Lecture 2: Algorithm Analysis

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

Announcements

- EX00
  - due monday
- EX01
  - released friday
- Please email me (if you haven’t already) if you need a makeup exam

Today

- What do we care about?
- Analyzing Code
  - Counting code constructs
  - Best Case vs. Worst Case
- Asymptotic Analysis
- Big-Oh Definition

What do we care about?

- Correctness:
  - Does the algorithm do what is intended?
- Performance:
  - Speed   time complexity
  - Memory   space complexity
- Why analyze?
  - To make good design decisions
  - Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

Q: How should we compare two algorithms?

Problem: Sort list of all students has ever taken CSE332

A: How should we compare two algorithms?

- Why not run and time the program?
  - Too much variability, dependent on:
    - Hardware, OS, exact implementation, etc.
  - Miss worst-case input
  - What happens when n doubles in size?
- We want to evaluate the algorithm, not the implementation
  - i.e., evaluate even before coding it
- What does "better" mean?
  - Many answers: clarity, security, simplicity, etc.
  - **Performance**: for big inputs, runs in less **time** (our focus) or less **space**

Today

- What do we care about?
- Analyzing Code
    - Counting code constructs
    - Best Case vs. Worst Case
- Asymptotic Analysis
- Big-Oh Definition

Analyzing Code: Counting code constructs

Assume basic operations take some amount of constant time

- Arithmetic (1+1), assignment (int b = 3), array index(arr[i]), etc.

This approximates reality: a very useful "lie"

Code Construct	How much Time?
Consecutive Statements	Sum of time of each statement
Loops	Sum of time of each iteration
Conditionals	Time to evaluate conditional + whichever branch executes
Function (method) Calls	Time of function’s body
Recursion	Solve recurrence equation

Examples

```
b = b + 5
c = b / a
b = c + 100
```

```
for (i = 0; i < n; i++) {
  sum++;
}
```

```
if (j < 5) {
  sum++;
} else {
  for (i = 0; i < n; i++) {
    sum++;
  }
}
```

Number of operations? Big Oh?

```
int coolFunction(int n, int sum) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++)
      sum++;
    }
  }
  print "This program is great!"
  for (i = 0; i < n; i++) {
    sum++;
  }
  return sum
}
```

Loops: Using Summations

```
for (i = 0; i < n; i++) {
  sum++;
}
```

Any Questions?

Complexity Cases (e.g., Worst vs. Best Case)

- We'll start by focusing on two cases:
- **Worst-case complexity**:
  - max # steps algorithm takes on "most challenging" input of size n
- **Best-case complexity**:
  - min # steps algorithm takes on "easiest" input of size n

Other Complexity Cases

- **Average-case complexity**:
  - What does "average" mean?
  - What is an "average" dataset?
  - No agreement on a specific scenario
- **Amortized-case complexity**:
  - Worst-case in a sequence
    - Later

Linear search – Best Case & Worst Case

2	3	5	16	37	50	73	75	126

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   for(int i=0; i < arr.length; ++i)
      if(arr[i] == k)
        return true;
   return false;
}
```

Best case:

Worst case:

Remember a faster search algorithm?

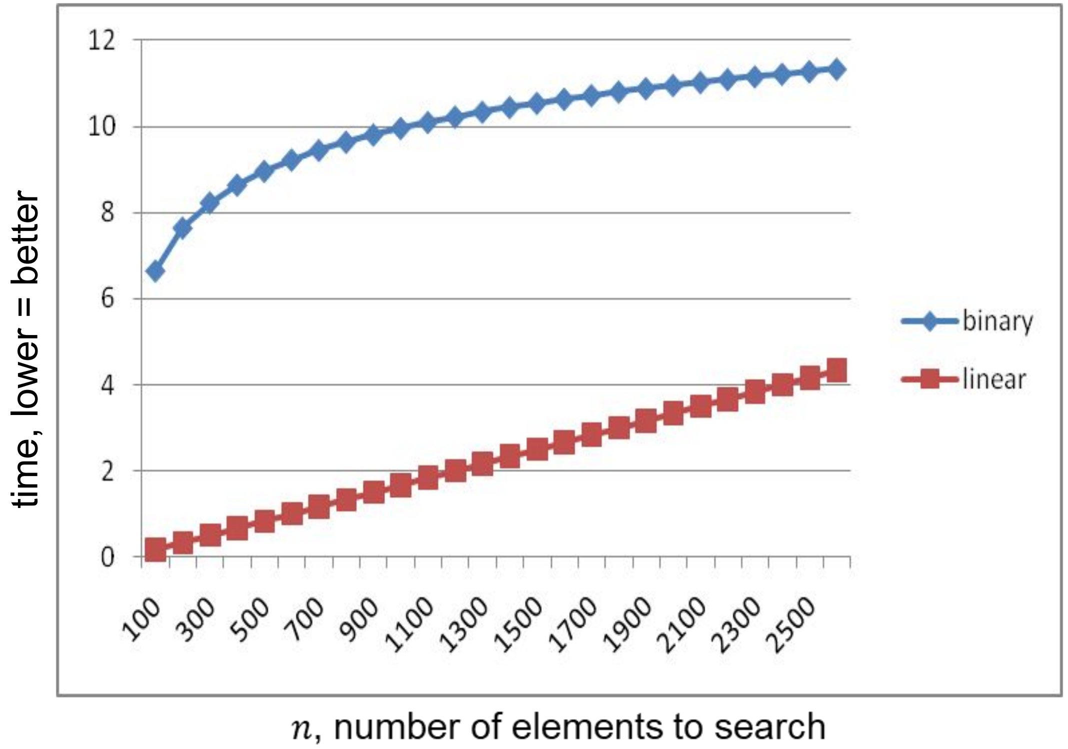
- Worst Cases:
- Binary Search – $\mathcal{O}(\log n)$
- Linear Search – $\mathcal{O}(n)$

Ignoring constant factors

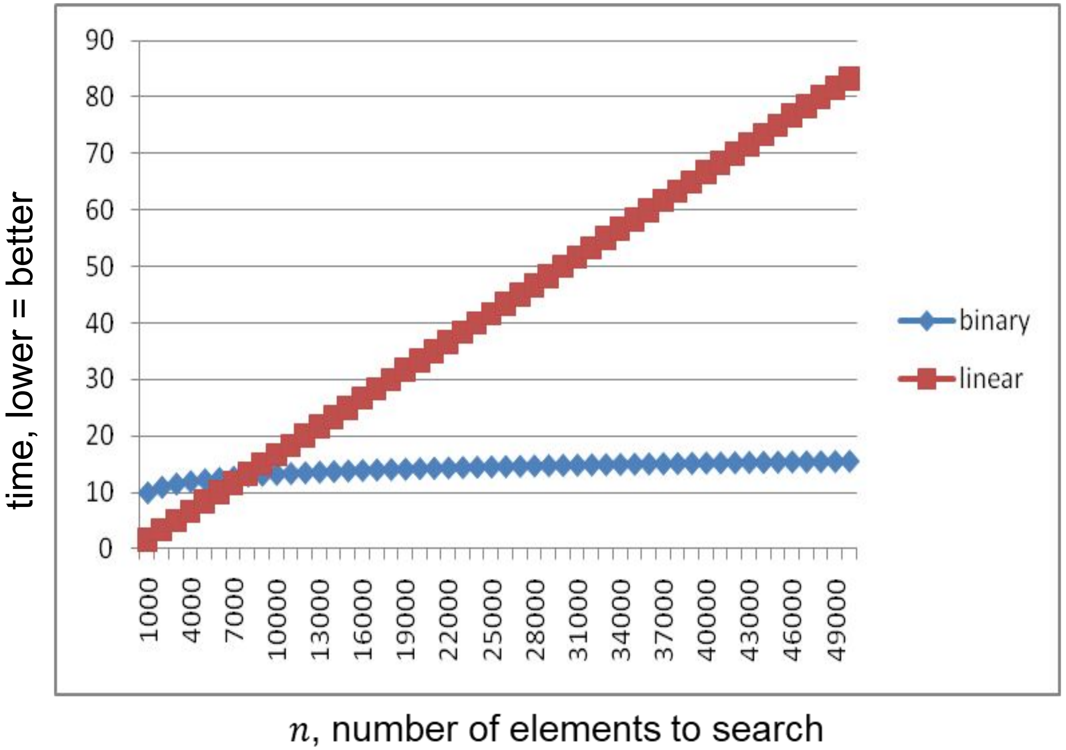
- So binary search is $\mathcal{O}(\log n)$ and linear is $\mathcal{O}(n)$
  - But which will be faster?
  - Depending on **constant factors** and **size of $\boldsymbol{n}$**, in a particular situation, linear search could be faster….
    - How many assignments, additions, etc. for each $\boldsymbol{n}$?
    - What if $\boldsymbol{n}$ is small?
- <u>But</u> there exists an n_0 such that for all $n \geq n_0$ binary search is faster
  - i.e., eventually n will get big enough that binary search is faster
- Let's look at a couple plots to get some intuition…

Example - Why we ignore constant factors

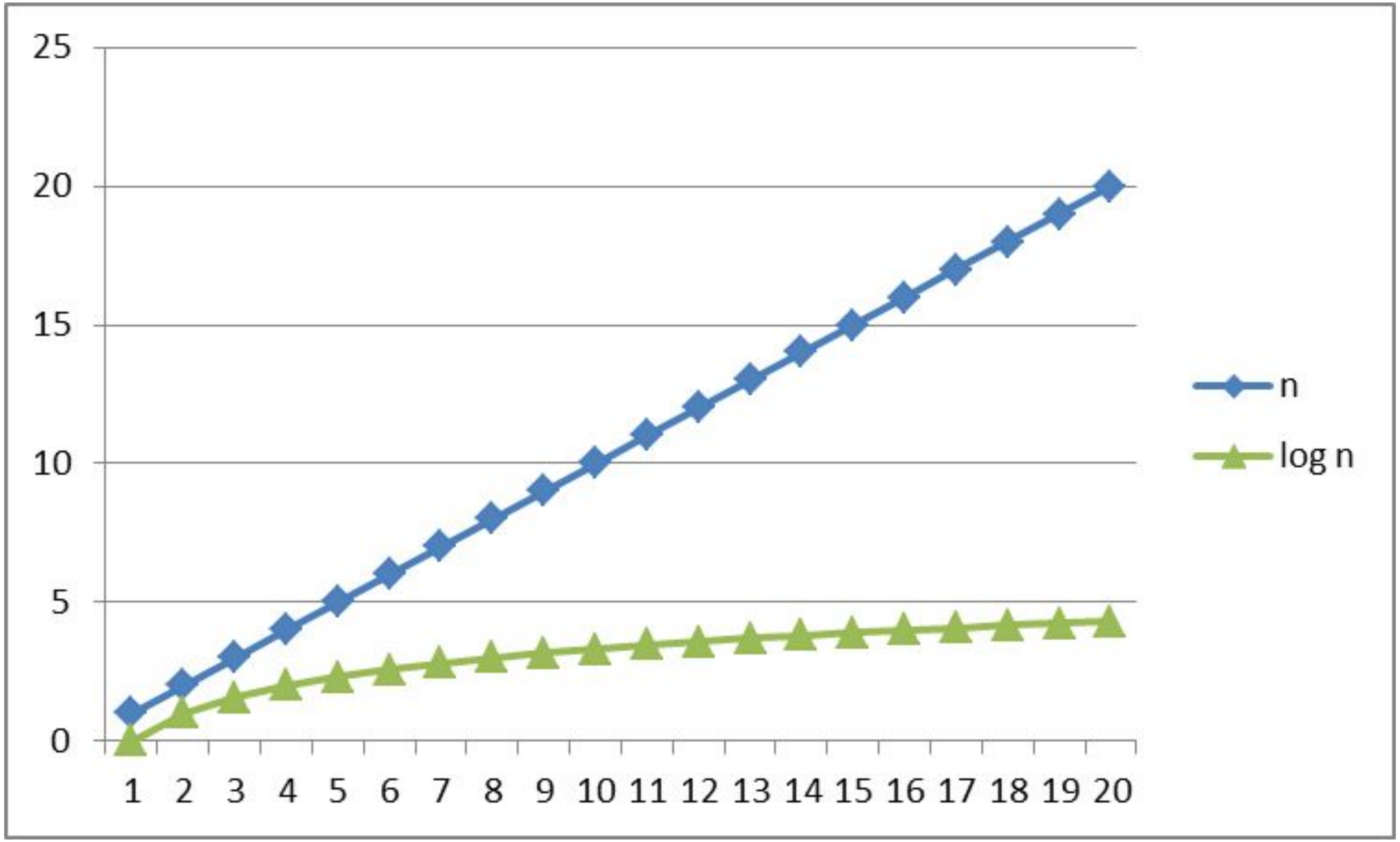
Small values of n



Big values of n

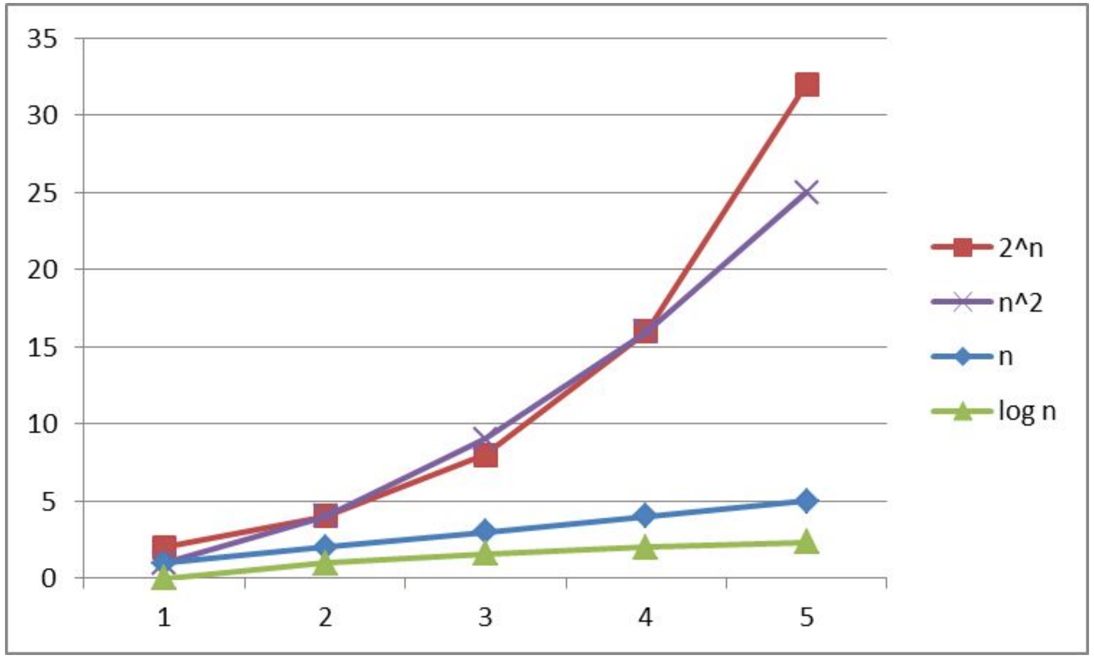


Logarithms and Exponents

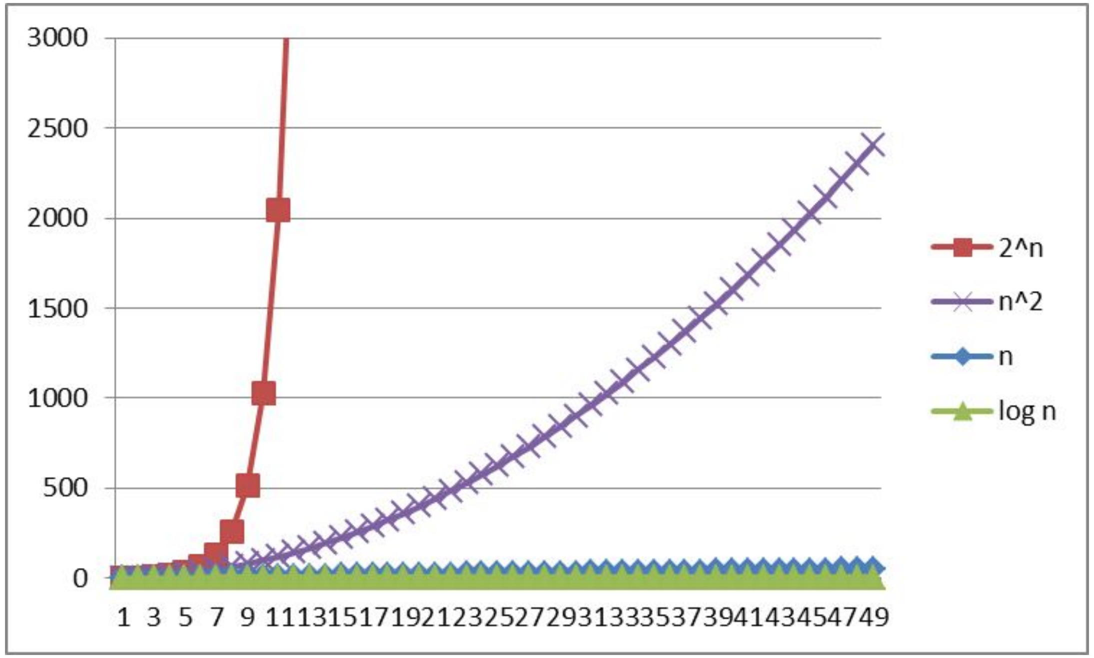


Logarithms and Exponents

Small values of n



Big values of n

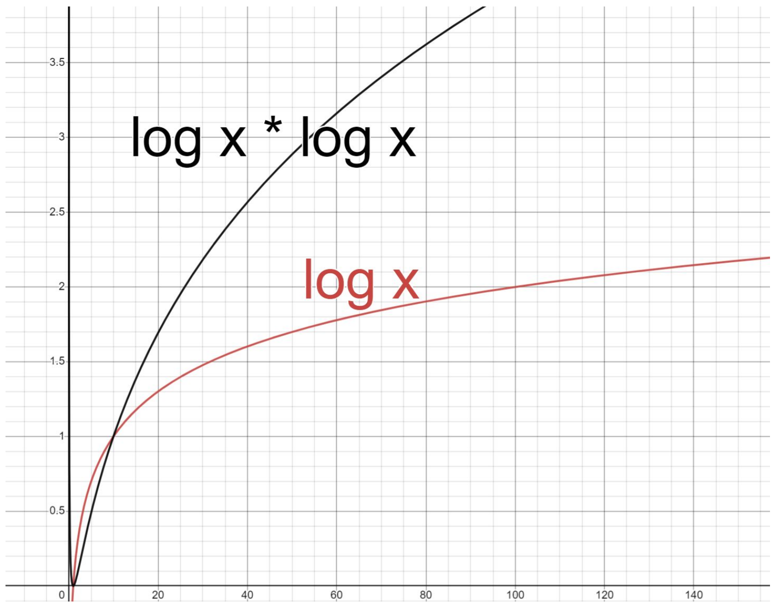


Speaking of Logarithms...

- Since so much is binary in CS, $\log$ almost always means $\log_2$
  - So, $\log_2(1{,}000{,}000) =$ "a little under 20"
  - Just as exponents grow very quickly, logarithms grow very slowly
- They don't matter much!
  - "Any base B $\log$ is equivalent to $\log_2$ with a constant factor"
  - e.g., $\log_2(x) = 3.22 \log_{10}(x)$
  - Can convert $\log_B(x) = \frac{1}{\log_2(B)} \cdot \log_2(x) = constant \cdot \log_2(x)$

Review: Logarithms

- $\log(A \cdot B) = \log(A) + \log(B)$
- $\log\left(\frac{A}{B}\right) = \log(A) - \log(B)$
- $\log(N^k) = k\log(N)$
- $\log_2(2^x) = x$
- $\log(\log(x))$ or $\log\log x$
  - Grows slower than $\log(x)$
- $\log(x) \cdot \log(x)$ or $\log^2(x)$
  - Grows faster than $\log(x)$



Today

- What do we care about?
- Analyzing Code
  - Counting code constructs
  - Best Case vs. Worst Case
- Asymptotic Analysis
- Big-Oh Definition

Asymptotic Analysis

Formal definition soon, intuition is:

1. Eliminate low-order terms
2. Eliminate constant coefficients

Examples:

- $4n + 5$
- $0.5n \log n + 2n + 7$
- $n^3 + 2^n + 3n$
- $n \log(10n^2)$

Asymptotic Analysis: Big-Oh

We use $\mathcal{O}$ on a function $f(n)$ (e.g., n^2) to mean *the set of functions with asymptotic behavior* ***less than or equal to*** $f(n)$

e.g., $(3n^2 + 17)$ **is in** $\mathcal{O}(n^2)$ (or in math, $(3n^2 + 17) \in \mathcal{O}(n^2)$)

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

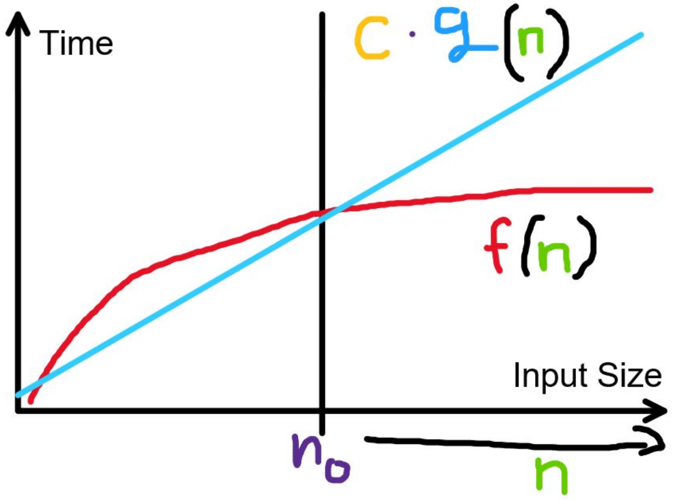
Confusingly, we also say/write:

- $(3n^2 + 17)$ **is** $\mathcal{O}(n^2)$
- $(3n^2 + 17) = \mathcal{O}(n^2)$
    - But we would never say $\mathcal{O}(n^2) = (3n^2 + 17)$

Today

- What do we care about?
- Analyzing Code
  - Counting code constructs
  - Best Case vs. Worst Case
- Asymptotic Analysis
- Big-Oh Definition

Formally Big-Oh



Formal Definition:

Suppose $f: \mathbb{N} \to \mathbb{R}$, $g: \mathbb{N} \to \mathbb{R}$ are two functions, then,

$$f(n) \in \mathcal{O}(g(n)) \equiv \exists_{c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}} \forall_{n \in \mathbb{N} \geq n_0} f(n) \leq c \cdot g(n)$$

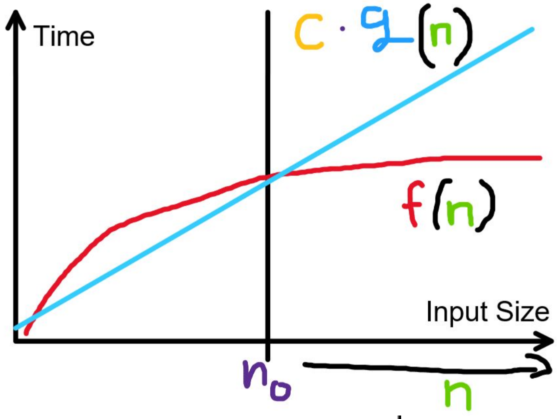
In English:

$f(n)$ is in $\mathcal{O}(g(n))$ iff there exists constants c and n_0 such that:

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Note: c is a positive real number, n_0 and n are natural numbers

Why n_0? Why c?



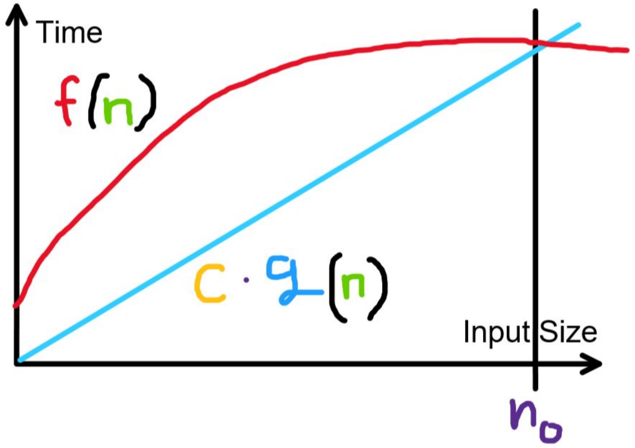
In English:

$f(n)$ is in $\mathcal{O}(g(n))$ iff there exists constants c and n_0 such that:

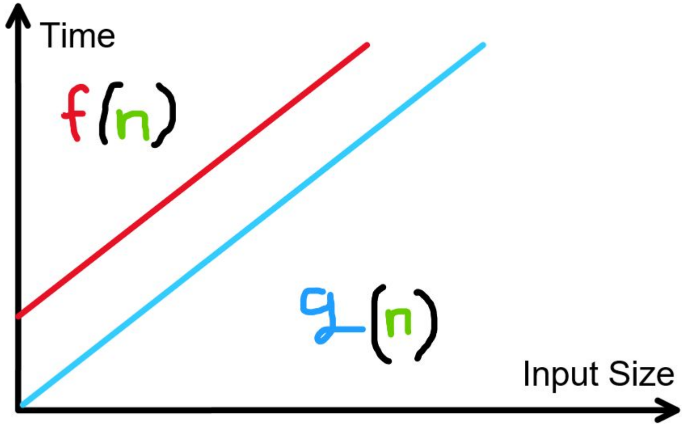
$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Note: c is a positive real number, n_0 and n are natural numbers

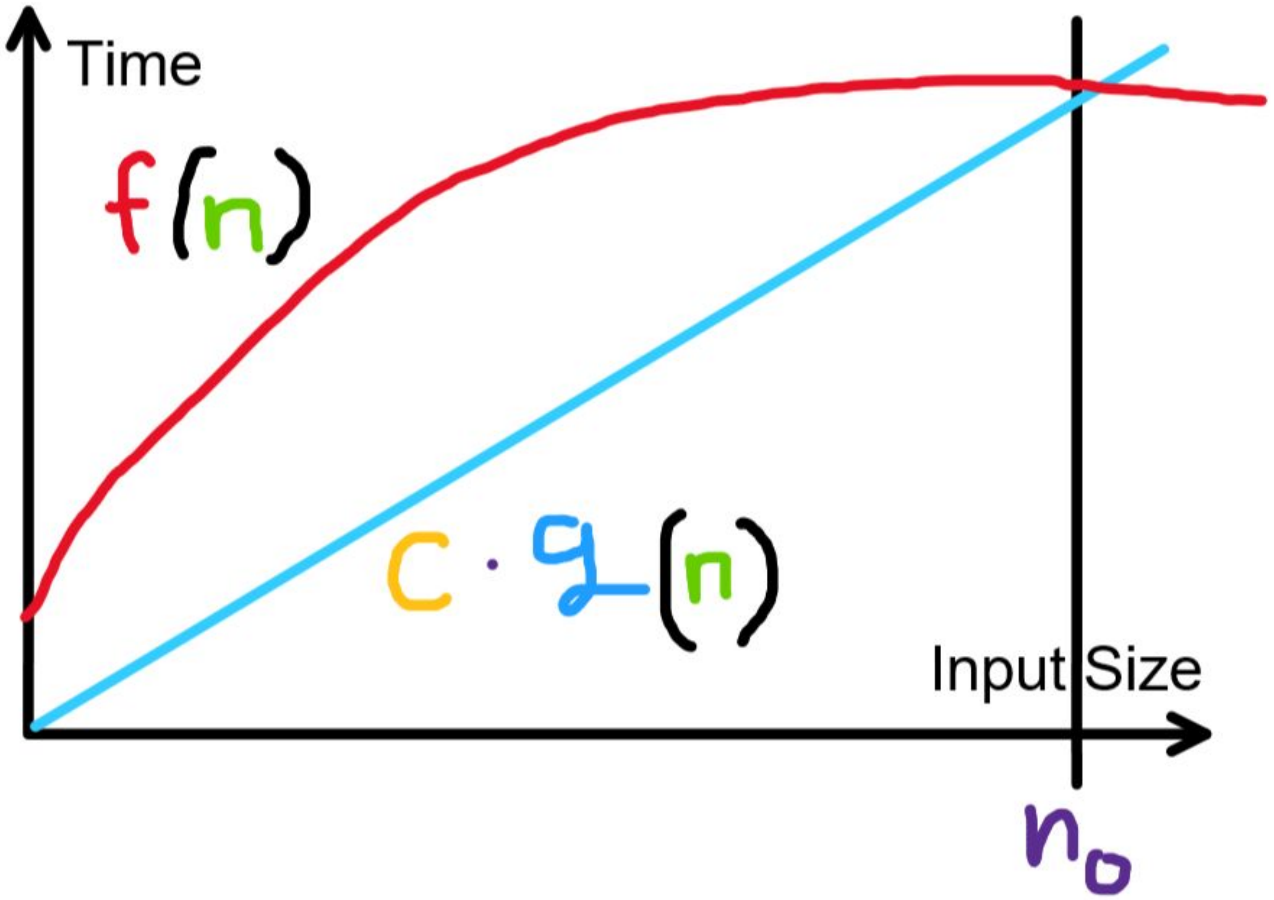
Why n_0?



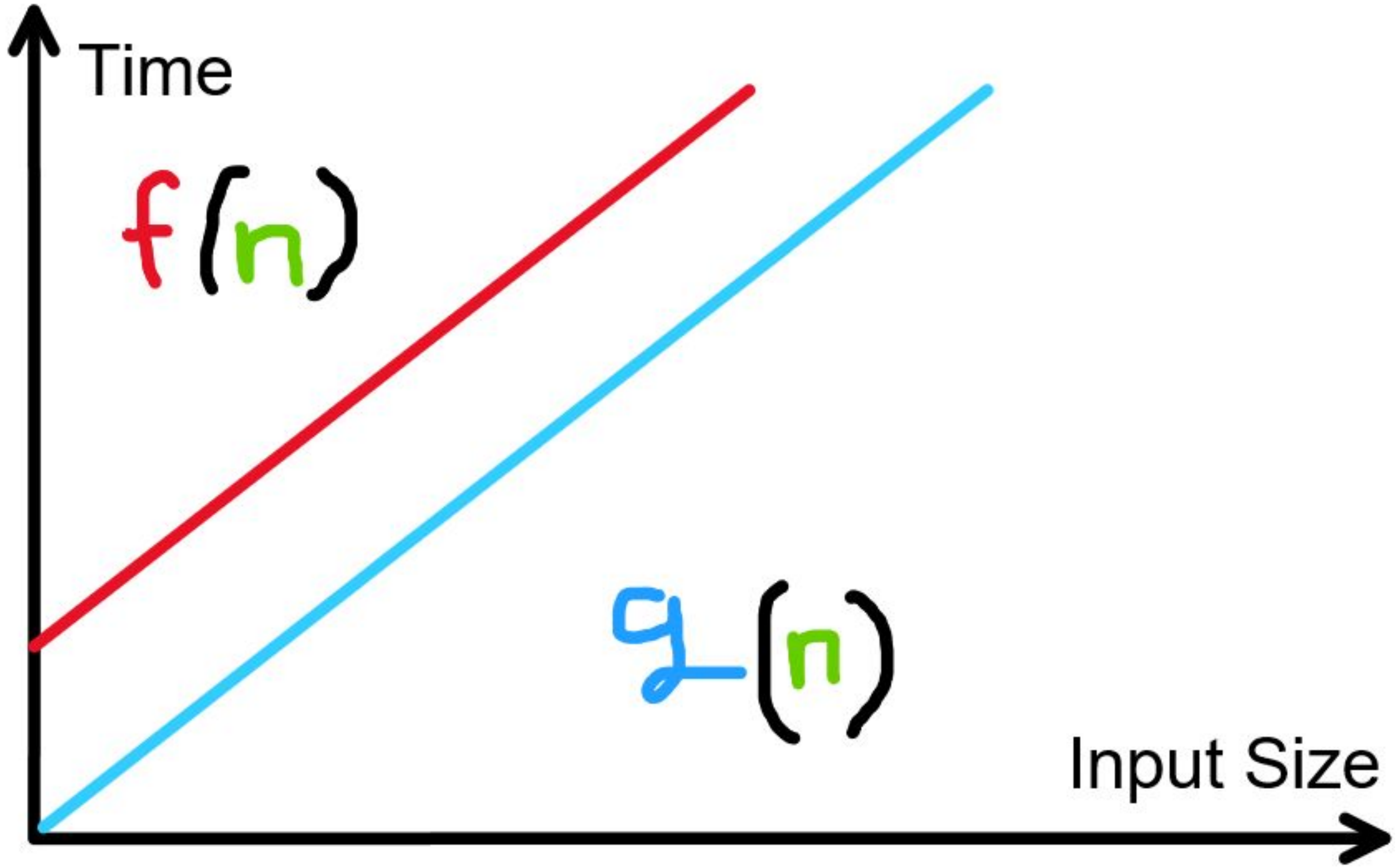
Why c?



Why n_0?



Why c?



Timeline

- What do we care about?
- Analyzing Code
    - Counting code constructs
    - Best Case vs. Worst Case
- Asymptotic Analysis
    - Big-Oh (and Big-) Definition
- Big-Oh Summary
- Cases vs Asymptotics
- Amortization