CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Practice with Graphs and Asymptotic Complexity Notation

- Consider a graph G = (V,E) with *n* nodes, i.e., |V|=n.
- How many edges might it have in edge set E?
- What can we say about its number of edges, using asymptotic complexity notation (O, θ, Ω)?

Practice with Graphs and Asymptotic Complexity Notation

- Consider a graph G = (V,E) with *n* nodes, i.e., |V|=n.
- How many edges might it have in edge set E?
- What can we say about its number of edges, using asymptotic complexity notation (O, θ, Ω)?

Any edge has the form (v_i, v_j), where both v_i, v_j are in V.
How many possible edges could there be? [They may not all be in E! We're just seeing what's possible]

There are n possible choices for v_i. [n possible vertices / nodes]
Similarly, for each of those n choices, there are n choices for v_j.
So, there are n*n = n^2 possible pairs—that is, n^2 possible edges.

In asymptotic complexity notation, then, there are ??? edges. [What can we say here?]

Try it where G has 3 nodes—we see 9 *possible* edges, including self-loops from a node to itself

Practice with Graphs and Asymptotic Complexity Notation

- Consider a graph G = (V,E) with *n* nodes, i.e., |V|=n.
- How many edges might it have in edge set E?
- What can we say about its number of edges, using asymptotic complexity notation (O, θ, Ω)?

Any edge has the form (v_i, v_j), where both v_i, v_j are in V.
How many possible edges could there be? [They may not all be in E! We're just seeing what's possible]

There are n possible choices for v_i. [n possible vertices / nodes]
Similarly, for each of those n choices, there are n choices for v_j.
So, there are n*n = n^2 possible pairs—that is, n^2 possible edges.

Try it where G has 3 nodes—we see 9 *possible* edges, including self-loops from a node to itself

In asymptotic complexity notation, then, there are $O(n^2)$ edges.
We cannot make any statement using θ or Ω notation—we only have enough information for an upper bound!

Business

- Grading update:
  - SA1 grading not before the weekend
- Another SA may be out soon—I'll email if so
- Problem Set 0 due already
  - PS1 out soon, perhaps tonight, or perhaps Friday
  - I'll email when it's out
  - …but you have things to work on in the meantime… such as…
- Project 1 due Sept. 28
  - Please direct project-specific questions to me, rather than to TAs
    - Questions about general concepts that show up on the project (e.g., Theta notation), though, rather than specifics, can go to TAs

Business, pt. 2

- Class will be cancelled Monday, Sept. 26
  - Will be an optional make-up class later in the semester
- General grading notes:
  - If we leave a note "See Prof. if you'd like"—it's recommended
  - If we leave a note "See Prof."—please see me, I think it's important
- If you come to office hours, please let me know you're there
  - Even if I'm with another student, please say hi and let me know
  - It gives me info so I can multitask efficiently and help more people

A tiny bit about the course, the Remix: Some Main Ideas

- Recall important elements for any course on algorithms:
  - Classic algorithms (which you might use or adapt for your work)
  - Algorithm design techniques and paradigms
    - Creating and working with algorithm specifications
  - Analyzing and explaining an algorithm's correctness
  - Analyzing and explaining an algorithm's complexity
- We'll spend a lot of the semester on these important elements

Design Paradigm	Analysis	
	Complexity (Efficiency)	Correctness
Iterative	*Counting* (Counting number of operations / amount of space used)	*Loop invariants*
Recursive	Solving *recurrences*	*Induction*

Problem-Solving Warmup

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
  1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.

Problem-Solving Warmup

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
  1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
  2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.

Loop Invariants—a Warmup

(Game description repeated from prev. slide)

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
    1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
    2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.
- What useful ***loop invariant*** do you have in mind when designing / explaining your algorithm?
  - Maybe it involves the odd / even-ness (the *parity*) of the number of red or green balls?

- **A *loop invariant* is something that's true about the variables every time the algo goes through the loop ("every time"—so, invariant)…**
- **…But is false once the loop ends, in a way that gets to the problem's solution**

Loop Invariants—a Warmup

(Game description repeated from prev. slide)

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
    1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
    2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.
- What useful ***loop invariant*** do you have in mind when designing / explaining your algorithm?
  - Maybe it involves the odd / even-ness (the *parity*) of the number of red or green balls?

Loop Invariant property: Every time through the game, until there's one ball left…
- **The overall number of balls decreases by 1; AND**
- **The number of red balls remains odd—either decreases by 0 or by 2**

Loop Invariants—a Warmup

(Game description repeated from prev. slide)

- A problem-solving warmup—consider the following game:
  - An odd number of red balls and any number of green balls are put in a bag. An infinite supply of green balls is available. A move consists of removing two balls from the bag and applying the following rule: If the balls are the same color, they are both thrown away and a new green ball is placed in the bag. If the balls are of different colors, the red one is returned to the bag and the green one is discarded. The game ends when it is no longer possible to pick two balls from the bag.
  - Consider a one-player version where the player can look in the bag before removing two balls from it.
  1. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is green.
  2. Give a method (strategy / algorithm) so that when the game is over, there is one ball left in the bag, and it is red.
- What useful ***loop invariant*** do you have in mind when designing / explaining your algorithm?
  - Maybe it involves the odd / even-ness (the *parity*) of the number of red or green balls?

Loop Invariant property: Every time through the game, until there's one ball left…
- **The overall number of balls decreases by 1; AND**
- **The number of red balls remains odd—either decreases by 0 or by 2**

How does this help you solve the problem?

Log It: Questions about exponents

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]

- So, how many iterations until it reaches 1?

Log It: Questions about exponents

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]
- **So, reaches 1 = (n/n) on iteration *k* when $n=2^k$**

- So, how many iterations until it reaches 1?

Log It: Questions about exponents

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true
  - *How could we even phrase that question?*

Recall one of our CS "pro tips":
If you want to talk about something, give yourself a word / name for it
- **E.g., declaring a variable rather than recomputing something repeatedly**

So, this question—what's the exponent?—is something we want to ask repeatedly…
We need to have a word / function for it

Log It: Questions about exponents

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true
  - *How could we even phrase that question?*

- The *logarithm* function lets us ask the question
  - So, for $2^x = 375$, we'd say $x = log_2\ 375$ (read as "log base 2 of 375")
  - Examples: $log_3\ 81 = 4$; $log_4\ 16 = 2$; $log_2\ 1024 = 10$
- Logarithms *are* exponents, so rules of exponentiation apply
  - E.g., $log_b\ (m*n) = log_b\ m + log_b\ n$

If $b^x = m$ and $b^y = n$,
then $b^{x}*b^{y} = b^{x+y} = m*n$

Log It: Questions about exponents

Getting back to where we left off a couple of slides ago…

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true. Notation: $x = log_2 375$ ←

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]
- **So, reaches 1 = (n/n) on iteration *k* when $n=2^k$**

- So, how many iterations until it reaches 1?

Log It: Questions about exponents

Getting back to where we left off a couple of slides ago…

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true. Notation: $x = log_2 375$ ←

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]
- **So, reaches 1 = (n/n) on iteration *k* when $n=2^k$**

- So, how many iterations until it reaches 1? $log_2 n + 1$

Notation: this is not $log_2 (n+1)$

Log It: Questions about exponents

Getting back to where we left off a couple of slides ago…

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true. Notation: $x = log_2 375$ ←

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]
- **So, reaches 1 = (*n*/*n*) on iteration *k* when $n=2^k$**

- So, how many iterations until it reaches 1? ***$log_2 n + 1$***

[Watch for off-by-one errors!]

Log It: Questions about exponents

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true. Notation: $x = log_2 375$ ←

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]
- **So, reaches 1 = (*n*/*n*) on iteration *k* when $n=2^k$**
- **That is, *$log_2 n + 1$* iterations [Watch for off-by-one errors!]**

- Can you think of an algo that works like this on its input?

Log It: Questions about exponents

- When solving equations, we may want to know the value of an exponent
  - E.g., in equation $2^x=375$, we might want to ask what value of x makes that true. Notation: $x = log_2 375$ ←

How does this come up in analyzing algorithm complexity?

Imagine an algo that takes input of size *n* and then goes through a loop (or recursion), getting rid of half the remaining input each time, until there's only 1 left
- **Input of size *n* on iteration 0; *n/2* on iteration 1; *n/4* on iteration 2; …**
- **How many iterations until input of size 1?**

On iteration *k*, input is of size ($n/(2^k)$) [do you see that pattern?]
- **So, reaches 1 = (*n/n*) on iteration *k* when $n=2^k$**
- **That is, $log_2 n + 1$ iterations [Watch for off-by-one errors!]**

Exercise: If input size was *n*, *n/3*, *n/9*, … , then how many iterations until input size 1?

These are common complexity classes, but there are many others!

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*
O(1)	Constant
O(lg n)	Log time
O(n)	Linear
O(n lg n)	n lg n
O(n^2)	n-squared; quadratic
O(n^3)	n-cubed; cubic
O(2^n)	Exponential
O(n!)	Factorial

What algos do you know in each complexity class?

Do you have any favorites?

These are common complexity classes, but there are many others!

Time Complexity Classes Illustrated!

Complexity Class	*What we call it*	*Example algorithms / objects*
O(1)	Constant	Print "Hello, World!"; stack operations [and much, much more—be careful!]
O(lg n)	Log time	Binary search
O(n)	Linear	Exhaustive search of an array (linear search); Merge (as used in Mergesort)
O(n lg n)	n lg n	Mergesort; Heapsort [Recall: sorting can be done in θ(n lg n)]
O(n^2)	n-squared; quadratic	Insertion / selection / bubble sort; several graph algos
O(n^3)	n-cubed; cubic	*My favorite algorithm*! (a graph algo)
O(2^n)	Exponential	Number of *subsets* of a set of size n
O(n!)	Factorial	Number of orderings / permutations of elements of a list of length n