# CE203
# ساختمان داده ها و الگوریتم ها

**سجاد شیرعلی شهرضا**

**پاییز 1400**

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

# جلسه دوم:
# ضرب

**شنبه، 3 مهر 1400**

# MULTIPLICATION: THE PROBLEM

**Input**: 2 non-negative numbers, x and y (n digits each)

**Output**: the product $x \cdot y$

$$\begin{array}{r} 5678 \\ \times\ \ 1234 \\ \hline \mathbf{7006652} \end{array}$$

# GRADE-SCHOOL MULTIPLICATION

$$
\begin{array}{r}
45 \\
\times\ 63 \\
\hline
135 \\
2700 \\
\hline
\mathbf{2835}
\end{array}
$$

# GRADE-SCHOOL MULTIPLICATION

**Algorithm description (informal*):**
compute partial products (using multiplication & “carries” for digit overflows), and add all (properly shifted) partial products together

```
   45
x  63
-----
  135
 2700
-----
 2835
```

** This is not a good example of what your algorithm descriptions should look like on HW/exams*

# GRADE-SCHOOL MULTIPLICATION

```
    45123456678093420581217332421
  x 63782384198347750652091236423
  -------------------------------
                               ):
```

# GRADE-SCHOOL MULTIPLICATION

$$
\begin{array}{r}
\overbrace{45123456678093420581217332421}^{\boldsymbol{n}\ \textit{digits}} \\
\times\ 63782384198347750652091236423 \\
\hline
\end{array}
$$

):

**How efficient is this algorithm?**

(How many single-digit operations are required?)

# GRADE-SCHOOL MULTIPLICATION

$$\overbrace{\begin{array}{r} 4512345667809342058121733 2421 \\ \times\ 6378238419834775065209123 6423 \\ \hline ):\end{array}}^{\textbf{\textit{n}}\ \textit{digits}}$$

**How efficient is this algorithm?**

(How many single-digit operations *in the worst case*?)

**n partial products: ~2n² ops** (at most n multiplications & n additions per partial product)

**adding n partial products: ~2n² ops** (a bunch of additions & "carries")

# GRADE-SCHOOL MULTIPLICATION

$$\begin{array}{r} \overbrace{4512345667809342058121733 2421}^{\textbf{\textit{n}}\ \textit{digits}} \\ \times\ 6378238419834775065209123 6423 \\ \hline ):\end{array}$$

**How efficient is this algorithm?**

(How many single-digit operations *in the worst case*?)

**n partial products: ~$2n^2$ ops** (at most n multiplications & n additions per partial product)

**adding n partial products: ~$2n^2$ ops**

(a bunch of additions & "carries")

**~ $4n^2$ operations in the worst case**

# GRADE-SCHOOL MULTIPLICATION

*n digits*

4512345 … 17332421

x 63782 … 236423

): 

*THE QUESTION IS…*

## *CAN WE DO BETTER?*

**How efficient is this algo…**

(How many single-digit opera…

*in the worst case*?)

…oducts: ~$2n^2$ ops (at most n

… & n additions per partial product)

… n partial products: ~$2n^2$ ops

(a bunch of additions & "carries")

**~ $4n^2$ operations in the worst case**

# WHAT EXACTLY DOES “BETTER” MEAN?

Is **$1000000n$** operations better than $4n^2$?

Is **$0.000001n^3$** operations better than $4n^2$?

Is **$3n^2$** operations better than $4n^2$?

# WHAT EXACTLY DOES “BETTER” MEAN?

Is **1000000n** operations better than $4n^2$?

Is **$0.000001n^3$** operations better than $4n^2$?

Is **$3n^2$** operations better than $4n^2$?

- **The answers for the first two depend on what value n is…**
  - $1000000n < 4n^2$ only when n exceeds a certain value (in this case, 250000)
- **These constant multipliers are too environment-dependent...**
  - An operation could be faster/slower depending on the machine, so $3n^2$ ops on a slow machine might not be “better” than $4n^2$ ops on a faster machine

# WHAT EXACTLY DOES “BETTER” MEAN?

*INTRODUCING…*

**ASYMPTOTIC ANALYSIS**

# WHAT EXACTLY DOES "BETTER" MEAN?

*INTRODUCING...*

## ASYMPTOTIC ANALYSIS

- **Some guiding principles:** we care about how the running time/number of operations *scales* with the size of the input (i.e. the runtime's *rate of growth*), and we want some measure of runtime that's independent of hardware, programming language, memory layout, etc.

# WHAT EXACTLY DOES "BETTER" MEAN?

*INTRODUCING...*

## ASYMPTOTIC ANALYSIS

- **Some guiding principles:** we care about how the running time/number of operations *scales* with the size of the input (i.e. the runtime's *rate of growth*), and we want some measure of runtime that's independent of hardware, programming language, memory layout, etc.
  - Note: details like hardware/language/memory/compiler/etc. could totally be important to real world engineers, but in TheoryLand™, we want to reason about high-level algorithmic approaches rather than lower-level details

# ASYMPTOTIC ANALYSIS (High Level Idea)

*We'll express the asymptotic runtime of an algorithm using*

**BIG-O NOTATION**

*"big-oh of n squared"*
*or*
*"Oh of n squared"*

- We would say Grade-school Multiplication **"runs in time $O(n^2)$"**
    - Informally, this means that the runtime "scales like" $n^2$
    - We'll discuss the formal definition of Big-O (math-y stuff) next week

# ASYMPTOTIC ANALYSIS (High Level Idea)

*We'll express the asymptotic runtime of an algorithm using*

**BIG-O NOTATION**

*"big-oh of n squared"*
*or*
*"Oh of n squared"*

- We would say Grade-school Multiplication **"runs in time $O(n^2)$"**
  - Informally, this means that the runtime "scales like" $n^2$
  - We'll discuss the formal definition of Big-O (math-y stuff) next week

*THE POINT OF ASYMPTOTIC NOTATION*

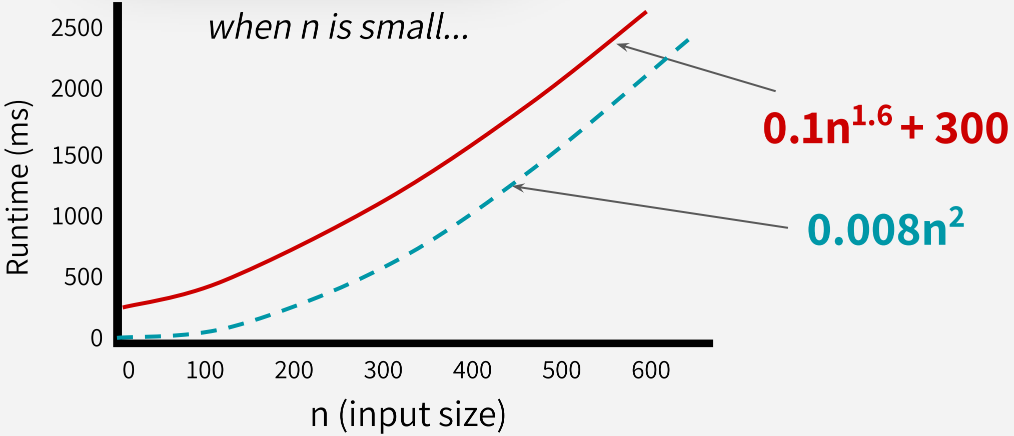
**suppress constant factors and lower-order terms**

*too system dependent* — *irrelevant for large inputs*

# ASYMPTOTIC ANALYSIS (High Level Idea)

*THE POINT OF ASYMPTOTIC NOTATION*

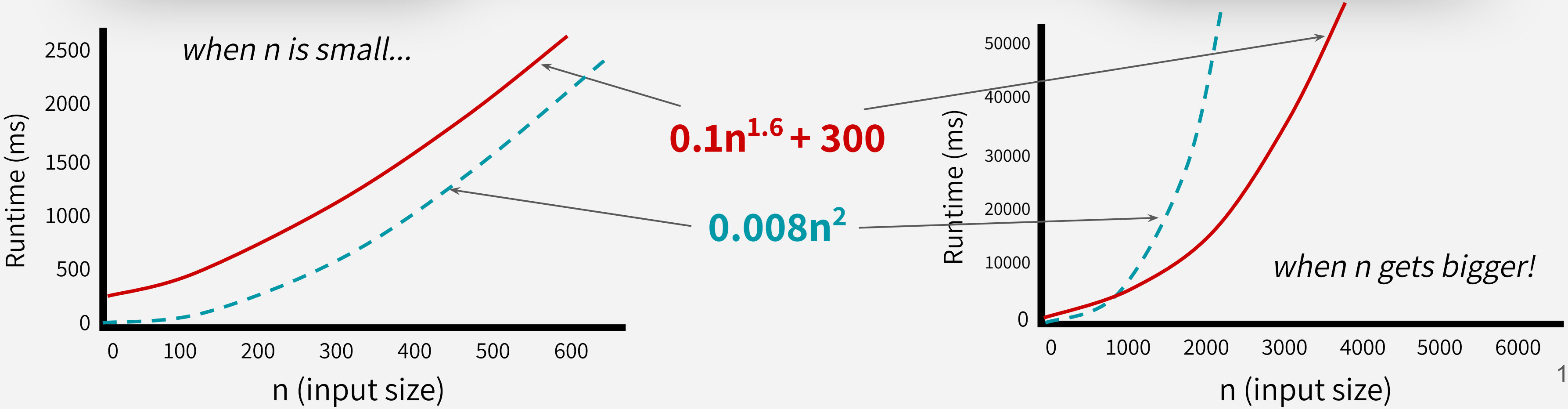
**suppress $\underbrace{\text{constant factors}}_{\textit{too system dependent}}$ and $\underbrace{\text{lower-order terms}}_{\textit{irrelevant for large inputs}}$**



# ASYMPTOTIC ANALYSIS (High Level Idea)

*THE POINT OF ASYMPTOTIC NOTATION*

**suppress $\underbrace{\text{constant factors}}_{\text{too system dependent}}$ and $\underbrace{\text{lower-order terms}}_{\text{irrelevant for large inputs}}$**

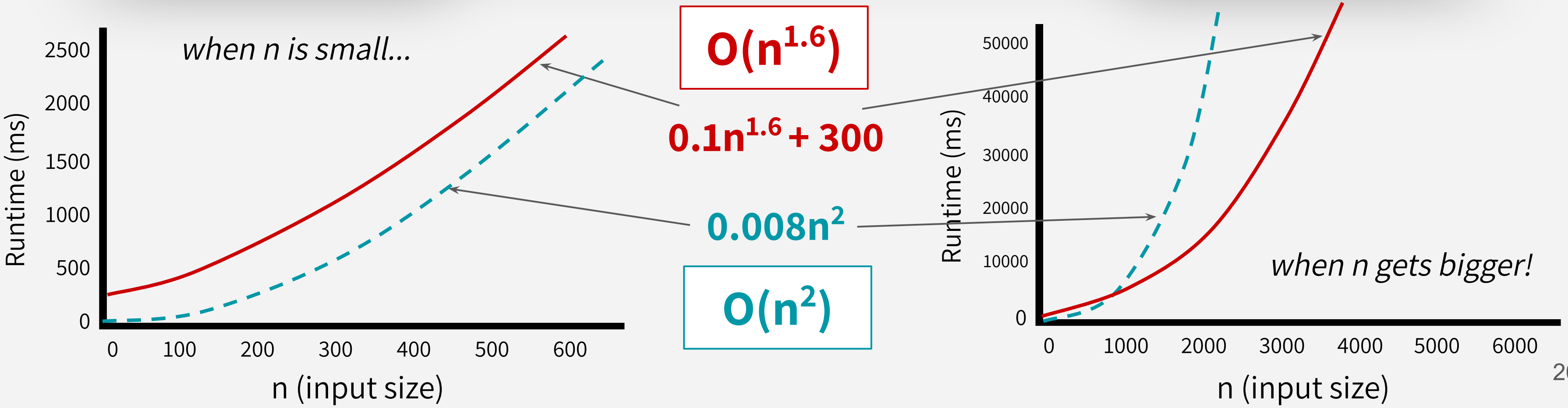


# ASYMPTOTIC ANALYSIS (High Level Idea)

*THE POINT OF ASYMPTOTIC NOTATION*

**suppress constant factors and lower-order terms**

*too system dependent* — *irrelevant for large inputs*



# ASYMPTOTIC ANALYSIS (High Level Idea)

- To compare algorithm runtimes in this class, we compare their Big-O runtimes
  - Ex: a runtime of $O(n^2)$ is considered “better” than a runtime of $O(n^3)$
  - Ex: a runtime of $O(n^{1.6})$ is considered “better” than a runtime of $O(n^2)$
  - Ex: a runtime of $O(1/n)$ is considered “better” than $O(1)$

# ASYMPTOTIC ANALYSIS (High Level Idea)

- To compare algorithm runtimes in this class, we compare their Big-O runtimes
  - Ex: a runtime of $O(n^2)$ is considered "better" than a runtime of $O(n^3)$
  - Ex: a runtime of $O(n^{1.6})$ is considered "better" than a runtime of $O(n^2)$
  - Ex: a runtime of $O(1/n)$ is considered "better" than $O(1)$

*So the question is:*

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

***Don't worry,*** *we'll revisit Asymptotic Analysis & Big-O stuff more formally next week!*

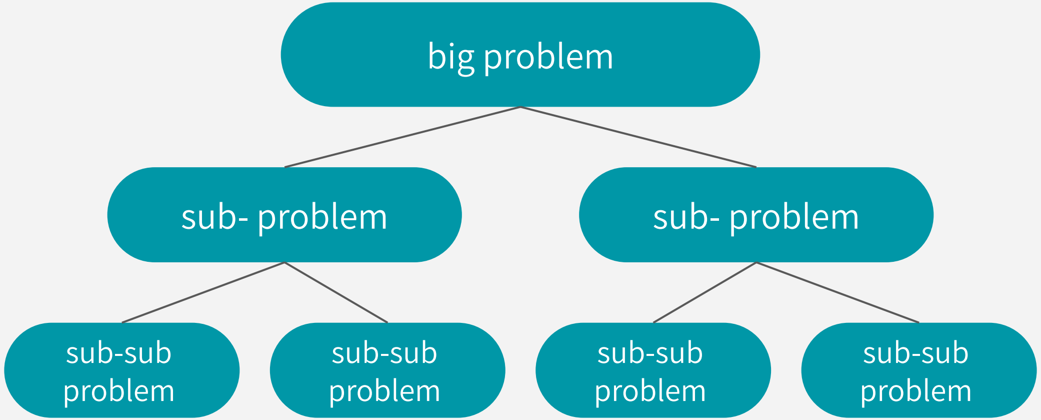

# سوال؟

# تقسیم و غلبه

Our first algorithm design paradigm

# DIVIDE AND CONQUER

- **An algorithm design paradigm:**
  1. break up a problem into smaller subproblems
  2. solve those subproblems *recursively*
  3. combine the results of those subproblems to get the overall answer



# MULTIPLICATION SUBPROBLEMS

- **Original large problem:** multiply 2 n-digit numbers
- **What are the subproblems?** Let's unravel some stuff...

# MULTIPLICATION SUBPROBLEMS

- **Original large problem:** multiply two 4-digit numbers
- **What are the subproblems?** Let's unravel some stuff...

$$1234 \times 5678$$

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

$$= ( 12 \times 56 ) 100^2 + ( 12 \times 78 + 34 \times 56 ) 100 + ( 34 \times 78 )$$

# MULTIPLICATION SUBPROBLEMS

- **Original large problem:** multiply two 4-digit numbers
- **What are the subproblems?** Let's unravel some stuff...

$$1234 \times 5678$$

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

$$= (\underbrace{12 \times 56}_{1})100^2 + (\underbrace{12 \times 78}_{2} + \underbrace{34 \times 56}_{3})100 + (\underbrace{34 \times 78}_{4})$$

*One 4-digit problem* → *Four 2-digit subproblems*

# MULTIPLICATION SUBPROBLEMS

- **Original large problem:** multiply 2 n-digit numbers
- **What are the subproblems?** More generally:

$$[x_1 \dots x_{n/2} x_{n/2+1} \dots x_n] \times [y_1 \dots y_{n/2} y_{n/2+1} \dots y_n]$$

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

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

1 2 3 4

*One n-digit problem* → *Four (n/2)-digit subproblems*

# LET'S SEE SOME PSEUDOCODE

```
MULTIPLY( x, y ):
```

x & y are n-digit numbers

***Note**: we're making a big assumption that n is a power of 2 just to make the pseudocode simpler*

# LET'S SEE SOME PSEUDOCODE

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y
```

x & y are n-digit numbers

**Base case**: we can just reference some memorized 1-digit multiplication tables

***Note***: *we're making a big assumption that n is a power of 2 just to make the pseudocode simpler*

# LET'S SEE SOME PSEUDOCODE

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y

    write x as a·10^(n/2) + b
    write y as c·10^(n/2) + d
```

x & y are n-digit numbers

**Base case**: we can just reference some memorized 1-digit multiplication tables

a, b, c, & d are (n/2)-digit numbers

***Note**: we're making a big assumption that n is a power of 2 just to make the pseudocode simpler*

# LET'S SEE SOME PSEUDOCODE

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y

    write x as a·10^(n/2) + b
    write y as c·10^(n/2) + d

    ac = MULTIPLY(a,c)
    ad = MULTIPLY(a,d)
    bc = MULTIPLY(b,c)
    bd = MULTIPLY(b,d)
```

x & y are n-digit numbers

**Base case**: we can just reference some memorized 1-digit multiplication tables

a, b, c, & d are (n/2)-digit numbers

These are **recursive** calls that provide subproblem answers

***Note***: *we're making a big assumption that n is a power of 2 just to make the pseudocode simpler*

# LET'S SEE SOME PSEUDOCODE

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y

    write x as a·10^(n/2) + b
    write y as c·10^(n/2) + d

    ac = MULTIPLY(a,c)
    ad = MULTIPLY(a,d)
    bc = MULTIPLY(b,c)
    bd = MULTIPLY(b,d)

    return ac·10^n + (ad + bc)·10^(n/2) + bd
```

x & y are n-digit numbers

**Base case**: we can just reference some memorized 1-digit multiplication tables

a, b, c, & d are (n/2)-digit numbers

These are **recursive** calls that provide subproblem answers

Add them up to get our overall answer!

***Note**: we're making a big assumption that n is a power of 2 just to make the pseudocode simpler*

# HOW EFFICIENT IS THIS ALGORITHM?

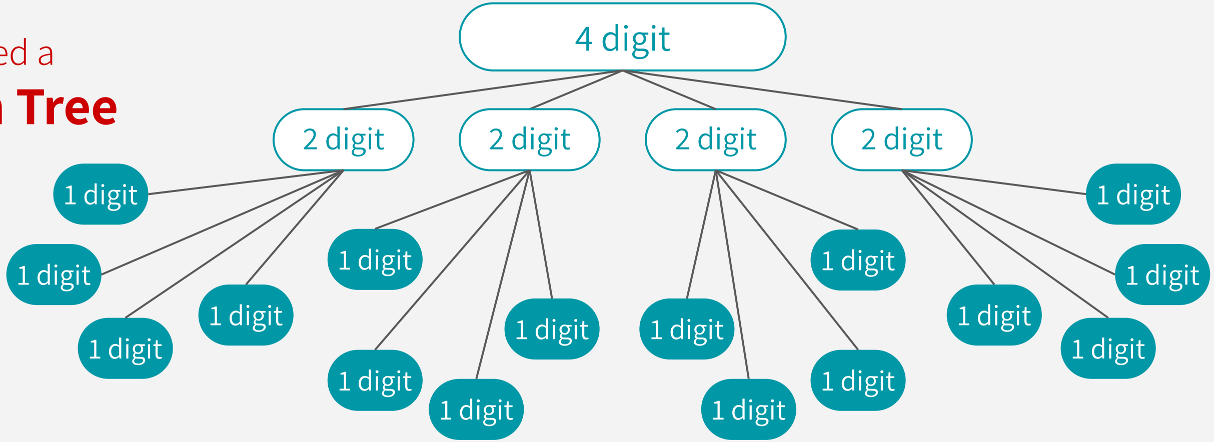
- **Let's start small:** if we're multiplying two 4-digit numbers, how many 1-digit multiplications does the algorithm perform?
    - In other words, how many times do we reach the base case where we actually perform a "multiplication" (a.k.a. a table lookup)?
    - This at least lower bounds the number of operations needed overall

# HOW EFFICIENT IS THIS ALGORITHM?

- **Let's start small:** if we're multiplying two 4-digit numbers, how many 1-digit multiplications does the algorithm perform?
    - In other words, how many times do we reach the base case where we actually perform a "multiplication" (a.k.a. a table lookup)?
    - This at least lower bounds the number of operations needed overall

This is called a

**Recursion Tree**

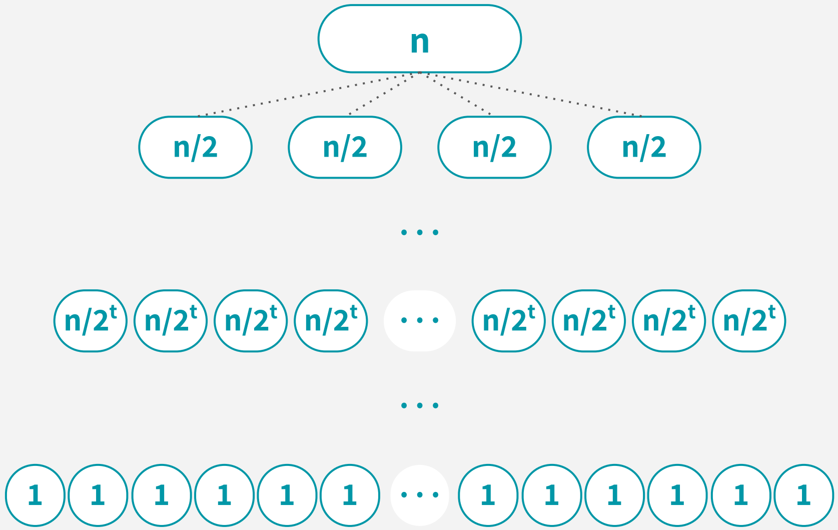



***Sixteen 1-digit multiplications!***

# HOW EFFICIENT IS THIS ALGORITHM?

- **Now let's generalize:** if we're multiplying two n-digit numbers, how many 1-digit multiplications does the algorithm perform?

**Recursion Tree**




**Level 0**: 1 problem of size n

**Level 1**: $4^1$ problems of size n/2

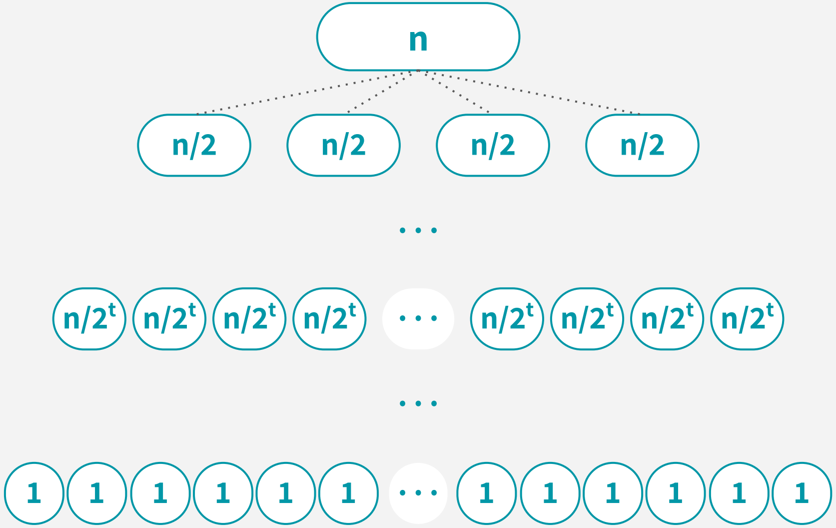
**Level t**: $4^t$ problems of size $n/2^t$

**Level $\log_2 n$**: ____ problems of size 1

# HOW EFFICIENT IS THIS ALGORITHM?

- **Now let's generalize:** if we're multiplying two n-digit numbers, how many 1-digit multiplications does the algorithm perform?

**Recursion Tree**




**Level 0**: 1 problem of size n

**Level 1**: $4^1$ problems of size n/2

**Level t**: $4^t$ problems of size $n/2^t$

**Level $\log_2 n$**: __**$n^2$**__ problems of size 1

**$\log_2 n$ levels**
(you need to cut n in half $\log_2 n$ times to get to size 1)

**# of problems on last level (size 1)**

$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

# HOW EFFICIENT IS THIS ALGORITHM?

The running time of this Divide-and-Conquer multiplication algorithm is **at least $O(n^2)$**!

We know there are already $n^2$ multiplications happening at the bottom level of the recursion tree, so that's why we say "at least" $O(n^2)$

# HOW EFFICIENT IS THIS ALGORITHM?

The running time of this Divide-and-Conquer multiplication algorithm is **at least $O(n^2)$**!

We know there are already $n^2$ multiplications happening at the bottom level of the recursion tree, so that's why we say "at least" $O(n^2)$

Wait, our grade-school algorithm was already $O(n^2)$!

Is Divide-and-Conquer really that useless?

# HOW EFFICIENT IS THIS ALGORITHM?

The running time of this Divide-and-Conquer multiplication algorithm is **at least $O(n^2)$**!

We know there are already $n^2$ multiplications happening at the bottom level of the recursion tree, so that's why we say "at least" $O(n^2)$

Wait, our grade-school algorithm was already $O(n^2)$!
Is Divide-and-Conquer really that useless?

**Karatsuba says no!!!**

# روش ضرب اعداد صحیح کاراتسوبا

Three subproblems instead of four!

# CHOOSING SUBPROBLEMS WISELY

$$[x_1 \ldots x_{n/2} x_{n/2+1} \ldots x_n] \times [y_1 \ldots y_{n/2} y_{n/2+1} \ldots y_n]$$

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

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

**The subproblems we choose to solve just need to provide these quantities:**

$ac$ $\qquad$ $ad + bc$ $\qquad$ $bd$

*Originally, we assembled these quantities by computing FOUR things: ac, ad, bc, and bd.*

# KARATSUBA'S TRICK

$$\text{end result} = (\,ac\,)10^{n} + (\,ad + bc\,)10^{n/2} + (\,bd\,)$$

# KARATSUBA'S TRICK

$$\text{end result} = (\,ac\,)10^{n} + (\,ad + bc\,)10^{n/2} + (\,bd\,)$$

$ac$ & $bd$ can be recursively computed as usual

$ad + bc$ is equivalent to $(a+b)(c+d) - ac - bd$

$$= (ac + ad + bc + bd) - ac - bd$$
$$= ad + bc$$

# KARATSUBA'S TRICK

end result = ( ac )$10^n$ + ( ad + bc )$10^{n/2}$ + ( bd )

**ac** & **bd** can be recursively computed as usual

**ad** + **bc** is equivalent to **(a+b)(c+d) - ac - bd**

= (ac + ad + bc + bd) - ac - bd
= ad + bc

So, instead of computing **ad** & **bc** as two separate subproblems, let's just compute **(a+b)(c+d)** instead!

# OUR THREE SUBPROBLEMS

These *three* subproblems give us everything we need to compute our desired quantities:

| | |
|---|---|
| 1 | **ac** |
| 2 | **bd** |
| 3 | **(a+b)(c+d)** |

Assemble our overall product by combining these three subproblems:

$$(\ ac\ )10^n + (\ ad + bc\ )10^{n/2} + (\ bd\ )$$

$$\underbrace{ac}_{1} \qquad \underbrace{ad+bc}_{3-1-2} \qquad \underbrace{bd}_{2}$$

# OUR THREE SUBPROBLEMS

These *three* subproblems give us everything we need to compute our desired quantities:

| | |
|---|---|
| 1 | **ac** |
| 2 | **bd** |
| 3 | **(a+b)(c+d)** |

(a+b) and (c+d) are both going to be n/2-digit numbers!

⇩

This means we still have half-sized subproblems!

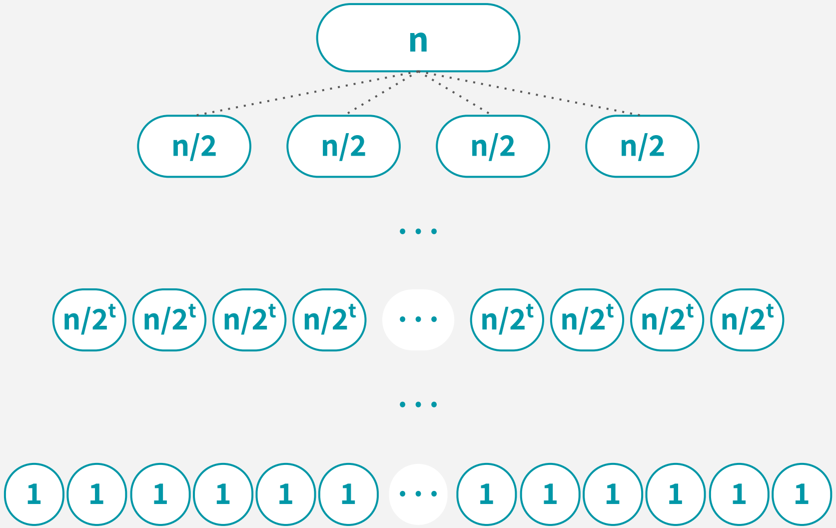
Assemble our overall product by combining these three subproblems:

$$(\ ac\ )10^{n} + (\ ad + bc\ )10^{n/2} + (\ bd\ )$$

$$\underbrace{ac}_{1} \qquad \underbrace{ad + bc}_{3 - 1 - 2} \qquad \underbrace{bd}_{2}$$

# WHAT'S THE RUNTIME?

***This was the Recursion Tree + Analysis from Divide-and-Conquer Attempt 1:***




**Level 0**: 1 problem of size n

**Level 1**: $4^1$ problems of size n/2

**Level t**: $4^t$ problems of size $n/2^t$

**Level $\log_2 n$**: __**$n^2$**__ problems of size 1

**$\log_2 n$ levels**

(you need to cut n in half $\log_2 n$ times to get to size 1)

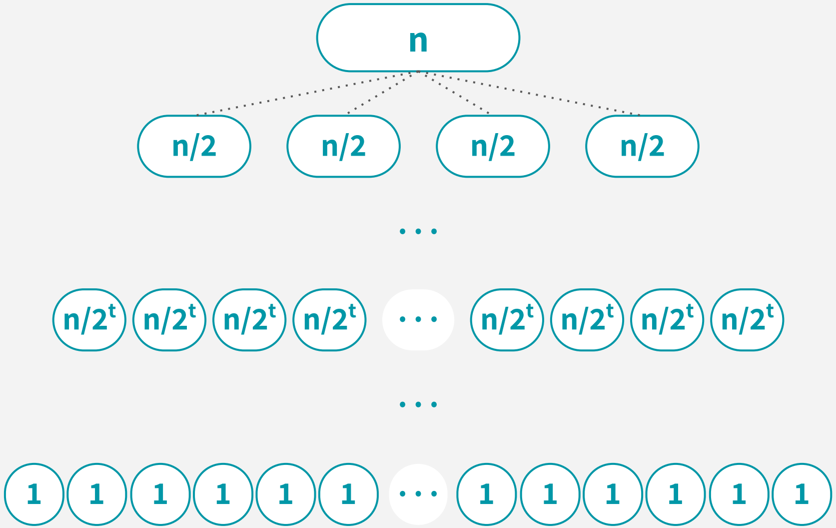
**# of problems on last level (size 1)**

$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

# WHAT'S THE RUNTIME?

***This was the Recursion Tree + Analysis from Divide-and-Conquer Attempt 1:***




**Level 0**: 1 problem of size n

**Level 1**: $4^1$ problems of size n/2

**Level t**: $4^t$ problems of size $n/2^t$

**Level $\log_2 n$**: _**$n^2$**_ problems of size 1

**$\log_2 n$ levels**

(you need to cut n in half $\log_2 n$ times to get to size 1)

**# of problems on last level (size 1)**

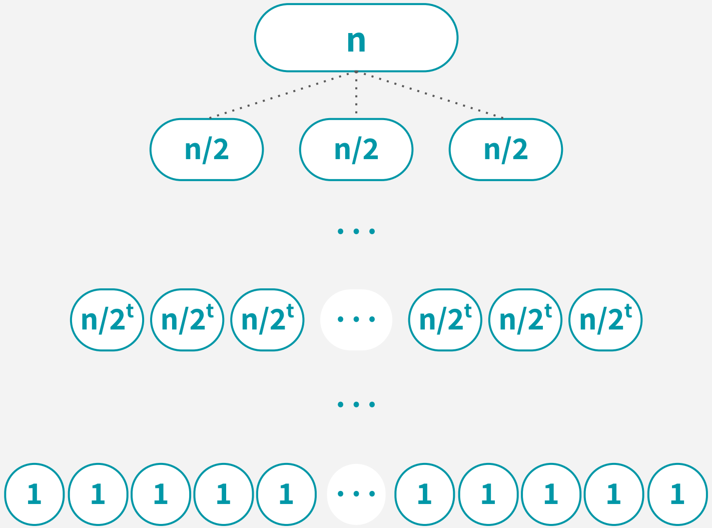
$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

**For Karatsuba's, we'll replace the branching factor of 4 with a 3! ⇒**

# WHAT'S THE RUNTIME?

## Karatsuba Multiplication Recursion Tree




**Level 0**: 1 problem of size n

**Level 1**: $3^1$ problems of size n/2

**Level t**: $3^t$ problems of size $n/2^t$

**Level $\log_2 n$**: __**$n^{1.6}$**__ problems of size 1

**$\log_2 n$ levels**

(you need to cut n in half $\log_2 n$ times to get to size 1)

**# of problems on last level (size 1)**

$= 3^{\log_2 n} = n^{\log_2 3}$

$\approx \mathbf{n^{1.6}}$

**Thus, the runtime is $O(n^{1.6})$!**

# WHAT'S THE RUNTIME?

## Karatsuba Multiplication Recursion Tree

**NOTE**: I know it looks like we didn't account for the work done on higher levels in the recursion tree, but as we'll learn later, the work on the last level actually dominates *in this particular recursion tree*!


**Level 0**: 1 problem of size n

**Level 1**: $3^1$ problems of size n/2

**Level t**: $3^t$ problems of size $n/2^t$

**Level $\log_2 n$**: $\underline{n^{1.6}}$ problems of size 1

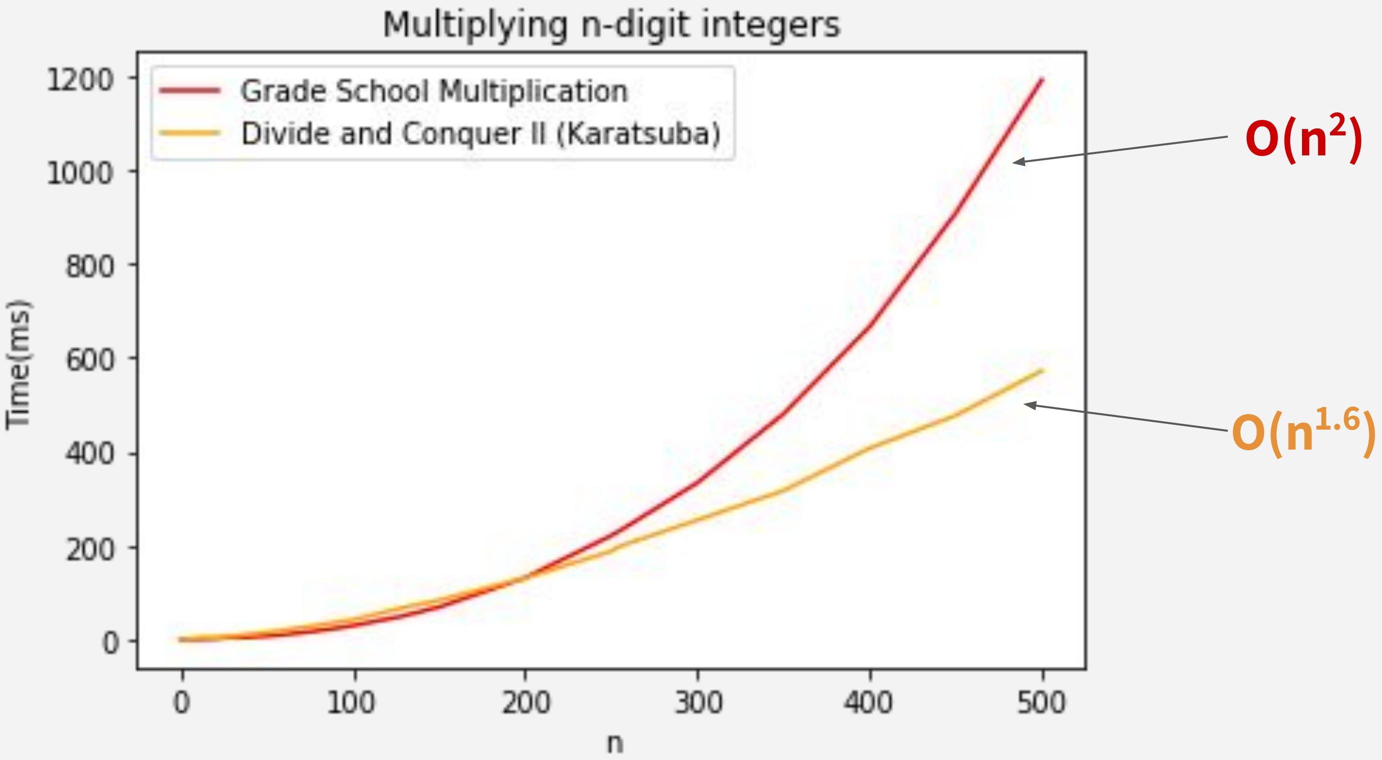
**$\log_2 n$ levels**
(you need to cut n in half $\log_2 n$ times to get to size 1)

**# of problems on last level (size 1)**
$= 3^{\log_2 n} = n^{\log_2 3}$

$\approx \mathbf{n^{1.6}}$

**Thus, the runtime is $O(n^{1.6})$!**

# IT WORKS IN PRACTICE TOO!



# CAN WE DO BETTER?

- **Toom-Cook (1963):** another Divide & Conquer! Instead of breaking into three (n/2)-sized problems, break into five (n/3)-sized problems.
  - Runtime: $O(n^{1.465})$
- **Schönhage–Strassen (1971):** uses fast polynomial multiplications
  - Runtime: $O(n \log n \log \log n)$
- **Fürer (2007):** uses Fourier Transforms over complex numbers
  - Runtime: $O(n \log(n)\, 2^{O(\log^*(n))})$
- **Harvey and van der Hoeven (2019!):** crazy stuff
  - Runtime: $O(n \log(n))$

We won't expect you to know any of these algorithms by the way!


سوال؟