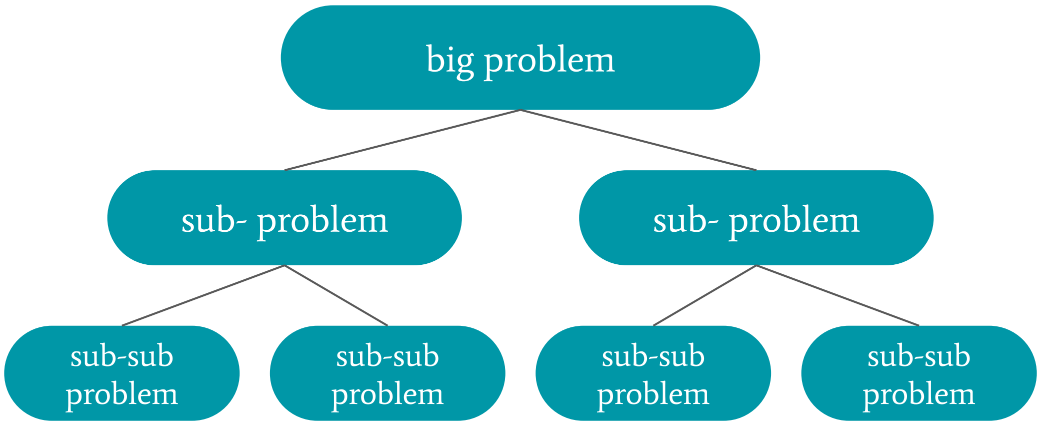
Divide & Conquer

algorithm *design* paradigm

Divide & 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?

Multiplication Subproblems

Original large problem: multiply two 4-digit numbers

What are the subproblems?

$$1234 \text{ x } 5678$$

$$= (12\text{x}100 + 34) \text{ x } (56\text{x}100 + 78)$$

$$= (12\text{x}56)100^2 + (12\text{x}78 + 34\text{x}56)100 + (34\text{x}78)$$

Multiplication Subproblems

Original large problem: multiply two 4-digit numbers

What are the subproblems?

$$1234 \times 5678$$

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

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

1 2 3 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

$$[x_1 x_2 \ldots x_{n-1} x_n] \times [y_1 y_2 \ldots y_{n-1} y_n]$$

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

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

1 2 3 4

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

Some Pseudocode

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

x & y are n-digit numbers

making an assumption that n is a power of 2 just to make the pseudocode simpler

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

making an assumption that n is a power of 2 just to make the pseudocode simpler

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

making an assumption that n is a power of 2 just to make the pseudocode simpler

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

making an assumption that n is a power of 2 just to make the pseudocode simpler

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

making an assumption that n is a power of 2 just to make the pseudocode simpler

Efficiency of the algorithm?

Start small: if we're multiplying two 4-digit numbers, how many 1-digit multiplications does the algorithm perform?

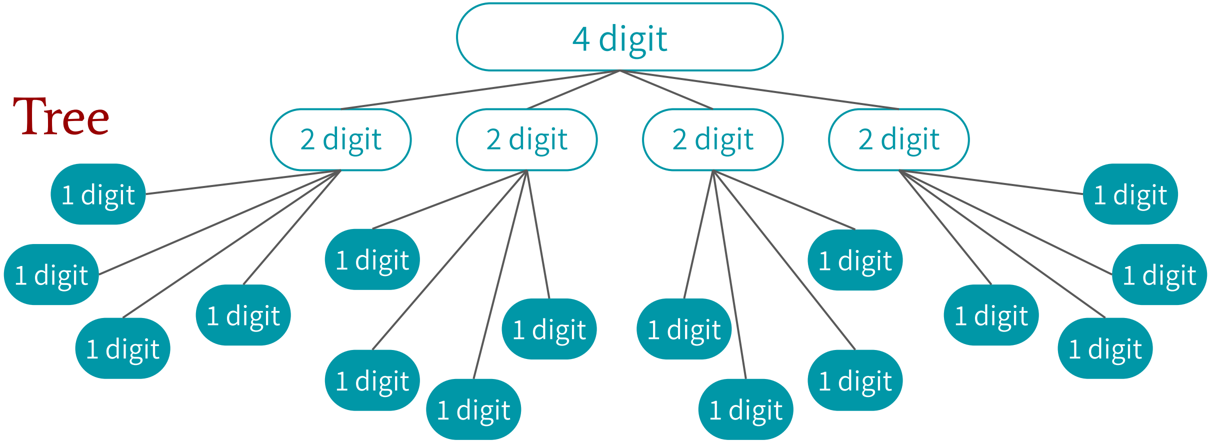
- In other words, how many times do we reach the base case where we actually perform a "multiplication" (a.k.a. a table lookup)?
- This at least lower bounds the number of operations needed overall

Efficiency of the 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

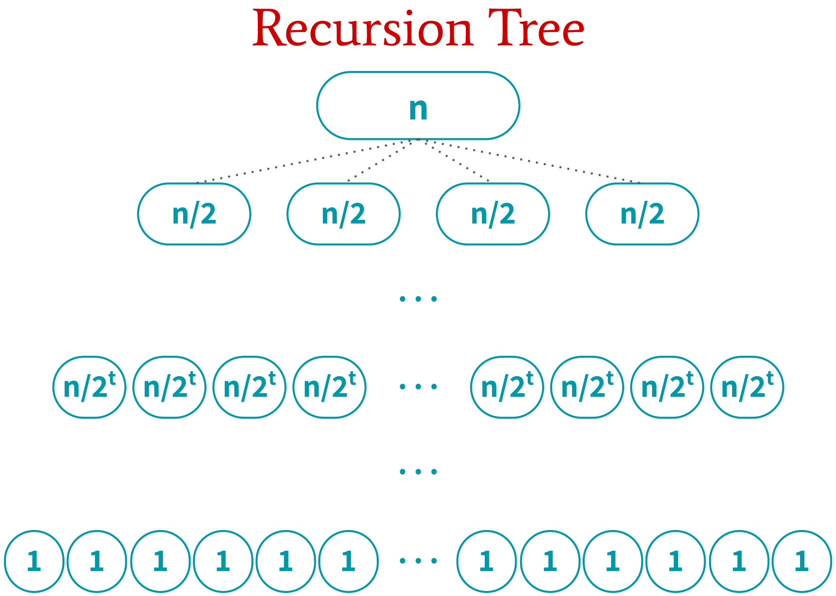
Recursion Tree



Sixteen 1-digit multiplications

Efficiency of the algorithm?

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



Level 0: 1 problem of size n

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

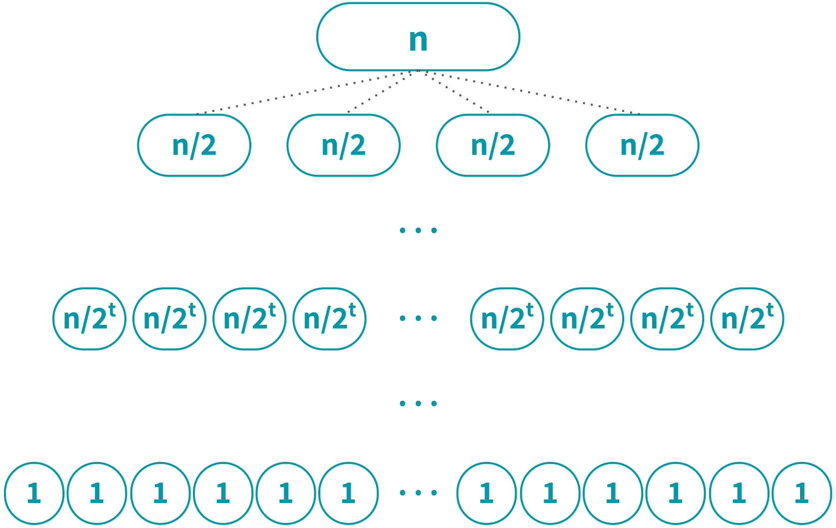
Level t: 4^t problems of size $n/2^t$

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

Efficiency of the algorithm?

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 $\log_2 n$: $\underline{\mathbf{n^2}}$ 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)

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

$= \mathbf{n^2}$

Efficiency of the algorithm?

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" $O(n^2)$

KARATSUBA's INTEGER MULTIPLICATION

Three subproblems instead of four

Choose Subproblems Wisely

$$[x_1 x_2 \ldots x_{n-1} x_n] \times [y_1 y_2 \ldots y_{n-1} y_n]$$

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

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

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

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

KARATSUBA'S idea

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

KARATSUBA'S idea

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 idea

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

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

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

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

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

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:

$$\underbrace{(\ ac\ )}_{1}10^{n} + \underbrace{(\ ad + bc\ )}_{3\,-\,1\,-\,2}10^{n/2} + \underbrace{(\ bd\ )}_{2}$$

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!

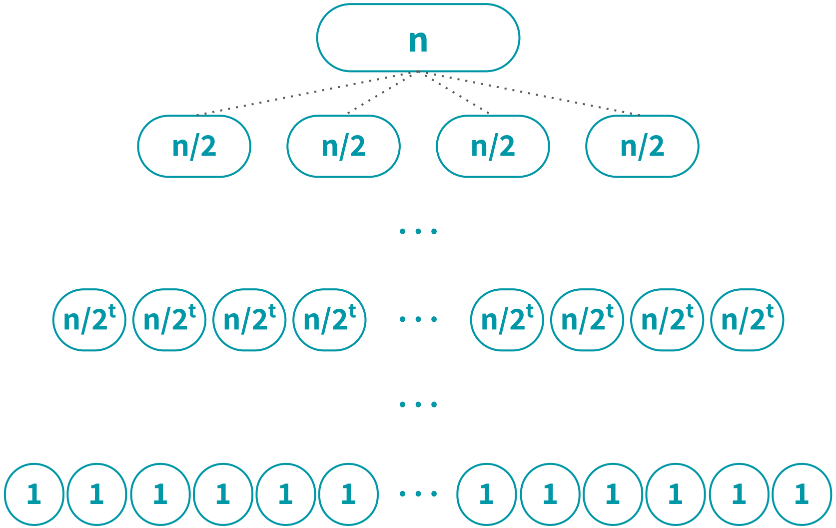
Assemble our overall product by combining these three subproblems:

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

1 | 3 - 1 - 2 | 2

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

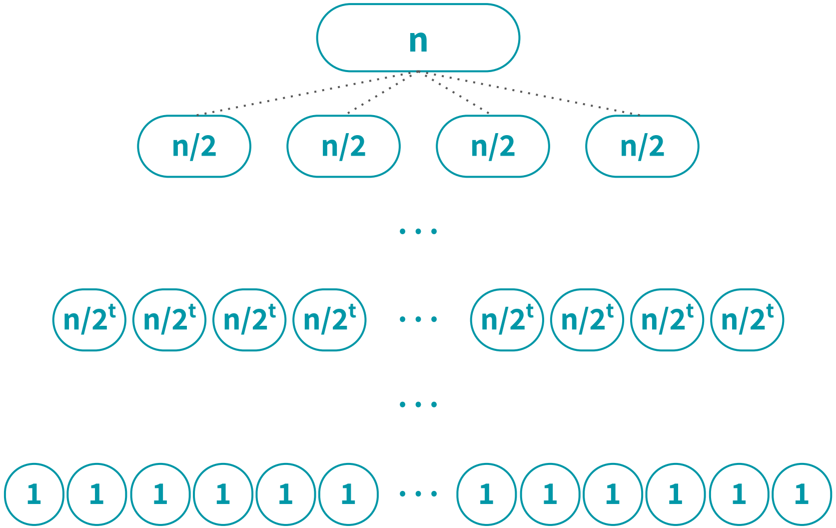
$\log_2 n$ levels
(you need to cut n in half $\log_2 n$ times to get to size 1)

# of problems on last level (size 1)
$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

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)

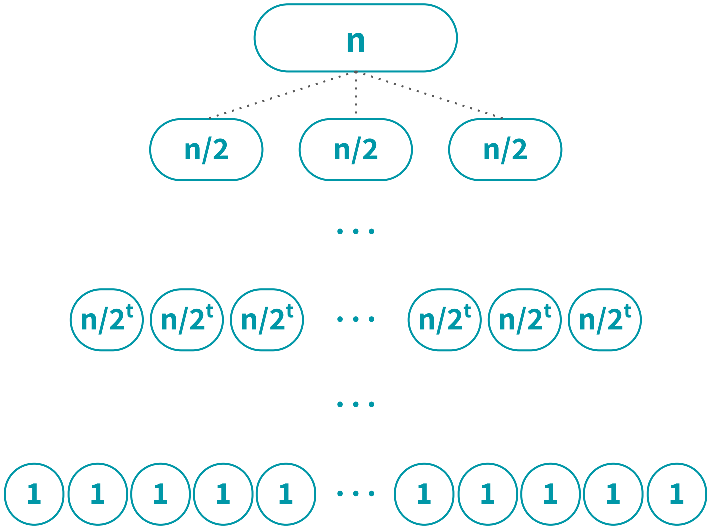
# of problems on last level (size 1)
$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

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

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)

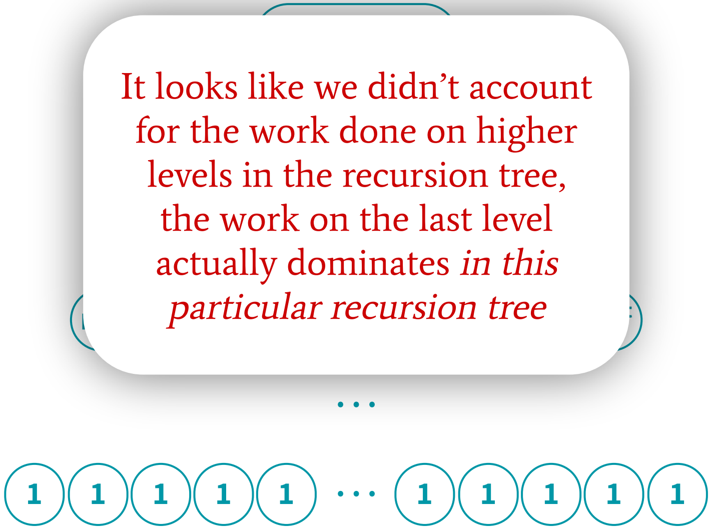
# of problems on last level (size 1)
$= 3^{\log_2 n} = n^{\log_2 3}$

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

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

Runtime?

Karatsuba Multiplication Recursion Tree



It looks like we didn't account for the work done on higher levels in the recursion tree, 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 $n/2^t$

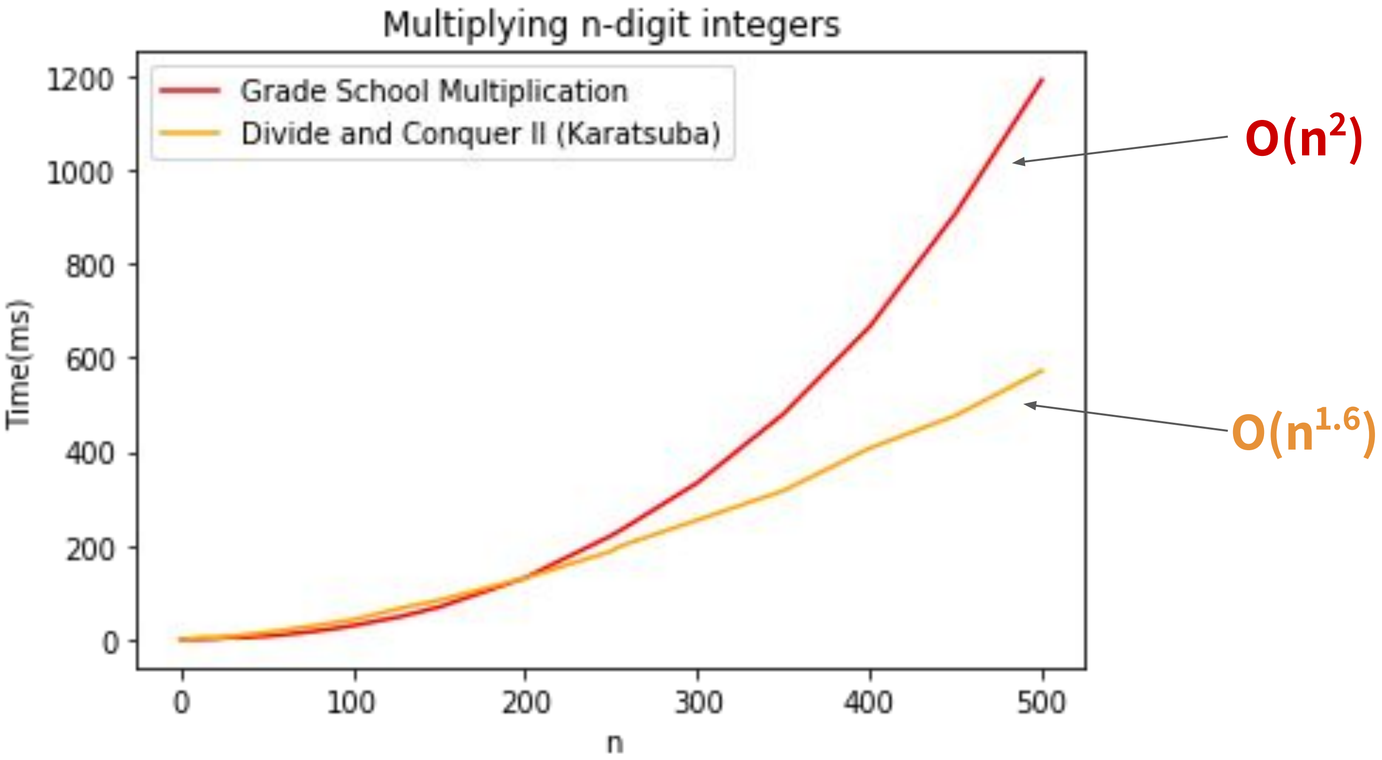
Level $\log_2 n$: $\underline{n^{1.6}}$ problems of size 1

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

# of problems on last level (size 1)
$= 3^{\log_2 n} = n^{\log_2 3}$
$\approx \mathbf{n^{1.6}}$

The runtime is $O(n^{1.6})$!

Runtime


Multiplying n-digit integers
Grade School Multiplication
Divide and Conquer II (Karatsuba)
Time(ms)
n
0
200
400
600
800
1000
1200
0
100
200
300
400
500
$O(n^2)$
$O(n^{1.6})$

Researchers always want to do better ...

Before 1960 Runtime: $O(n^2)$

Karatsuba (1960) Runtime: $O(n^{1.6})$

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: $O(n \log n \log \log n)$

Harvey and van der Hoeven (2019)

- Runtime: $O(n \log(n))$