CSE 332: Data Structures & Parallelism
Lecture 2: Algorithm Analysis

Arthur Liu

Summer 2022

Announcements

- ”About you” survey! (See Ed)
- EX01
  - Resubmit as many times before deadline!
  - No late days!!
  - **Due Sunday night**
- Post-lecture PollEV make-up (must submit before next lecture)
- P1 Released
  - If you did not fill out partner form, fill it out ASAP
- EX02 releasing later today

Today – Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- Analyzing Code
- 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?

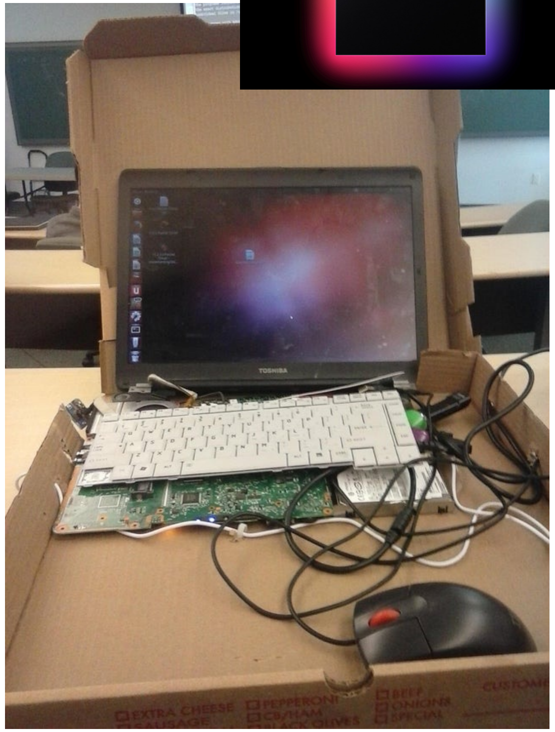
I have some problem I need solved.

I ask Dara and Hans. They both have different ideas for how to solve the problem. How do we know which is better?

Easy. Have them both write the code and run it and see which is faster.

THIS IS A TERRIBLE IDEA

A: How should we compare two algorithms?



- Uh, why NOT just run the program and time it??
  - Too much *variability*, not reliable or *portable*:
    - Hardware: processor(s), memory, etc.
    - OS, Java version, libraries, drivers
    - Other programs running
    - Implementation dependent
  - Choice of input (dataset)
    - Testing (inexhaustive) may *miss* worst-case input
    - Timing does not *explain* relative timing among inputs (what happens when *n* doubles in size)

- Often want to evaluate an *algorithm*, not an implementation
  - Even *before* creating the implementation ("coding it up")

A better strategy?

What we want:

Answer is *independent* of CPU speed, programming language, coding tricks, etc.

Large inputs (n) because probably any algorithm is "plenty good" for small inputs (if n is 10, probably anything is fast enough)

Answer is general and rigorous, complementary to "coding it up and timing it on some test cases"

- Can do analysis before coding!

Analyzing code ("worst case")... let's count!

Assume basic operations take "some amount of" constant time

- Arithmetic
- Assignment
- Access one Java field or array index
- Etc.

This is an approximation of reality: a very useful "lie"

A = B + C[i]

F = F + G

if (cond)
  stmt1
else
  stmt2

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Conditionals	Time of condition plus time of *slower* branch
Function Calls	Time of function's body
Recursion	Solve *recurrence equation*

Examples

worst case

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

6 operations → O(1)

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

1 + 5·n → O(n)

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

1+5n

1 + 1+5n → O(n)

What is the number of operations in this code? What is the 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
}
```

5n

n · 5n

1   100

n

1

$n \cdot 5n + 1 + n + 1$

$5n^2 + \cancel{1}+1+\cancel{n}$

$O(n^2)$

Examples

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

10,000,000

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

→ O(1)

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

Using Summations for Loops

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

1

inclusive

$$1 + \sum_{i=0}^{n-1} 5 = 1 + \underbrace{5 + 5 + 5}_{n \text{ times}} \ldots = 1 + 5n = O(n)$$

When math is helpful

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

$$\sum_{i=0}^{n-1} \underbrace{\sum_{j=0}^{i-1} 5} = \sum_{i=0}^{n-1} 5i = 5\sum_{i=0}^{n-1} i = 5\,\frac{n(n-1)}{2}$$

$$= \frac{5n^2}{2} - \frac{5n}{2}$$

$$\boxed{O(n^2)}$$

Complexity Cases

scenarios

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

What is the dataset like? What are the best/worst paths through our code?

✗ Incorrect to say: Best case is when N = 0

✓ Correct to say: Best case is…
 …when data is sorted
 …our algorithm gets lucky

Other Complexity Cases

Average-case complexity: what does "average" case even mean? What is an "average" dataset? Depends on your scenario

Amortized analysis: we'll talk about this one later in this course.

Example

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){
   ???
}
```

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;
}
```

n

Best case: find(2) - 6 - O(1)

Worst case: find(4), find(126)

$1 + 5 \cdot n \rightarrow O(n)$

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;
}
```

AGAIN, best case / worst case is INDEPENDENT of N (size of dataset)

Best case: 6 “ish” operations = O(1)

Worst case: 5 “ish” * (arr.length) = O(N)

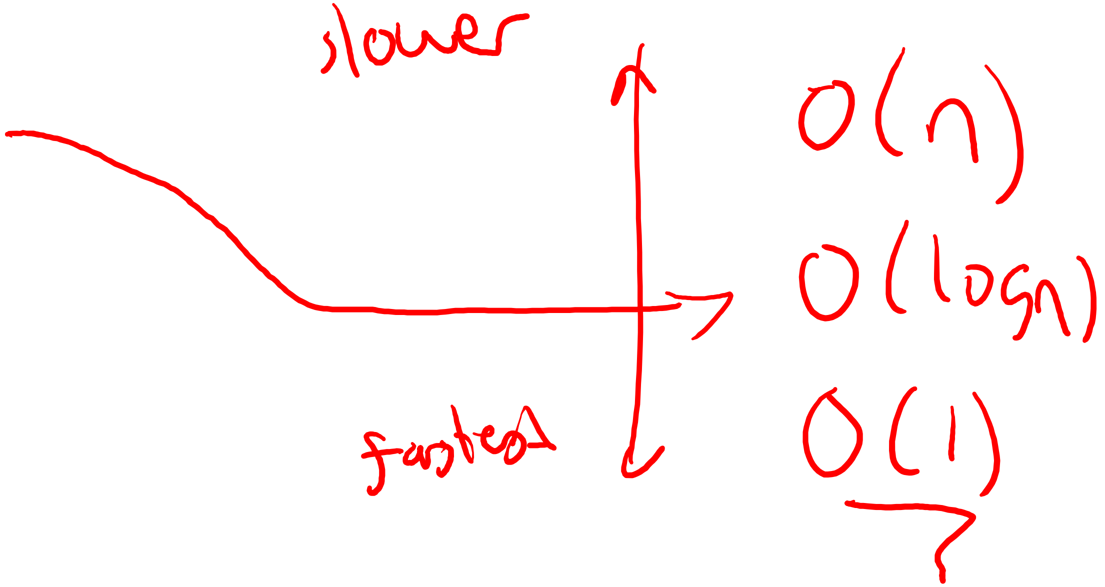
Remember a faster search algorithm?

Remember a faster search algorithm?

Worst Cases:

Binary Search – O(logn)

Linear Search – O(n)



Ignoring Constant Factors

- So binary search is O(log n) and linear is O(n)
  - But which will actually be faster?
  - Depending on **constant factors** and **size of n**, in a particular situation, linear search could be faster….
    - How *many* assembly instructions, assignments, additions, etc. for each *n*
- And could depend on size of *n*

- **But** there exists some n_0 such that for all $n > n_0$ binary search "wins"

- Let's play with a couple plots to get some intuition…

Example

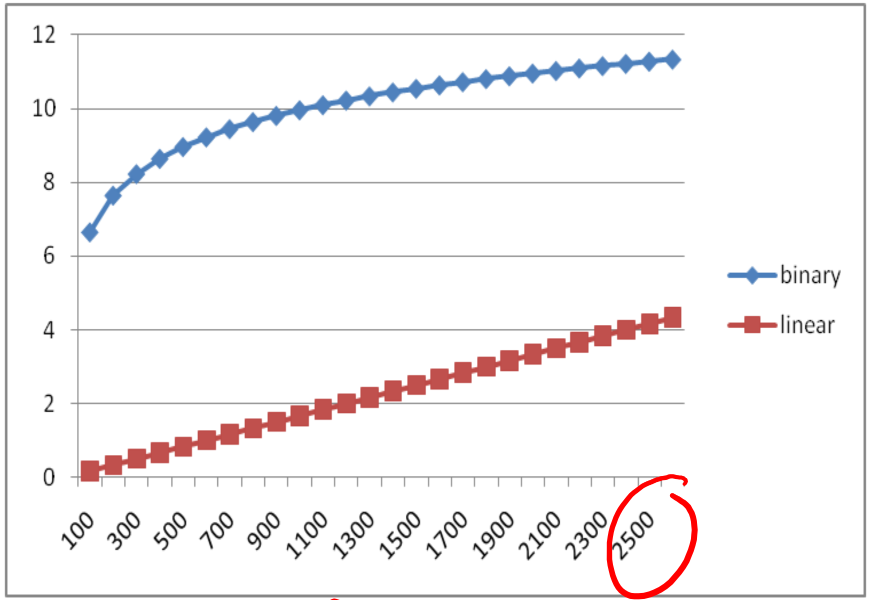
Let's "help" linear search

- Run it on a computer 100x as fast
- Use a new compiler/language that is 3x as fast
- Be a clever programmer to eliminate half the work

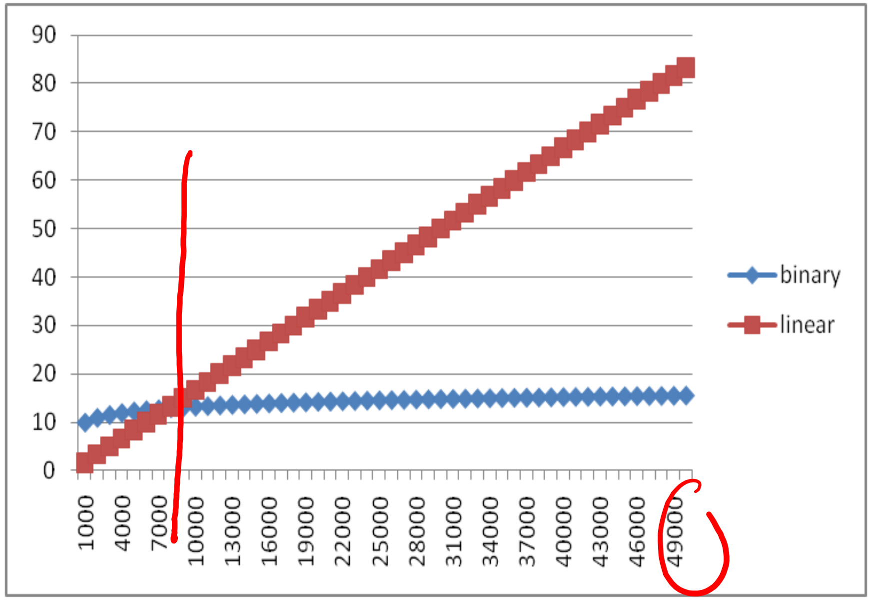
600x speedup!

- Note: 600x is still helpful for problems! (esp. when no better algorithm)

time



n



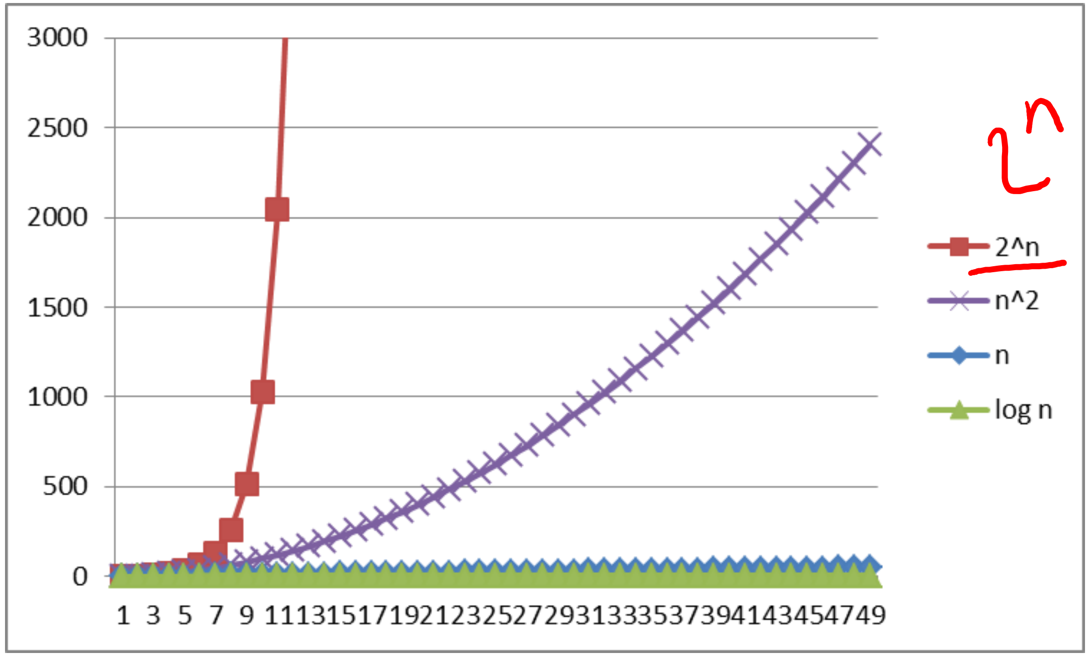
Logarithms and Exponents

Definition: $\log_2 x = y$ if $x = 2^y$

Logarithms *grow as slowly* as exponents grow *quickly*

So, $\log_2$ 1,000,000 = "a little under 20"

Since so much is binary in CS, $\log$ *almost always means* $\log_2$



2^n k^n

Log base doesn't matter much

"Any base *B* log is equivalent to base 2 log within a constant factor"

- **And we are about to stop worrying about constant factors!**
- In particular, $\log_2 x = 3.22 \log_{10} x$
- In general, we can convert log bases via a constant multiplier
- Say, to convert from base B to base A:

$$\boxed{\log_B x = (\log_A x) \mathbin{/} \underbrace{(\log_A B)}_{\text{constant}}}$$

Review: Properties of logarithms

- $\log(A*B) = \log A + \log B$
  - So $\log(N^k) = k \log N$
- $\log(A/B) = \log A - \log B$
- $\log_2 2^x = x$

Other functions with log

- $\texttt{log(log x)}$ is written $\texttt{log log x}$
    - Grows as slowly as 2^{2^x} grows fast
    - Ex: $\texttt{log log 4billion} \sim \texttt{log log } 2^{32} = \texttt{log 32} = 5$
- $\texttt{(log x)(log x)}$ is written $\texttt{log}^2\texttt{x}$
    - It is greater than $\texttt{log x}$ for all $\texttt{x} > 2$

$\log^3 x$ $(\log x)^2$

NOT THE SAME

Today

- What do we care about?
- How to compare two algorithms
- Analyzing Code
- Asymptotic Analysis
- Big-Oh Definition

Asymptotic Analysis

About to show formal definition, which amounts to saying:

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

Examples:

- $\cancel{4}n + \cancel{5}$ $\rightarrow$ $O(n)$
- $\cancel{0.5}n\ \mathbf{log}\ n + \cancel{2}n + \cancel{7}$ $\quad n \lg n + n \quad O(n \lg n)$
- $n^3 + 2^n + 3n$ $\quad O(2^n)$
- $n\ \mathbf{log}\ (10n^2)$

$n \lg(10n^2) \rightarrow n(\cancel{\log 10} + \log n^2)$

$n \cdot 2 \cdot \lg n \in O(n \lg n)$

Big-Oh relates functions

We use O on a function f(n) 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**

Less ideal:

Confusingly, we also say/write:

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

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

Formally Big-Oh

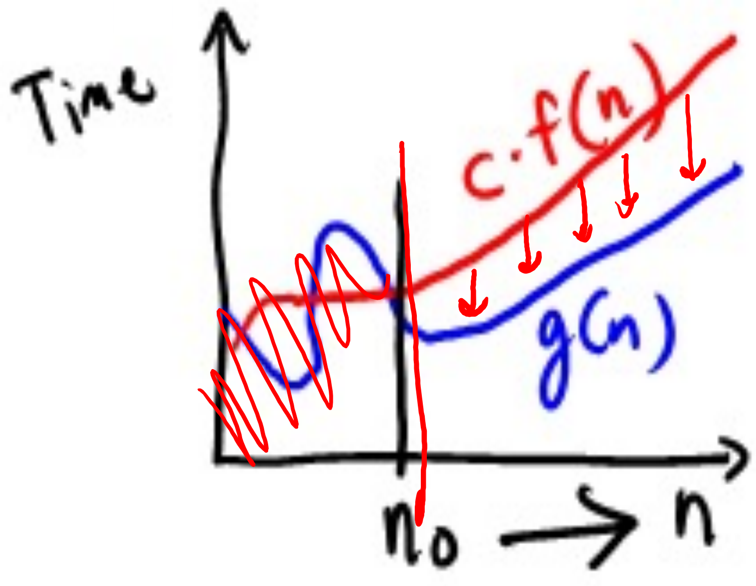
$O(n)$ n^2

$3n+4$

Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

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

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$



Formally Big-Oh

Definition: g(n) is in O(f(n)) iff there exist positive constants c and n_0 such that

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

1, 2, 3, ...

Note: $n_0 \geq 1$ (and a natural number) and $c > 0$

Example: Let $g(n) = 3n + 4$ and $f(n) = n$

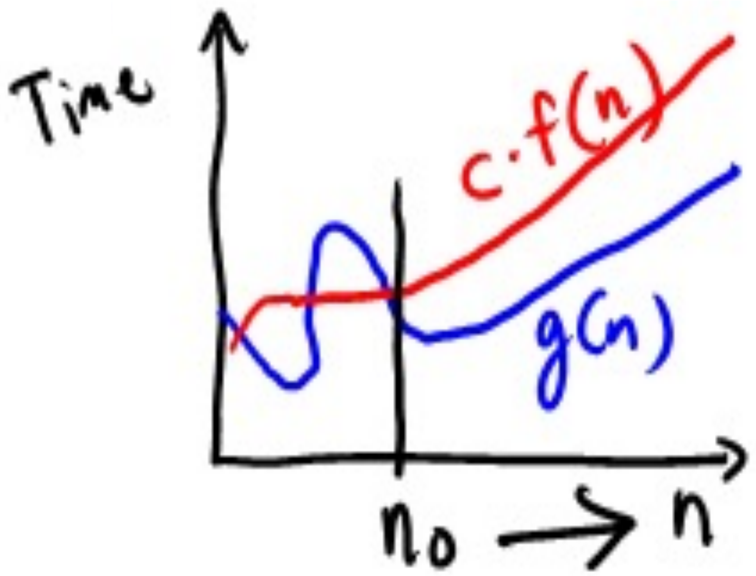
$c = 4$ and $n_0 = 5$ is one possibility

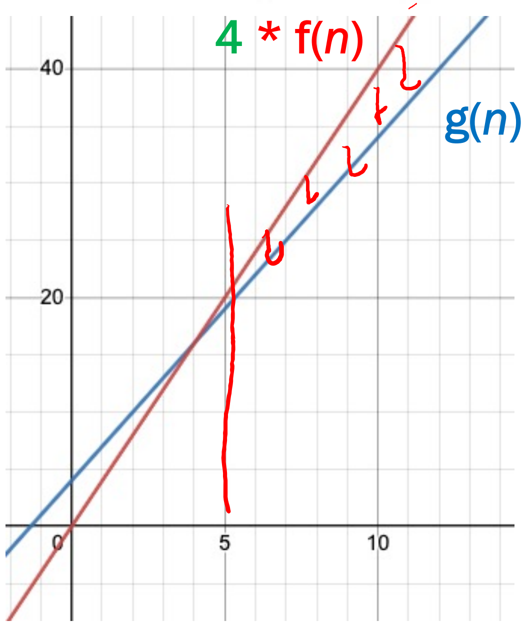
$3n+4 \leq c \cdot n$

$3n+4 \leq 4n$

$19 \leq 20$

try:
$c = 4$
$n_0 = 5$

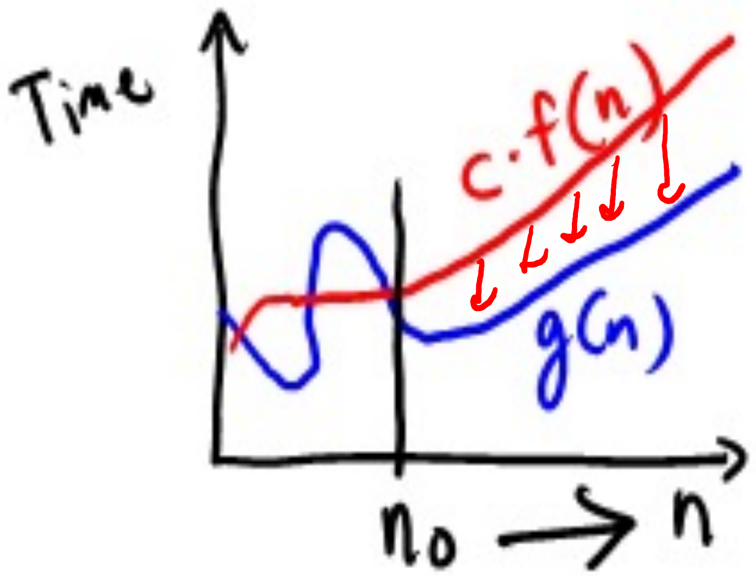




Formally Big-Oh

Definition: $g(n)$ is in $O(\ f(n)\ )$ iff there exist positive constants c and n_0 such that

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

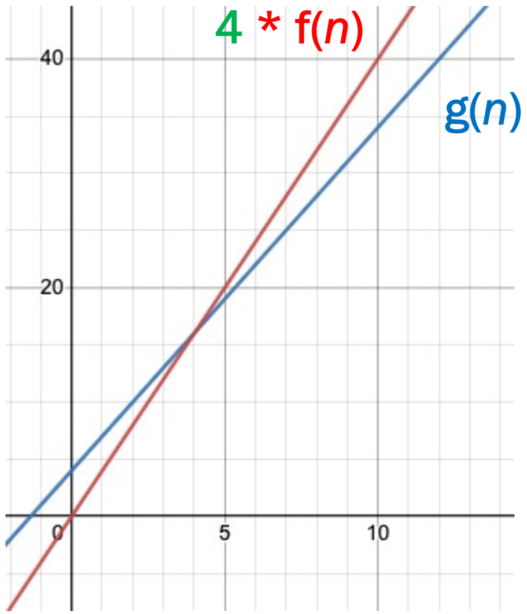


Note: $n_0 \geq 1$ **(and a natural number) and** $c > 0$

Example: Let $g(n) = 3n + 4$ and $f(n) = n$

$c = 4$ and $n_0 = 5$ is one possibility

~~$O(n^1)$~~



This is "less than or equal to"

- So $3n + 4$ is also $O(n^5)$ and $O(2^n)$ etc.

$O(n)$

Why n_0?

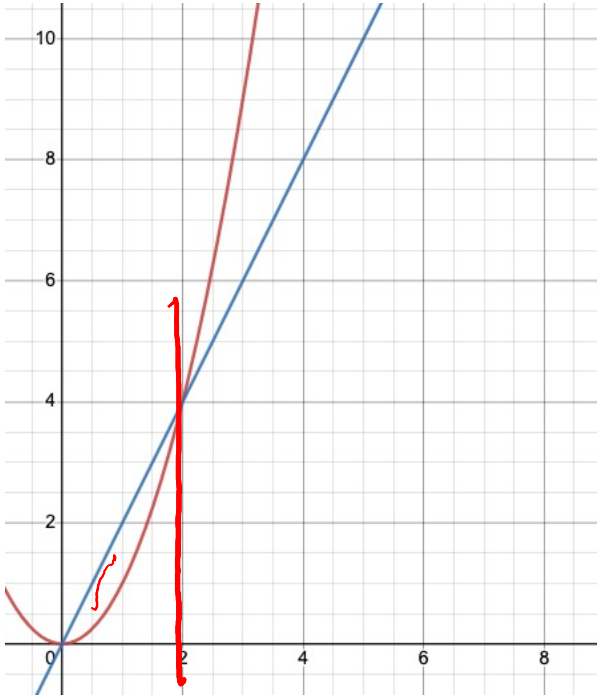
n_0 gives time for the higher-order terms to cover the lower-order ones

Example:

$g(n) = 2n$

$f(n) = n^2$

$2n$ is in $O(n^2)$, but $2n$ is only smaller when n exceeds 2

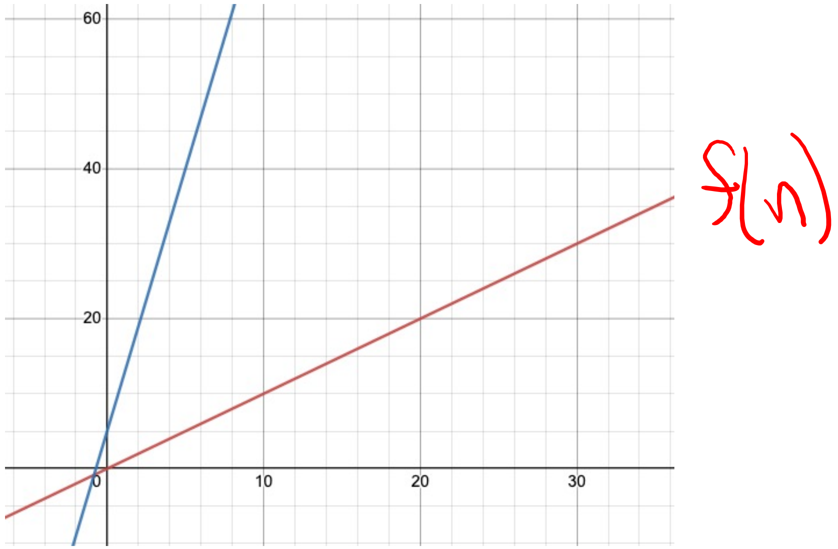


Why c?

- The constant multiplier (called c) allows functions with the same asymptotic behavior to be grouped together
  - Pick a c large enough to "cover the dropped constant factors"

$g(n) = 7n+5$
$f(n) = n$

It's true:
$g(n)$ is in $O(\ f(n)\ )$



- There is no positive n_0 such that $g(n) \leq f(n)$ for all $n \geq n_0$

Why c?

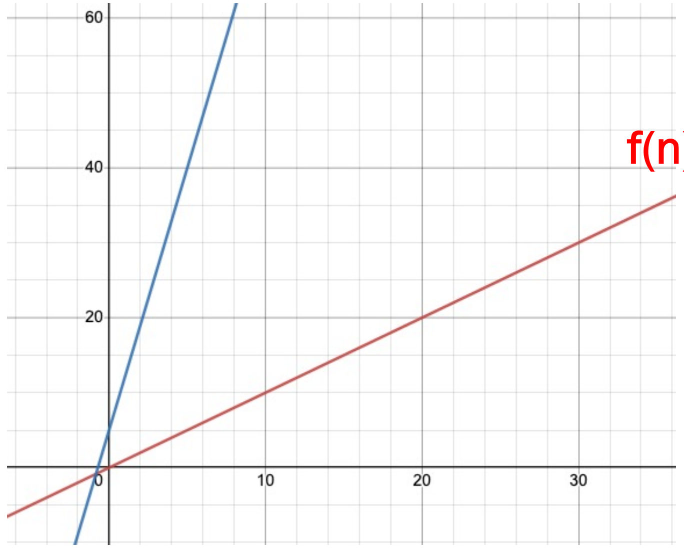
$g(n) = 7n+5$

$f(n) = n$

- The 'c' in the definition fixes this! for that:

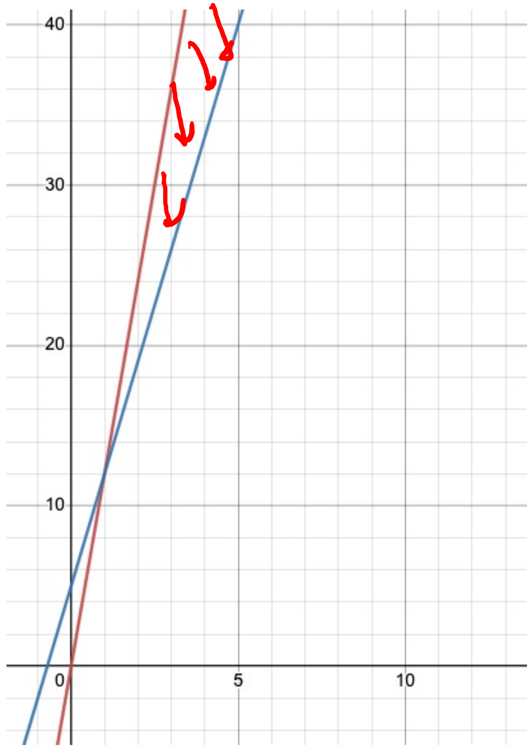
$$g(n) \leq c\, f(n) \qquad \text{for all } n \geq n_0$$

- To show g(n) is in O(f(n)), have $c = 12$, $n_0 = 1$



f(n) = n

f(n) = 12n



Working through an example

n_0 c
4 4

2

To show g(n) is in O(f(n)), pick a c large enough to "cover the constant factors" and n_0 large enough to "cover the lower-order terms"

- Example: Let g(n) = $4n^2 + 3n + 4$ and f(n) = n^3

$$4n^2 + 3n + 4 \le 4n^3 + 3n^3 + 4n^3 \le 11n^3 \le c \cdot n^3$$

$n \ge 1$

$\boxed{c = 11 \quad n_0 = 1}$ → 12

$n_0 = 2$

Big Oh: Common Categories

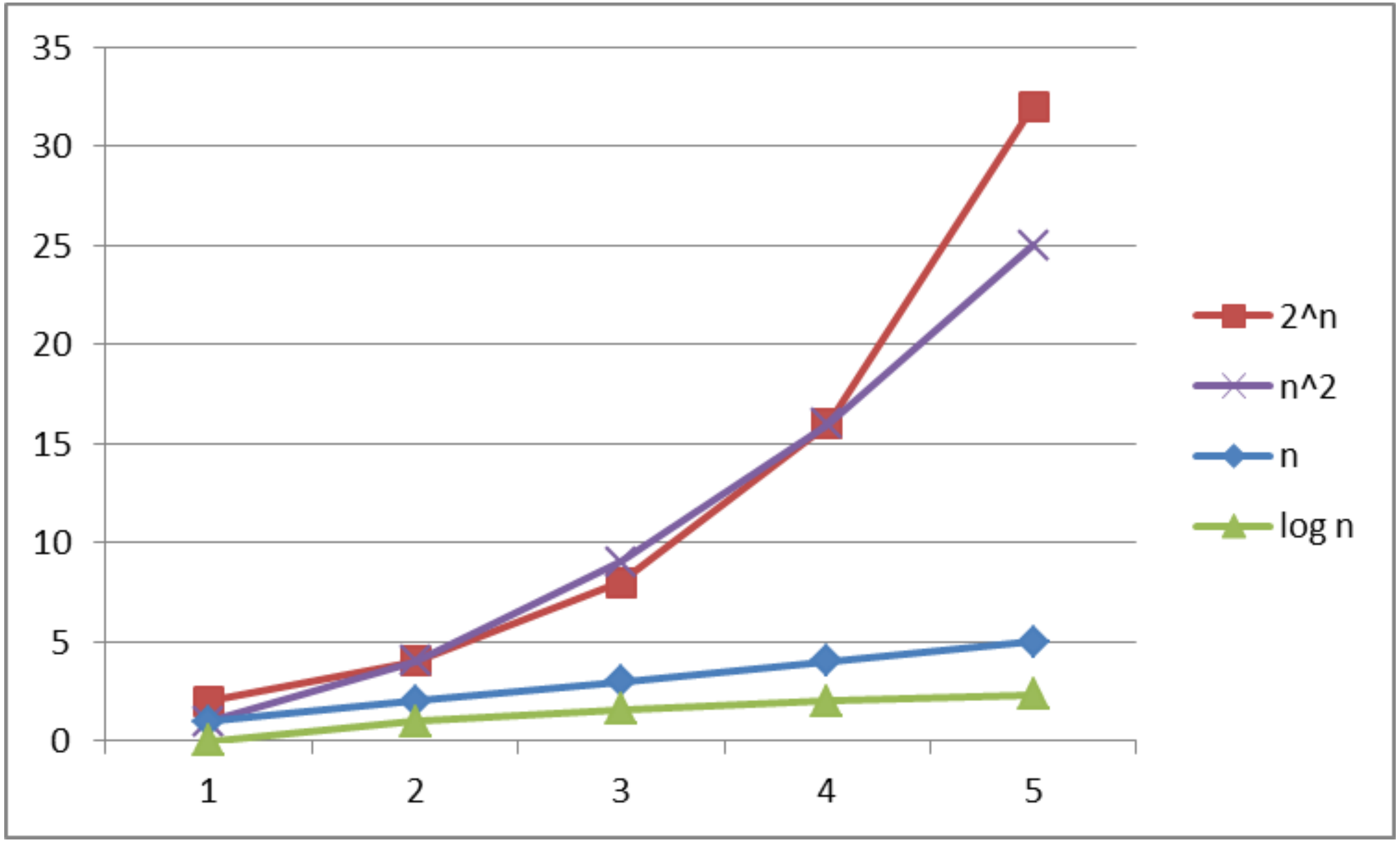
From fastest to slowest

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\mathtt{log}\ n)$	logarithmic
$O(n)$	linear
$O(\mathrm{n}\ \mathtt{log}\ n)$	"n **log** *n*"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant > 1)
$O(k^n)$	exponential (where k is any constant > 1)

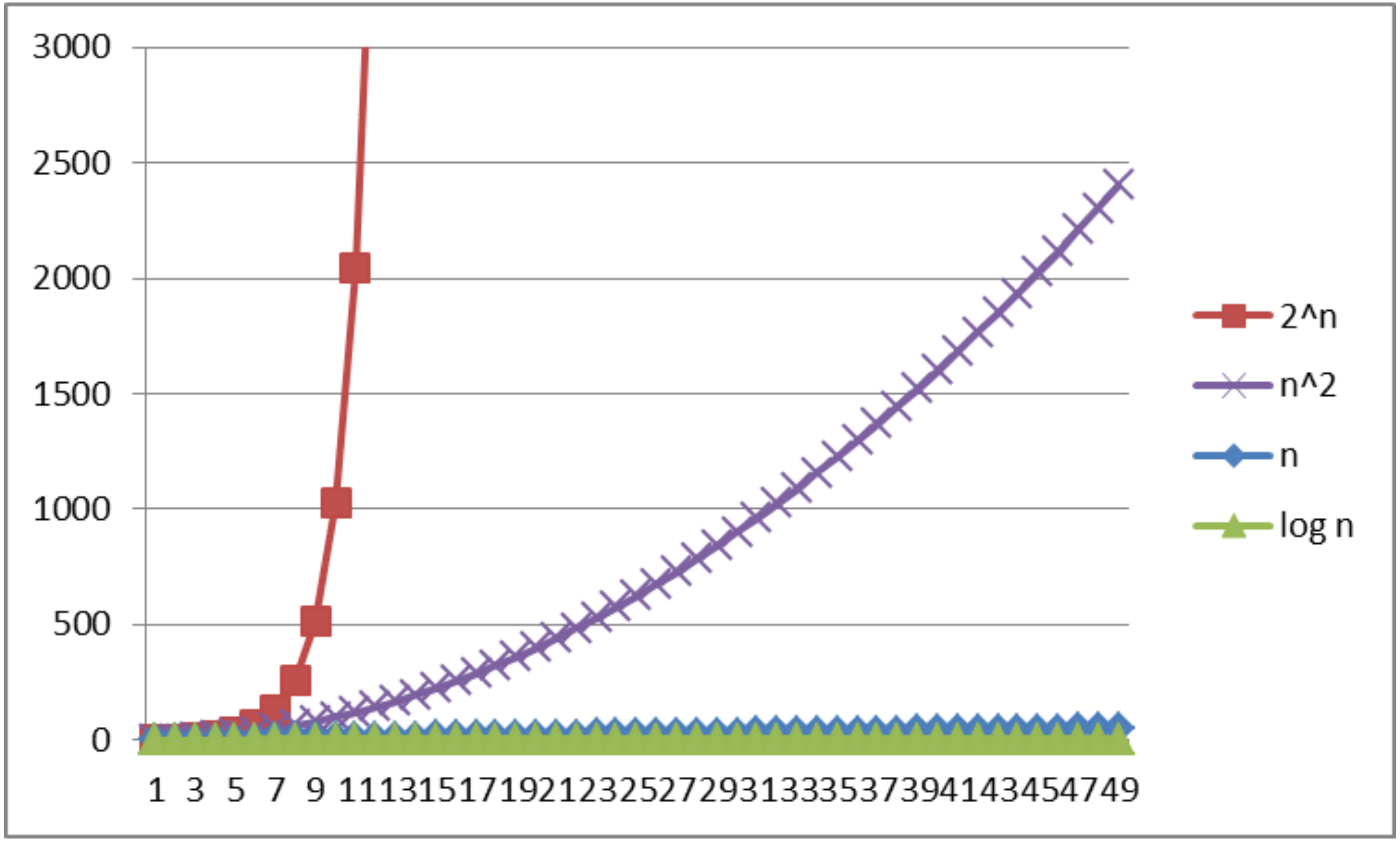
Note: *Don't write* $O(5n)$ instead of O(n) – same thing!
It's like writing 6/2 instead of 3. Looks weird

Usage note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1"

Big Oh: Common Categories



Big Oh: Common Categories



Poll Everywhere

pollev.com/artliu

True or false? (If true, what is a possible c and n_0)

1. `4+3n` is in `O(n)`
2. `n+2logn` is in `O(logn)`
3. `logn+2` is in `O(1)`
4. n^{50} is in $O(1.1^n)$

Notes:

- Do NOT ignore constants that are not multipliers:
  - n^3 is $O(n^2)$: FALSE
  - 3^n is $O(2^n)$: FALSE
- When in doubt, refer to the definition

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) (We can prove it!)

More asymptotic analysis (more detail next time)

Upper bound: $O($ **f(n)** $)$

g(n) is in $O($ **f(n)** $)$ if there exist constants c and n_0 such that

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

Lower bound: $\Omega($ **f(n)** $)$

g(n) is in $\Omega($ **f(n)** $)$ if there exist constants c and n_0 such that

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

Tight bound: $\theta($ **f(n)** $)$

g(n) is in $\theta($ **f(n)** $)$ if it is in $O($ **f(n)** $)$ and it is in $\Omega($ **f(n)** $)$

Next

- More asymptotic analysis (theta, omega, little-oh)
- Mentioning Big-Oh proofs
- Heaps

- EX02 released after lecture