Timing & Algorithmic Complexity

Sam Wolfson

CSE 120, Winter 2020

Administrivia

- Portfolio Update 2 due tonight!
- Tic-Tac-Toe due Friday
    - The focus of section this week
    - Check off or submit on Canvas
- Quiz 4 on Friday
- Living Computers Museum Report due Monday
    - Go this weekend if you haven't yet!
    - TAs will post on Piazza about when they will be there.
- Also in section this week: brainstorming final project ideas!
- Section next week: Innovation Exploration Presentations

Outline

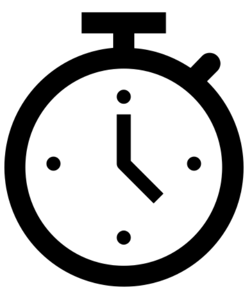
- **What is algorithm analysis?**
- How can we compare how long algorithms take?
- How can we “formalize” how long an algorithm takes?
- How can we optimize our algorithms?

Reprise: Algorithm Correctness

- Correctness is probably the most important aspect by which we can analyze algorithms.
- An algorithm is considered **correct** if, for every input, it:
  - returns the correct output,
  - doesn't run forever, and
  - doesn't cause an error.
- Incorrect algorithms could run forever, or crash, or not return the correct answer.

Algorithm Analysis: Timing

- One way to analyze algorithms: **computation time**
  - How long does it take to run and finish its job?
- We can use this to compare efficiency of two different algorithms that solve the same task.
  - Example: multiple ways to sort a list
- But how can we measure time?
  - Counting in your head
  - With a stopwatch
  - **Within the program itself**



Timing in Processing

- The function `millis()` returns the number of milliseconds since starting your program (as an `int`).
  - To start timing, call and store the value in a variable:

```
int startTime = millis();
```

  - Call again after your function is complete, and subtract:

```
void draw() {
  int startTime = millis();
  computeSomething();
  int totalTime = millis() - startTime;
  println(“Took “ + totalTime + “ ms”);
  noLoop();
}
```

Outline

- What is algorithm analysis?
- **How can we compare how long algorithms take?**
- How can we “formalize” how long an algorithm takes?
- How can we optimize our algorithms?

Algorithm Example: Fibonacci

- Function: Fibonacci
  - $\text{fibonacci}(1) = 1$
  - $\text{fibonacci}(2) = 1$
  - $\text{fibonacci}(3) = \text{fibonacci}(1) + \text{fibonacci}(2) = 1 + 1 = 2$
  - $\text{fibonacci}(n) = \text{fibonacci}(n-2) + \text{fibonacci}(n-1)$
- Code: Fibonacci

```
int fibonacci(int n) {
  if (n == 1 || n == 2) {
    return 1;
  }
  return fibonacci(n-1) + fibonacci(n-2);
}
```

- Let's see it in action...

Comparison: Fibonacci

- One of our Fibonacci functions seemed a lot faster than the other one – why?
- Let's look at a more concrete way to figure it out.

- We can analyze time without ever getting out `millis()`, just by reasoning our way through an algorithm!

Outline

- What is algorithm analysis?
- How can we compare how long algorithms take?
- **How can we “formalize” how long an algorithm takes?**
- How can we optimize our algorithms?

How To Analyze Algorithmic Time

- Silly Example Function: SumPlus1
  - **Input**: an array of `ints`
  - **Output**: the sum of all `ints` in the array, plus 1
- Code: SumPlus1

```
int sumPlus1(int[] array) {
  int sum = 0;
  int i = 0;
  while (i < array.length) {
    sum = sum + array[i];
    i = i + 1;
  }
  sum = sum + 1;
  return sum;
}
```

How To Analyze Algorithmic Time

- **Cost**: the amount of time it takes to do something.
  - The cost of a “simple” line of code (i.e., no function calls or loops) is 1 “time.”

    ```
    int z = x + y;  // cost: 1
    ```

  - The cost of a loop is the cost of all the lines of code inside of it, multiplied by the number of times it loops.

    ```
    int i = 0;
    while (i < n) {  // cost: 2 * n
        int x = 3;  // cost: 1
        i = i + 1;  // cost: 1
    }
    ```

  - The cost of a function is the sum of the cost of all the lines of code within the function.

Analysis of SumPlus1

```
int sumPlus1(int[] array) {
  int sum = 0;
  int i = 0;
  while (i < array.length) {
    sum = sum + array[i];
    i = i + 1;
  }
  sum = sum + 1;
  return sum;
}
```

Cost: 1

Cost: 1

Cost: 1

Cost: 1

Loop Cost:
2 * `array.length`

Cost: 1

Cost: 1

Let the length of `array` be equal to n. Then the total cost is:

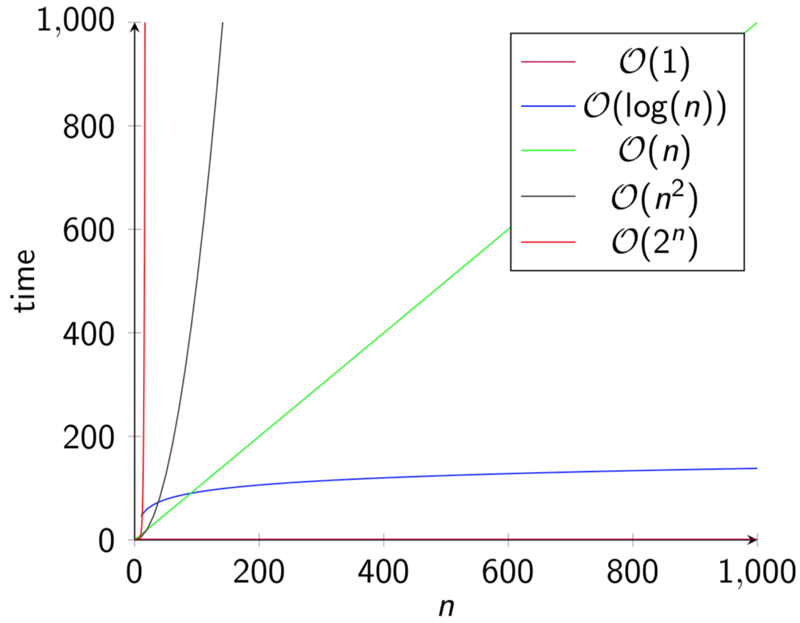
$$\text{cost}(n) = 1 + 1 + (2 * n) + 1 + 1 = 2n + 4$$

Analysis of SumPlus1

- When analyzing functions, we only care about the term that **grows the fastest**.
- For $\text{cost}(n) = 2n + \cancel{4}$, which term is this?
  - 4 is constant – it never grows no matter how large n becomes.
  - $2n$, however, grows linearly with n – so it is the fastest-growing term in this cost.
- $\text{cost}(n) \approx \cancel{2}n$
  - Furthermore, we don’t really care about the 2, since it’s constant.
- $\mathcal{O}(n) = n$
  - We call this “Big-Oh” notation – we’re only concerned with the **fastest-growing term**, and with the parts of it that **actually grow**.

Time Complexity

- The amount of time it take to run an algorithm.
  - The fastest-growing term in the cost function ("order of growth").
  - Written in terms of the size n of the input (e.g., number of elements in an array, n^{th} Fibonacci number) with "Big-Oh" notation.



Time Analysis: Fibonacci

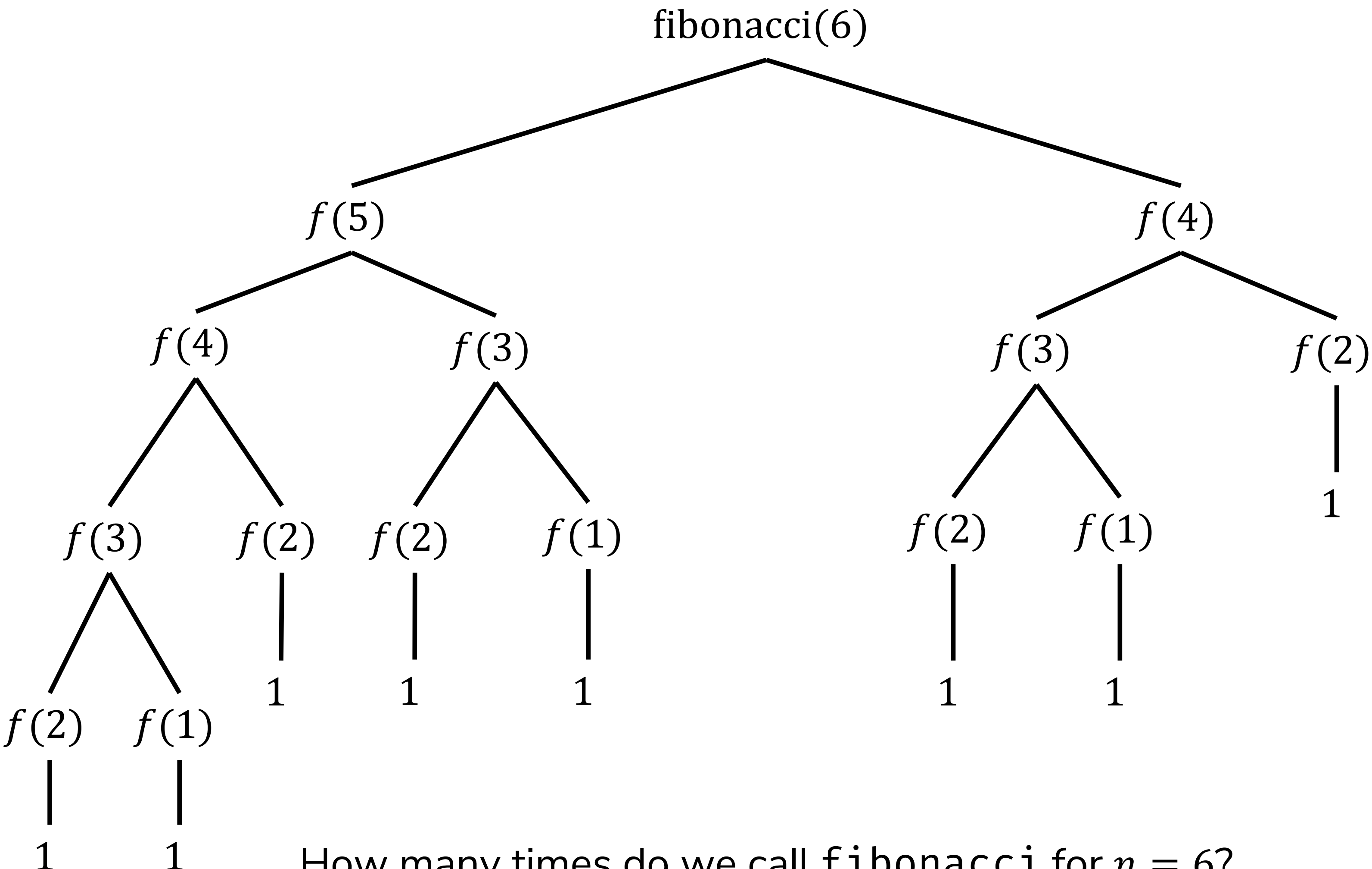
```
int fibonacci(int n) {
  if (n == 1 || n == 2) {
    return 1;
  }
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Cost: 1

Cost: ????

Everything inside `fibonacci` besides the recursive call is just $\mathcal{O}(1)$. This is also called “constant time” since it doesn’t grow as n grows.

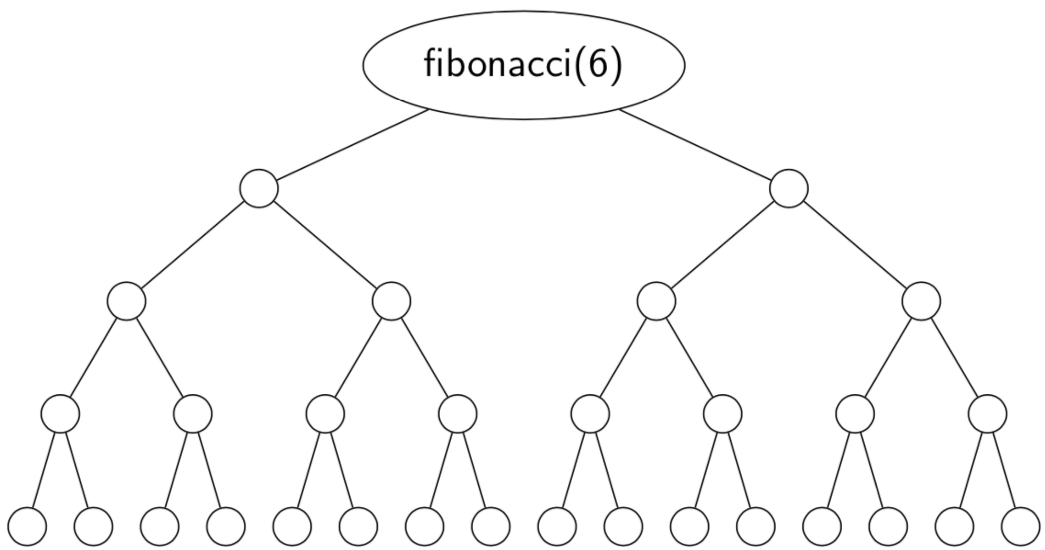
Time Analysis: Fibonacci



How many times do we call `fibonacci` for $n = 6$?

Relax!

- Let's relax this problem a bit.

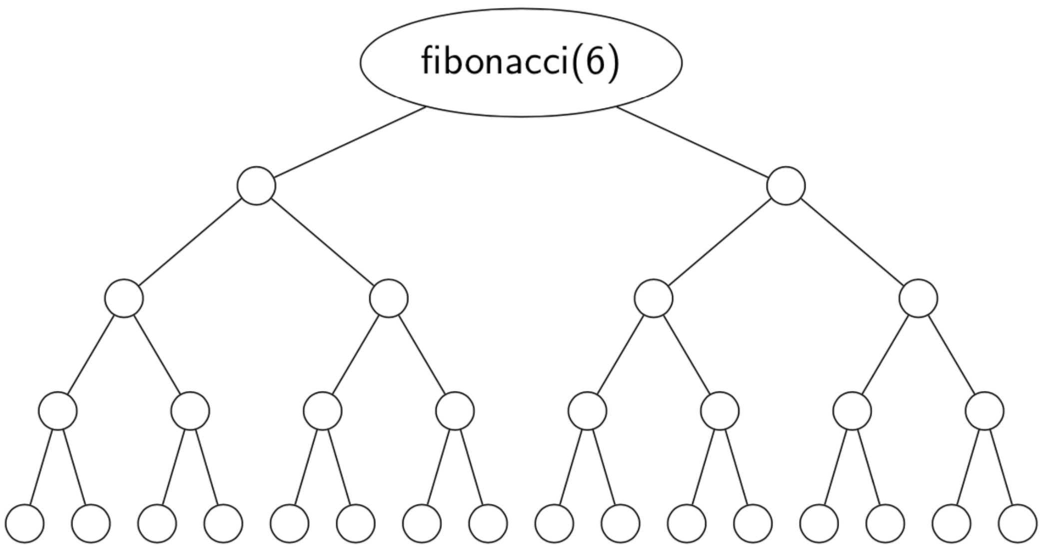


All I've done is filled in the missing nodes to make the tree "full"

- How many circles are there on this tree?
  - $\text{cost}(n) = ??? = 31$

Relax!

- Let's relax this problem a bit.

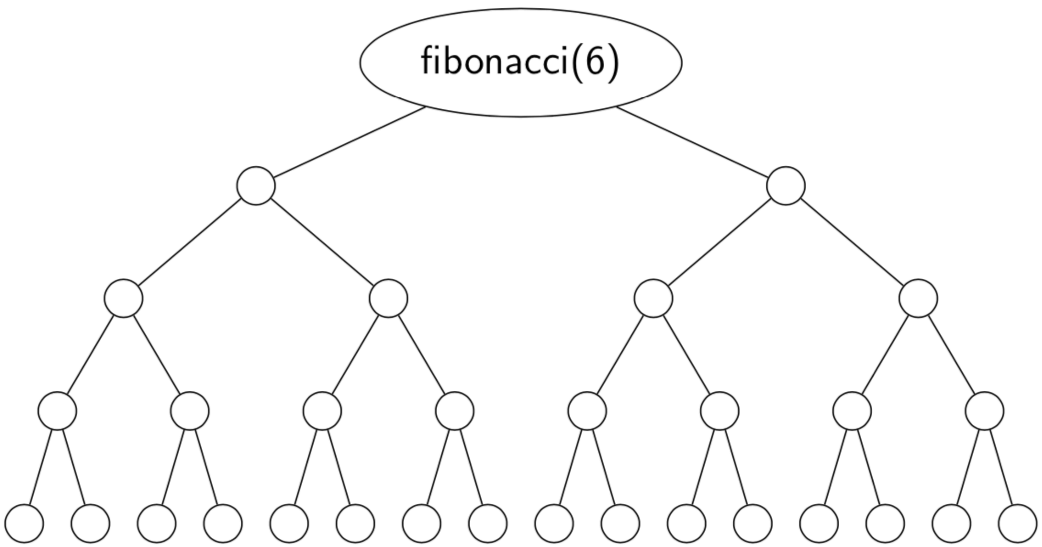


All I've done is filled in the missing nodes to make the tree "full"

- How many circles are there on this tree?
  - $\text{cost}(n) = 2^5 - 1 = 31$

Relax!

- Let's relax this problem a bit.

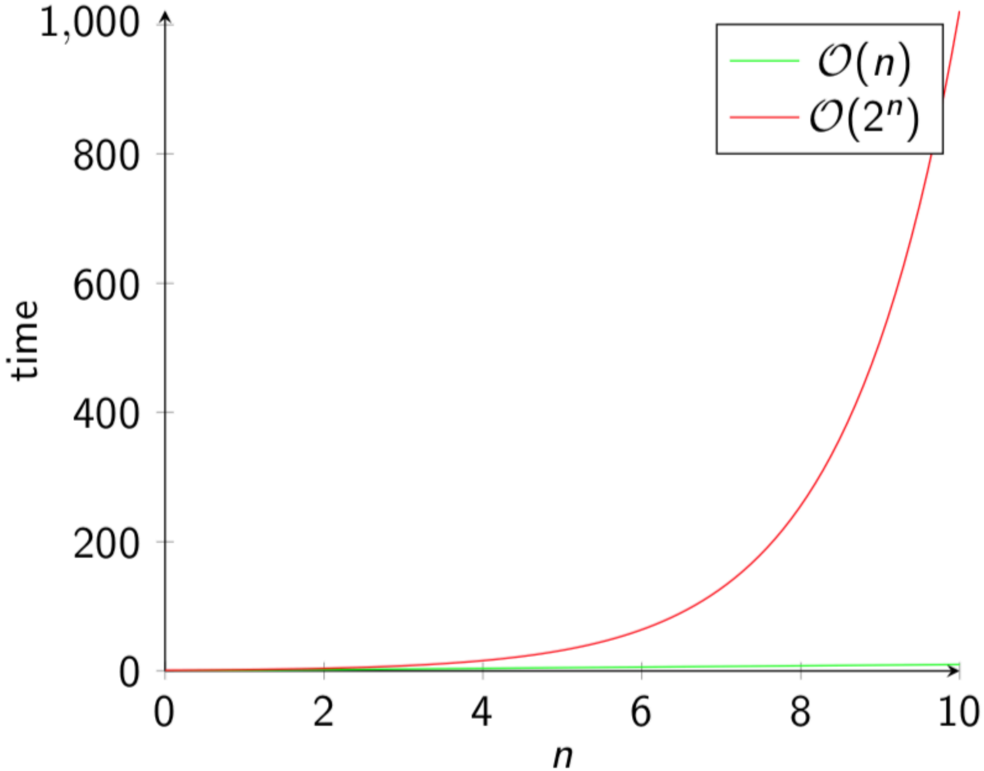


All I've done is filled in the missing nodes to make the tree "full"

- How many circles are there on this tree?
  - $\text{cost}(n) = 2^{n-1} - 1 = 31$
- So what's the time cost?
  - $\text{cost}(n) = 2^{n\cancel{-1}} \cancel{-1} \approx \mathcal{O}(2^n)$

Big oof...

- Remember that the time taken by SumPlus1 was $\mathcal{O}(n)$.



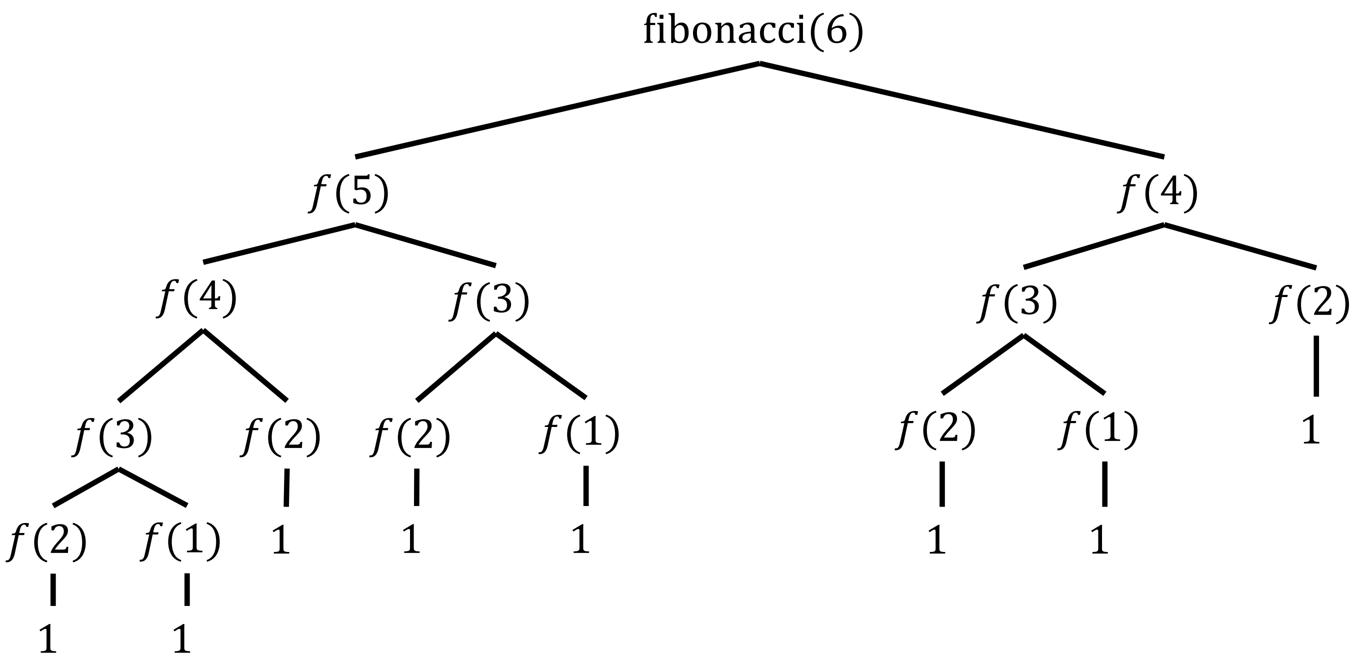
- Can we do better? (Yes!)

Outline

- What is algorithm analysis?
- How can we compare how long algorithms take?
- How can we “formalize” how long an algorithm takes?
- **How can we optimize our algorithms?**

Improving Fibonacci

- Which calculations here are redundant? **Most of them!**



By remembering the calculations we already performed, we can save a lot of time. $f(6)$ now only needs 6 function calls (not 15).

- This looks a lot more like $\mathcal{O}(n)$.

Speedy Fibonacci 🏎️

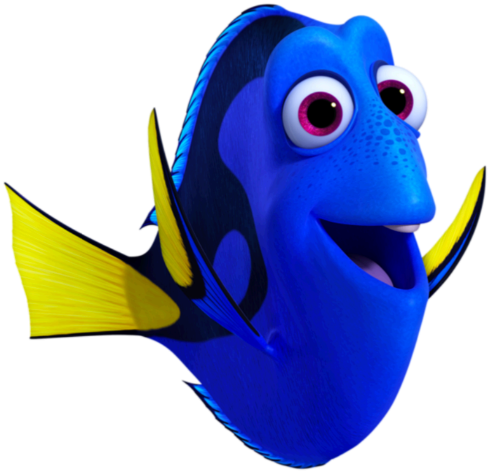
```
int fibonacci(int n) {
  if (isStored(n)) {
    return getStored(n);
  }
  if (n == 1 || n == 2) {
    return 1;
  }
  int fibN = fibonacci(n-1) + fibonacci(n-2);
  store(n, fibN);
  return fibN;
}
```

Assume that `isStored`, `getStored`, and `store` all have constant cost ($\mathcal{O}(1)$).

Now, we only need to compute each number once!

Memoization

- The programming technique of remembering previous calculations so we don't need to redo them every time.
  - As we saw with `fibonacci`, this can save a lot of time!



Who Cares???

- In The Real World™, most algorithms aren't as simple to optimize (or as bad when not optimized) as `fibonacci`.
- But some some applications, even small improvements can be helpful when n gets really large.
    - For Facebook, n (number of users) is ≈ 1 billion!
        - Want to generate a list of suggested friends? You'd better have a fast algorithm as a function of n.

Summary

- There are many ways we can analyze algorithms, such as for correctness.
- Analyzing the **time complexity** of an algorithm is useful for determining how long it will take when the input gets large.
  - Time complexity can be analyzed within your code using `millis()` to see how long a function takes to run.
  - It can also be analyzed by reasoning through the code and understanding how long each piece takes, then finding a cost function $\text{cost}(n)$ where n is the size of the input.
- Time complexity is expressed in “Big-Oh” notation, where we drop all the pieces of the cost function except the one that **grows the fastest**. We call the fastest-growing term **the order of growth**.