

# CS 561, Pre Lecture 1

Jared Saia  
University of New Mexico

# Today's Outline

- Background
- Asymptotic Analysis

# Why study algorithms?

*“Seven years of College down the toilet” - John Belushi in Animal House*

Can you get a job without knowing algorithms? Yes, but:

- You won't understand why software systems work the way they do
- You won't learn the fundamentals of systematic thinking (aka computation/problem solving)
- You'll have less fun!

## Why study algorithms? (II)

- Almost all big companies want programmers with knowledge of algorithms: Google, Facebook, Amazon, Oracle, Yahoo, Sandia, Los Alamos, etc.
- In most programming job interviews, they will ask you several questions about algorithms and/or data structures
- Your knowledge of algorithms will set you apart from the large masses of interviewees who know only how to program
- If you want to start your own company: many startups are successful because they've found better algorithms for solving a problem (e.g. Google, OpenAI, Akamai, etc.)

## Why Study Algorithms? (III)

- You'll improve your research skills in almost any area
- You'll write better, faster code
- You'll learn to think more abstractly and mathematically
- It's one of the most challenging and interesting area of CS!

# A Real Job Interview Question

The following is a real job interview question (thanks to Maksim Noy):

- You are given an array with integers between 1 and 1,000,000.
- All integers between 1 and 1,000,000 are in the array at least once, and one of those integers is in the array twice
- Q: Can you determine which integer is in the array twice?  
Can you do it while iterating through the array only once?

# Solution

- Ideas on how to solve this problem?? What if we allowed multiple iterations?

# Naive Algorithm

- Create a new array of ints between 1 and 1,000,000, which we'll use to count the occurrences of each number. Initialize all entries to 0
- Go through the input array and each time a number is seen, update its count in the new array
- Go through the count array and see which number occurs twice.
- Return this number



# Naive Algorithm Analysis

- Q: How long will this algorithm take?
- A: We iterate through the numbers 1 to 1,000,000 *three* times!
- Note that we also use up a lot of space with the extra array
- This is wasteful of time and space, particularly as the input array gets very large (e.g. it might be a huge data stream)
- Q: Can we do better?

## Ideas for a better Algorithm

- Note that  $\sum_{i=1}^n i = (n + 1)n/2$
- Let  $S$  be the sum of the input array
- Let  $x$  be the value of the repeated number
- Then  $S = (1,000,000 + 1)1,000,000/2 + x$
- Thus  $x = S - (1,000,000 + 1)1,000,000/2$

## A better Algorithm

- Iterate through the input array, summing up all the numbers, let  $S$  be this sum
- Let  $x = S - (1,000,000 + 1)1,000,000/2$
- Return  $x$

# Analysis

- This algorithm iterates through the input array just once
- It uses up essentially no extra space
- It is at least three times faster than the naive algorithm
- Further, if the input array is so large that it won't fit in memory, this is the only algorithm which will work!
- These time and space bounds are the best possible

## Take Away

- Designing good algorithms matters!
- Not always this easy to improve an algorithm
- However, with some thought and work, you can *almost always* get a better algorithm than the naive approach

# How to analyze an algorithm?

- There are several resource bounds we could be concerned about: time, space, communication bandwidth, logic gates, etc.
- However, we are usually most concerned about time
- Recall that algorithms are independent of programming languages and machine types
- Q: So how do we measure resource bounds of algorithms

# Random-access machine model

- We will use RAM model of computation in this class
- All instructions operate in serial
- All basic operations (e.g. add, multiply, compare, read, store, etc.) take unit time
- All “atomic” data (chars, ints, doubles, pointers, etc.) take unit space

# Worst Case Analysis

- We'll generally be pessimistic when we evaluate resource bounds
- We'll evaluate the run time of the algorithm on the worst possible input sequence
- Amazingly, in most cases, we'll still be able to get pretty good bounds
- Justification: The “average case” is often about as bad as the worst case.



## Example Analysis

- Let's consider the more general problem of the duplicate number problem, where the numbers are 1 to  $n$  instead of 1 to 1,000,000

# Algorithm 1

- Create a new “count” array of ints of size  $n$ , which we’ll use to count the occurrences of each number. Initialize all entries to 0
- Go through the input array and each time a number is seen, update its count in the “count” array
- As soon as a number is seen in the input array which has already been counted once, return this number

## Algorithm 2

- Iterate through the input array, summing up all the numbers, let  $S$  be this sum
- Let  $x = S - (n + 1)n/2$
- Return  $x$

## Example Analysis: Time

- Worst case: Algorithm 1 does  $5 * n$  operations ( $n$  inits to 0 in “count” array,  $n$  reads of input array,  $n$  reads of “count” array (to see if value is 1),  $n$  increments, and  $n$  stores into count array)
- Worst case: Algorithm 2 does  $2 * n + 4$  operations ( $n$  reads of input array,  $n$  additions to value  $S$ , 4 computations to determine  $x$  given  $S$ )

## Example Analysis: Space

- Worst Case: Algorithm 1 uses  $n$  additional units of space to store the “count” array
- Worst Case: Algorithm 2 uses 2 additional units of space

## A Simpler Analysis

- Analysis above can be tedious for more complicated algorithms
- In many cases, we don't care about constants.  $5n$  is about the same as  $2n + 4$  which is about the same as  $an + b$  for any constants  $a$  and  $b$
- However we do still care about the difference in space:  $n$  is very different from  $2$
- Asymptotic analysis is the solution to removing the tedium but ensuring good analysis

# Asymptotic analysis?

- A tool for analyzing time and space usage of algorithms
- Assumes input size is a variable, say  $n$ , and gives time and space bounds as a function of  $n$
- Ignores multiplicative and additive constants
- Concerned only with the *rate* of growth
- E.g. Treats run times of  $n$ ,  $10,000 * n + 2000$ , and  $.5n + 2$  all the same (We use the term  $O(n)$  to refer to all of them)

## What is Asymptotic Analysis?(II)

- Informally,  $O$  notation is the leading (i.e. quickest growing) term of a formula with the coefficient stripped off
- $O$  is sort of a relaxed version of " $\leq$ "
- E.g.  $n$  is  $O(n)$  and  $n$  is also  $O(n^2)$
- By convention, we use the smallest possible  $O$  value i.e. we say  $n$  is  $O(n)$  rather than  $n$  is  $O(n^2)$



## More Examples

- E.g.  $n$ ,  $10,000n - 2000$ , and  $.5n + 2$  are all  $O(n)$
- $n + \log n$ ,  $n - \sqrt{n}$  are  $O(n)$
- $n^2 + n + \log n$ ,  $10n^2 + n - \sqrt{n}$  are  $O(n^2)$
- $n \log n + 10n$  is  $O(n \log n)$
- $10 * \log^2 n$  is  $O(\log^2 n)$
- $n\sqrt{n} + n \log n + 10n$  is  $O(n\sqrt{n})$
- $10,000$ ,  $2^{50}$  and  $4$  are  $O(1)$

## More Examples

- Algorithm 1 and 2 both take time  $O(n)$
- Algorithm 1 uses  $O(n)$  extra space
- But, Algorithm 2 uses  $O(1)$  extra space

## Formal Defn of Big-O

- A function  $f(n)$  is  $O(g(n))$  if there exist positive constants  $c$  and  $n_0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$

## Example

- Let's show that  $f(n) = 10n + 100$  is  $O(g(n))$  where  $g(n) = n$
- We need to give constants  $c$  and  $n_0$  such that  $0 \leq f(n) \leq cg(n)$  for all  $n \geq n_0$
- In other words, we need constants  $c$  and  $n_0$  such that  $10n + 100 \leq cn$  for all  $n \geq n_0$

## Example

- We can solve for appropriate constants:

$$10n + 100 \leq cn \quad (1)$$

$$10 + 100/n \leq c \quad (2)$$

- So if  $n > 1$ , then  $c$  should be greater than 110.
- In other words, for all  $n > 1$ ,  $10n + 100 \leq 110n$
- So  $10n + 100$  is  $O(n)$

# Questions

Express the following in  $O$  notation

- $n^3/1000 - 100n^2 - 100n + 3$
- $\log n + 100$
- $10 * \log^2 n + 100$
- $\sum_{i=1}^n i$

# Relatives of big-O

The following are relatives of big-O:

$O$	" $\leq$ "
$\Theta$	" $=$ "
$\Omega$	" $\geq$ "
$o$	" $<$ "
$\omega$	" $>$ "

## Formal Defns

- $O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$
- $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$
- $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$



## Formal Defns (II)

- $o(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exists } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$
- $\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0 \text{ there exists } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

# Relatives of big-O

When would you use each of these? Examples:

- $O$  “ $\leq$ ” This algorithm is  $O(n^2)$  (i.e. worst case is  $\Theta(n^2)$ )
- $\Theta$  “ $=$ ” This algorithm is  $\Theta(n)$  (best and worst case are  $\Theta(n)$ )
- $\Omega$  “ $\geq$ ” Any comparison-based algorithm for sorting is  $\Omega(n \log n)$
- $o$  “ $<$ ” Can you write an algorithm for sorting that is  $o(n^2)$ ?
- $\omega$  “ $>$ ” This algorithm is not linear, it can take time  $\omega(n)$

## Rule of Thumb

- Let  $f(n)$ ,  $g(n)$  be two functions of  $n$
- Let  $f_1(n)$ , be the fastest growing term of  $f(n)$ , stripped of its coefficient.
- Let  $g_1(n)$ , be the fastest growing term of  $g(n)$ , stripped of its coefficient.

Then we can say:

- If  $f_1(n) \leq g_1(n)$  then  $f(n) = O(g(n))$
- If  $f_1(n) \geq g_1(n)$  then  $f(n) = \Omega(g(n))$
- If  $f_1(n) = g_1(n)$  then  $f(n) = \Theta(g(n))$
- If  $f_1(n) < g_1(n)$  then  $f(n) = o(g(n))$
- If  $f_1(n) > g_1(n)$  then  $f(n) = \omega(g(n))$

## More Examples

The following are all true statements:

- $\sum_{i=1}^n i^2$  is  $O(n^3)$ ,  $\Omega(n^3)$  and  $\Theta(n^3)$
- $\log n$  is  $o(\sqrt{n})$
- $\log n$  is  $o(\log^2 n)$
- $10,000n^2 + 25n$  is  $\Theta(n^2)$

# Problems

True or False? (Justify your answer)

- $n^3 + 4$  is  $\omega(n^2)$
- $n \log n^3$  is  $\Theta(n \log n)$
- $\log^3 5n^2$  is  $\Theta(\log n)$
- $10^{-10}n^2 + n$  is  $\Theta(n)$
- $n \log n$  is  $\Omega(n)$
- $n^3 + 4$  is  $o(n^4)$

## Another Example

- Let  $f(n) = 10 \log^2 n + \log n$ ,  $g(n) = \log^2 n$ . Let's show that  $f(n) = \Theta(g(n))$ .
- We want positive constants  $c_1, c_2$  and  $n_0$  such that  $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $n \geq n_0$

$$0 \leq c_1 \log^2 n \leq 10 \log^2 n + \log n \leq c_2 \log^2 n$$

Dividing by  $\log^2 n$ , we get:

$$0 \leq c_1 \leq 10 + 1/\log n \leq c_2$$

- If we choose  $c_1 = 1$ ,  $c_2 = 11$  and  $n_0 = 2$ , then the above inequality will hold for all  $n \geq n_0$

## At-Home Exercise

Show that for  $f(n) = n + 100$  and  $g(n) = (1/2)n^2$ , that  $f(n) \neq \Theta(g(n))$

- What statement would be true if  $f(n) = \Theta(g(n))$  ?
- Show that this statement can not be true.