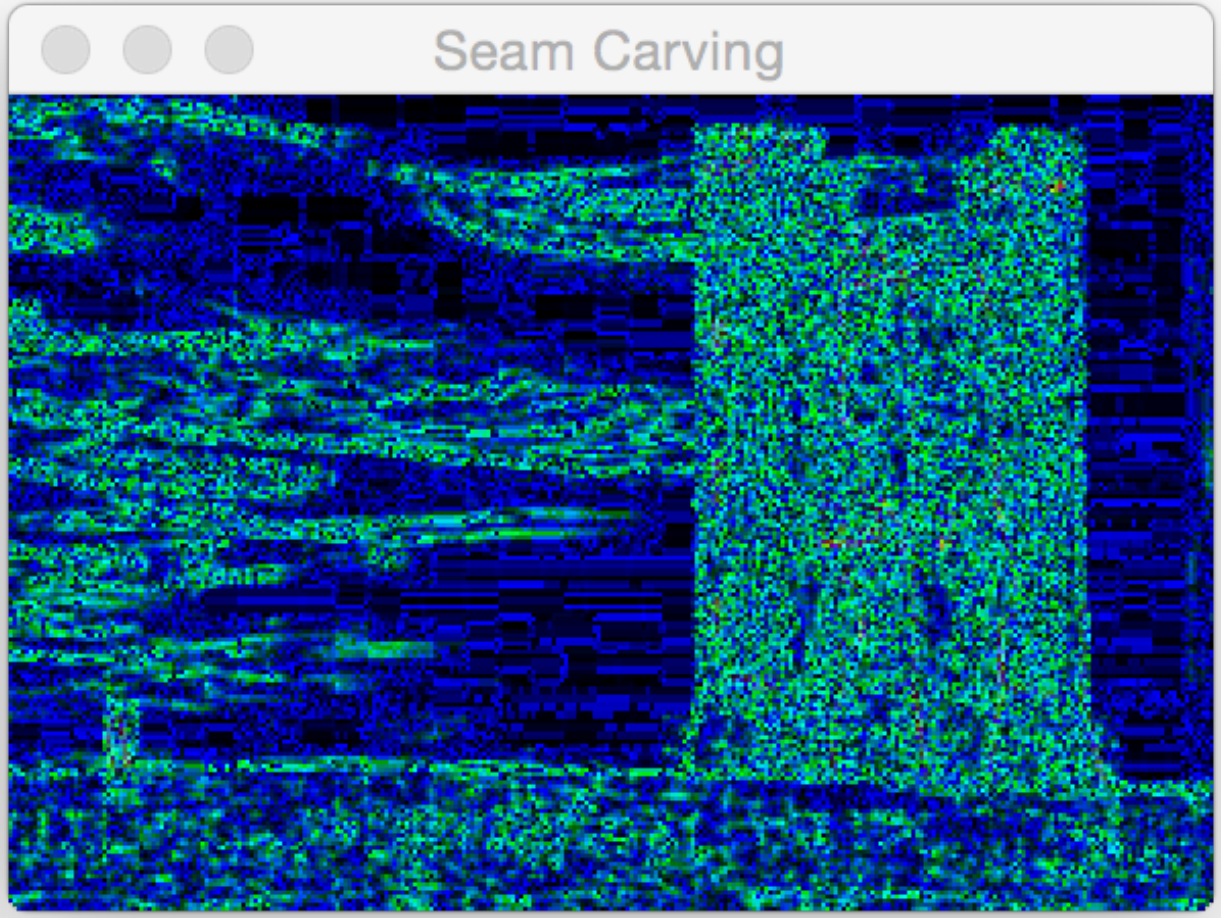
Fast Fib

$$\mathcal{O}(n)$$

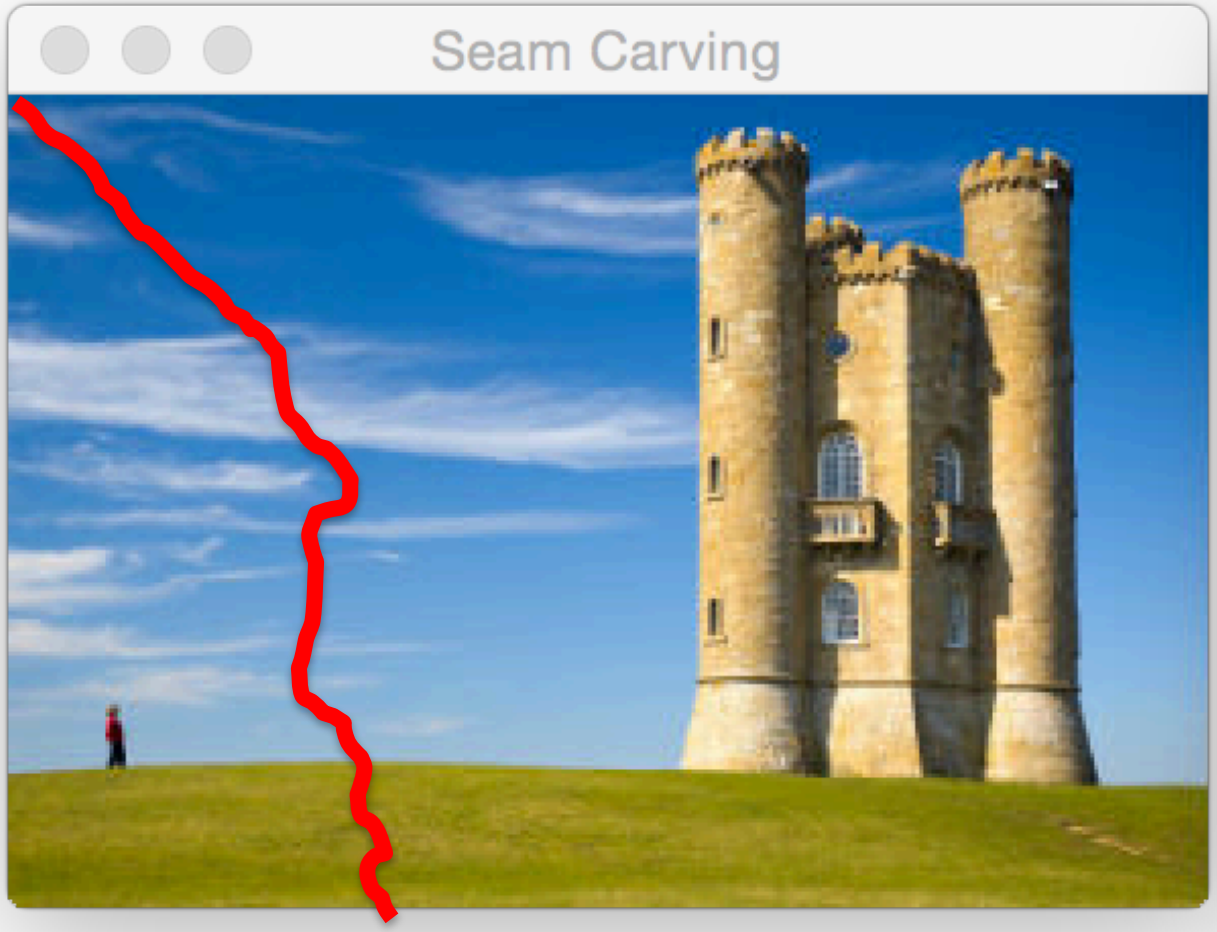
Seam Carving

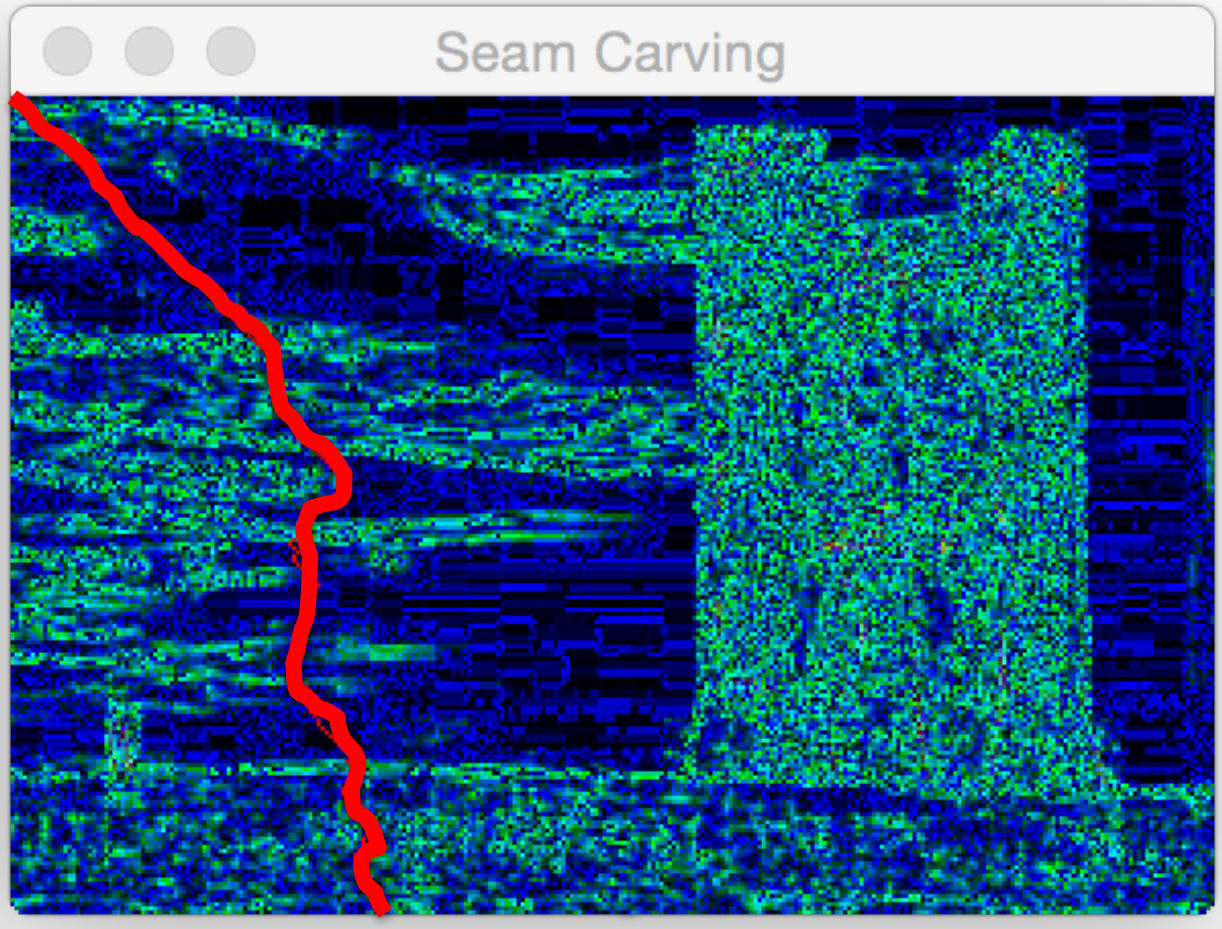
Seam Carving





Seam Carving





How to Represent the Path

Structs!

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

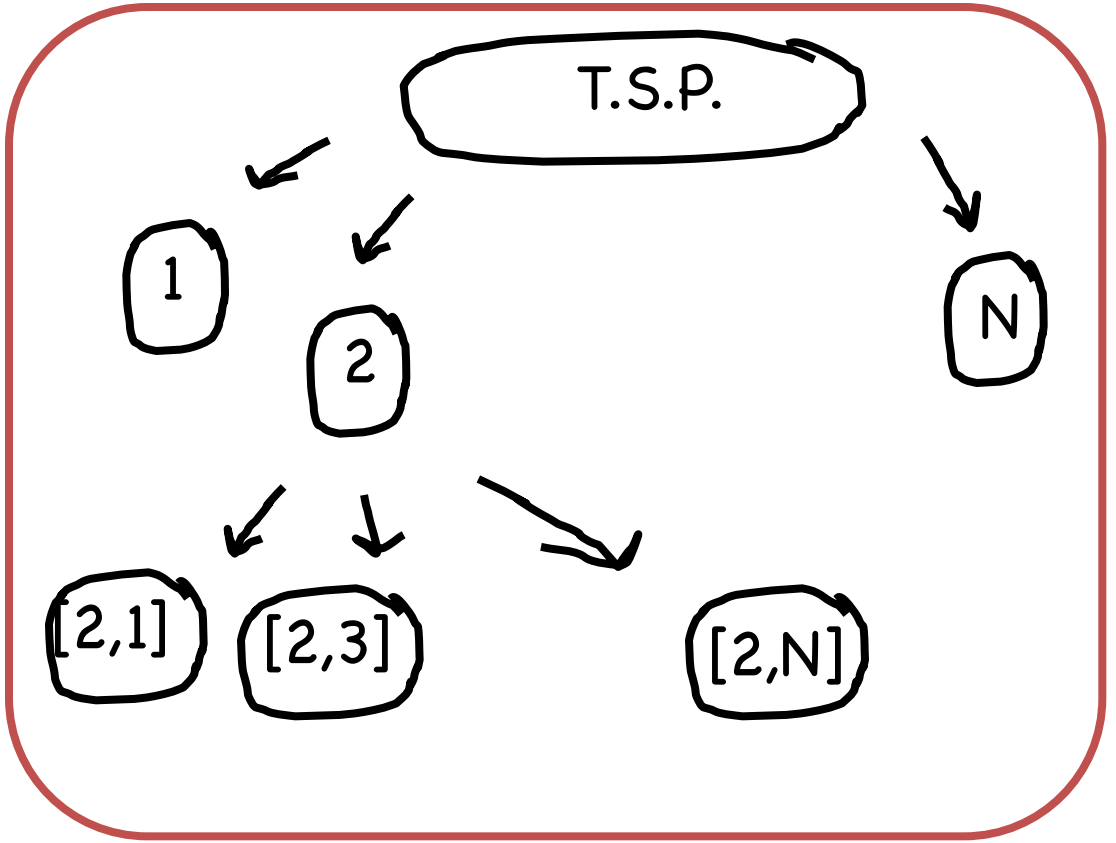
How to Represent the Path?

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



int main() {
    Coord myCord;
    myCoord.row = 5;
    myCoord.col = 7;

    cout << myCord.row << endl;

    Vector<Coord> path;
    return 0;
}
```

Lets do it!

Big O and Recursion

Count the number of recursive calls



$$\# \; calls = (N)\,(N-1) \times \ldots \times 1$$

Today's Goal

1. Feel Comfort with Big O
2. Feel Comfort with Recursion
3. Understand the benefit of memoization