CSE 373: Data Structures and Algorithms

Pep Talk; Algorithm Analysis

Riley Porter

Winter 2017

Announcements

- Optional Java Review Section: **PAA A102 Tuesday, January 10th, 3:30-4:30pm**. Any materials covered will be posted online. TAs will be around after the session to answer questions.

- TA office hours will be posted on the website later today

- HW1 released after class today
  - Extension: Due **Tuesday, January 17th at 11:00PM**

Assumed Knowledge (Pep Talk)

- **Objects in Java**: fields, methods, encapsulation, inheritance, interfaces, classes
- **Being the Client of Data Structures**: List, Set, Stack, Queue, Map, Trees
- **Being the implementer of Data Structures:** ArrayList, LinkedList, Binary Trees, ***testing your objects***
- **Binary Search, and some awareness of how sorting works**: merge sort, selection sort.
- **Some basic analysis about above**. Examples:
  - when to use a HashSet vs a TreeSet?
  - when does adding into an ArrayList become expensive?
  - why does sorting an array help for searching?

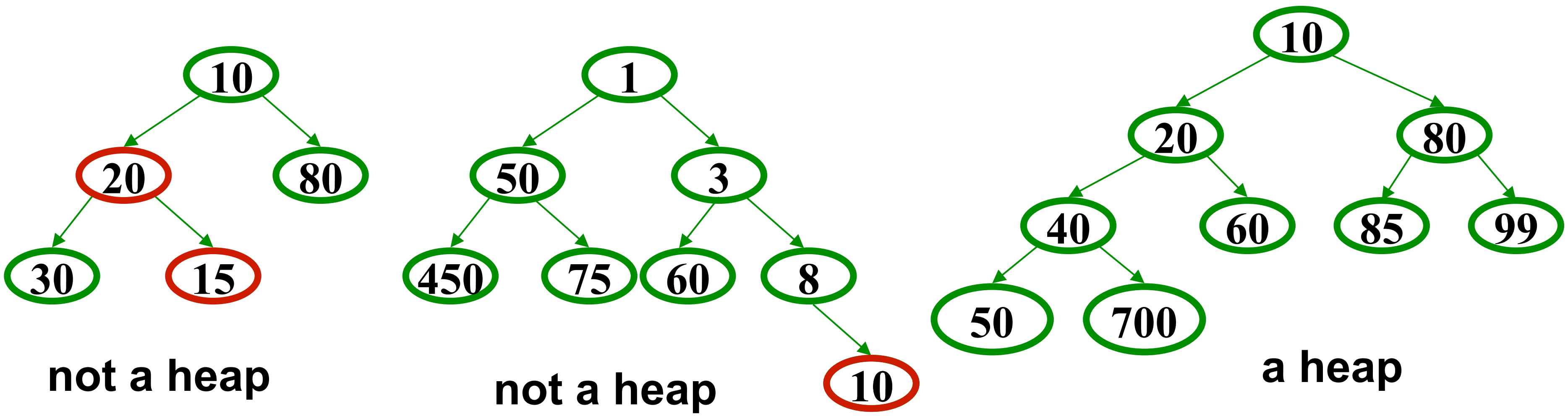
NOT Assumed Knowledge

- Full understanding of the difference between the definitions of **ADTs vs Data Structures vs Implementations**
- **Big O analysis**. Maybe you have some awareness, but we don't expect mastery of Big O yet.
- Indepth analysis or **mastery of sorting or hashing**.
- Anything at all about Graphs, Heaps, AVL Trees, Union Find, Disjoint Sets, Hashing, Topological Sort, Parallelism
- Any advanced algorithms, dynamic programming, P vs NP, complexity theory, proofs, induction

Review of last time: Heaps

Heaps follow the following two properties:

- Structure property: A *complete* binary tree
- Heap order property: The priority of the children is always a greater value than the parents (greater value means less priority / less importance)



Today – Algorithm Analysis

- Review math for algorithm analysis
  - Exponents and logarithms, floor and ceiling
- Analyzing code
- Big-O definition
- Using asymptotic analysis (continue next time)
- Set ourselves up to analyze why we use Heaps for Priority Queues (continue later this week)

Review of Logarithms

- $\mathbf{log_2 x=y}$ if $\mathbf{x=2^y}$ (so, $\mathbf{log_2}1{,}000{,}000$ = "a little under 20")
- Just as exponents grow *very* quickly, logarithms grow *very* slowly
- Log base B compared to log base 2 doesn't matter so much
  - In computer science we use base 2 because it works nicely with binary and how a computer does math.
  - we are about to stop worrying about constant factors
  - In particular, $\mathbf{log_2\ x = 3.22\ log_{10}\ x}$

Review of log properties

- **log(A*B) = log A + log B**
  - So $\log(N^k) = k \log N$
- **log(A/B) = log A – log B**
- **log(log x)** is written **log log x**
  - Grows as slowly as 2^{2^y} grows quickly
- **(log x)(log x)** is written $\log^2 x$
  - It is greater than **log x** for all **x > 2**
  - It is not the same as **log log x**

Review of floor and ceiling

$\lfloor X \rfloor$ Floor function: the largest integer $\leq$ X

$\lfloor 2.7 \rfloor = 2 \qquad \lfloor -2.7 \rfloor = -3 \qquad \lfloor 2 \rfloor = 2$

$\lceil X \rceil$ Ceiling function: the smallest integer $\geq$ X

$\lceil 2.3 \rceil = 3 \qquad \lceil -2.3 \rceil = -2 \qquad \lceil 2 \rceil = 2$

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 or less space
- Large inputs (n) because probably any algorithm is "fine" for small inputs
- Answer will be independent of CPU speed, programming language, coding tricks, etc.

Comparing Algorithms Example

4	2	5	1	8	6	10	9	3	7

- Given the above list, search for 3, which is better?
  - binary search
  - linear search

1	2	3	4	5	6	7	8	9	10

- Given the above list, search for 3, which is better?
  - binary search
  - linear search

Analyzing Algorithms

As the size of an algorithm's input grows:

- How much longer does the algorithm take (time)
- How much more memory does the algorithm need (space)

- Ignore constant factors, think about large input:
  - there exists some input size n_0, that for all input sizes n larger than n_0, binary search is better than linear search on sorted input
- Analyze code to compute runtime, then look at how the runtime behaves as **n gets really large** (asymptotic runtime)

Analyzing Code

"Constant time" operations:

- Arithmetic, Variable Assignment, Access one Java field or array index, etc

Complex operations (approximation):

- Consecutive Statements: *Sum of time of each statement*
- Conditionals: *Time of condition + max(ifBranch, elseBranch)*
- Loops: *Number of iterations* $*$ *Time for Loop Body*
- Function Calls: *Time of function's body*

Example

What is the runtime of this pseudocode:

```
x := 0
for i=1 to N do
  for j=1 to N do
     x := x + 3
return x
```

Example Solution

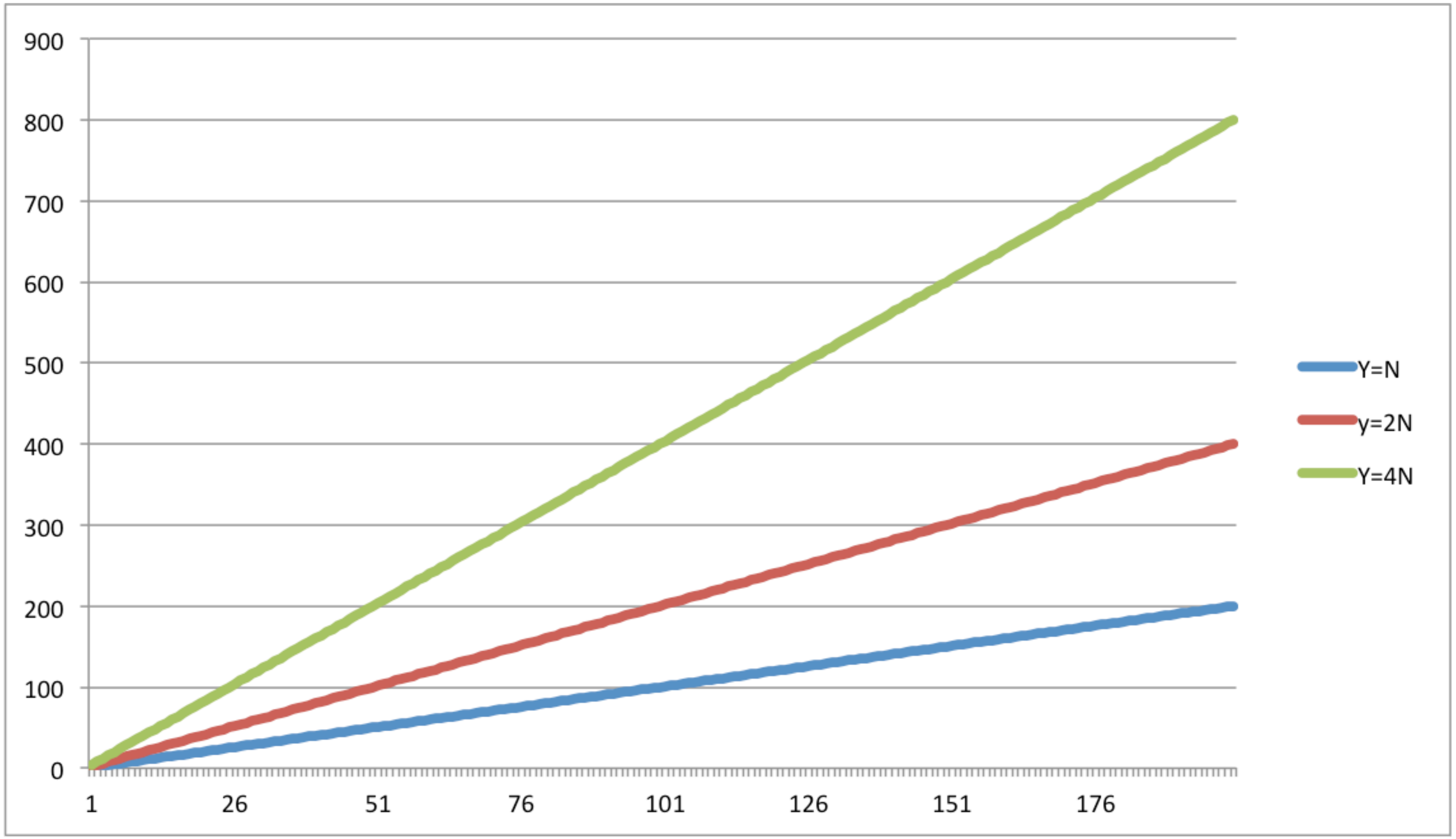
What is the runtime of this pseudocode:

```
x := 0
for i=1 to N do
  for j=1 to N do
     x := x + 3
return x
```

1 assignment +
(N iterations of loop *
  (N iterations of loop *
    1 assignment and math))
1 return

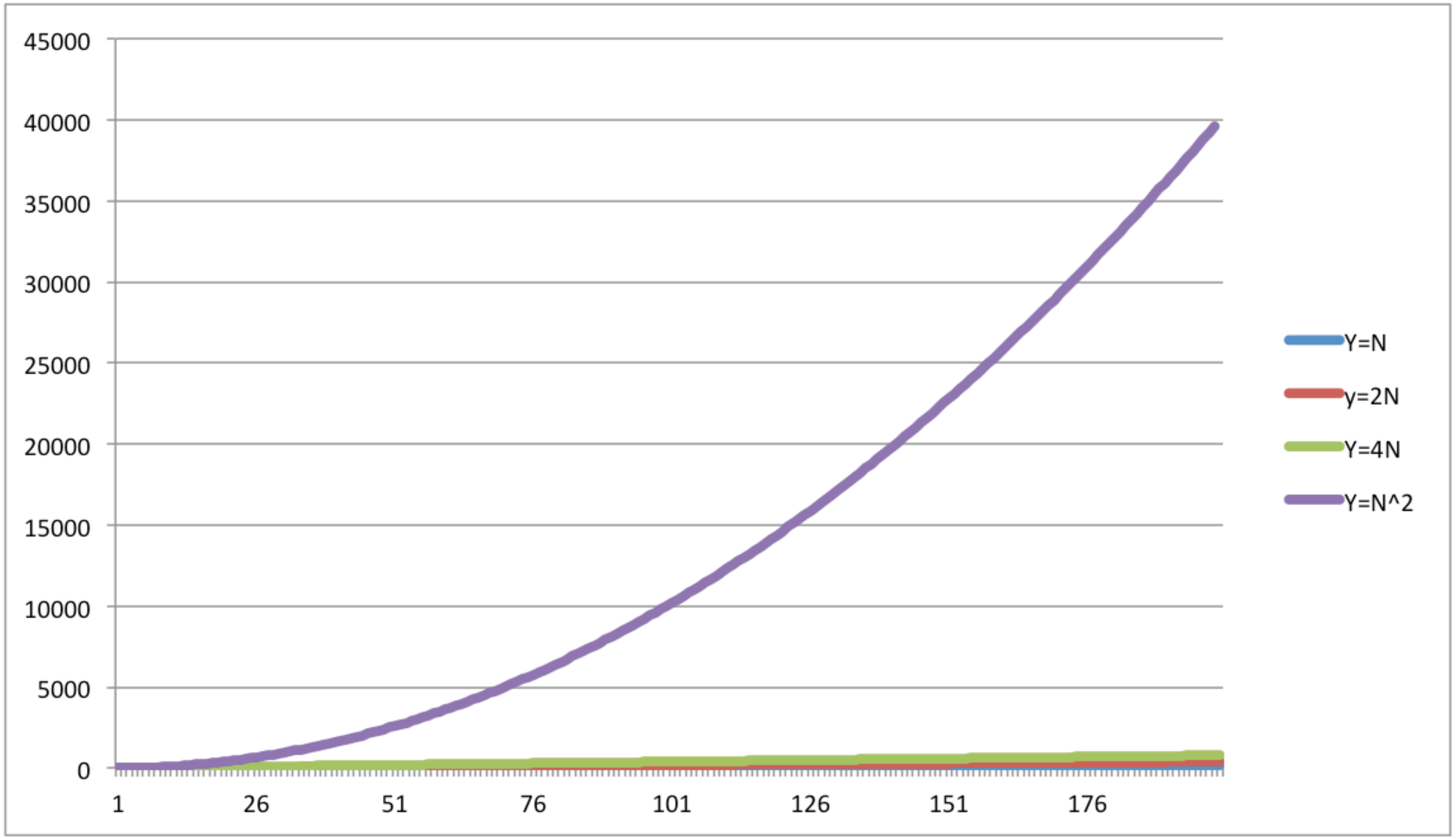
1 + (N * (N * 1)) + 1 = **N^2 + 2**
However, what we care about here is the N^2 part.
Let's look at asymptotic runtimes to see why.

Asymptotic Intuition with Pictures



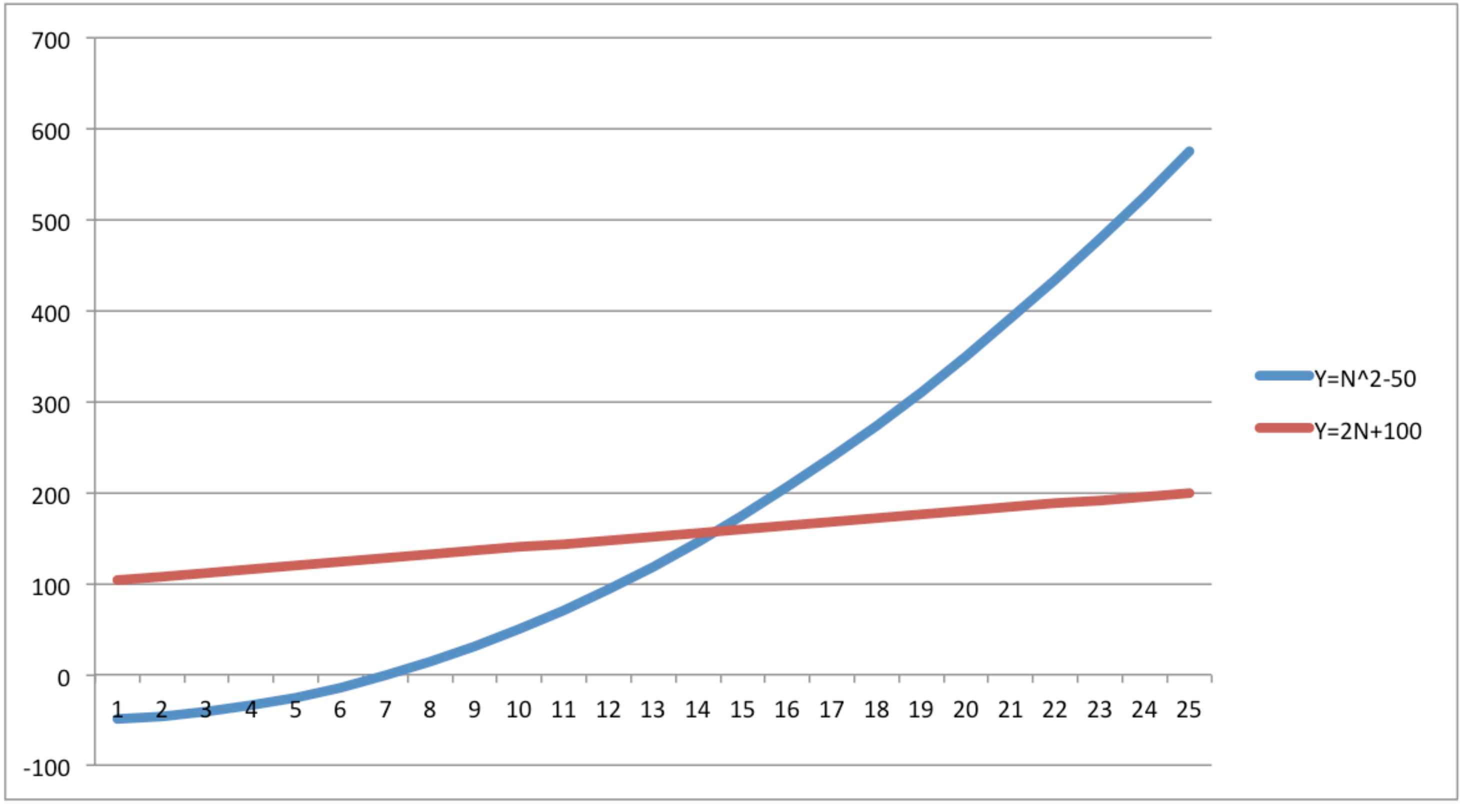
Are these the same?

Asymptotic Intuition with Pictures



What about now that we compare them to $y=N^2$?

Asymptotic Intuition with Pictures



What about these? One starts off much lower than the other one, but grows much faster.

Asymptotic Notation

About to show formal definition, which amounts to saying:

1. Calculate Runtime by analyzing code
2. Eliminate low-order terms
3. Ignore constants and coefficients

Examples:

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

Examples with Big-O Asymptotic Notation

True or False?

1. $3n+10 \in O(n)$
2. $4+2n \in O(1)$
3. $20-3n \in O(n^2)$
4. $n+2\log n \in O(\log n)$
5. $\log n \in O(n+2\log n)$

Examples with Big-O Asymptotic Notation Solutions

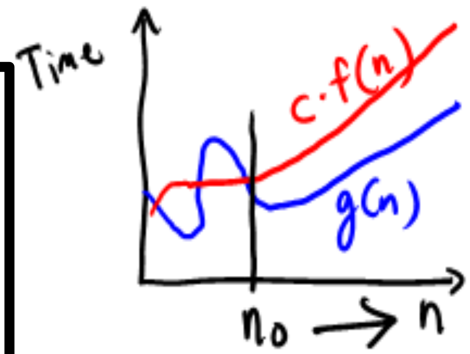
True or False?

1. `3n+10` $\in$ `O(n)` True (n = n)
2. `4+2n` $\in$ `O(1)` False: (n >> 1)
3. `20-3n` $\in$ `O(`n^2`)` True: ($n \le n^2$)
4. `n+2logn` $\in$ `O(logn)` False: (n >> logn)
5. `logn` $\in$ `O(n+2logn)` True: (logn ≤ n+2logn)

Formally Big-O

Definition:

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 $n \geq n_0$



- 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) = $3n^2+17$ and f(n) = n^2
    c=5 and n_0=10 is more than good enough

- This is "less than or equal to"
  - So $3n^2+17$ is also $O(n^5)$ and $O(2^n)$ etc.

Big-O

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

What it means:

- For your runtime, asymptotically, O(function) is the family of functions that defines the upper bound.
- There is a size of input (n_0) and a constant factor (c) you can use to make O(function) strictly larger than your runtime.

Examples using formal definition

A valid proof is to find valid *c* and n_0 :

- Let g(n) = 1000n and f(n) = n^2.
  - The "cross-over point" is n=1000
  - So we can choose n_0=1000 and c=1
    - Many other possible choices, e.g., larger n_0 and/or c
- Let g(n) = n^4 and f(n) = 2^n.
  - We can choose n_0=20 and c=1

> Definition:
>
> 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 $n \geq n_0$

What's with the c

- The constant multiplier c is what allows functions that differ only in their largest coefficient to have the same asymptotic complexity
- Example: $g(n) = 7n+5$ and $f(n) = n$
  - For any choice of n_0, need a $c > 7$ (or more) to show $g(n)$ is in $O(f(n))$

Definition:

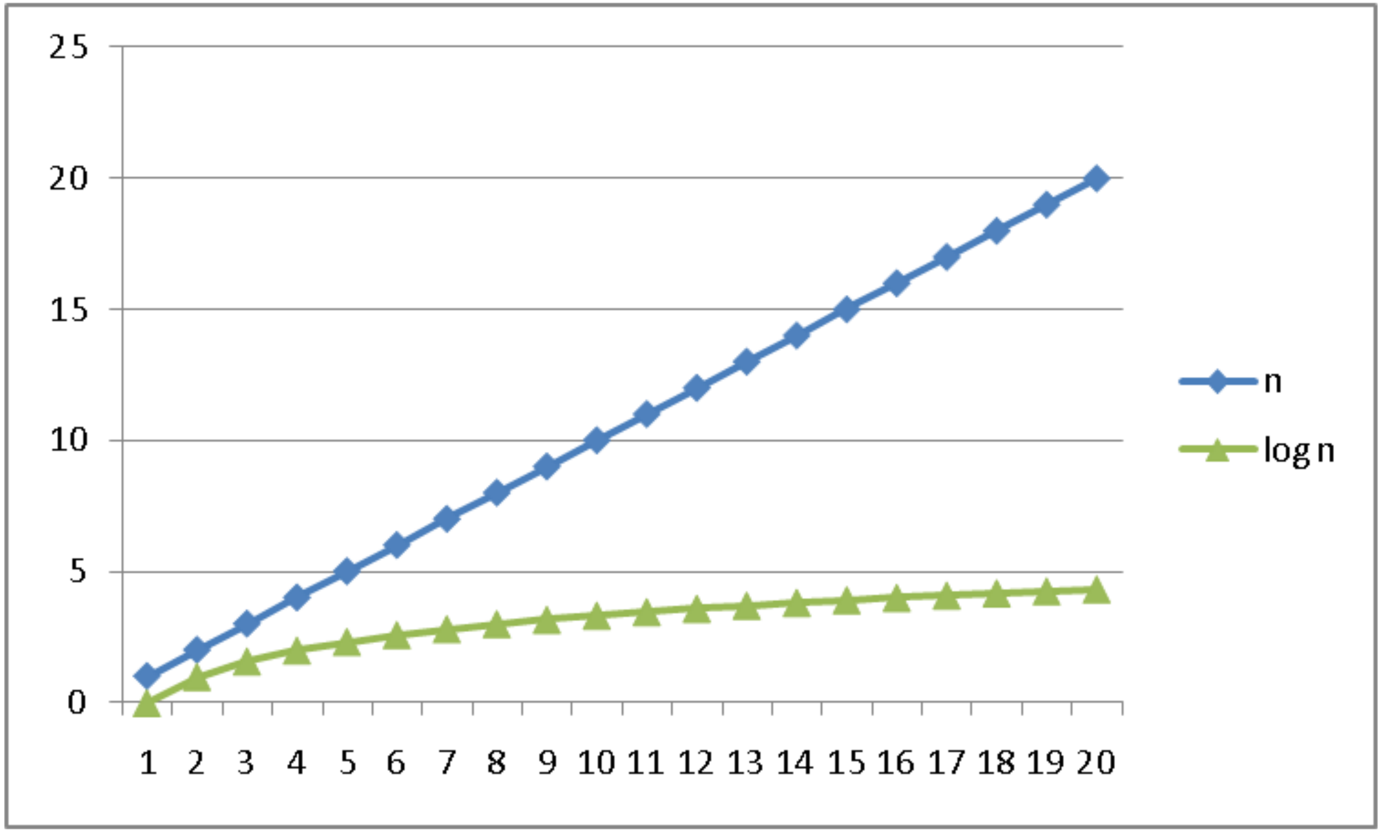
$g(n)$ is in $O(f(n))$ if there exist constants c and n_0 such that $g(n) \leq c\, f(n)$ for all $n \geq n_0$

Big-O: Common Names

$O(1)$	constant (same as $O(k)$ for constant k)
$O(\mathbf{log}\ n)$	logarithmic
$O(n)$	linear
O(n **log** n)	"n **log** n"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is any constant: linear, quadratic and cubic all fit here too.)
$O(k^n)$	exponential (where k is any constant > 1)

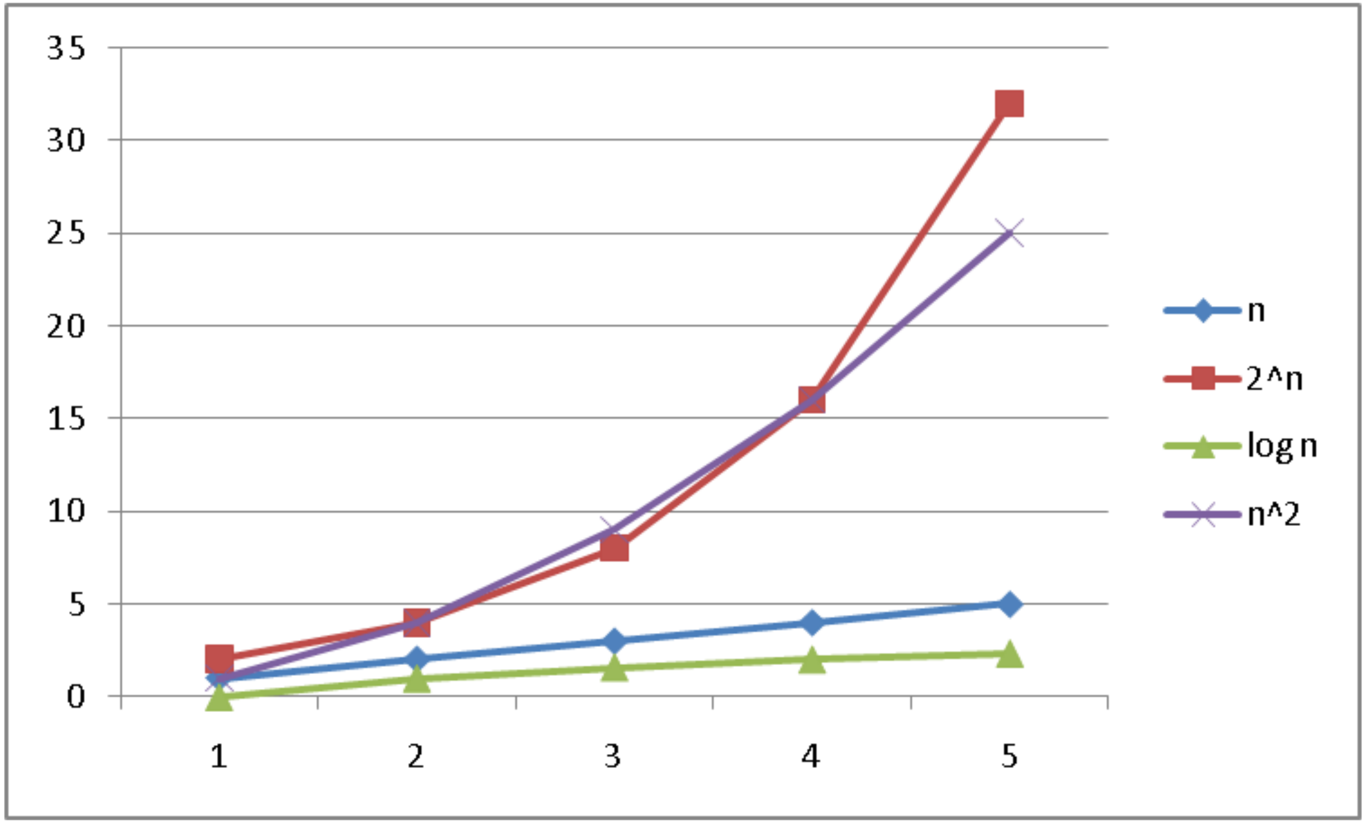
Note: "exponential" does not mean "grows really fast", it means "grows at rate proportional to k^n for some k>1". Example: a savings account accrues interest exponentially (k=1.01?).

Intuition of Common Runtimes



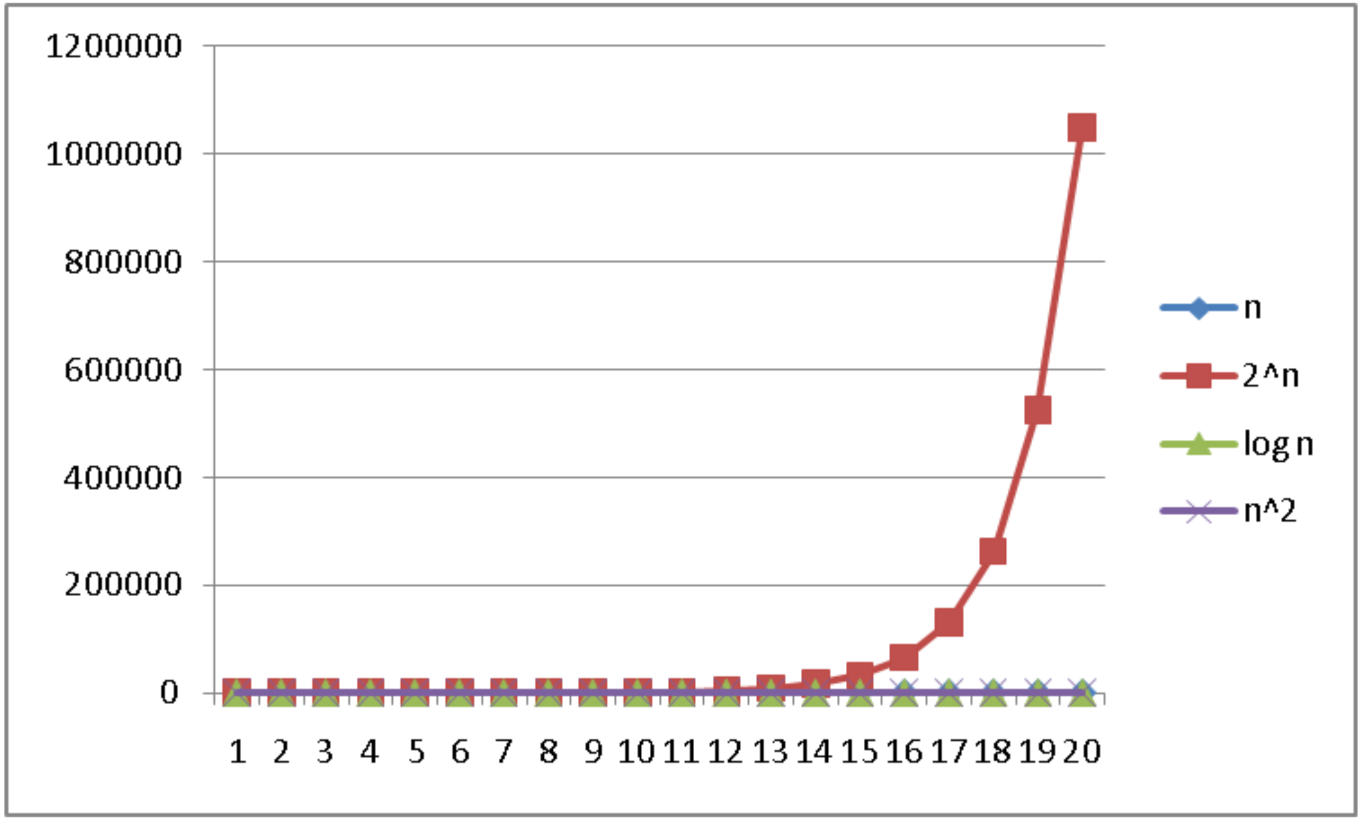
Even for small N, these look pretty different very quickly.

Intuition of Common Runtimes



Now y=N and y=logN look a lot more similar in comparison to other runtimes.

Intuition of Common Runtimes



Asymptotically, $y=2^N$ looks way different than the rest and the rest all look roughly the same.

More Asymptotic Notation

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

- Big-Omega 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
    $g(n) \geq c\ f(n)$ for all $n \geq n_0$

- Big-Theta Tight bound: $\theta(\ f(n)\ )$ is the set of all functions asymptotically equal to $f(n)$
  - Intersection of $O(\ f(n)\ )$ and $\Omega(\ f(n)\ )$ (use *different* c values)

A Note on Big-O Terms

- A common error is to say O(function) when you mean θ(function):
  - People often say Big-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 θ(n)
  - That means that it is not, for example O(log n)

What We're 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 (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)

Today's Takeaways – Algorithm Analysis

- Lots of ways to compare algorithms, today we analyzed runtime and asymptotic behavior

- Intuition of how the different types of runtimes compare asymptotically

- Big-O, Big-Theta, and Big-Omega definitions. Being able to prove them for a given runtime.