Complexity

Why is my program taking forever?

Sam Wolfson

CSE 120, Winter 2019

Computer Science In The News

Studies Keep Showing That the Best Way to Stop Piracy Is to Offer Cheaper, Better Alternatives

Study after study indicates that overly-aggressive anti-piracy efforts don't work, and the real solution lies in giving would-be pirates better, cheaper options.

motherboard.vice.com

Administrivia

- Portfolio Update 2 due **tonight**!
- Project Planning due Saturday (March 3rd)
- Tic-Tac-Toe due Saturday (March 3rd)

Innovation Exploration

- Mini-research project to let you explore a computing topic that is **interesting to you**
  - Pick a recent and relevant topic
  - Think of this as your "project" for the reading and writing portion of this course
- **Part 1**: Innovation Post (March 5th)
  - 4+ paragraphs, 550-750 words — posted to Canvas discussion board
  - Well-researched, insightful post, including 3+ citations
  - *Purpose, Effects and Impacts, Technical Aspects*
- **Part 2**: Respond to Posts (March 8th)
  - Comment on 3+ other students' posts

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 Correctness

- An algorithm is considered **correct** if, for every input, it:
  - reports the correct output,
  - doesn't run forever,
  - doesn't cause an error.
- Incorrect algorithms could run forever, or crash, or not return the correct answer.
  - But they can still be useful, e.g., for approximation.

Algorithm Analysis: Timing

- One way to analyze algorithms: **computation time**
  - How long does it take to run and finish its task?
  - We can use this to compare efficiency of two different algorithms that solve the same task.
- How to measure time?
  - Counting in my head
  - Stopwatch
  - Within the program itself

Timing in Processing

Definition: `millis()`

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 endTime = millis() - startTime;
    println("Took " + endTime + " ms to compute");
    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

- $\text{fibonacci}(1) = 1$
- $\text{fibonacci}(2) = 1$
- $\text{fibonacci}(3) = \text{fibonacci}(1) + \text{fibonacci}(2) = 1 + 1 = 2$
- $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

Code: Fibonacci

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

Let's start with a silly example.

Function: `SumPlus1`

- **Input**: an array of `int`s
- **Output**: the sum of all `int`s in the array, plus 1

Code: `SumPlus1`

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

How to analyze algorithmic time

Definition: Cost

The amount of time it takes to do something.

1. The cost of a "simple" line of code (i.e., no function calls or loops) is 1 "time."

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

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

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

3. The cost of a function is equal to the sum of the cost of all the lines of code within the function.

Analysis of our silly example

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

Analysis of our silly example

```
int sumPlus1(int[] array) {
    int sum = 0;  // cost: 1
    for (int i = 0; i < array.length; i = i + 1) {
        sum = sum + array[i];  // cost: 1
    }  // cost of loop: array.length * 1
    sum = sum + 1;  // cost: 1
    return sum;  // cost: 1
}
```

Overall Cost

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

$$\text{cost}(n) = 1 + (n * 1) + 1 + 1 = n + 3$$

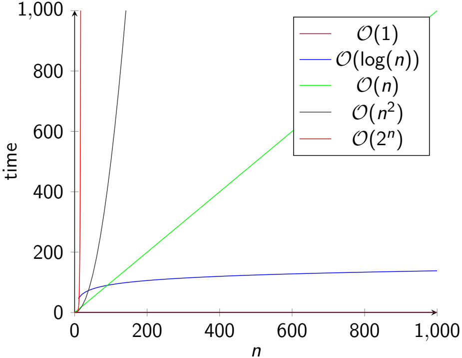
In computer science, we only really care about the part of this function that **costs the most**. In this case, the n term.
We call this "big-oh of n:" $\mathcal{O}(n)$.

Time Complexity

Definition: Time Complexity

The amount of time it takes to run an algorithm.

- The fastest-growing term in the cost function (the order of growth).
- Written in terms of the size n of the input (e.g., number of elements in an array, the n^{th} Fibonacci number) with "big-oh" notation.



Time Analysis: Fibonacci

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

Time Analysis: Fibonacci

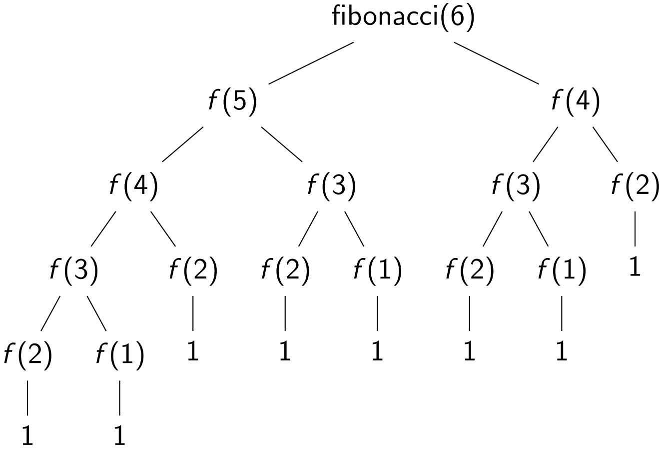
```
int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1; // cost: 1
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Time Analysis: Fibonacci

```
int fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1; // cost: 1
    } else {
        // cost: ????
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Everything inside `fibonacci` besides the recursive call is just $\mathcal{O}(1)$. This is sometimes referred to as "constant time" since it doesn't grow as n grows.

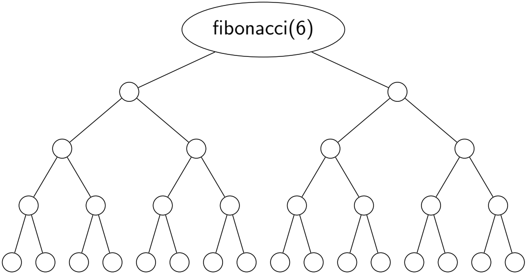
Time Analysis: Fibonacci



How many times do we end up calling `fibonacci` for $n = 6$?

Relax!

Let's relax this problem a little bit.



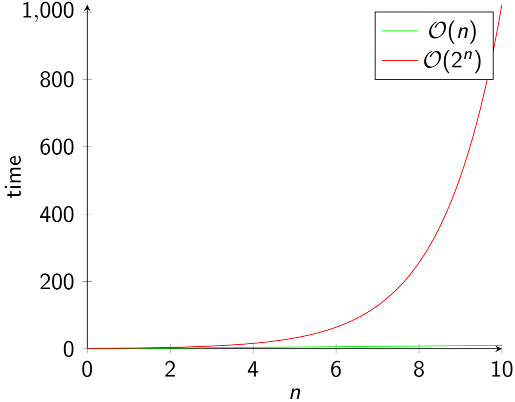
How many circles are there on this tree?

$\text{cost}(n) = ??? = 31$ $\text{cost}(n) = 2^5 - 1 = 31$ $\text{cost}(n) = 2^{n-1} - 1 = 31$

So what's the time cost? $\text{cost}(n) = 2^{n-1} - 1 \approx \mathcal{O}(2^n)$

Big oof. . .

Remember the time taken by our function `sumPlus1` was $\mathcal{O}(n)$.



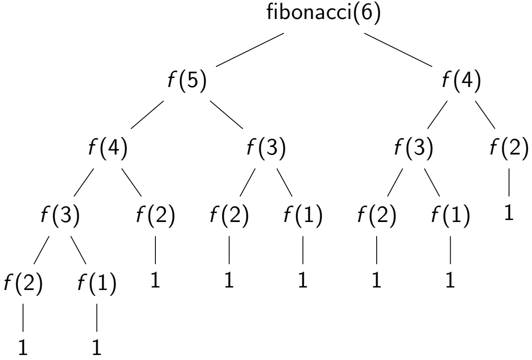
Can we do better? **Yes!**

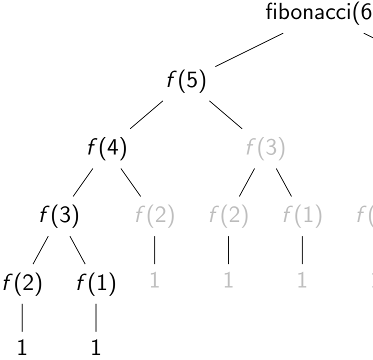
Outline

- 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

What calculations here are redundant?





By remembering the calculations we already performed, we can save a lot of time! $F(6)$ now only needs to make 6 total function calls (instead of 15). **This now looks a lot more like $\mathcal{O}(n)$.**

Speedy Fibonacci

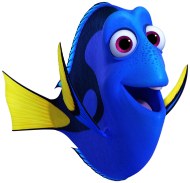
```
int fibonacci(int n) {
  if (isStored(n)) {
    return getStored(n);
  } else if (n == 1 || n == 2) {
    return 1;
  } else {
    int fibN = fibonacci(n-1) + fibonacci(n-2);
    store(n, fibN);
    return fibN;
  }
}
```

Assume that `isStored`, `getStored`, and `store` all have constant cost ($\mathcal{O}(1)$). Now, we only compute each number once.

Memoization

Definition: Memoization

The programming technique of remembering previous calculations so we don't need to recalculate them every time.



- As we saw with `fibonacci`, this can save a lot of time!

Who cares???

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

Summary

- There are many different 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 of the pieces of the cost function except the one that **grows the fastest**. We call the fastest-growing term the **order of growth**.