

CSE 332 Autumn 2026

Lecture 3: Algorithm Analysis pt.2

Ruth Anderson

<http://www.cs.uw.edu/332>

Announcements

- Preliminary Survey
 - Due Monday 10/05 – TONIGHT!
 - On Canvas
- Exercise 0
 - Due Friday 10/09/2026
 - There are 2 separate Gradescope submissions
- “Meet the staff” activity
 - Sometime during the first 4 weeks of class, visit a CSE 332 office hour (in person or on zoom)
 - Tell the staff member you want to get checked off
 - You do not have to have a question about course content
 - We just want to meet you!
- Exercise 1 Running Time: coming early next week, due Mon Oct 12

Review: Motivation for Algorithm Analysis

- We want to define algorithm running times in a way that's predictive:
 - We're not required to implement & run the algorithms to determine the time
 - Our conclusions do not depend on how the algorithm is implemented/run
 - E.g. variation in choice of programming language, machine, or other tasks the computer is doing in parallel
 - We can use it to compare running times of different algorithms
 - We do not need to select an input size in advance

Review: Process for Algorithm Analysis

- Identify a *function* which maps the algorithm's input size to a measure of resources used
 - Domain of the function: **sizes** of the input
 - Number of characters in a string, number of items in a list, number of pixels in an image
 - Codomain of the function: **counts** of resources used
 - Typically we think of this as “amount of time” but we will specifically count things like: Number of times the algorithm adds two numbers together, number times the algorithm does a $>$ or $<$ comparison, maximum number of bytes of memory the algorithm uses at any time
- Important note: Make sure you know the “units” of your domain and codomain!
 - Domain = inputs to the function
 - Codomain = outputs to the function

Review: Scenarios to analyze for an algorithm

- Worst-case complexity:
 - max number of steps algorithm takes on “most challenging” input
- Best-case complexity:
 - min number of steps algorithm takes on “easiest” input
- Average/expected complexity:
 - avg number of steps algorithm takes on random inputs (context-dependent)
- Amortized complexity:
 - max total number of steps algorithm takes on M “most challenging” *consecutive* inputs, divided by M (i.e., divide the max total sum by M).

Searching in a Sorted List

5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```
public static boolean contains(List<Integer> a, int k){  
    for(int i=0; i< a.size(); i++){  
        if (a.get(i) == k)  
            return true;  
    }  
    return false;  
}
```

Faster way?

5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

Can you think of a faster algorithm to solve this problem?

Binary Search

5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```
public static boolean binary_contains(List<Integer> a, int k){
    int start = 0;
    int end = a.size();
    while(start < end){
        int mid = start + (end-start)/2;
        if(a.get(mid) == k)
            return true;
        else if(a.get(mid) < k)
            start = mid+1;
        else
            end = mid;
    }
    return false;
}
```

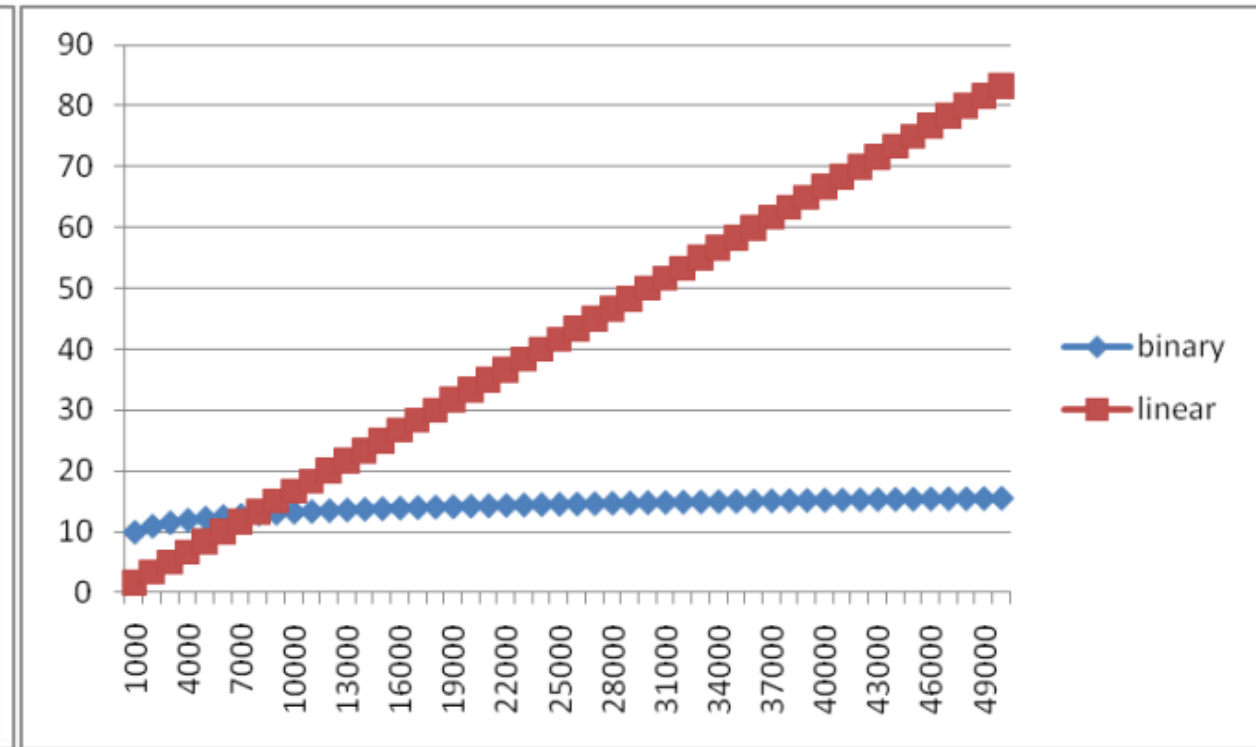
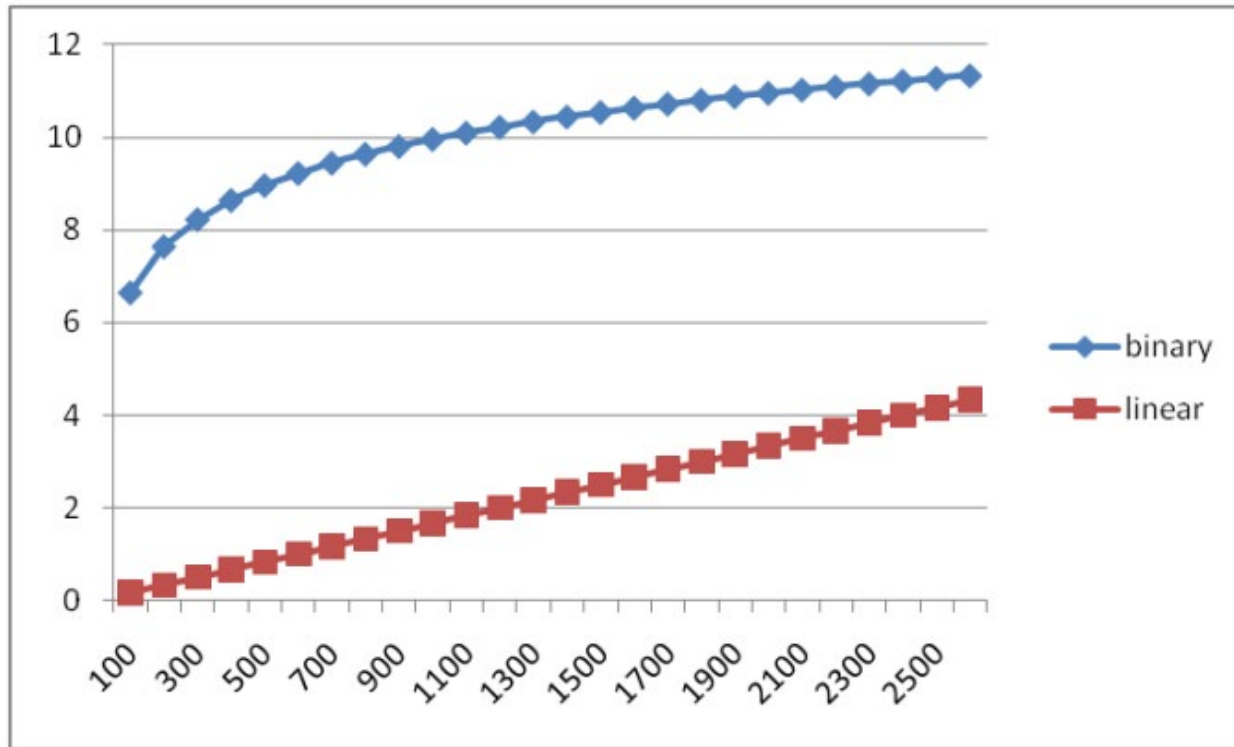
Why is this $\log_2 n$?

- In the beginning: $\text{end} - \text{start} = n$
- After 1 iteration: $\text{end} - \text{start} = \frac{n}{2}$
 - $\text{mid} - \text{start} = (\text{start} + (\text{end} - \text{start}) / 2) - \text{start}$
 - $\text{end} - \text{mid} = \text{end} - (\text{start} + (\text{end} - \text{start}) / 2)$
- Each iteration cuts the “gap” in half!
 - x iterations the size is $\frac{n}{2^x}$
- We stop when the gap is 1
 - Solve for x : $\frac{n}{2^x} = 1$

Aside: What's the base of the log?

- If I write $\log n$, without specifying a base, I usually mean $\log_2 n$.
- But does it matter for big-O?
- Suppose we found an algorithm with running time $\log_5 n$ instead.
- Is that different from $O(\log_2 n)$?
- No!
- $\log_c n = \frac{\log_2 n}{\log_2 c}$ If c is a constant, then $\log_2 c$ is just a constant, and we can hide it inside the $O()$.

Running Time Graphed



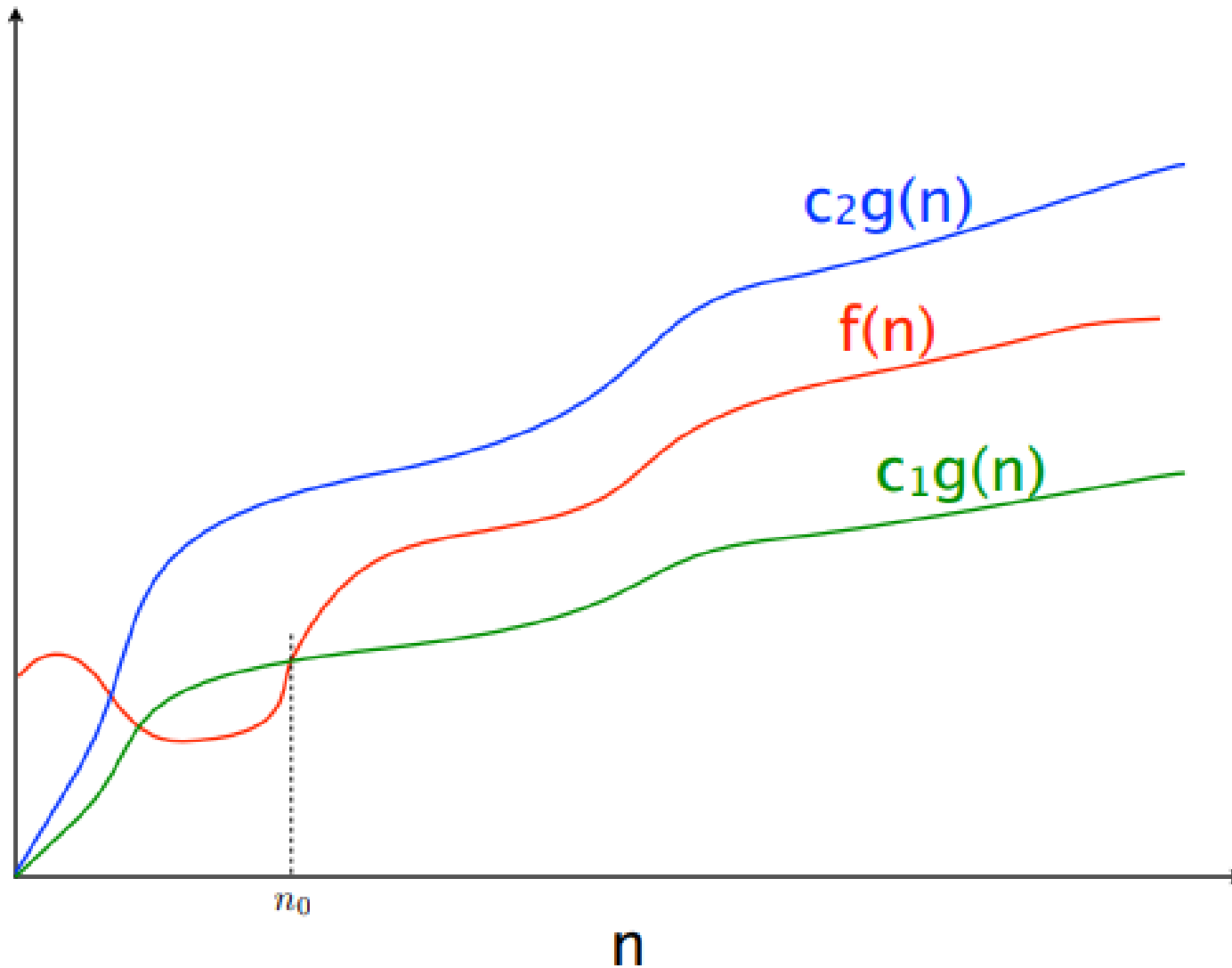
Comparing Running Times

- Suppose I have these algorithms, all of which have the same input/output behavior:
 - Algorithm A's worst case running time is $10n + 900$
 - Algorithm B's worst case running time is $100n - 50$
 - Algorithm C's worst case running time is $\frac{n^2}{100}$
- Which algorithm is best?

Wish List for Way to Compare Running Times

- To compare two algorithms' running times, we need a way to compare functions
- We want this comparison to have the following properties:
 - We answer based on long-term behavior (i.e. large inputs)
 - In most cases, all algorithms are fast on small inputs, so it's more important to compare for large inputs
 - We ignore constant coefficients and non-dominant terms
 - Different people might count different subsets of operations, which would give different constant coefficients in the running time
 - Even if we count the same operations, different operations take different amounts of time on different computers
 - Constant coefficients matter more than the non-dominant terms (e.g. $2n^2 > n^2 + n$)

Asymptotically comparing $f(n)$ with $g(n)$



$$f(n) = O(g(n))$$

$$f(n) = \Theta(g(n))$$

$$f(n) = \Omega(g(n))$$

Asymptotic Notation

- $O(g(n))$
 - The **set of functions** with asymptotic behavior less than or equal to $g(n)$
 - **Upper-bounded** by a constant times g for large enough values n
 - $f \in O(g(n)) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \leq c \cdot g(n)$
- $\Omega(g(n))$
 - the **set of functions** with asymptotic behavior greater than or equal to $g(n)$
 - **Lower-bounded** by a constant times g for large enough values n
 - $f \in \Omega(g(n)) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \geq c \cdot g(n)$
- $\Theta(g(n))$
 - **“Tightly”** within constant of g for large n
 - $\Omega(g(n)) \cap O(g(n))$

Idea of Θ

- $x = y$
 - $x \leq y \wedge x \geq y$

Aside: Big-Oh relates functions

We use O on a function $f(n)$ (for example n^2) to mean *the set of functions with asymptotic behavior less than or equal to $f(n)$*

So $(3n^2+17)$ **is in** $O(n^2)$

- $3n^2+17$ and n^2 have the same **asymptotic behavior**

Confusingly, we also say/write:

- $(3n^2+17)$ **is** $O(n^2)$
- $(3n^2+17)$ **=** $O(n^2)$

But we would never say $O(n^2) = (3n^2+17)$

Warning!! O , Ω , Θ vs. Best, Worst, Average

- It's a common misconception that $\Omega()$ is “best-case” and $O()$ is “worst-case”. This is a misconception!! (GenAI often says the wrong thing)
- $O()$ says “the complexity of this algorithm is at most” (think $\leq$)
- $\Omega()$ says “the complexity of this algorithm is at least” (think $\geq$)
- You can use $\leq$ on worst-case or best case; you can use $\geq$ on worst-case or best-case.
- Best/Worst/Average say “what function f am I analyzing?”
- O , Ω , Θ say “let me summarize what I know about the function f , it's $\leq$, $\geq$, $=$...”

Proving Big-O, Formally

- Big-O is an $\exists c, n_0 \forall n$ statement.
- I.e., an exists statement with a “forall” inside.
- How do you prove an exists statement?
- How do you prove a for-all statement?

Proving Big-O, Formally (answers)

- Big-O is an $\exists c, n_0 \forall n$ statement.
- I.e., an exists statement with a “forall” inside.
- How do you prove an exists statement?
 - Show the c, n_0 that will work. Give specific values.
- How do you prove a for-all statement?
 - Introduce an arbitrary n .

Asymptotic Notation Example 1

- Show: $10n + 100 \in O(n^2)$
 - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n > n_0. 10n + 100 \leq c \cdot n^2$
 - **Proof:**

Asymptotic Notation Example 1 (Scratchwork)

- Show: $10n + 100 \in O(n^2)$
 - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n > n_0. 10n + 100 \leq c \cdot n^2$
 - **Scratchwork:**
 - We need $10n + 100 \leq c \cdot n^2$
 - The algebra looks like it might be easier if we select $c = 10$
 - Now we need $10n + 100 \leq 10n^2$
 - Equivalently, $n + 10 \leq n^2$
 - To select n_0 we need to consider what values of n will cause “add 10” to have less impact than “multiply by n ”.
 - If $n \geq 10$ then $n + 10 \leq 2n$, and also $2n \leq 10n$, and also $10n \leq n^2$, so $n + 10 \leq n^2$
 - Thus it should work to use $c = 10, n_0 = 10$

Asymptotic Notation Example 1 (Proof)

- Show: $10n + 100 \in O(n^2)$
 - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0. 10n + 100 \leq c \cdot n^2$
 - **Proof:** Let $c = 10$ and $n_0 = 10$. Show $\forall n \geq 10 \quad 10n + 100 \leq 10n^2$
$$\begin{aligned}10n + 100 &\leq 10n^2 \\ \equiv n + 10 &\leq n^2 \\ \equiv 10 &\leq n^2 - n \\ \equiv 10 &\leq n(n - 1)\end{aligned}$$
This is True because $n(n - 1)$ is strictly increasing and $10(10 - 1) > 10$

Asymptotic Notation Example 2

- Show: $13n^2 - 50n \in \Omega(n^2)$
 - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0. 13n^2 - 50n \geq c \cdot n^2$

Asymptotic Notation Example 2 (Scratchwork)

- Show: $13n^2 - 50n \in \Omega(n^2)$
 - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0. 13n^2 - 50n \geq c \cdot n^2$
 - **Scratchwork:**
 - we need $13n^2 - 50n \geq c \cdot n^2$
 - equivalently, $13n - 50 \geq c \cdot n$
 - It looks like things might simplify if we pick $c = 13$
 - Now we need $13n - 50 \geq 13n$
 - equivalently we need $-50 \geq 0$
 - hmmm..... That didn't work! But, it tells us that to make the left hand side bigger, it needs to be 50 larger than the right! This means we definitely need $c < 13$
 - let's instead use $c = 12$, now we need $13n - 50 \geq 12n$
 - This simplifies to $n \geq 50$, so we can use 50 as n_0

Asymptotic Notation Example 2 (Proof)

- Show: $13n^2 - 50n \in \Omega(n^2)$

- **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0. 13n^2 - 50n \geq c \cdot n^2$

- **Proof:**

Let $c = 12$ and $n_0 = 50$. show $\forall n \geq 50$ we have $13n^2 - 50n \geq 12 \cdot n^2$

Applying algebra:

$$13n^2 - 50n \geq 12n^2$$

$$\equiv n^2 - 50n \geq 0$$

$$\equiv n - 50 \geq 0 \text{ (allowed because } n \neq 0 \text{ since } n \geq 50)$$

$$\equiv n \geq 50$$

Because we chose $n_0 = 50$, the inequality holds for all values of $n \geq n_0$

Gaining Intuition

- When doing asymptotic analysis of functions:
 - If multiple expressions are added together, ignore all but the “biggest”
 - If $f(n)$ grows asymptotically faster than $g(n)$, then $f(n) + g(n) \in \Theta(f(n))$
 - Ignore all multiplicative constants
 - $c \cdot f(n) \in \Theta(f(n))$ for any constant $c \in \mathbb{R}$
 - Ignore bases of logarithms
 - Do NOT ignore:
 - Non-multiplicative and non-additive constants (e.g. in exponents, bases of exponents)
 - Logarithms themselves
- Examples:
 - $4n + 5$
 - $0.5n \log n + 2n + 7$
 - $n^3 + 2^n + 3n$
 - $n \log(10n^2)$

What you can drop

- Eliminate coefficients because we don't have units anyway
 - $3n^2$ versus $5n^2$ doesn't mean anything when we cannot count operations very accurately
- Eliminate low-order terms because they have vanishingly small impact as n grows
- Do NOT ignore constants that are not multipliers
 - n^3 is not $O(n^2)$
 - 3^n is not $O(2^n)$

(This all follows from the formal definition)

Common Categories

- $O(1)$ “constant”
- $O(\log n)$ “logarithmic”
- $O(n)$ “linear”
- $O(n \log n)$ “log-linear”
- $O(n^2)$ “quadratic”
- $O(n^3)$ “cubic”
- $O(n^k)$ “polynomial” (where k is a constant > 1)
- $O(k^n)$ “exponential” (where k is a constant > 1)

Addendum and Caveats

More Examples

- Is each of the following True or False?
 - $4 + 3n \in O(n)$
 - $n + 2 \log n \in O(\log n)$
 - $\log n + 2 \in O(1)$
 - $n^{50} \in O(1.1^n)$
 - $3^n \in \Theta(2^n)$

More Examples: Solutions

- Is each of the following True or False? Justifications are “semi-intuitive” rather than rigorous:
 - $4 + 3n \in O(n)$
 - True because for large inputs, $4 + 3n < n + 3n = 4n$
 - $n + 2 \log n \in O(\log n)$
 - False because for any constant c eventually $n > c \log n$ (specifically, when $\frac{2^n}{n} > c$)
 - $\log n + 2 \in O(1)$
 - False because for any constant c eventually $\log n > c$ (specifically when $n > 2^c$)
 - $n^{50} \in O(1.1^n)$
 - True because for large n it will be that $50 \log_{1.1} n \leq n$
 - $3^n \in \Theta(2^n)$
 - False because $3^n \leq c 2^n$ is true iff $(\log_2 3) n \leq \log_2 c + n$, and eventually multiplying by $\log_2 3$ has more impact than adding $\log_2 c$ (for example, when $n > \frac{\log_2 c}{\log_2 3 - 1}$)

Regarding use of terms

A common error is to say $O(f(n))$ when you mean $\theta(f(n))$

- People often say $O()$ to mean a tight bound
- Say we have $f(n)=n$; we could say $f(n)$ is in $O(n)$, which is true, but only conveys the upper-bound
- Since $f(n)=n$ is *also* $O(n^5)$,
it's tempting to say “this algorithm is *exactly* $O(n)$ ”
- Somewhat incomplete; instead say it is $\theta(n)$
- That means that it is not, for example $O(\log n)$

What we are analyzing

- The most common thing to do is give an O or θ **bound** to the **worst-case** running **time** of an **algorithm**
- Example: True statements about binary-search algorithm
 - Common: $\theta(\log n)$ running-time in the worst-case
 - Less common: $\theta(1)$ in the best-case (item is in the middle)
 - Less common: Algorithm is $\Omega(\log \log n)$ in the worst-case (it is not really, really, really fast asymptotically)
 - Less common (but good to know): the find-in-sorted-array **problem** is $\Omega(\log n)$ in the worst-case
 - No algorithm can do better (without parallelism)
 - A **problem** cannot be $O(f(n))$ since you can always find a slower algorithm, but can mean **there exists** an algorithm

Summary

Analysis can be about:

- The problem or the algorithm (usually algorithm)
- Time or space (usually time)
 - Or power or dollars or ...
- Best-, worst-, or average-case (usually worst)
- Upper-, lower-, or tight-bound (usually upper or tight)

Big-Oh Caveats

- Asymptotic complexity (Big-Oh) focuses on behavior for **large n** and is independent of any computer / coding trick
 - But you can “abuse” it to be misled about trade-offs
 - Example: $n^{1/10}$ vs. **log** n
 - Asymptotically $n^{1/10}$ grows more quickly
 - But the “cross-over” point is around $5 * 10^{17}$
 - So if you have input size less than 2^{58} , prefer $n^{1/10}$
- Comparing $O()$ for **small n** values can be misleading
 - Quicksort: $O(n \log n)$ (expected)
 - Insertion Sort: $O(n^2)$ (expected)
 - Yet in reality Insertion Sort is faster for small n 's
 - We'll learn about these sorts later

Addendum: Timing vs. Big-Oh?

- At the core of CS is a backbone of theory & mathematics
 - Examine the algorithm itself, mathematically, not the implementation
 - Reason about performance as a function of n
 - Be able to mathematically prove things about performance
- Yet, timing has its place
 - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
 - Ex: Benchmarking graphics cards
- Evaluating an algorithm? Use asymptotic analysis
- Evaluating an implementation of hardware/software? Timing can be useful