CSE 101: Introduction to Algorithms

Professor: Daniel Kane
dakane@ucsd.edu

Course webpage:
http://cseweb.ucsd.edu/~dakane/CSE101/

Lecture Zoom Meeting Link:
https://ucsd.zoom.us/j/93843292201?pwd=a1FDQS9Da0ZWWjRKUnZ3a3FQWHo5QT09

Basic Logistical Information

Course Technology Guide:

http://cseweb.ucsd.edu/~dakane/CSE101/Technology.pdf

Course Syllabus:

http://cseweb.ucsd.edu/~dakane/CSE101/Syllabus.pdf

Practice Quiz

What is your favorite number?

(A) 0

(B) e

(C) π

(D) 17

(E) 154

Office Hours

Daniel Kane: Thursday and Friday 2:30-4:00pm or by appointment

https://ucsd.zoom.us/my/dankane

TAs:

Jiabei Han:Monday, Wednesday, Friday 4:00-5:00pm pacific over zoom at https://ucsd.zoom.us/j/92571674513.

Vaishakh Ravindrakumar:Monday, Wednesday, Friday 11:00am-12:00pm pacific over zoom at https://ucsd.zoom.us/j/7577412678.

Manish Kumar Singh:Tuesday 4:00-6:00pm and Thursday 5:00-6:00pm pacific over zoom at https://ucsd.zoom.us/j/9029365896.

Chutong Yang:Tuesday 8:00-9:00pm and Thursday 7:00-9:00pm pacific over zoom at https://ucsd.zoom.us/s/5785340529.

Tutor:

Harrison Matt Ku:Tuesday, Thursday 1:00-2:30pm pacific over zoom at https://ucsd.zoom.us/my/harrisonku.

Introduction

- What kinds of problems will we consider in this course?
- Fibonacci numbers.
- Asymptotic Runtimes.
- Levels of algorithm design.

Straightforward Programming Problems

- Display text
- Copy a file
- Count number of occurrences of a given word

Each has a straightforward algorithm that is hard to improve upon.

Algorithms Problems

- Find a shortest path in a city
- Find the best pairing of students to dorm rooms
- Find the best schedule of classes

These problems are

- Well defined mathematically
- Still not easy to solve

AI Problems

- Image recognition
- Game playing
- Understanding natural language

For these problems, much of the difficulty is in formalizing exactly what you are trying to do.

This Class

- We will focus on algorithms problems

Problem: Fibonacci Numbers

Definition:

The Fibonacci numbers are the sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…

Defined by

$F_0 = F_1 = 1$

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

Problem: Given n, compute F_n.

Naïve Algorithm

There is an easy recursive algorithm.

```
Fib(n)
  If n ≤ 1
    Return 1
  Else
    Return Fib(n-1)+Fib(n-2)
```

Essentially turned definition into an algorithm.

Runtime

Lets compute T(n) = number of lines of code `Fib`(n) needs to execute.

```
Fib(n)
  If n ≤ 1
    Return 1
  Else
    Return Fib(n-1)+Fib(n-2)
```

2 lines

1+T(n-1)+T(n-2) lines

$$T(n) = \begin{cases} 2 & \text{if } n \leq 1 \\ T(n-1)+T(n-2)+3 & \text{else} \end{cases}$$

Question: Runtime

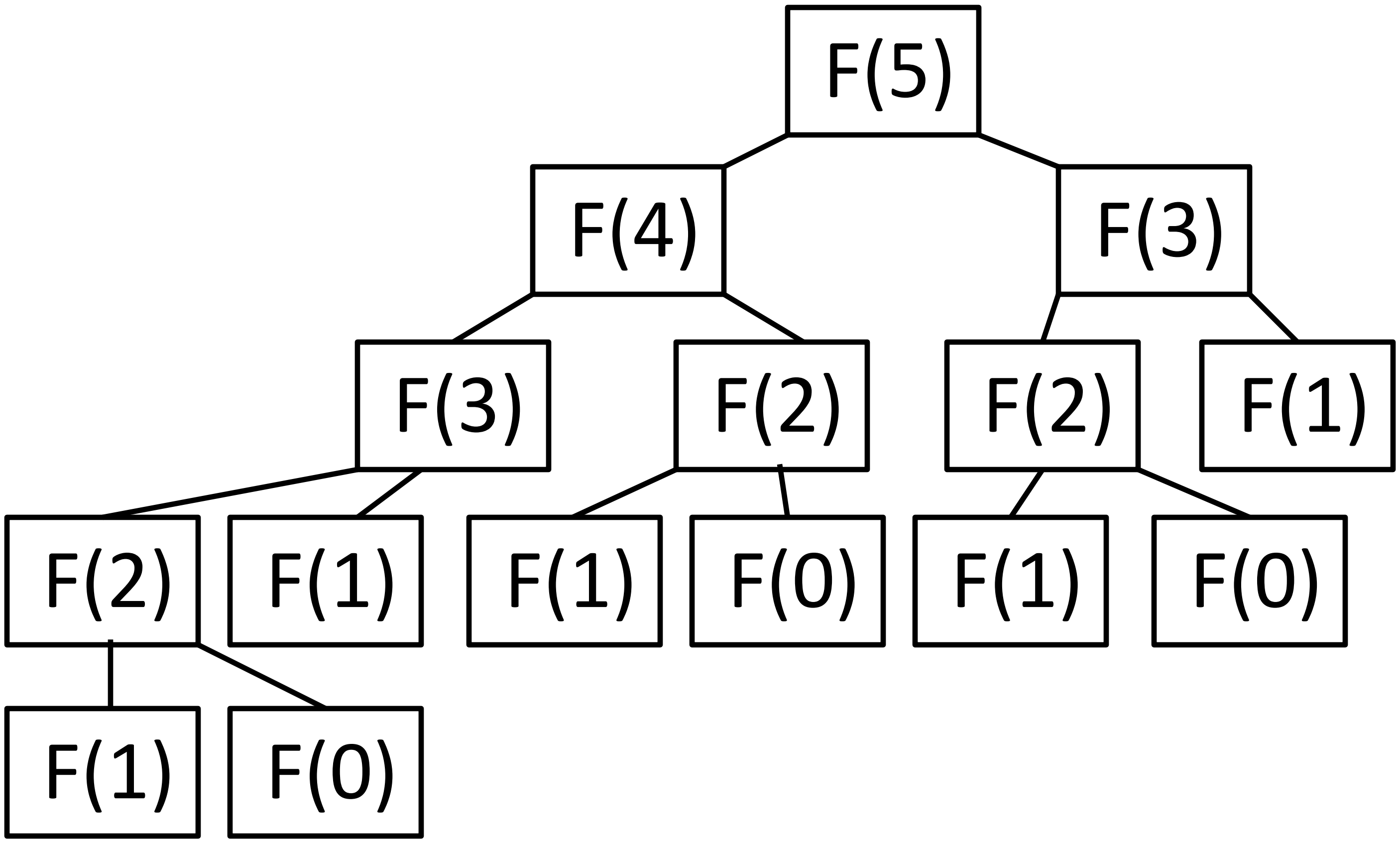
If your computer executes a billion lines of code per second, approximately how long does it take to compute F(100)?

A) A millisecond
B) A second
C) A year
D) 100,000 years
E) Forever

$T(100) \approx 2.87 \cdot 10^{21}$
At a billion lines of code per second, this is just over 90,000 years.

Why So Slow?

Too many recursive calls.



How do we improve this?

Avoid having to recompute things.

How would you do it by hand?

$F_0 = 1$

$F_1 = 1$

$F_2 = 2$

$F_3 = 3$

$F_4 = 5$

As long as you have records of the previous answers, you can just write down the next one.

Improved Algorithm

Simulate the above idea using array A to store values A[n] = F_n.

$T(n) = 2n+1$

```
Fib2(n)
  Initialize A[0..n]
  A[0] = A[1] = 1
  For k = 2 to n
     A[k] = A[k-1] + A[k-2]
  Return A[n]
```

2 lines

2(n-1) lines

1 line

Runtime

With the new algorithm T(100) = 201. Easily runable on almost any computer.

The power of algorithms: Sometimes the right algorithm is the difference between something working and not finishing in your lifetime.

Question: Runtime

Is $T(n) = 2n+1$ a good description of this program's runtime?

A) Yes
B) No

```
Fib2(n)
  Initialize A[0..n]
  A[0] = A[1] = 1
  For k = 2 to n
     A[k] = A[k-1] + A[k-2]
  Return A[n]
```

2 lines

2(n-1) lines

1 line

Discussion of Runtimes

Is $T(n) = 2n+1$ really an accurate description of that program's runtime?

- Is initializing an array one operation or several?
- What about adding large integers?
- Should we count machine ops?
  - Doesn't this depend on implementation?

Bottom Line

What we really care about is how long it takes program to run on a real machine.

Unfortunately, this depends on:

- CPU speed
- Memory architecture
- Compiler optimizations
- Background processes

Too much to consider for every analysis

Asymptotic Analysis

- These issues usually just constant factors.
- If we analyze runtime in a way that *ignores* constant factors (like big-O), we don't have to deal with them.
- But ignoring constant factors 1 second is the same as 100,000 years.
- On the other hand, we can still compare things *asymptotically*. A $\Theta(n)$ algorithm will beat an $\Theta(n^2)$ algorithm for n large enough.

Advantages and Disadvantages of Asymptotic Analysis

Disadvantages:

- Cannot tell you whether algorithm is practical on given inputs.
- Ignores constant factor runtime improvements which are important in practice.

Advantages:

- Independent of hardware and implementation.
- Allows you to compare behavior on sufficiently large inputs.
- *Usually* an algorithm with better asymptotic behavior will do better in practice (though there are notable exceptions).

Because of this, this class will almost exclusively use big-O analysis.