Overview

CSE 101: Design and Analysis of Algorithms

Lecture 1

CSE 101: Design and analysis of algorithms

- Course overview
- Logistics

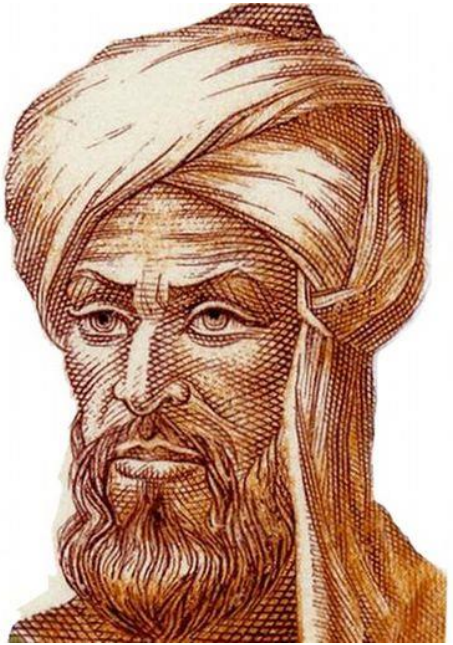
First, relevant prerequisites

- Advanced Data Structures (CSE 100)
  - Mathematics for Algorithm and Systems (CSE 21)
    - Introduction to Discrete Mathematics (CSE 20)
- You can describe algorithms and prove their correctness using precise mathematical terminology and techniques. For example:
  - Basic math notation (logic, sets, functions, etc.)
  - Proofs (contradiction, induction, etc.)
  - Asymptotic notation (big O, etc.)
  - Pseudocode
  - Basic counting strategies
  - Graphs and special types of graphs

Based on slides courtesy of Miles Jones

Algorithm

- Muhammad ibn Musa al-Khwarizmi
- Authored *The Compendious Book on Calculation by Completion and Balancing*
- The terms Algorism and algorithm are based on his name



What is an algorithm?

- An algorithm is a method for solving a problem
- What are some examples of non-computer science algorithms?

Course objectives

- Introduction to the design and analysis of algorithms
  - Going beyond the obvious algorithm
- Problem solving

What is an algorithm?

- Procedure for performing a computation
- Broken into a well-specified series of steps
- Each step is completely determined
- Input: instance X
- Output: solution Y
- Both X and Y should be finitely describable
- N = number of bits to describe input

Questions about algorithms

- Does the algorithm work?
- Does the algorithm terminate?
- How many operations does the algorithm take?
- How much memory does it require?

Algorithm time

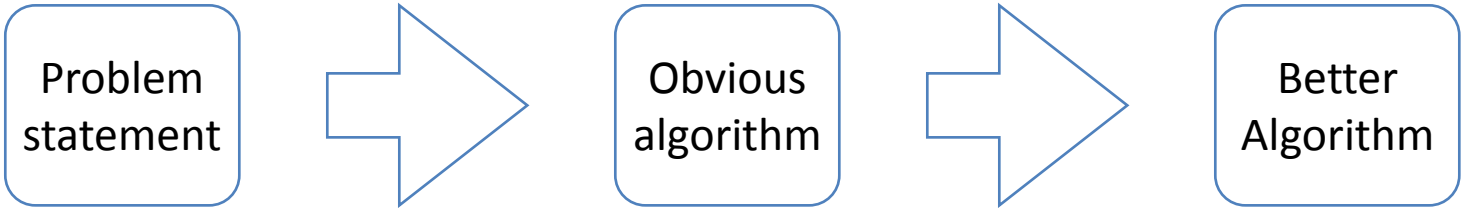
- Why do we care about algorithm time?
  - Computers are fast. Don't they have plenty of time to solve problems?
  - When do we need to worry about the time algorithms take?

Algorithm time

- When is algorithm time important?
  - When an algorithm is used billions of times
    - Examples: sorting, searching, data structures, encryption, data compression
  - When a problem seems to be really hard to solve
    - Examples: optimization, cryptanalysis
  - When the input is really big (i.e., “big data”)
  - Many of the most important problems are about technology
    - Examples: compilers, VLSI, networking, databases
  - As computers and networks get faster, they also get more complex, which means instances get larger

Course objective

Beyond the obvious



While most problems have an obvious algorithm implicit in their statement, one object of this class is going beyond the obvious algorithm and getting to a better algorithm. Usually, this means "asymptotically faster"

Hierarchy of obviousness

- Obvious algorithms: Implicit in the problem statement
- Methodical algorithms: Applying general principles and paradigms that improve algorithms for a wide variety of problems
- Clever algorithms: Stretching the general paradigms in a way to best fit a particular problem
- Miraculous algorithms: How did anyone ever think of this?

- 80% of this class: How to use the general paradigms to go from the obvious to the methodical algorithm
- 20% of this class: Appreciating some clever or even miraculous algorithms

The most useful algorithm design methods

- Graph search: First two weeks
- Using data structures: Mixed between graph search and greedy
- Greedy algorithms: Third and fourth week
- Divide and conquer: Fifth and sixth week
- Dynamic programming: Seventh and eighth week
- Reducing to a previously solved problem: throughout and again in P vs. NP
- Iterative improvement (e.g., network flow): time permitting

Themes

- Cut across different methods
- General principles that apply to many situations
  - Reuse computation, do not repeat
  - Balance costs
  - Only track what changes
  - Keep your options open

Fibonacci sequence

- Leonardo Bonacci introduced many of the ideas from Al-Khwarizmi's work to Europe
- Leonardo Bonacci (Fibonacci) is mostly known for the sequence of integers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - Each number is the sum of the previous two numbers

Fibonacci sequence definition

- $F(1) = 1$
- $F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$

Fibonacci sequence, algorithm 1

- Obvious algorithm

function fib1(*n, positive integer*)
if $n = 1$ then return 1
if $n = 2$ then return 1
return fib1(n-1) + fib1(n-2)

- How long does it take?
- Can we do better?

Fibonacci sequence, algorithm 1

function fib1(*n*)

if *n* = 1 then return 1

if *n* = 2 then return 1

return fib1(*n*-1) + fib1(*n*-2)

- Let T(n) be the number of computer steps it takes to calculate fib1(*n*)

Fibonacci sequence, algorithm 1

function fib1(*n*)

if *n* = 1 then return 1

if *n* = 2 then return 1

return fib1(*n*-1) + fib1(*n*-2)

- Let T(n) be the number of computer steps it takes to calculate fib1(*n*)

If $n < 3$ then $0 < T(n) < 3$

If $n > 3$ then $T(n) > T(n-1) + T(n-2)$

So we have that $T(n) > F(n)$

Fibonacci numbers grow fast!

$F(n) \sim 1.6^n$

Computer speed

- A very fast computer can do about 2^45 = 3,518,372,088,832 calculations in one second

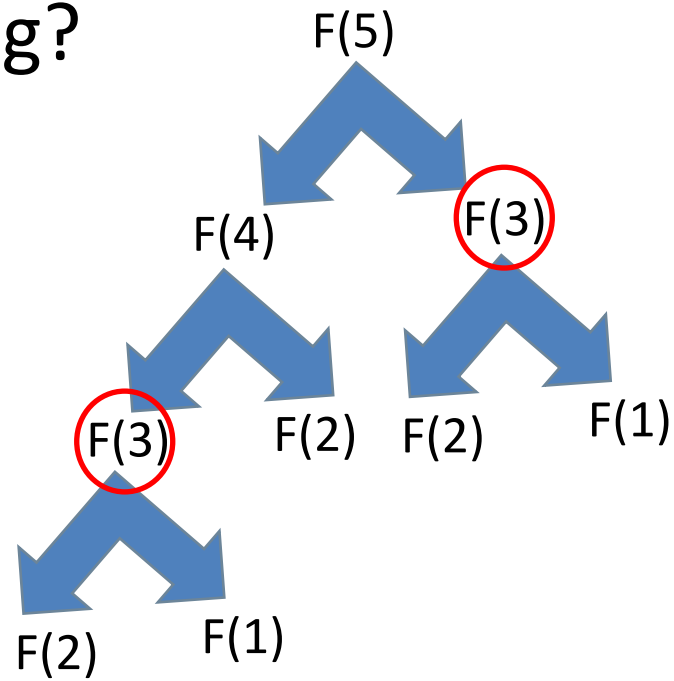
- 2^45 ~ 1.6^66, so it would take this very fast computer one second to calculate F(66)

Computer speed

Fibonacci number	Time for fastest computer
F(66)	1 second
F(75)	1 minute
F(85)	1 hour
F(95)	1 week
F(100)	3 months
F(105)	2 and a half years
F(110)	27 years
F(125)	Cave paintings
F(150)	Age of the Universe

Fibonacci sequence, algorithm 1

- Why does it take so long?
  - Recomputing



General principle: store and re-use

- If an algorithm is **recomputing** the same thing many times, we should **store and re-use** instead of recomputing
- Basis for dynamic programming, but also comes up in data structures

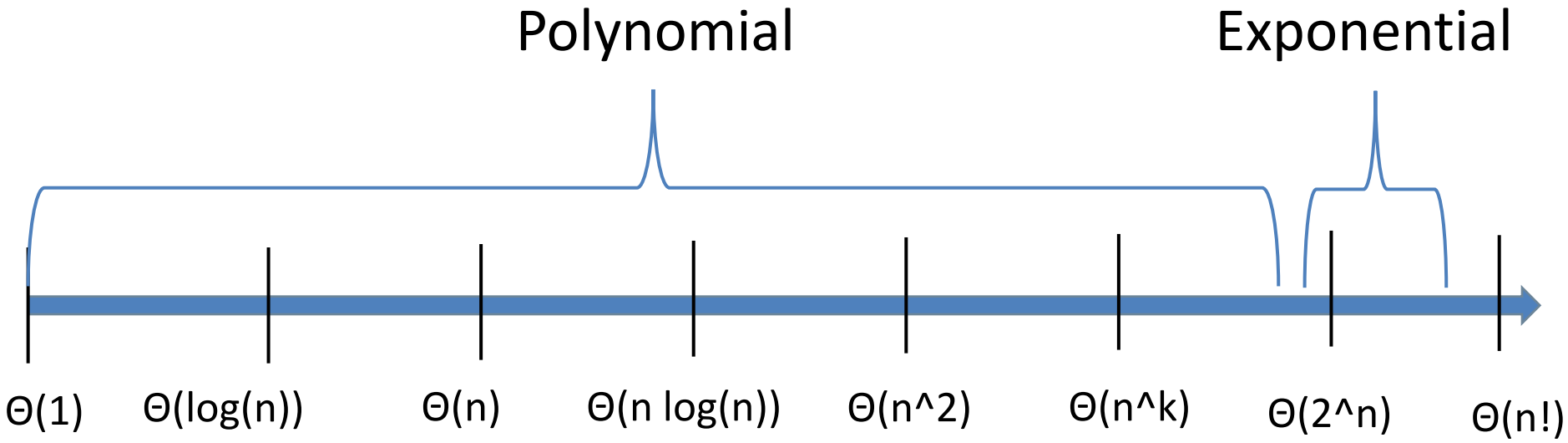
Fibonacci sequence, algorithm 2

function fib2(*n*)
if *n* = 1 then return 1
if *n* = 2 then return 1
create array f[1…n]
f[1] := 1
f[2] := 1
for i =3 … n:
        f[i] := f[i-1] + f[i-2]
return f[n]

The **for** loop consists of a single computer step, so in order to compute f[*n*], you need *n* – 1 + 2 computer steps!

This is a huge improvement: *linear time (O(n)) vs. exponential time (O(1.6^n))*

Exponential time vs. polynomial time



Basic computer steps

- Primitives
  - Branching
  - Storing
  - Comparing (small numbers)
  - Simple addition (small numbers)
  - Array look up
- Are we cheating?

Constant time

- A constant is a fixed number
  - 2 is a constant. The time an algorithm takes on a specific input such as 1 is a constant. The time for a chip to perform a hardware instruction is usually a constant.
- A constant does not depend on the input
  - “The maximum value in an array” is not a constant, though it is also not a function of the size of the array
- When should operations such as addition, multiplication, and indirect addressing be viewed as constant time?

Computer architecture and time

- Most computer architectures have a “word size”, e.g., 32 bits, 64 bits, or similar
- Memory locations hold this amount of information and the name of a memory location must be less than the word size
- The CPU is designed to process instructions on word sized inputs
- Operations on inputs less than word size are performed on the CPU in a single access; larger inputs need to be broken into word size chunks

Model operations as constant time

- We assume elements in an array fit in a single register, so are less than word size. As such, operations on single array elements are constant time.
- Array indices need to be memory locations, so operations on these are constant time
- If we do not require perfect precision, we can use floating point arithmetic, which returns a word size number of bits of accuracy, so is constant time

Addition

- When we add two numbers that fit in word size, such as indices in an array or elements of an array, we can view this as constant time
- When we are using floating point, we can view addition as constant time
- But, when we need to add two n bit numbers, where n is larger than the word size, we need to perform the addition in software. This takes linear time in n, O(n) time.

Fibonacci example

- While we can assume input n fits in one word, we should not assume Fib(n) does
- n fits in one word means log n < w or n < 2^w, not n < w
- Fib(n) is about 1.6^n, which grows quickly
- For example, Fib(50) = 12,586,269,025
  - How many bits do we need to write this down?
  - How many bits for Fib(n)?

Runtime of fib2

- The procedure **fib2** takes O(n) additions. But each addition involves integers of size O(n) bits each.
- If we use floating point, and only approximate the value, then the total time will be O(n)
- But, if we need exact precision, then each of n additions takes O(n) time so the runtime of **fib2** is O(n^2)

Runtime of fib1 and fib2

- The n-th Fibonacci number is F(n)<2^n so it has less than n bits and
  - the procedure **fib1** takes O(2^n) additions and each addition takes O(n) time so the runtime of **fib1** is O(n*2^n)
  - the procedure **fib2** takes O(n) additions and each addition takes O(n) time so the runtime of **fib2** is O(n*n)

Big O, big Ω, and big Θ

- $f(n) = O(g(n))$ means that there exists a constant c such that $f(n) \leq cg(n)$ for all n large enough
- $f(n) = \Omega(g(n))$ means that $g(n) = O(f(n))$
- $f(n) = \Theta(g(n))$ means that $f(n) = O(g(n))$ and $g(n) = O(f(n))$

Quick rules for big O

- Any polynomial is big O of its highest power
- Exponentials dominate polynomials
- If f/g goes to 0 then $f \in O(n)$: f grows slower than g
- If f/g goes to infinity then $f \in \Omega(n)$: f grows faster than g
- If f/g goes to c>0 then f is Θ(g), i.e., both O(g) and Ω(g), f and g are of the same order of growth

How to approach problems

- When can we use an algorithm developed for one problem to solve another?
- Modifying algorithms vs. using algorithms in reductions
- Defining problems precisely

Next lecture

- We'll start with a familiar algorithm (graph search) and try to re-use it for a new problem (max bandwidth path)

Logistics

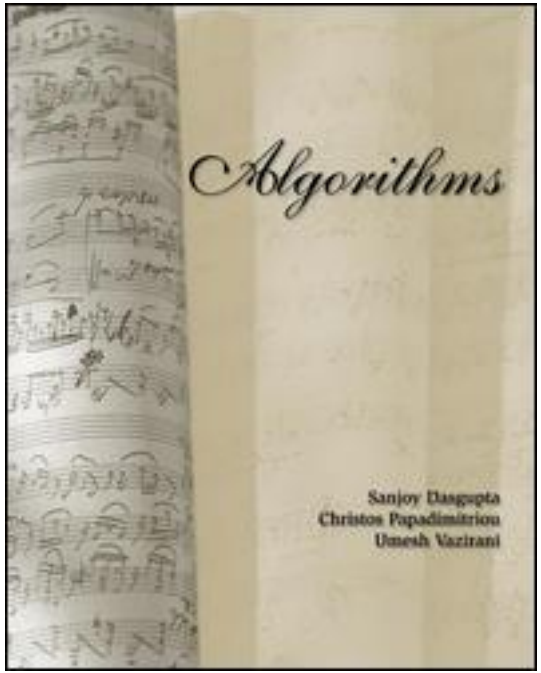
- Instructor: Ben Ochoa
- TAs: Yiyuan Ma, Harsh Lal, Waquar Ahmad, and Sameeksha Khillan
- Tutors: Toan Bui and Eric Liu
- Course website
  - https://cseweb.ucsd.edu/classes/fa18/cse101-b/
- 20 lecture meetings
  - 1 university holiday (Nov 22)
- Discussion sections on Fridays (not meeting tomorrow)
- Class discussion
  - Piazza

Logistics

- Grading
  - 7 homework assignments (drop lowest one; 30% of grade)
  - 3 quizzes (each 10% of grade)
  - Final exam (40% of grade, or 50% of grade and drop lowest quiz, whichever results in maximum grade)
  - Piazza
    - Ask (and answer) questions using Piazza, not email
    - Good participation could raise your grade (e.g., raise a B+ to an A-)

Textbook

- Algorithms
  - Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani



Collaboration policy

You are encouraged to collaborate. As such, homework assignments will be completed in groups of size 1-4. However, collaboration or copying on exams of any kind is not allowed.

Academic integrity policy

Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual or group to whom it is assigned, without unauthorized aid of any kind.

Outside resources

- You should not attempt to search for homework solutions online or in sources outside of the course text. If you accidentally stumble upon a homework solution in an outside source you must cite it in your homework solution. If your solution proves to be too similar to the cited one, you may lose credit on the problem; however, failure to cite the other solution will be treated as an academic integrity violation.

Academic integrity violation

If the work you submit is determined to be other than your own or your group, you will be reported to the Academic Integrity Office for violating UC San Diego's Policy on Integrity of Scholarship. In accordance with the CSE department academic integrity guidelines, ***students found committing an academic integrity violation will receive an F in the course***.