Algorithmic Complexity

CSE 120 Winter 2018

Instructor:

Justin Hsia

Teaching Assistants:

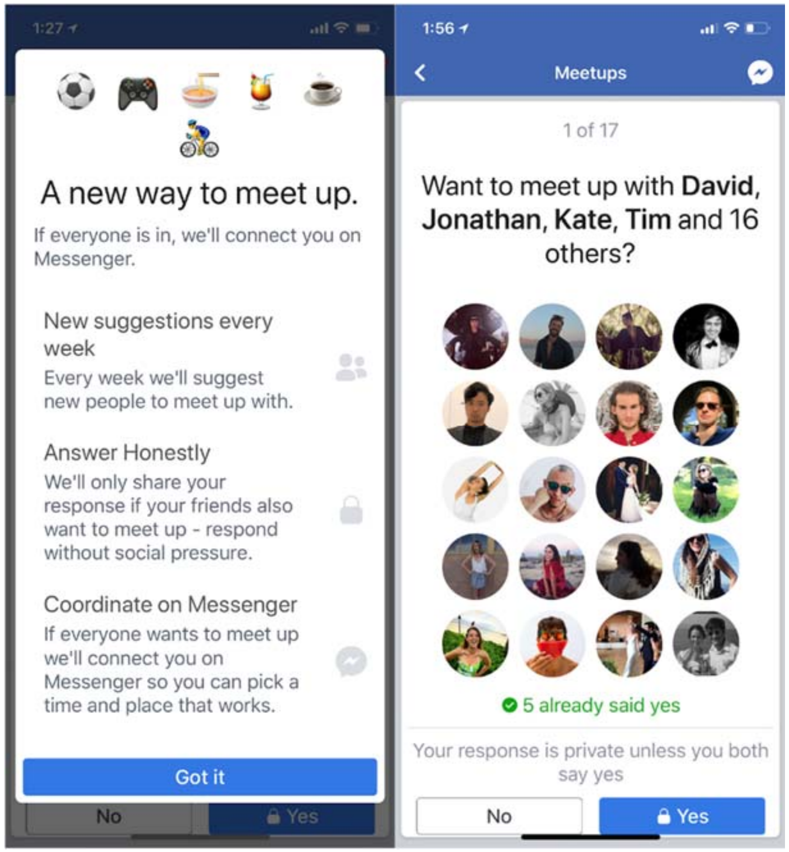
Anupam Gupta, Cheng Ni, Eugene Oh, Sam Wolfson, Sophie Tian, Teagan Horkan

Facebook should actually be Tinder too

"The idea behind Meetups is smart, but the execution is a mess. Because Meetups ambiguously shows multiple people at once, sends aggressive notifications to participate and encompasses all kinds of relationships, the results are meaningless.

"Facebook's made it easier than ever to "feel connected," endlessly scrolling through friends' photos, while actually allowing us to isolate ourselves."

- https://techcrunch.com/2018/02/14/facebook-matchmaker/



LCM Report Wrap-Up

living computers
museum + labs

Administrivia

- Assignments:
  - Innovation Exploration post (2/27)
  - Project Update in lab on Thursday (3/1)
  - Innovation Exploration comments (3/2)

- "Big Ideas" lecture: Proofs and Computation

Outline

- **Algorithm Analysis: The Basics**
- Comparing Algorithms
- Orders of Growth

Algorithm Correctness

- An algorithm is considered correct if for every input, it reports the correct output and doesn't run forever or cause an error
- Incorrect algorithms may run forever, crash, or not return the correct answer
  - But they could still be useful!
  - *e.g.* an approximation algorithm e.g. approximate value of π
    approximate shortest path
- Showing correctness
  - Mathematical proofs for algorithms
  - Empirical verification of implementations ← today's lecture

Algorithm Analysis

- One commonly used criterion for analyzing algorithms is computation time
  - How long does the algorithm take to run and finish its task?
  - Can be used to compare different algorithms for the same computational problem

- How to measure this time?
  - Counting in my head ← inconsistent, inaccurate, not precise
  - Stopwatch ← inconsistent, still inaccurate
  - Within your program ← much more accurate
    inconsistent?

Aside: Computation Time

- Computers take time to complete their tasks
  - Under the hood, it's sort of like a bunch of buckets of water filling up – you have to wait for water to reach the top of a bucket for a single computation to complete transistors
  - Buckets take about a billionth of a second to fill (~ 1 nanosecond)
    - There are billions of them on a single chip!

- A CPU can generally only execute one instruction at a time and tasks take many instructions

Timing in Processing

- The function **`millis()`** returns the number of milliseconds since starting your program (as an `int`)
  - To start timing, call and store the value in a variable
  - Call again after your computation and subtract the values

```
void draw() {
  int time = millis();   // stores start time in variable
  someComputation();
  println("Took " + (millis()-time) + " milliseconds to compute.");
  noLoop();
}
```

this expression calculates time taken by someComputation() and is evaluated before the string is printed

Outline

- Algorithm Analysis: The Basics
- **Comparing Algorithms**
- Orders of Growth

Algorithm: Searching A Sorted Array

- Input: Numbers in a sorted array, desired number
- Output: If desired number is in the array (**true**/**false**)

computational problem

- Algorithm 1:
  - Check each index starting from 0 for desired number
    - If equal, then report **true**
    - If not equal, then move to next index
    - If at end of array, then report **false**
  - Called Linear Search (also works for unsorted array)

algorithm

```
boolean linearSearch(int num) {
  for(int i = 0; i < intArr.length; i = i + 1) {
    if(intArr[i] == num) {
      return true;
    }
  }
  return false;
}
```

implementation

Algorithm: Searching A Sorted Array

- Input: Numbers in a sorted array, desired number
- Output: If desired number is in the array (`true`/`false`)

- Algorithm 2:
  - Check "middle" index for desired number
    - If equal, then report `true`
    - If less than desired number, check *halfway* forwards next
    - If greater than desired number, check halfway backwards next
  - If no halfway point left, then report `false`
  - Called Binary Search
    - http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html

Question

- On average, which algorithm should take less time to complete a search?
  - Vote at http://PollEv.com/justinh

A. Algorithm 1 (Linear Search)

B. Algorithm 2 (Binary Search)

but how do you "prove" this?

C. They'd take about the same amount of time

Measuring Linear Search

- Let's time Linear Search:

```
void draw() {
  int n = 3;
  println("Is " + n + " in intArr?");
  int time = millis();
  println(linearSearch(n));
  println("Took " + (millis()-time) + " milliseconds to compute.");
  noLoop();
}
```

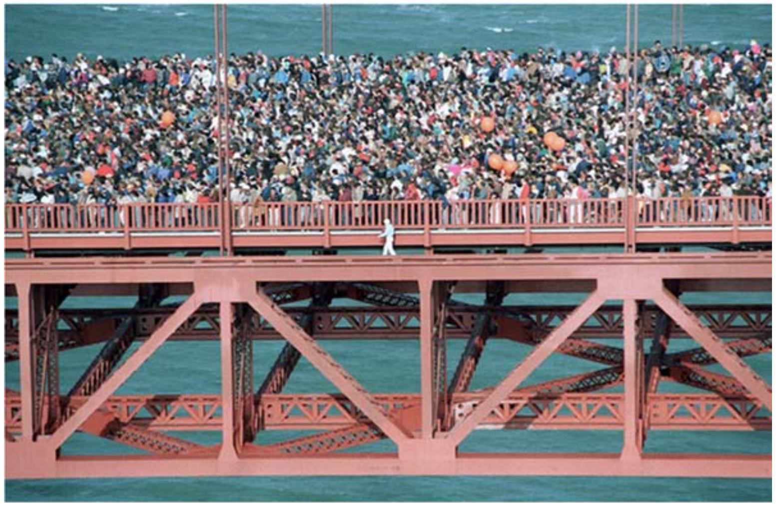
- One issue: our algorithm seems to be too fast to measure! (keeps showing 0 milliseconds)
  - How can we fix this? → try longer arrays
    → try different n

Which Case Do We Care About?

- We were measuring close to the best case!
  - Didn't matter how long our array was

3 is always at the front of the array

- Could measure average case instead
  - Run many times on random numbers and average results
- Instead, we'll do worst case analysis. Why?
  - Nice to know the most time we'd ever spend
  - Worst case may happen often
  - An algorithm is often judged by its worst case behavior



What is the Worst Case?

- Discuss with your neighbor (no voting):
  - Assume `intArr.length` is `1000000` and `intArr[i] = i;`
  - What is a worst case argument for `num` for Linear Search? C/D
    best case: A
  - What is a worst case argument for `num` for Binary Search? A/C/D
    best case: B

A. 1 (beginning)

B. 500000 (middle)

C. 1000000 (end)

D. 1000001 (not in array)

E. Something else (else)

```java
boolean linearSearch(int num) {
  for(int i = 0; i < intArr.length; i = i + 1) {
    if(intArr[i] == num) {
      return true;
    }
  }
  return false;
}
```

Timing Experiments

- For array length 100,000,000 and `intArr[i] = i;` searching for 100,000,001

- Let's try running Linear Search on a worst case argument value
  - Results: 47, 47, 47, 31, 47 ms

- Now let's run Binary Search on a worst case argument value
  - Results: 0, 0, 0, 0 ms

Runtime Intuition

- Does it seem reasonable that the runtimes were inconsistent?

- Some reasons:
  - Your computer isn't just running Processing – there's a lot of other stuff running (*e.g.* operating system, web browser)
  - The computer hardware does lots of fancy stuff to avoid slowdown due to physical limitations
    - These may not work as well each execution based on other stuff going on in your computer at the time

Empirical Analysis Conclusion

- We've shown that Binary Search is seemingly much faster than Linear Search
  - Similar to having two sprinters race each other

Usain Bolt
vs.
Justin Gatlin

- Limitations:
  - Different computers may have different runtimes
  - Same computer may have different runtime on same input
  - Need to implement the algorithm in order to run it

- Goal: come up with a "universal algorithmic classifier"
  - Analogous to coming up with a metric to compare all athletes

better "athlete": speed skater or bobsledder?

Outline

- Algorithm Analysis: The Basics
- Comparing Algorithms
- **Orders of Growth**

Characterizing Algorithms

- The method computer scientists use is roughly:

1) Measure the algorithm's runtime on many different input sizes N (*e.g.* arrays of length 100, 200, 400, 800, ...)
   - To avoid runtime issues, can also count number of "steps" involved

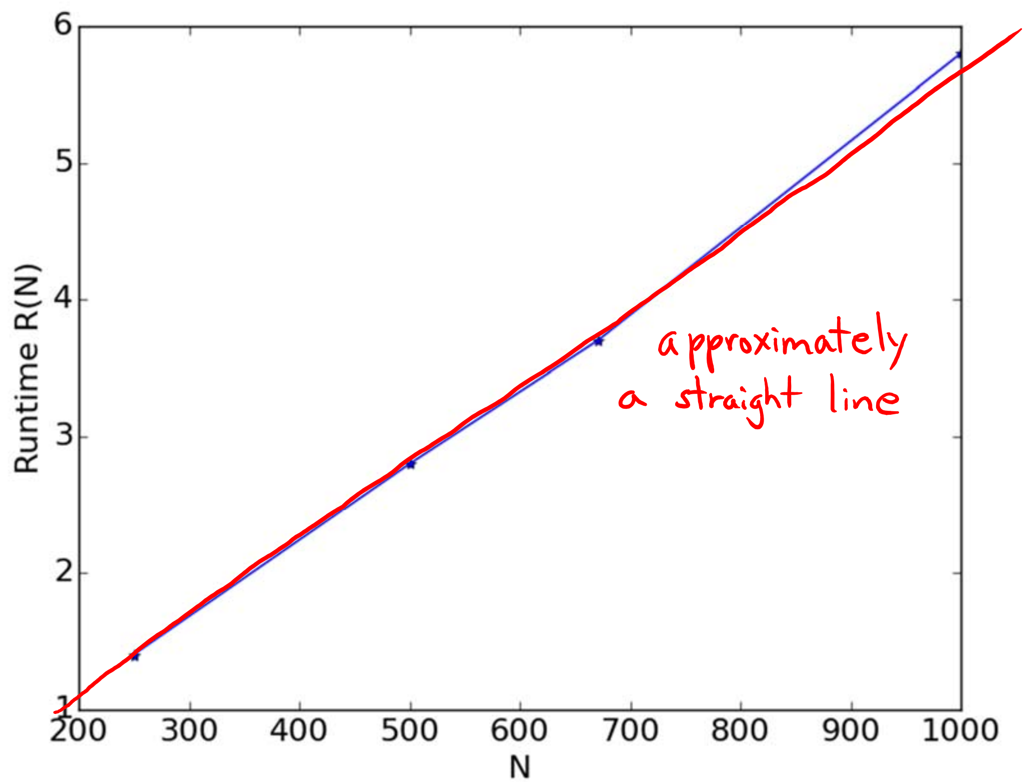
2) Make a plot of the runtime as a function of N, which we'll call $\mathrm{R(N)}$

3) Determine the general *shape* of $\mathrm{R(N)}$
   - Does $\mathrm{R(N)}$ look like N (linear), N^2 (quadratic), N^3 (cubic), $\log \mathrm{N}$ (logarithmic), etc.

Linear Search

- As the name implies, Linear Search is linear
  - If you double N, then R(N) should roughly double

N (input size)	R(N) (time)
x 250 items	1.4 sec y
2x 500 items	2.8 sec 2y
671 items	3.8 sec
4x 1000 items	5.7 sec ≈4y

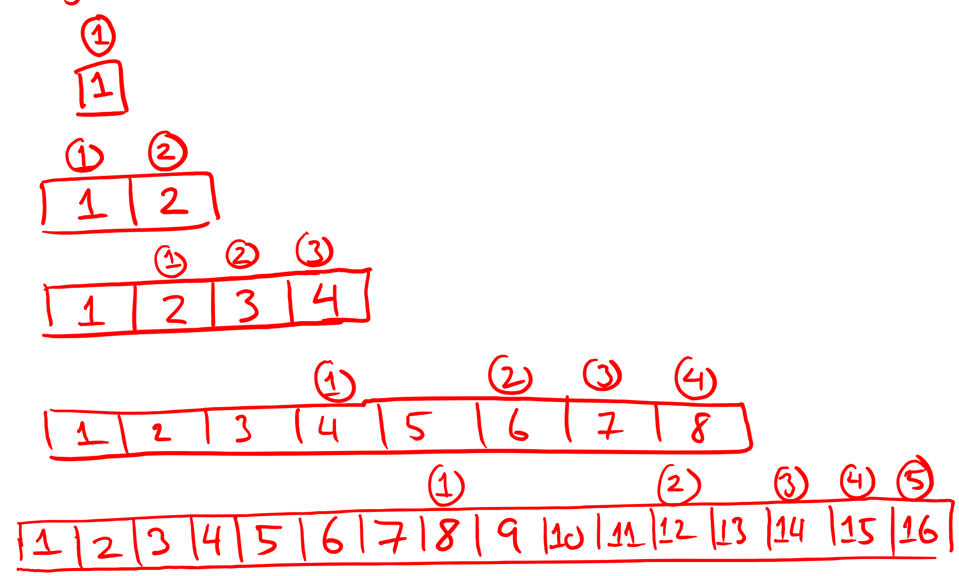


Binary Search

- What order of growth is Binary Search?
  - Analyze using number of "steps" in worst case

	N (input size)	Indices to Check	worst case n
x	1 items	1	2
2x	2 items	2	3
4x	4 items	3	5
8x	8 items	4	9
16x	16 items	5	17
	32 items	6	
	64 items	7	
	⋮	⋮	

Algorithm Illustration

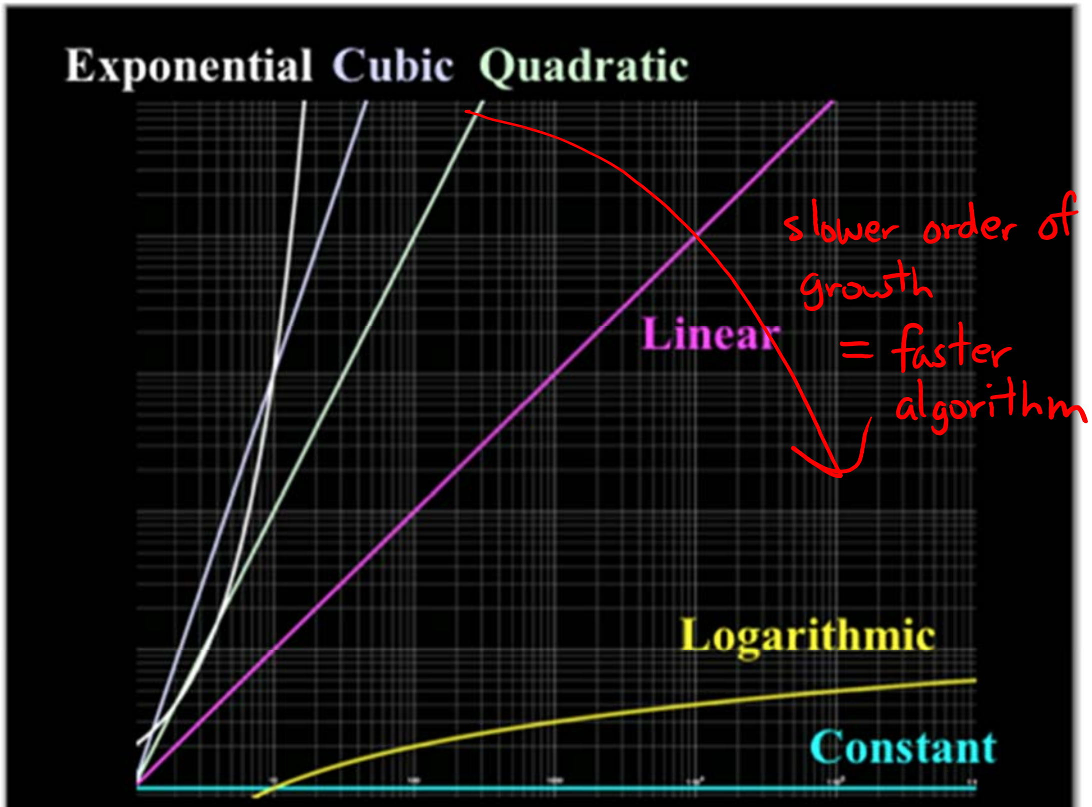


$$\boxed{R(N) = \log_2(N)}$$

logarithmic!

Orders of Growth

- The order of growth of $R(N)$ is its general shape:
  - Constant 1
  - Logarithmic $\log N$
  - Linear N
  - Quadratic N^2
  - Cubic N^3
  - Exponential 2^N
  - Factorial $N!$

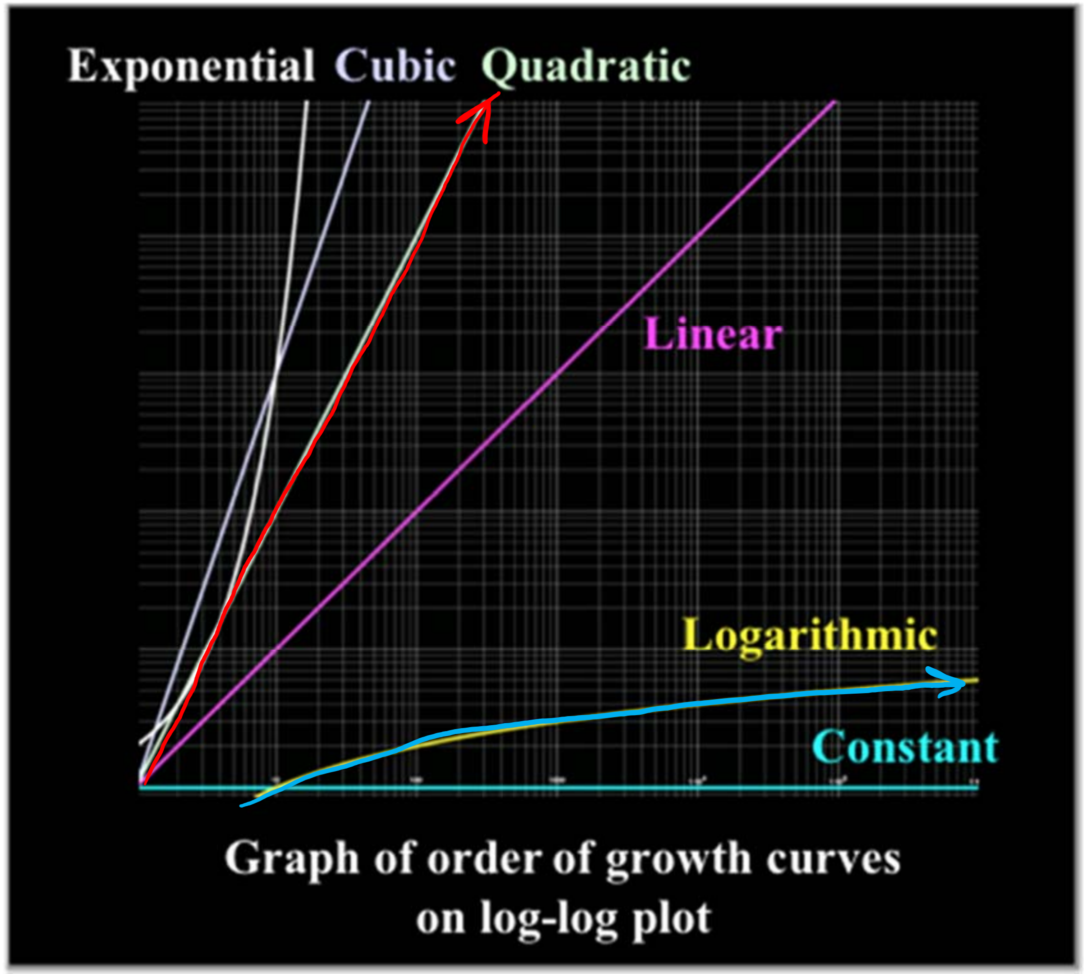


Graph of order of growth curves on log-log plot

Orders of Growth

- The order of growth of $R(N)$ is its general shape:
  - Use *dominant* term
  - *e.g.* $10N^2 + 4\log N$ is quadratic

grows much faster than $4 \log N$ as $N \to \infty$



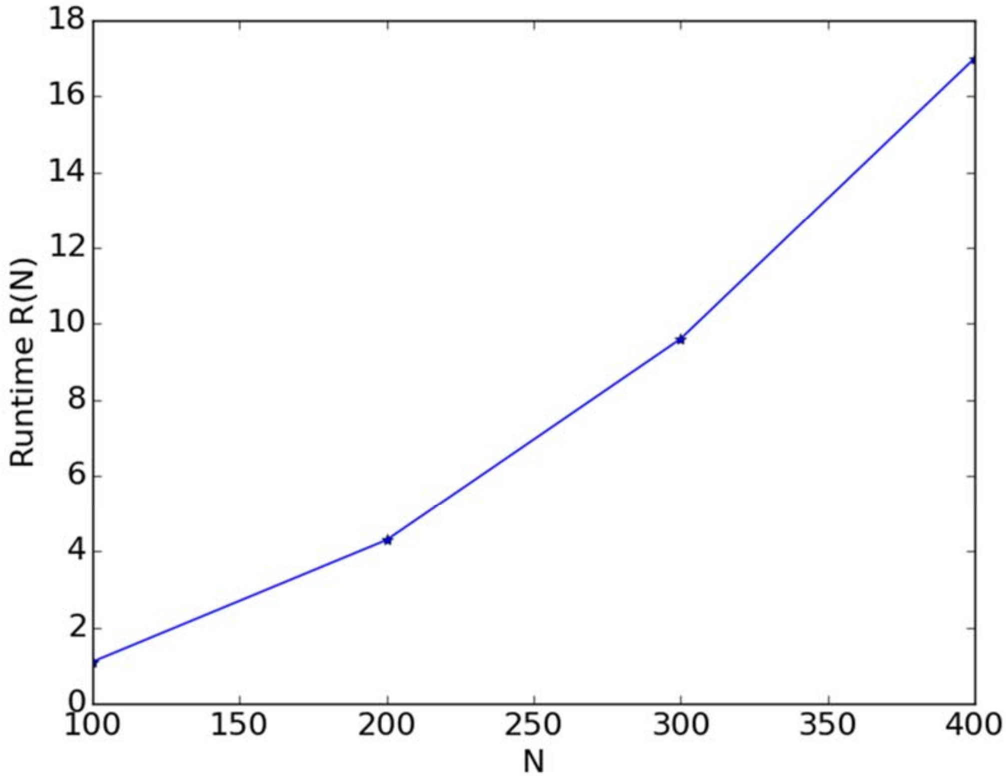
Graph of order of growth curves on log-log plot

Peer Instruction Question

- Algorithm for: do any pairs in array sum to zero?
- Which function does R(N) look like?
  - Vote at http://PollEv.com/justinh

A. sqrt(N)
B. log(N)
C. N
D. N^2
E. 2^N

N (input size)	R(N) (time)
100 items	1.1 seconds
200 items	4.3 seconds
300 items	9.6 seconds
400 items	17.0 seconds



The Reason Order of Growth Matters

- Roughly speaking, we care about really big N in real world applications
  - *e.g.* For Facebook, N (users) is ~ 1 billion
    - Want to generate list of suggested friends? Better be a fast algorithm as a function of N
- Order of growth is just a rough rule of thumb
  - There are limited cases where an algorithm with a worse order of growth can actually be faster
  - In almost all cases, order of growth works very well as a representation of an algorithm's speed

Orders of Growth Comparison

- The numbers below are rough estimates for a "typical" algorithm on a "typical" computer – provides a qualitative difference between the orders of growth

	Linear	Linearithmic	Quadratic	Cubic	Exponential	Exponential	Factorial
	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.