CSE100: Design and Analysis of Algorithms
Lecture 02 – Introduction (cont)

Multiplication

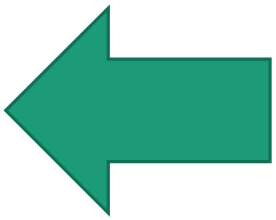
The big questions

- Who are we?
  - Professor, TA's, students?
- Why are we here?
  - Why learn about algorithms?
- What is going on?
  - What is this course about?
  - Logistics?
- Can we multiply integers?
  - And can we do it quickly?

Course goals

- Think analytically about algorithms
- Flesh out an "algorithmic toolkit"
- Learn to communicate clearly about algorithms

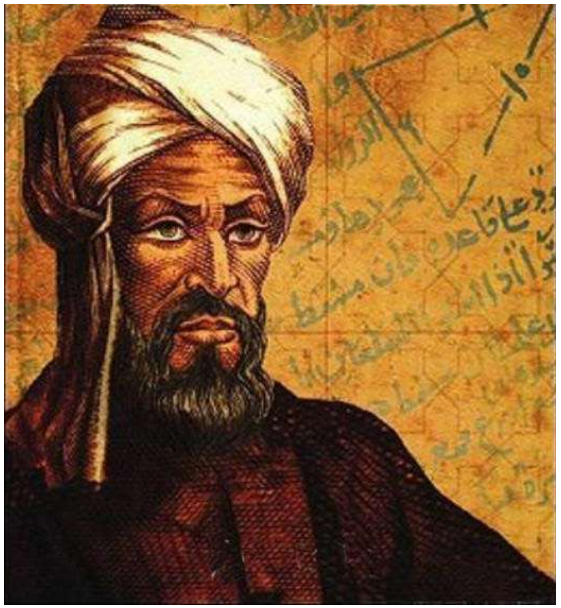
Today's goals

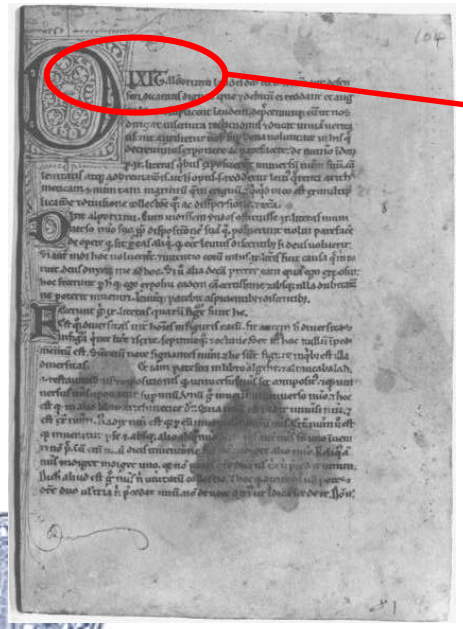
- Integer Multiplication 
- Algorithmic Technique:
  - Divide and conquer
- Algorithmic Analysis tool:
  - Intro to asymptotic analysis

Let’s start at the beginning

Etymology of "Algorithm"

- Al-Khwarizmi was a 9th-century scholar, born in present-day Uzbekistan, who studied and worked in Baghdad during the Abbassid Caliphate.
- Among many other contributions in mathematics, astronomy, and geography, he wrote a book about how to multiply with Arabic numerals.
- His ideas came to Europe in the 12th century.





Dixit algorizmi
(so says Al-Khwarizmi)

- Originally, "Algorisme" [old French] referred to just the Arabic number system, but eventually it came to mean "Algorithm" as we know today.

This was kind of a big deal

XLIV × XCVII = ?

$$\begin{array}{r} 44 \\ \times\ 97 \\ \hline \end{array}$$

A problem you all know how to solve:

Integer Multiplication

$$\begin{array}{r} 12 \\ \times\ 34 \\ \hline \end{array}$$

A problem you all know how to solve:

Integer Multiplication

$$
\begin{array}{rr}
 & 12345678959314 13 \\
\times & 4563823520395533 \\
\hline
\end{array}
$$

A problem you all know how to solve:

Integer Multiplication

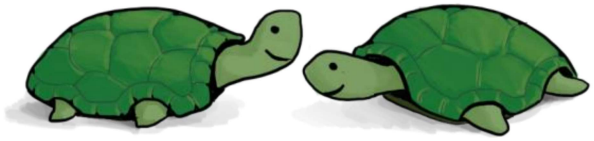
n

```
  12339257207527523846237642835683649183745238562 98
x 45623235823423952856234672350191307501353500137 53
```

???

How fast is the grade-school multiplication algorithm?

(How many one-digit operations?)



Think-pair-share Terrapins

About n^2 one-digit operations

Plucky the Pedantic Penguin

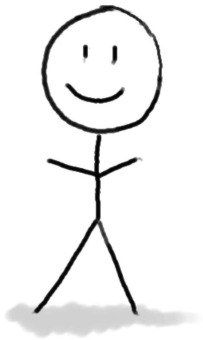
At most n^2 multiplications,
and then at most n^2 additions (for carries)
and then I have to add n different 2n-digit numbers…

Does that answer the question?

- What does it mean for an algorithm to be “fast”?

All running the same algorithm…

python™



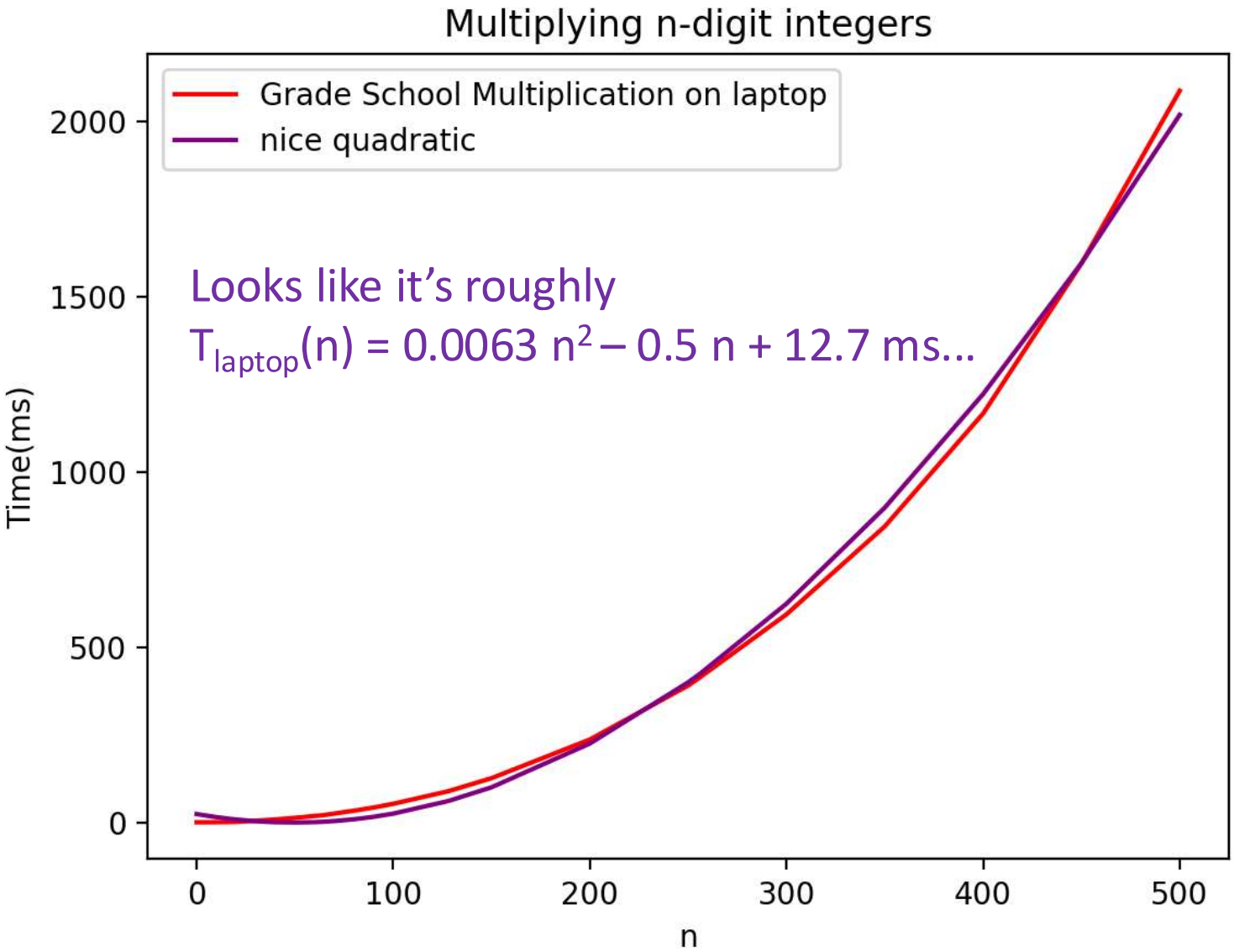
C++

- When we say “About n^2 one-digit operations”, we are measuring how the runtime scales with the size of the input.

highly non-optimized

Implemented in Python, on my laptop

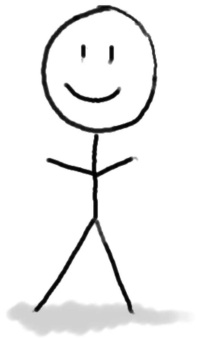
The runtime "scales like" n^2

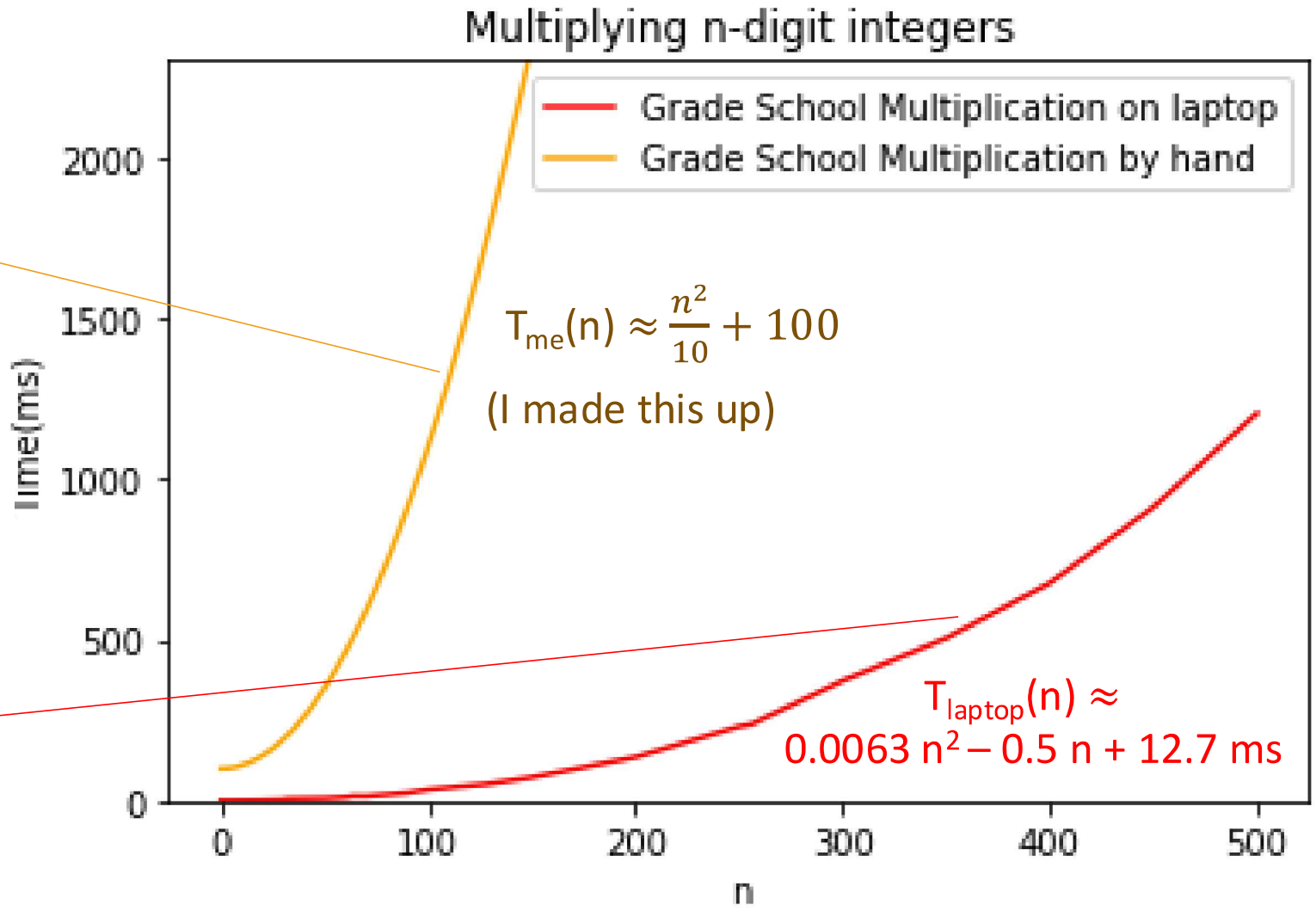


python™

Implemented by hand

The runtime still "scales like" n^2





python™

Super Informally…

Asymptotic Analysis

- We will say Grade School Multiplication "runs in time $O(n^2)$"
- Formalizes what it means to "scale like n^2"
- We will see a formal definition next lecture.
- Informally, definition-by-example:

Number of milliseconds on an input of size n	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\,n + 12.7$	$O(n^2)$
$100 \cdot n^2 - 10^{10000}\sqrt{n}$	$O(n^2)$
$\frac{1}{10} n^{1.6} + 100$	$O(n^{1.6})$

(Only pay attention to the largest power of n that appears.)

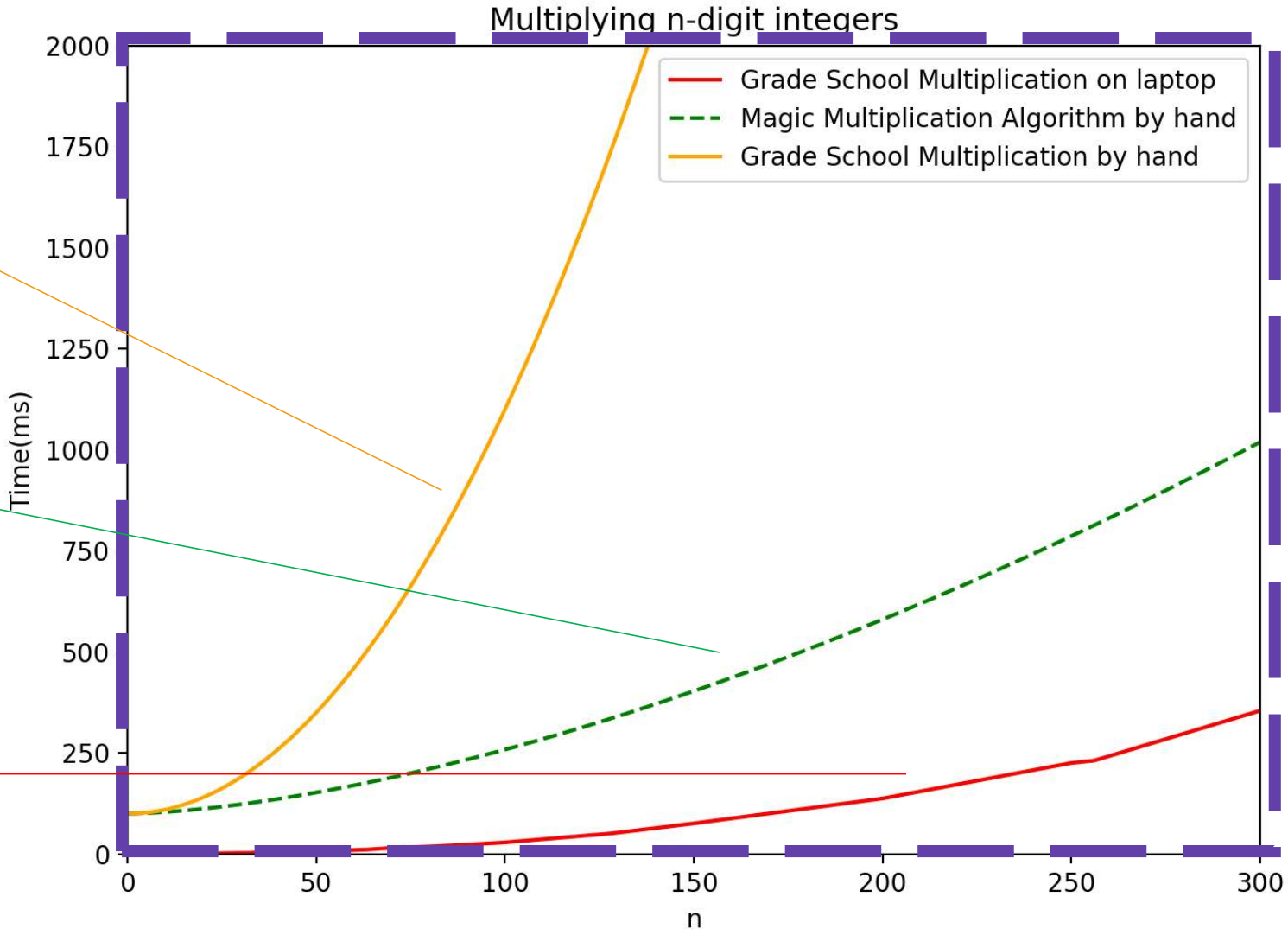
We say this algorithm is "asymptotically faster" than the others.

Why is asymptotic analysis meaningful?

$\approx \frac{n^2}{10} + 100$

$\approx \frac{n^{1.6}}{10} + 100$

$\approx .0063n^2$

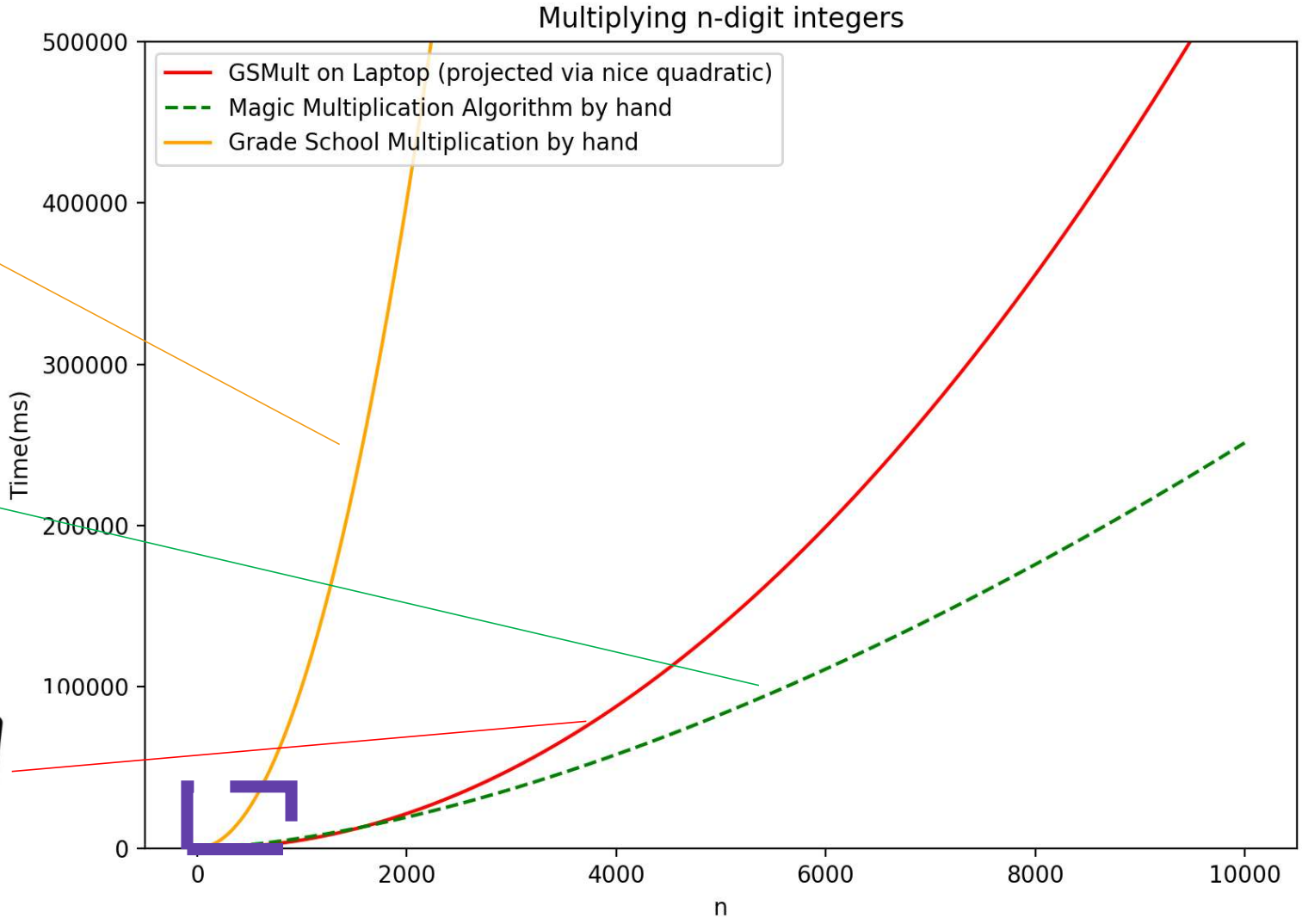


Let n get bigger…

$\approx \frac{n^2}{10} + 100$

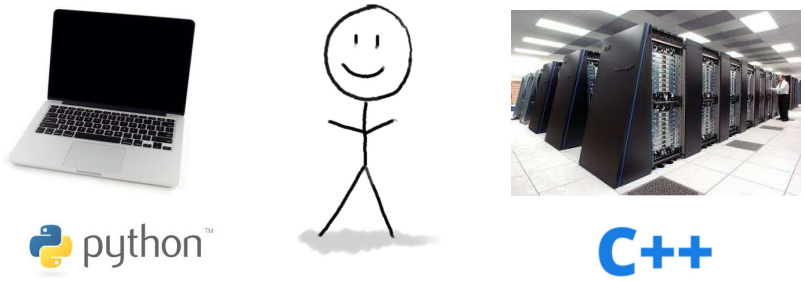
$\approx \frac{n^{1.6}}{10} + 100$

$\approx .0063n^2$



Asymptotic analysis is meaningful

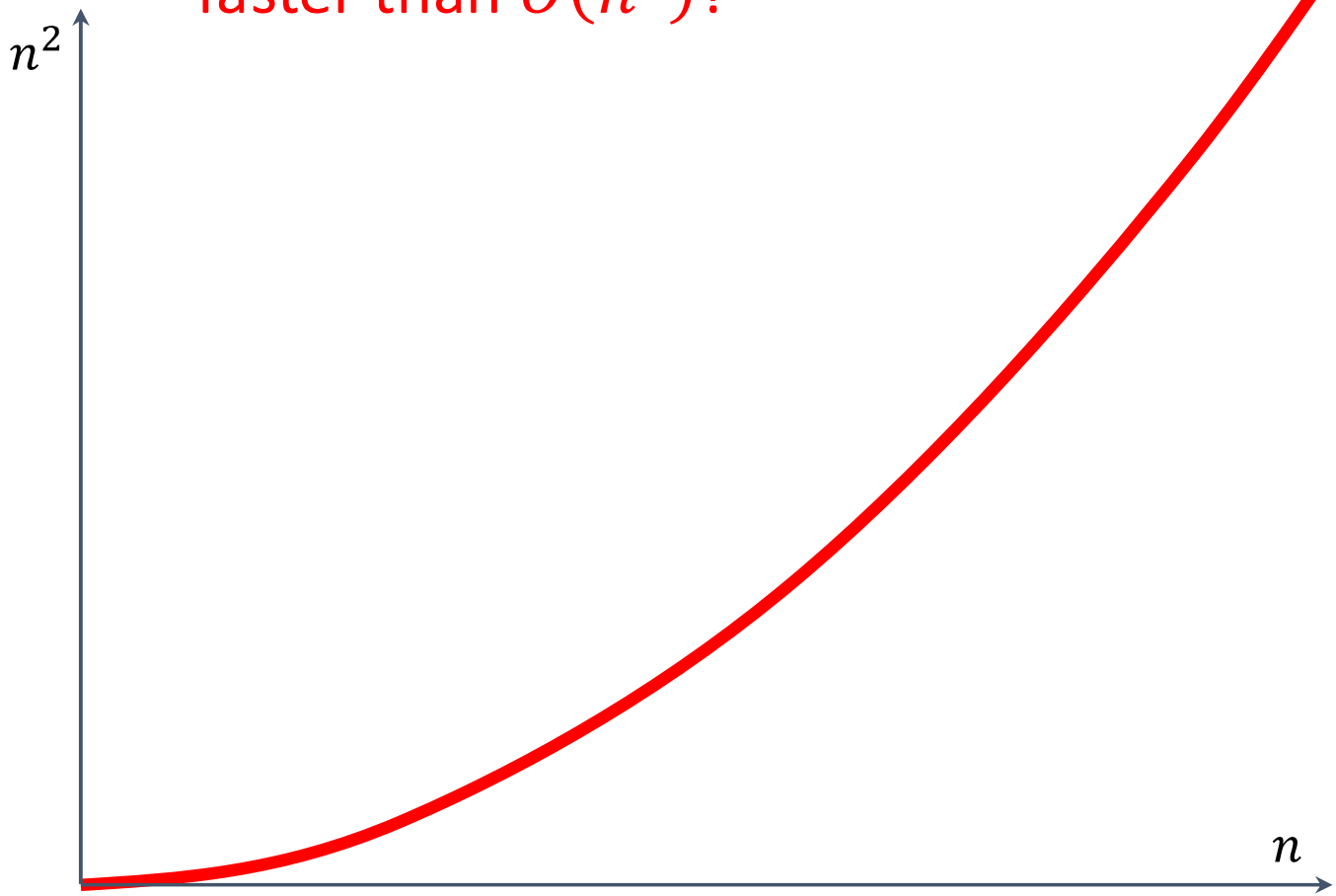
- For large enough input sizes, the “asymptotically faster” will be faster than the “asymptotically slower” one, no matter what your computational platform.



- So the question is…

Can we do better?

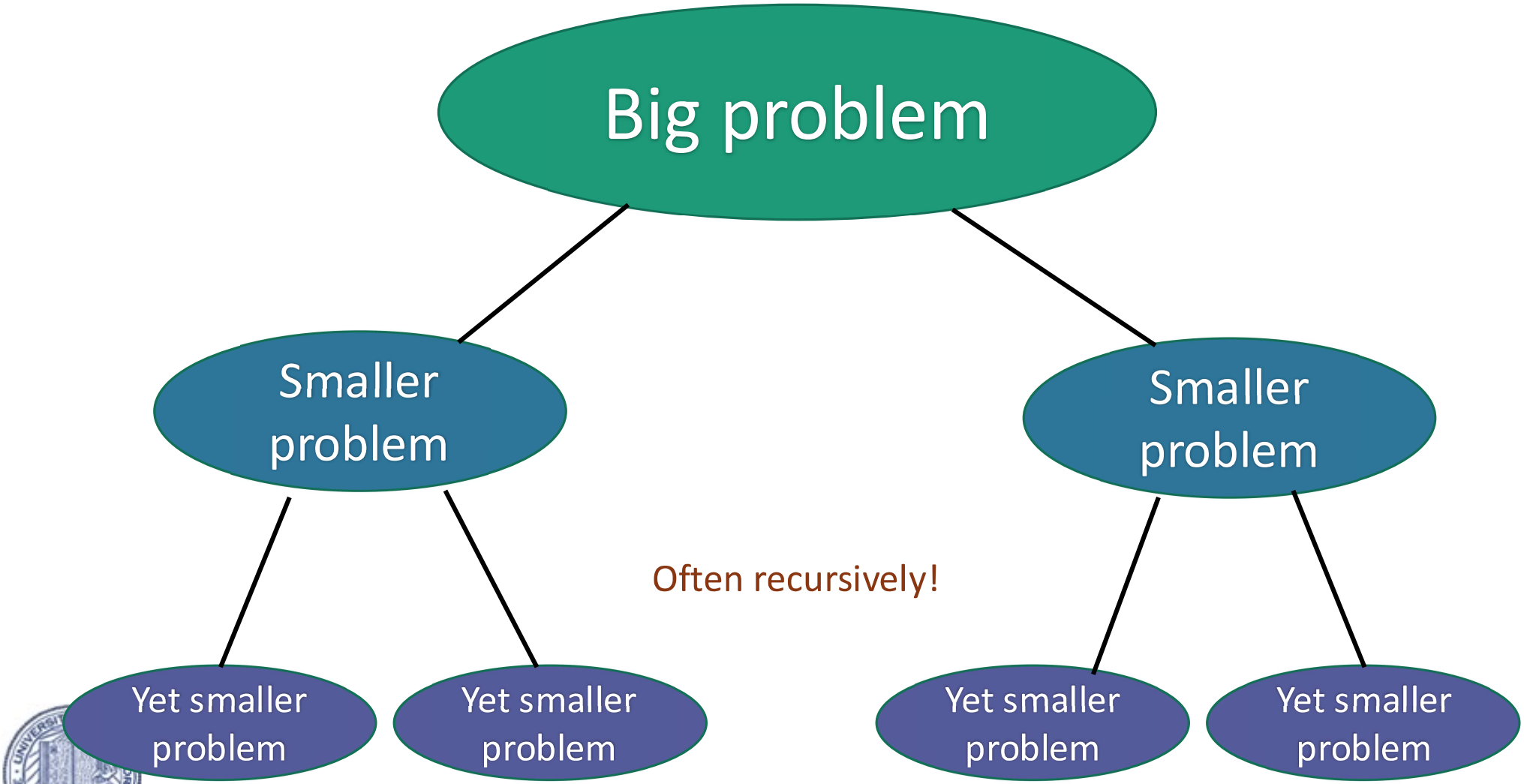
Can we multiply n-digit integers faster than $O(n^2)$?



Let’s dig into our algorithmic toolkit…

Divide and conquer

Break problem up into smaller (easier) sub-problems



Divide and conquer for multiplication

Break up an integer:

$$1234 = 12 \times 100 + 34$$

$$1234 \times 5678$$
$$= (12 \times 100 + 34) (56 \times 100 + 78)$$
$$= (12 \times 56)10000 + (34 \times 56 + 12 \times 78)100 + (34 \times 78)$$

1 2 3 4

One 4-digit multiply → Four 2-digit multiplies

More generally

Break up an n-digit integer:

$$[x_1 x_2 \cdots x_n] = [x_1 x_2 \cdots x_{n/2}] \times 10^{n/2} + [x_{n/2+1} x_{n/2+2} \cdots x_n]$$

$$x \times y = (a \times 10^{n/2} + b)(c \times 10^{n/2} + d)$$

$$= \underbrace{(a \times c)}_{1} 10^n + \underbrace{(a \times d}_{2} + \underbrace{c \times b)}_{3} 10^{n/2} + \underbrace{(b \times d)}_{4}$$

One n-digit multiply → Four (n/2)-digit multiplies

Divide and conquer algorithm

not very precisely…

(Assume n is a power of 2…)

x, y are n-digit numbers

Multiply(x, y):

- **If** $n = 1$:
  - **Return** xy

Base case: I've memorized my 1-digit multiplication tables…

- Write $x = a\ 10^{\frac{n}{2}} + b$
- Write $y = c\ 10^{\frac{n}{2}} + d$

a, b, c, d are $n/2$-digit numbers

- Recursively compute ac, ad, bc, bd:
  - ac = **Multiply**(a, c), etc…
- Add them up to get xy:
  - $xy = ac10^n + (ad + bc)10^{n/2} + bd$

Make this pseudocode more detailed! How should we handle odd n? How should we implement "multiplication by 10^n"?

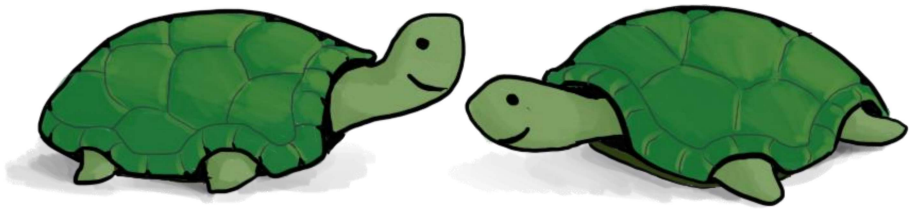
Siggi the Studious Stork

Think-Pair-Share

- We saw that this 4-digit multiplication problem broke up into four 2-digit multiplication problems

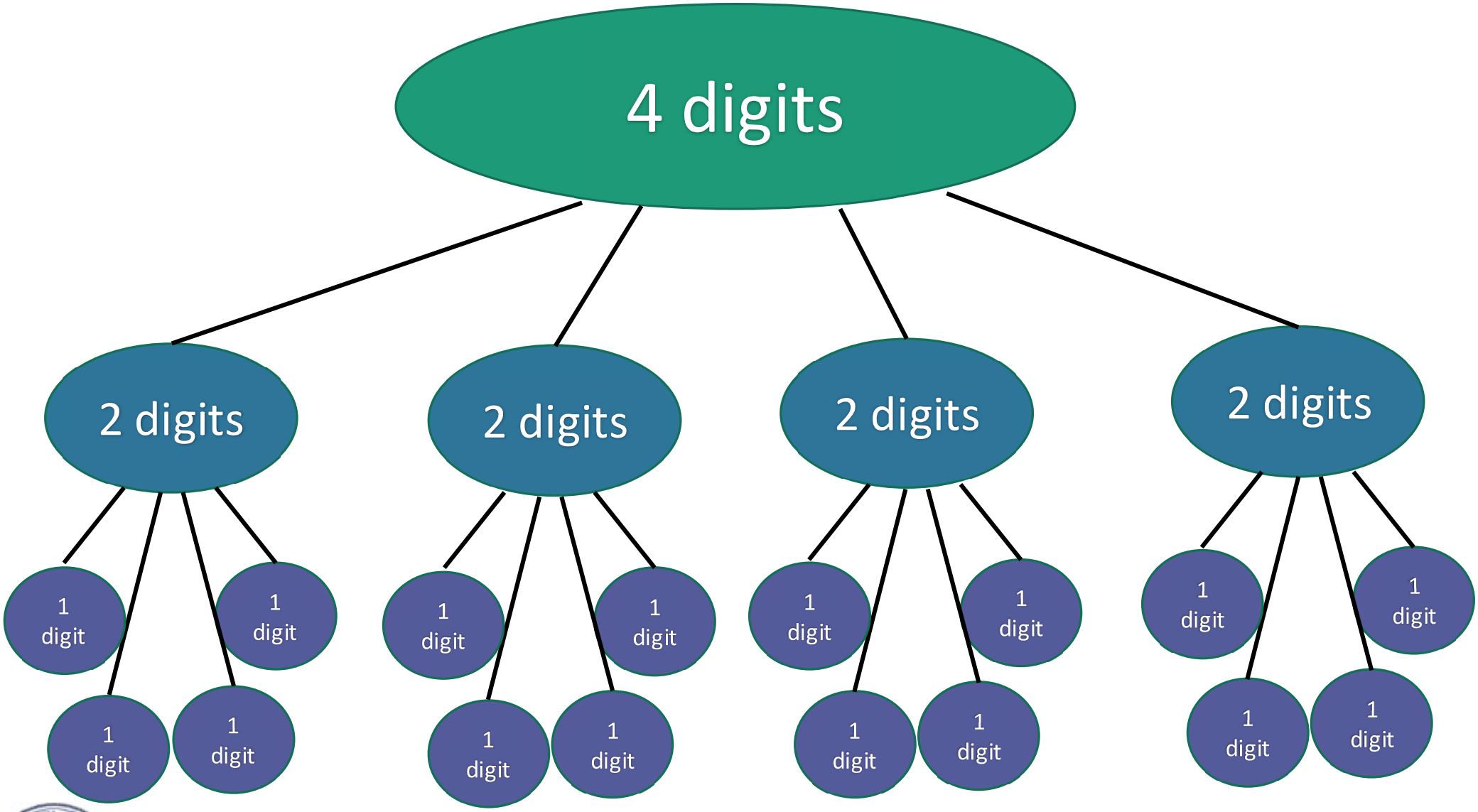
$$1234 \times 5678$$

- If you recurse on those 2-digit multiplication problems, how many 1-digit multiplications do you end up with in total?



Recursion Tree

16 one-digit multiplies!

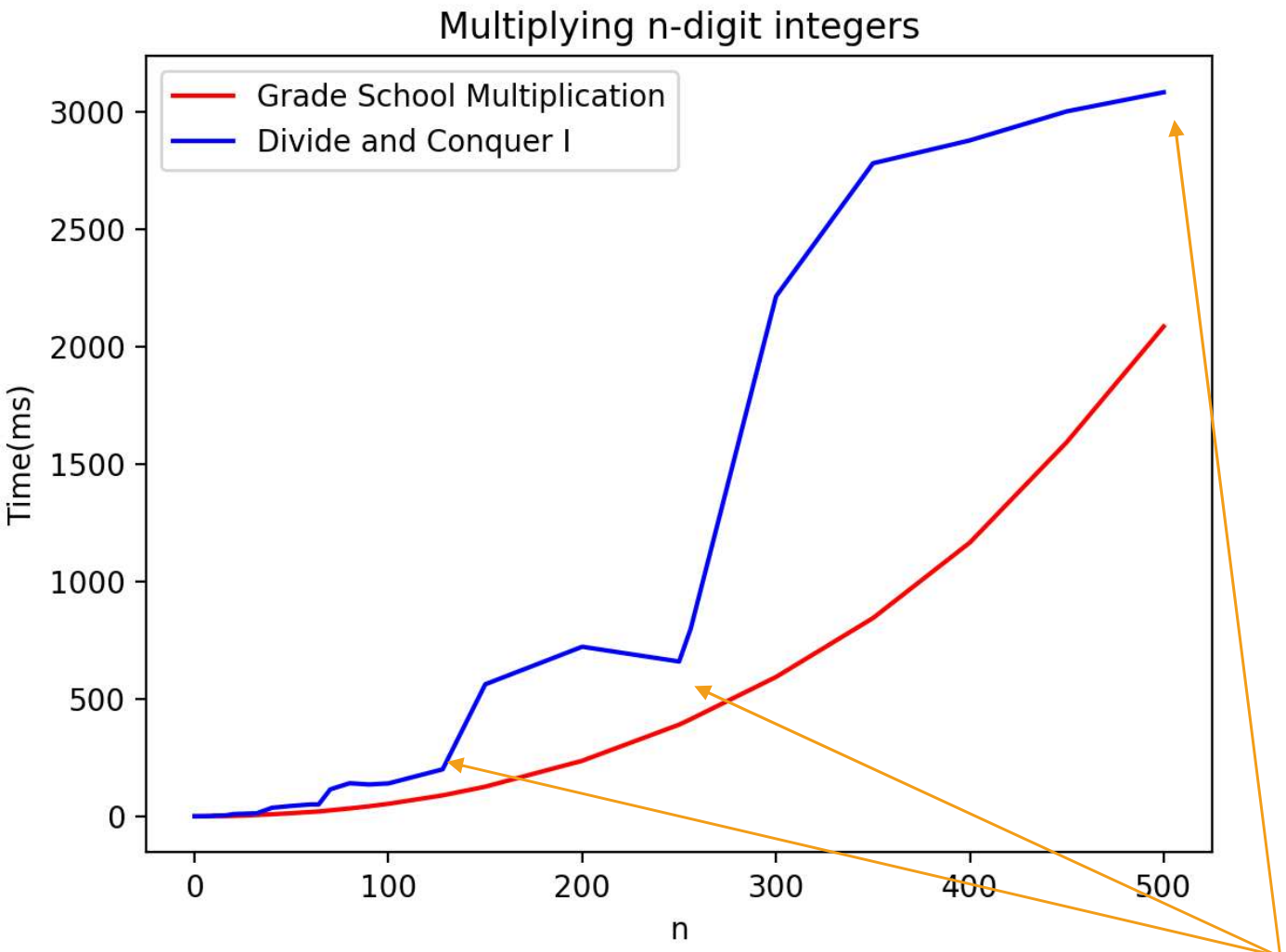


What is the running time?

- Better or worse than the grade school algorithm?

- How do we answer this question?
  1. Try it.
  2. Try to understand it analytically.

1. Try it.



Conjectures about running time?

Doesn't look too good but hard to tell…

Concerns with the conclusiveness of this approach?

Maybe one implementation is slicker than the other?

Maybe if we were to run it to n=10000, things would look different.

Something funny is happening at powers of 2…

2. Try to understand the running time analytically

- Proof by meta-reasoning:

It must be faster than the grade school algorithm, because we are learning it in an algorithms class.

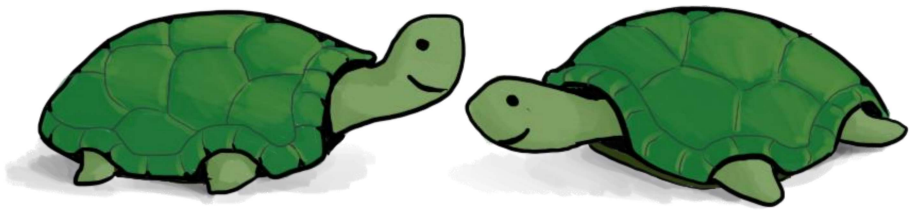
Not sound logic!

Plucky the Pedantic Penguin

2. Try to understand the running time analytically

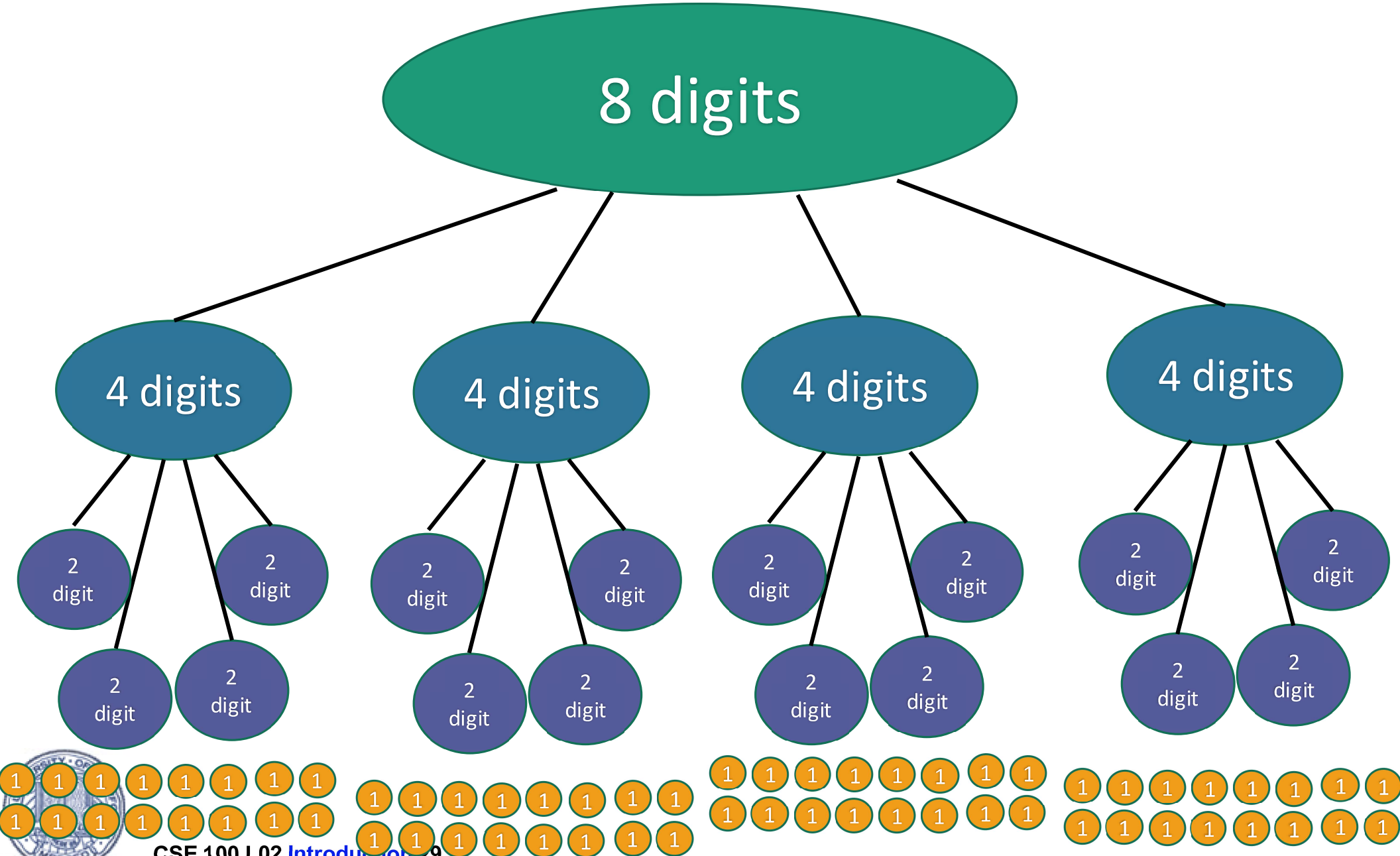
Think-Pair-Share:

- We saw that multiplying 4-digit numbers resulted in 16 one-digit multiplications.
- How about multiplying 8-digit numbers?
- What do you think about n-digit numbers?



Recursion Tree

64 one-digit multiplies!



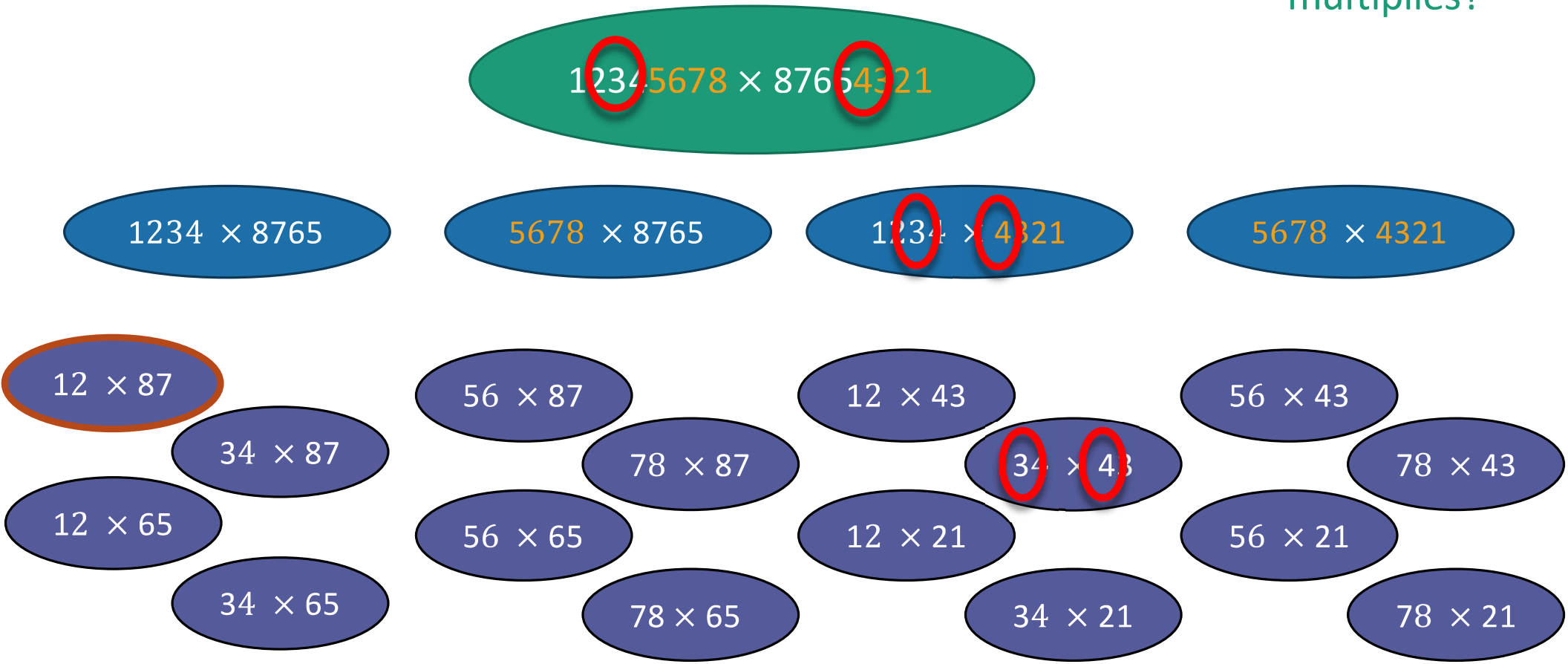
2. Try to understand the running time analytically

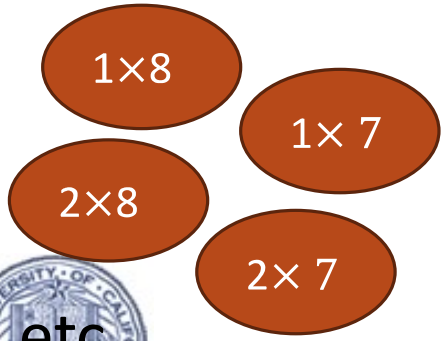
Claim:

The running time of this algorithm is AT LEAST n^2 operations.

Let's do an example

How many one-digit multiplies?





… 3× 4 …

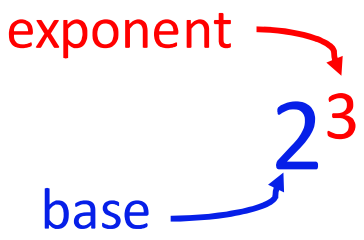
Claim: there are n^2 one-digit problems.

Every pair of digits still gets multiplied together separately.

So the running time is still at least n^2.

Review of exponents & logarithms (1)

What is an Exponent?



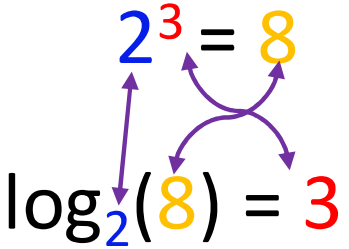
The exponent of a number says how many times to use the number in a multiplication

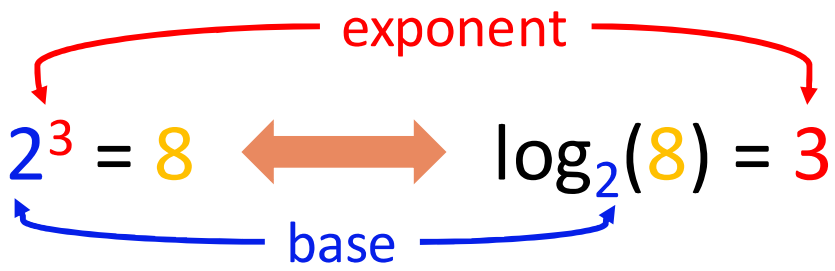
In this example: $2^3 = 2 \times 2 \times 2 = 8$

What is an Logarithm?

A Logarithm goes the other way. It asks the question "what exponent produced this?":

$2^? = 8$ and answers like this:

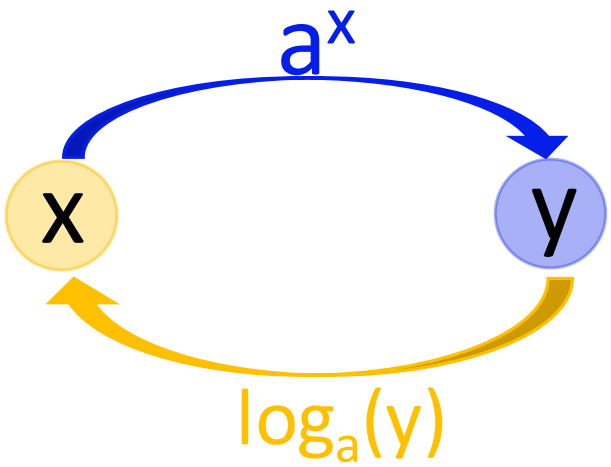




Review of exponents & logarithms (2)

Working Together

Exponents and Logarithms work well together because they "undo" each other (so long as the base "a" is the same):



The Logarithmic Function is "undone" by the Exponential Function.
(and vice versa)

Doing one, then the other, gets you back where you started:

- Doing $\mathbf{a^x}$ then $\mathbf{\log_a}$ gives you **x** back again: $\log_a(a^x) = x$
- Doing $\mathbf{\log_a}$ then $\mathbf{a^x}$ gives you **x** back again: $a^{\log_a(x)} = x$