



CSE332: Data Structures & Parallelism

Lecture 2: Algorithm Analysis

Ruth Anderson
Winter 2022

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?

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
 - 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”)

Comparing algorithms

When is one *algorithm* (not *implementation*) better than another?

- Various possible answers (clarity, security, ...)
- But a big one is *performance*: for sufficiently large inputs, runs in less time (our focus) or less space

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

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

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

- Can do analysis before coding!

Today – Algorithm Analysis

- What do we care about?
- How to compare two algorithms
- **Analyzing Code**
 - **How to count different code constructs**
 - **Best Case vs. Worst Case**
 - **Ignoring Constant Factors**
- Asymptotic Analysis
- Big-Oh Definition

Analyzing code (“worst case”)

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”.)

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

```
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++;  
    }  
}
```

Another Example

```
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
}
```

Using Summations for Loops

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

Complexity cases

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

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;
}
```

Best case:

Worst case:

Linear search – Running Times

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: 6 “ish” steps = $O(1)$
Worst case: 5 “ish” * (arr.length)
= $O(\text{arr.length})$

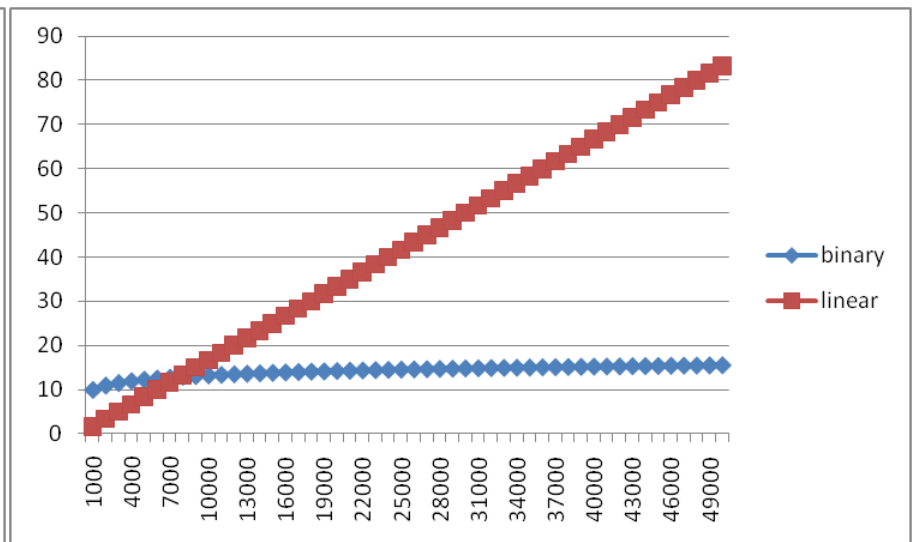
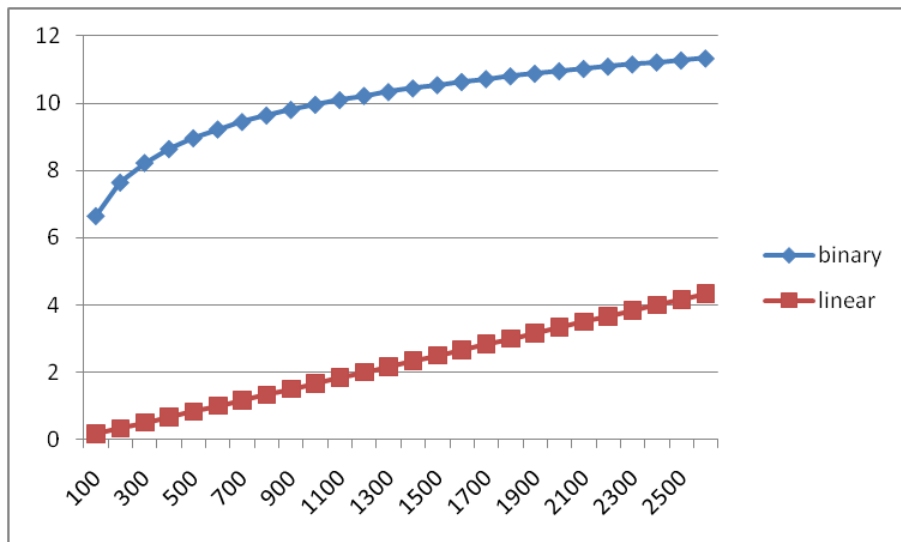
Remember a faster search algorithm?

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**....
- Could depend on constant factors
 - How *many* 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

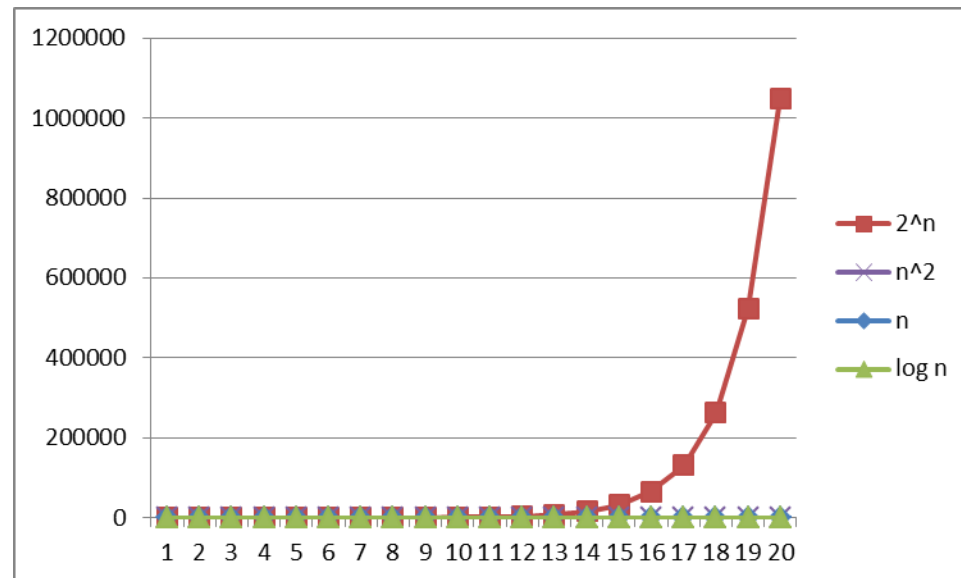
- Let's try to “help” linear search
 - Run it on a computer 100x as fast (say 2018 model vs. 1990)
 - Use a new compiler/language that is 3x as fast
 - Be a clever programmer to eliminate half the work
 - So doing each iteration is 600x as fast as in binary search
- Note: 600x still helpful for problems without logarithmic algorithms!



Logarithms and Exponents

- Since so much is binary in CS, \log almost always means \log_2
- Definition: $\log_2 x = y$ if $x = 2^y$
- So, $\log_2 1,000,000 =$ “a little under 20”
- Just as exponents grow *very* quickly, logarithms grow *very* slowly

See Excel file
for plot data –
play with it!



Aside: Log base doesn't matter (much)

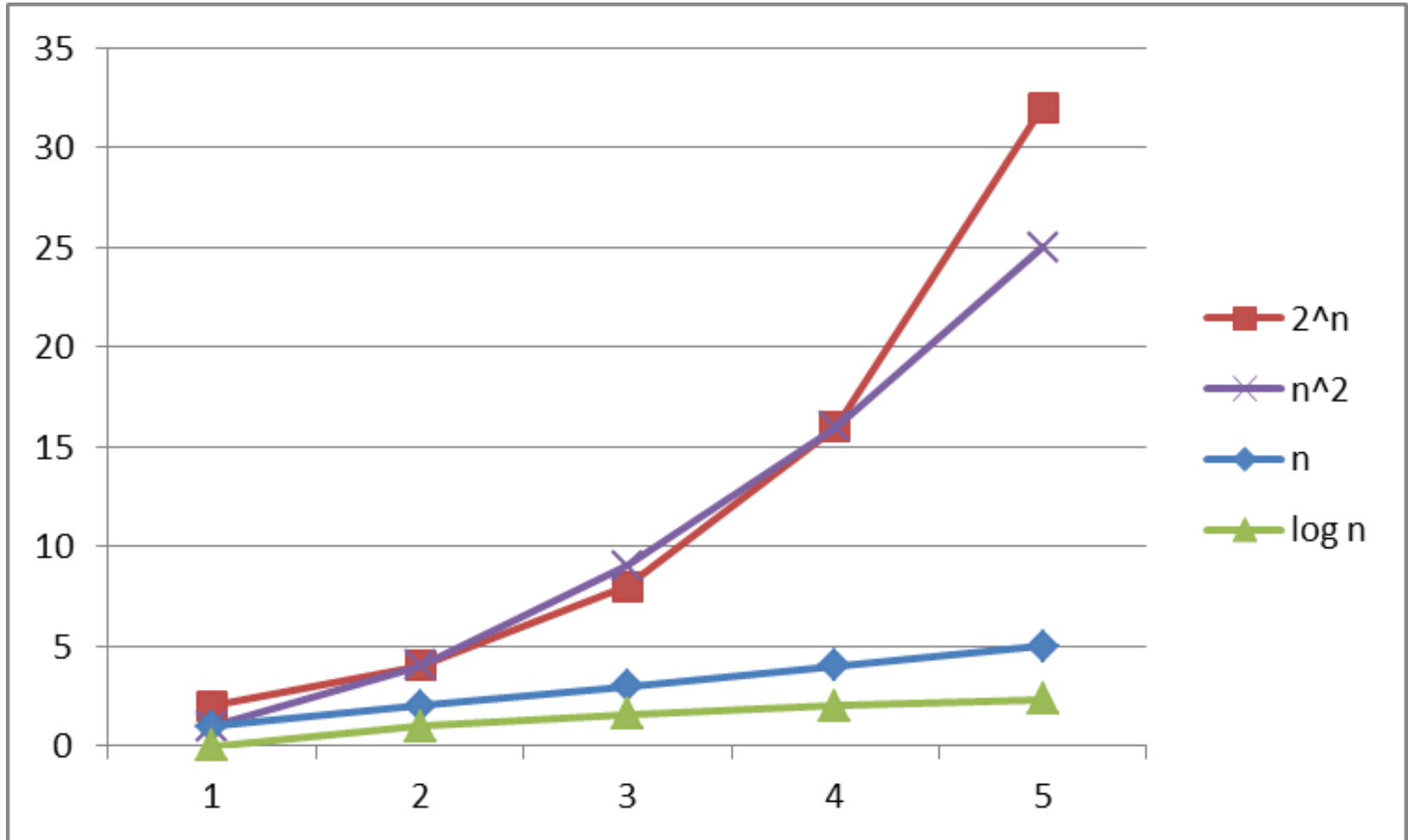
- “Any base B log is equivalent to base 2 log within a constant factor”
- **And we are about to stop worrying about constant factors!**
 - In particular, $\log_2 x = 3.22 \log_{10} x$
 - In general, we can convert log bases via a constant multiplier
 - Say, to convert from base B to base A :

$$\log_B x = (\log_A x) / (\log_A B)$$

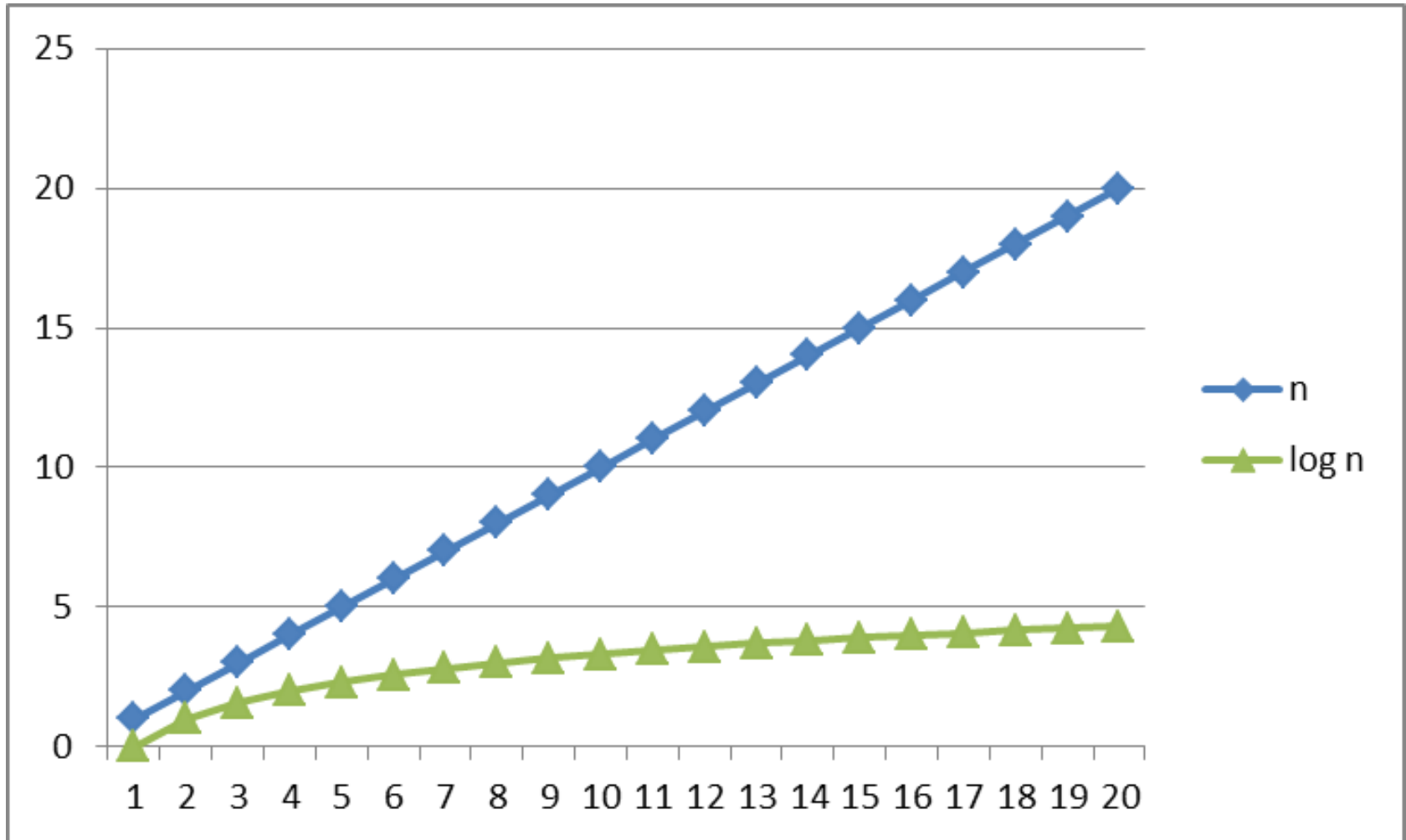
Review: Properties of logarithms

- $\log(A*B) = \log A + \log B$
 - So $\log(N^k) = k \log N$
- $\log(A/B) = \log A - \log B$
- $x = \log_2 2^x$
- $\log(\log x)$ is written $\log \log x$
 - Grows as slowly as 2^{2^y} grows fast
 - Ex:
$$\log_2 \log_2 4\text{billion} \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$$
- $(\log x)(\log x)$ is written $\log^2 x$
 - It is greater than $\log x$ for all $x > 2$

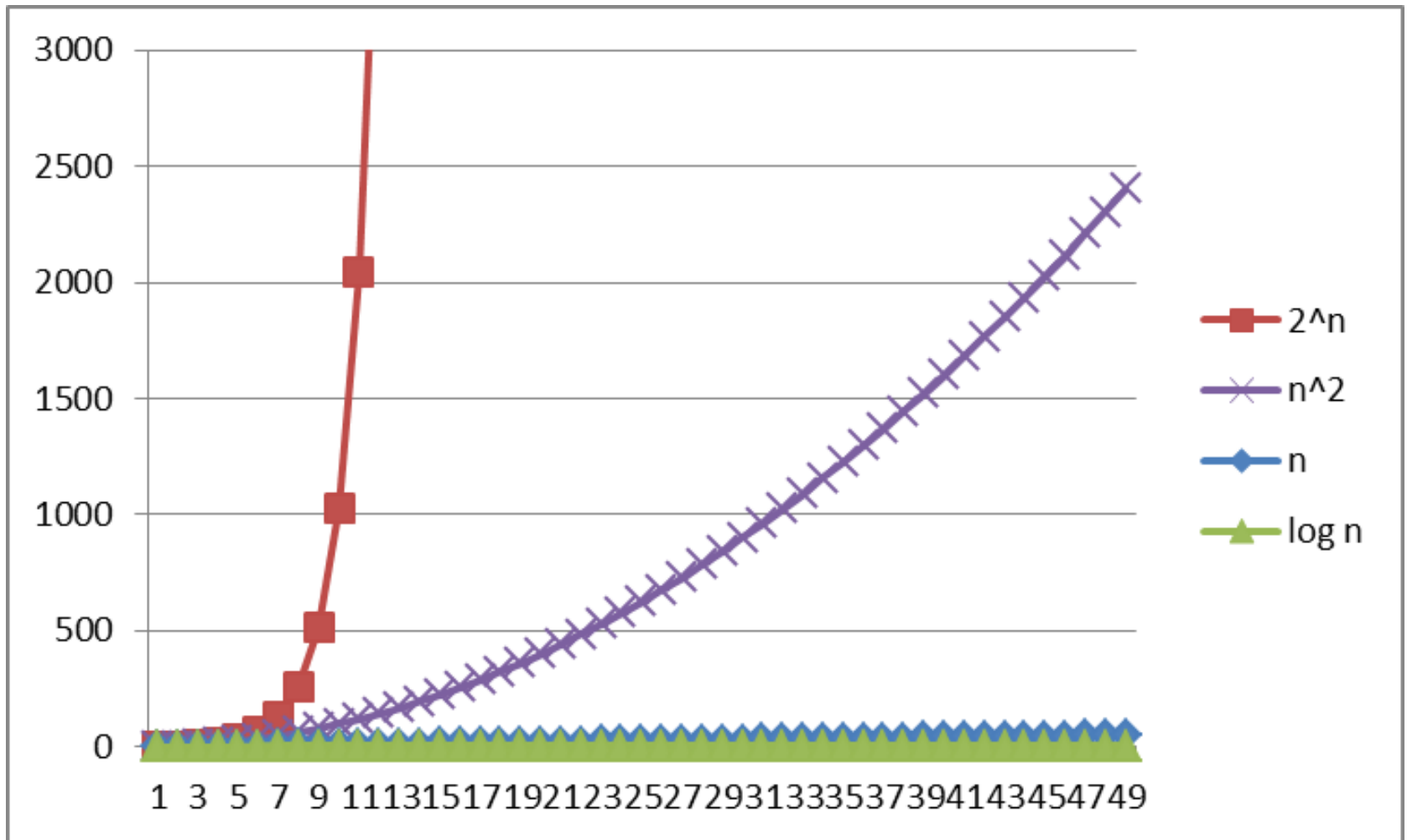
Logarithms and Exponents



Logarithms and Exponents



Logarithms and Exponents



Today – Algorithm Analysis

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

Asymptotic notation

About to show formal definition, which amounts to saying:

1. Eliminate low-order terms
2. Eliminate coefficients

Examples:

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

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

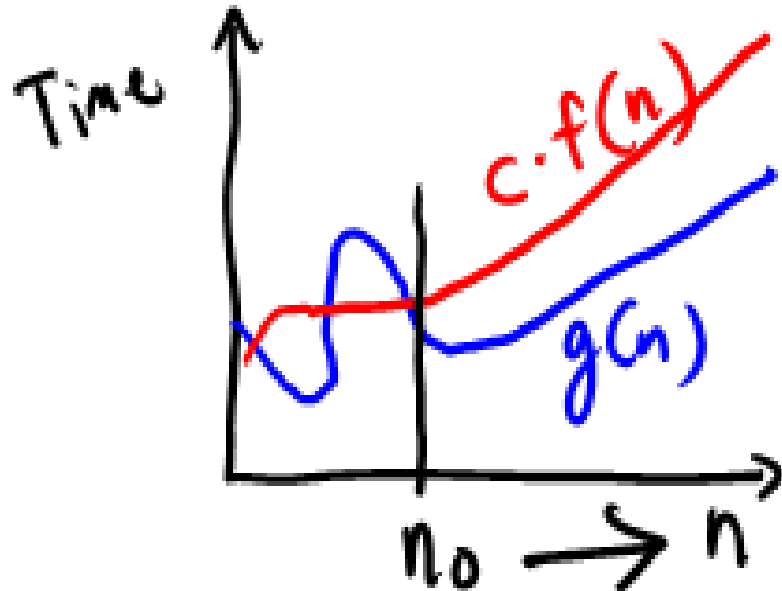
But we would never say $O(n^2) = (3n^2+17)$

Formally Big-Oh

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

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

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



Why n_0 ? Why c ?

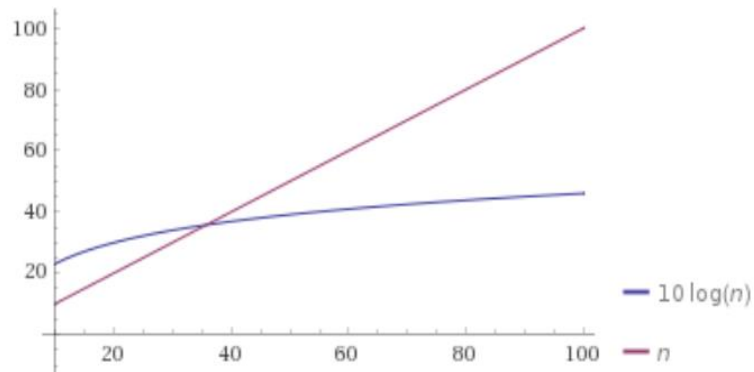
Definition: $g(n)$ is in $O(f(n))$ iff there exist positive constants c and n_0 such that

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

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

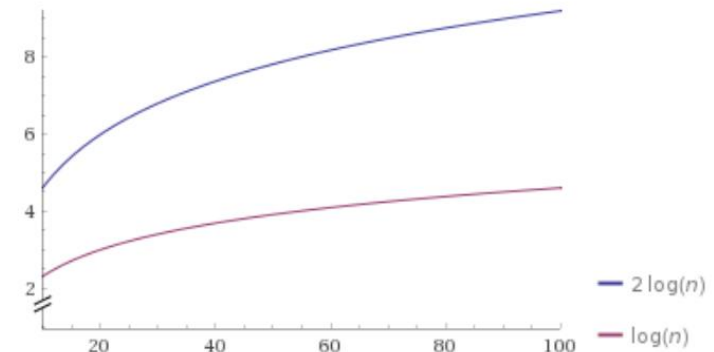
Why n_0 ?

Plot:



Why c ?

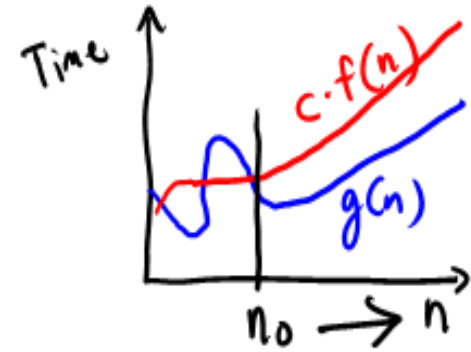
Plot:



Formally Big-Oh

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

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



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

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 $g(n) = 3n + 4$ and $f(n) = n$
 $c = 4$ and $n_0 = 5$ is one possibility

This is “less than or equal to”

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

What's with the **c**?

- To capture this notion of similar asymptotic behavior, we allow a constant multiplier (called **c**)
- Consider:
$$\mathbf{g(n)} = 7n+5$$
$$\mathbf{f(n)} = n$$
- These have the same asymptotic behavior (linear), so **g(n)** is in $O(\mathbf{f(n)})$ even though **g(n)** is always larger
- There is no positive n_0 such that $\mathbf{g(n)} \leq \mathbf{f(n)}$ for all $n \geq n_0$
- The '**c**' in the definition allows for that:
$$\mathbf{g(n)} \leq \mathbf{c f(n)} \quad \text{for all } n \geq n_0$$
- To show **g(n)** is in $O(\mathbf{f(n)})$, have $\mathbf{c} = 12$, $n_0 = 1$

An Example

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 $f(n) = n^3$

Examples

True or false?

1. $4+3n$ is $O(n)$
2. $n+2\log n$ is $O(\log n)$
3. $\log n+2$ is $O(1)$
4. n^{50} is $O(1.1^n)$

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)

Big Oh: Common Categories

From fastest to slowest

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

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

More Asymptotic Notation

- **Upper bound:** $O(f(n))$ is the set of all functions asymptotically less than or equal to $f(n)$
 - $g(n)$ is in $O(f(n))$ if there exist constants c and n_0 such that
$$g(n) \leq c f(n) \text{ for all } n \geq n_0$$
- **Lower bound:** $\Omega(f(n))$ is the set of all functions asymptotically greater than or equal to $f(n)$
 - $g(n)$ is in $\Omega(f(n))$ if there exist constants c and n_0 such that
$$g(n) \geq c f(n) \text{ for all } n \geq n_0$$
- **Tight bound:** $\theta(f(n))$ is the set of all functions asymptotically equal to $f(n)$
 - Intersection of $O(f(n))$ and $\Omega(f(n))$ (can use *different* c values)

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 $O(\log n)$

Less common notation:

- “little-oh”: like “big-Oh” but strictly less than
 - Example: sum is $o(n^2)$ but not $o(n)$
- “little-omega”: like “big-Omega” but strictly greater than
 - Example: sum is $\omega(\log n)$ but not $\omega(n)$

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 very 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

Other things to analyze

- Space instead of time
 - Remember we can often use space to gain time
- Average case
 - Sometimes only if you assume something about the distribution of inputs
 - See CSE312 and STAT391
 - Sometimes uses randomization in the algorithm
 - Will see an example with sorting; also see CSE312
- Sometimes an *amortized guarantee*

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