Multiplication + Master Theorem

CS/MCS 401: Computer Algorithms I
Lecture 6
Professor: Jonathan Liu

Sit next to someone! If you don't do it now, you'll have to move later.

Introduce yourself!
Icebreaker: You find a $10 bill, but it expires tomorrow. What do you spend it on?

Lesson Goals

Today we'll see another divide and conquer algorithm, and we'll use it to talk about how to formally write proofs of correctness and runtime for D/C algorithms.

Basic Arithmetic

Arithmetic

Arithmetic is important!

In this lecture, we discuss two operations:

- Addition
- Multiplication

We'll discuss exponentiation on Tuesday.

Binary Numbers Refresher

Remember that all data is stored on computers, and all computers store data in bits. This means that all numbers are represented in **binary**, consisting only of 0's and 1's.

For example:

$$53 \text{ in binary} = 53_2 = 2^5 + 2^4 + 2^2 + 2^0 = 110101.$$

Note: multiplying a number by 2 is the same as adding a 0 to the end! For example, $106_2 = 110101\mathbf{0}$.

Binary Arithmetic Recap

```
  1       1  1  1
    1 1 0 1 0 1
+ 1 0 0 0 1 1
-------------
1 0 1 1 0 0 0
```

Binary addition works the same as addition in Base 10 – don’t forget to carry!

Addition: Pseudocode

Algorithm 1 Addition

```
1: carry_bit ← 0
2: Solution ← [0] ∗ n.
3: for i from n to 1 do
4:     value = A[i] + B[i]+ carry_bit
5:     if A[i] + B[i]+ carry_bit ≥ 2 then:
6:         value = value - 2, carry_bit = 1
7:     else
8:         carry_bit = 0
9:     end if
10:    Solution[i] ← value
11: end for
12: return solution
```

Go bit by bit from right to left, adding the top and bottom.

If those two and the carry add up to 2 or more, make the carry 1 and store the new value. Else, carry is 0.

Store the value of that bit in the solution.

Return the solution.

Addition: Pseudocode

What is the runtime? Can we do better? How, or why not?

Algorithm 1 Addition

```
1: carry_bit ← 0
2: Solution ← [0] * n.
3: for i from n to 1 do
4:     value = A[i] + B[i]+ carry_bit
5:     if A[i] + B[i]+ carry_bit ≥ 2 then:
6:         value = value - 2, carry_bit = 1
7:     else
8:         carry_bit = 0
9:     end if
10:    Solution[i] ← value
11: end for
12: return solution
```

Go bit by bit from right to left, adding the top and bottom.

If those two and the carry add up to 2 or more, make the carry 1 and store the new value. Else, carry is 0.

Store the value of that bit in the solution.

Return the solution.

Addition: Pseudocode

Algorithm 1 Addition

```
1:  carry_bit ← 0
2:  Solution ← [0] * n.
3:  for i from n to 1 do
4:      value = A[i] + B[i]+ carry_bit
5:      if A[i] + B[i]+ carry_bit ≥ 2 then:
6:          value = value - 2, carry_bit = 1
7:      else
8:          carry_bit = 0
9:      end if
10:     Solution[i] ← value
11: end for
12: return solution
```

What is the runtime? Can we do better? How, or why not?

It's O(n). We CAN'T do better – it takes O(n) to read every bit!

Go bit by bit from right to left, adding the top and bottom.

If those two and the carry add up to 2 or more, make the carry 1 and store the new value. Else, carry is 0.

Store the value of that bit in the solution.

Return the solution.

Multiplication

Remember that, for an n-bit number, “dividing by 2” is the same as removing the last term!

Our classical multiplication algorithm!

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & & \\
+ & 1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
$$

(1101 times 1)
(1101 times 1, shifted once)
(1101 times 0, shifted twice)
(1101 times 1, shifted thrice)

(binary 143)

Multiplication

Our classical multiplication algorithm!

What is the runtime?

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & & \\
+ & 1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
$$

(1101 times 1)
(1101 times 1, shifted once)
(1101 times 0, shifted twice)
(1101 times 1, shifted thrice)

(binary 143)

Multiplication

Our classical multiplication algorithm!

What is the runtime?

n rows, each does an addition -> $O(n^2)$

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & & \\
+ & 1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
$$

(1101 times 1)

(1101 times 1, shifted once)

(1101 times 0, shifted twice)

(1101 times 1, shifted thrice)

(binary 143)

Multiplication on the Worksheet

Algorithm 1 Multiplication

```
1:  procedure MULTIPLY(x, y)
2:      if y = 0 then
3:          return 0
4:      end if
5:      z ← MULTIPLY(x, ⌊y/2⌋)
6:      if y is even then
7:          return 2z
8:      else
9:          return x + 2z
10:     end if
11: end procedure
```

What is the runtime?

Each recursive call divides by 2, so it decreases the length by 1. So there can only be n calls in total.

Each call does at most one addition before returning, so it takes at most O(n) time.

n recursive calls, each doing O(n) work, leaves us with **$O(n^2)$**

Multiplication on the Worksheet

What is the runtime?

Write x and y next to each other

Divide the first number by 2 (rounding down), double the second number

Repeat until the first number is 1.

O(1)

n times

Add the numbers on the right where the left number is odd.

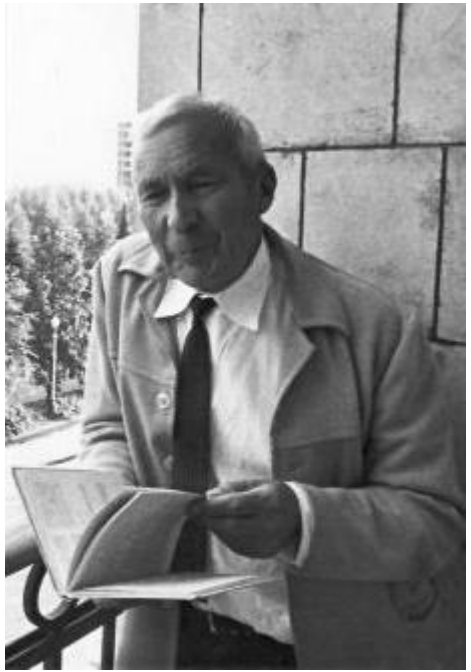
n additions, each O(n)

$=O(n^2)$

Takeaway

All of our algorithms take $\mathbf{O(n^2)}$ time!

In 1960, Kolmogorov (right, father of probability theory) conjectured at a seminar that it was impossible to do better than this.

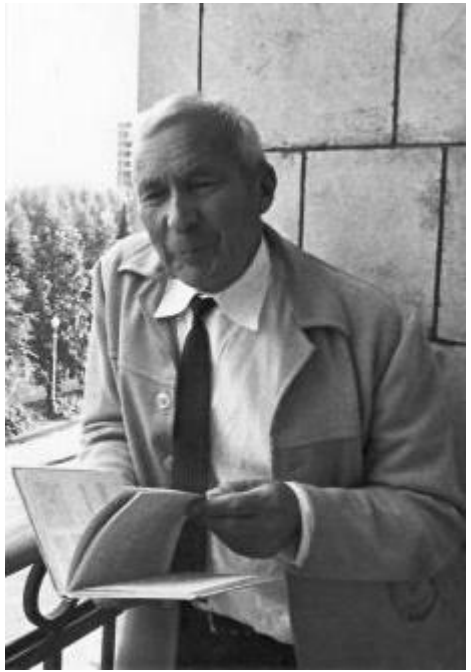


Takeaway?

All of our algorithms take **$O(n^2)$** time!

In 1960, Kolmogorov (right, father of probability theory) conjectured at a seminar that it was impossible to do better than this.

A week later, a student attending the seminar proved him wrong.



Karatsuba Multiplication

Let’s try divide and conquer!

Say we’re multiplying two numbers x and y.

$$x = 2307$$

$$y = 1154$$

Let’s try divide and conquer!

Say we’re multiplying two numbers x and y.
Let’s try splitting them in half.

$$x = 2307$$

$$y = 1154$$

Let’s try divide and conquer!

Say we’re multiplying two numbers x and y.
Let’s try splitting them in half.

$$x = 2307 = 23 \cdot 100 + 07$$

$$y = 1154 = 11 \cdot 100 + 54$$

Let’s try divide and conquer!

Say we’re multiplying two numbers x and y.
Let’s try splitting them in half.

$$x = 2307 = 23 \cdot 100 + 07$$

$$y = 1154 = 11 \cdot 100 + 54$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07) + 100(23 \cdot 54) + (07 \cdot 54)$$

Let's try divide and conquer!

Say we're multiplying two numbers x and y.
Let's try splitting them in half.

$$x = 2307 = 23 \cdot 100 + 07$$

$$y = 1154 = 11 \cdot 100 + 54$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07) + 100(23 \cdot 54) + (07 \cdot 54)$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07 + 23 \cdot 54) + (07 \cdot 54)$$

Let’s try divide and conquer!

Say we’re multiplying two numbers x and y.

$$x = 2307,\ y = 1154$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07 + 23 \cdot 54) + (07 \cdot 54)$$

Multiplying by a power of 10 is free.

Let's try divide and conquer!

Say we're multiplying two numbers x and y.

$$x = 2307,\ y = 1154$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07 + 23 \cdot 54) + (07 \cdot 54)$$

Multiplying by a power of 10 is free.

Multiplying two n-digit numbers takes **4 multiplications of size n/2** and **3 additions of size n**.

So if T(n) is the time it takes to multiply numbers of size n, then...

Let's try divide and conquer!

Say we're multiplying two numbers x and y.

$$x = 2307,\ y = 1154$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07 + 23 \cdot 54) + (07 \cdot 54)$$

Multiplying by a power of 10 is free.

Multiplying two n-digit numbers takes **4 multiplications of size n/2** and **3 additions of size n**.

So if T(n) is the time it takes to multiply numbers of size n, then...

$$\mathbf{T(n) = 4T(n/2) + O(n)}.$$

Let's try divide and conquer!

Say we're multiplying two numbers x and y.

$$x = 2307,\ y = 1154$$

$$x \cdot y = 10000(23 \cdot 11) + 100(11 \cdot 07 + 23 \cdot 54) + (07 \cdot 54)$$

Multiplying by a power of 10 is free.

Multiplying two n-digit numbers takes **4 multiplications of size n/2** and **3 additions of size n**.

So if T(n) is the time it takes to multiply numbers of size n, then...

$$\mathbf{T(n) = 4T(n/2) + O(n)}.$$

$\mathbf{T(n) = O(n^2)}$

Let's try again.

Say we're multiplying two numbers x and y.

$$x = AB,\ y = CD$$

$$x \cdot y = 10^{n}(A \cdot C) + 10^{n/2}(B \cdot C + A \cdot D) + (B \cdot D)$$

We need to know $(A \cdot C)$, $(B \cdot D)$, and $(B \cdot C + A \cdot D)$.

This is expensive!
Two multiplications!

Let's try again.

Say we're multiplying two numbers x and y.

$$x = AB,\ y = CD$$

$$x \cdot y = 10^{n}(A \cdot C) + 10^{n/2}(B \cdot C + A \cdot D) + (B \cdot D)$$

We need to know $(A \cdot C)$, $(B \cdot D)$, and $(B \cdot C + A \cdot D)$.

Here's the trick:

$$(A+B) \cdot (C+D) = (A \cdot C) + (B \cdot D) + (B \cdot C) + (A \cdot D)$$

Let's try again.

Say we're multiplying two numbers x and y.

$$x = AB,\ y = CD$$

$$x \cdot y = 10^{n}(A \cdot C) + 10^{n/2}(B \cdot C + A \cdot D) + (B \cdot D)$$

We need to know $(A \cdot C)$, $(B \cdot D)$, and $(B \cdot C + A \cdot D)$.

Here's the trick:

$$(A+B) \cdot (C+D) = (A \cdot C) + (B \cdot D) + (B \cdot C) + (A \cdot D)$$

One multiplication! We can use this for the last term.

Karatsuba Multiplication Algorithm (1960)

Algorithm 2 Karatsuba Multiplication on Numbers of Length n

```
1: procedure MULTIPLY(x, y)
2:     if n=1 then
3:         return x · y
4:     end if
5:
6:     Write x as x_L x_R, where x = 10^{n/2} x_L + x_R
7:     Write y as y_L y_R, where x = 10^{n/2} y_L + y_R
8:
9:     a ← MULTIPLY(x_L, y_L)
10:    c ← MULTIPLY(x_R, y_R)
11:    b ← MULTIPLY(x_L + x_R, y_L + y_R) − a − c
12:
13:    return 10^n · a + 10^{n/2} · b + c.
14: end procedure
```

Karatsuba Multiplication Algorithm (1960)

Algorithm 2 Karatsuba Multiplication on Numbers of Length n

```
1: procedure MULTIPLY(x, y)
2:     if n=1 then
3:         return x · y
4:     end if
5:
6:     Write x as x_L x_R, where x = 10^{n/2} x_L + x_R
7:     Write y as y_L y_R, where x = 10^{n/2} y_L + y_R
8:
9:     a ← MULTIPLY(x_L, y_L)
10:    c ← MULTIPLY(x_R, y_R)
11:    b ← MULTIPLY(x_L + x_R, y_L + y_R) − a − c
12:
13:    return 10^n · a + 10^{n/2} · b + c.
14: end procedure
```

$10^2 \cdot 7 + 10 \cdot 29 + 24 = 1014$

$x = 78, y = 13$

$x_L=7, x_R=8$
$y_L=1, y_R=3$

$x_L \cdot y_L = 7 \cdot 1 = 7$

$x_R \cdot y_R = 8 \cdot 3 = 24$

$(x_L+x_R)\cdot(y_L+y_R) = (7+8)(1+3)=60$

$60 - 7 - 24 = 29$

So how fast is it?

Algorithm 2 Karatsuba Multiplication on Numbers of Length n

```
1: procedure MULTIPLY(x, y)
2:     if n=1 then
3:         return x · y
4:     end if
5:
6:     Write x as x_L x_R, where x = 10^{n/2} x_L + x_R
7:     Write y as y_L y_R, where x = 10^{n/2} y_L + y_R
8:
9:     a ← MULTIPLY(x_L, y_L)
10:    c ← MULTIPLY(x_R, y_R)
11:    b ← MULTIPLY(x_L + x_R, y_L + y_R) − a − c
12:
13:    return 10^n · a + 10^{n/2} · b + c.
14: end procedure
```

Three multiplications, each of size n/2. 6 additions.

So if T(n) is the time it takes to multiply numbers of size n, then...

T(n) = 3T(n/2) + O(n).

This is already better than before! But how much better?

The Master Theorem

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then c = ???, d = ???, so

T(n) = ???

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then c = ???, d = ???, so

T(n) = ???

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then c = ???, d = ???, so

T(n) = ???

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then $c = 2$, $d = 1$, so

$T(n) = ???$

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then $c = ???$, $d = ???$, so

$T(n) = ???$

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then $c = ???$, $d = ???$, so

$T(n) = ???$

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then $c = 2$, $d = 1$, so

$T(n) = O(n^2)$

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then $c =$???, $d =$???, so

$T(n) =$???

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then $c =$???, $d =$???, so

$T(n) =$???

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then $c = 2$, $d = 1$, so

$T(n) = O(n^2)$

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then $c = 1$, $d = 1$, so

$T(n) = ???$

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then $c = ???$, $d = ???$, so

$T(n) = ???$

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then $c = 2$, $d = 1$, so

$T(n) = O(n^2)$

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then $c = 1$, $d = 1$, so

$T(n) = O(n \log n)$

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then c = ???, d = ???, so

T(n) = ???

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then $c = 2$, $d = 1$, so

$T(n) = O(n^2)$

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then $c = 1$, $d = 1$, so

$T(n) = O(n \log n)$

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then $c = \log_2 3 = 1.59$, $d = 1$, so

$T(n) = ???$

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Our original recurrence:
$T(n) = 4T(n/2) + O(n)$
Then $c = 2$, $d = 1$, so
$T(n) = O(n^2)$

Mergesort recurrence:
$T(n) = 2T(n/2) + O(n)$
Then $c = 1$, $d = 1$, so
$T(n) = O(n \log n)$

Karatsuba Multiplication:
$T(n) = 3T(n/2) + O(n)$
Then $c = \log_2 3 = 1.59$, $d = 1$, so
$T(n) = O(n^{1.59})$

The Master Theorem

To determine a runtime from a recurrence:

Say your recurrence is $T(n) = a \cdot T(n/b) + O(n^d)$.

- Say $c = \log_b a$.
- If $d > c$, then $T(n) = O(n^d)$.
- If $d = c$, then $T(n) = O(n^d \log n)$.
- If $d < c$, then $T(n) = O(n^c)$.

Caution: If the recurrence doesn't follow this form, Master Theorem doesn't tell us anything!

Our original recurrence:

$T(n) = 4T(n/2) + O(n)$

Then $c = 2$, $d = 1$, so

$T(n) = O(n^2)$

Mergesort recurrence:

$T(n) = 2T(n/2) + O(n)$

Then $c = 1$, $d = 1$, so

$T(n) = O(n \log n)$

Karatsuba Multiplication:

$T(n) = 3T(n/2) + O(n)$

Then $c = \log_2 3 = 1.59$, $d = 1$, so

$T(n) = O(n^{1.59})$

So how fast is it?

Algorithm 2 Karatsuba Multiplication on Numbers of Length n

```
1:  procedure MULTIPLY(x, y)
2:      if n=1 then
3:          return x · y
4:      end if
5:
6:      Write x as x_L x_R, where x = 10^{n/2} x_L + x_R
7:      Write y as y_L y_R, where x = 10^{n/2} y_L + y_R
8:
9:      a ← MULTIPLY(x_L, y_L)
10:     c ← MULTIPLY(x_R, y_R)
11:     b ← MULTIPLY(x_L + x_R, y_L + y_R) − a − c
12:
13:     return 10^n · a + 10^{n/2} · b + c.
14: end procedure
```

Three multiplications, each of size n/2. 6 additions.

So if T(n) is the time it takes to multiply numbers of size n, then...

$T(n) = 3T(n/2) + O(n)$.

$T(n) = O(n^{1.59})$

A look at the paradigm

Algorithm 2 Karatsuba Multiplication on Numbers of Length n

```
1:  procedure MULTIPLY(x, y)
2:      if n=1 then
3:          return x · y
4:      end if
5:
6:      Write x as x_L x_R, where x = 10^{n/2} x_L + x_R
7:      Write y as y_L y_R, where x = 10^{n/2} y_L + y_R
8:
9:      a ← MULTIPLY(x_L, y_L)
10:     c ← MULTIPLY(x_R, y_R)
11:     b ← MULTIPLY(x_L + x_R, y_L + y_R) − a − c
12:
13:     return 10^n · a + 10^{n/2} · b + c.
14: end procedure
```

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

So what happened with multiplication?

What happens next?

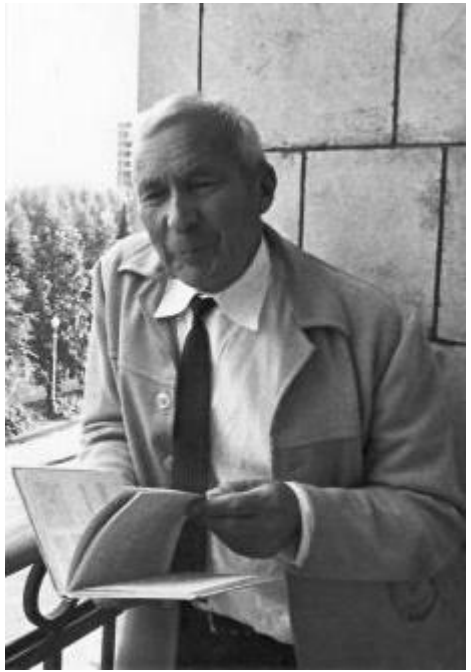
In 1960, Karatsuba, still a student, shows that you can do multiplication in $\mathbf{O(n^{1.59})}$ time.

Kolmogorov is shocked. He published the result under Karatsuba's name.

УМНОЖЕНИЕ МНОГОЗНАЧНЫХ ЧИСЕЛ НА АВТОМАТАХ

(Представлено академиком А. Н. Колмогоровым 13 II 1962)

Translation: Multiplication of many-digital numbers by automatic computers



So… how fast can we get?

Name(s)	Year	Runtime	How?
[everyone]	<1960	$O(n^2)$	All human ways
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT

Some practical details.

Name(s)	Year	Runtime	How?
[everyone]	<1960	$O(n^2)$	All human ways
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT

Which one do you think we use in practice?

Some practical details.

Name(s)	Year	Runtime	How?
[everyone]	<1960	$O(n^2)$	All human ways
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT

Which one do you think we use in practice?

Some practical details.

Name(s)	Year	Runtime	How?
[everyone]	<1960	$O(n^2)$	All human ways
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT

What is the minimum exponent here?

Some practical details.

Name(s)	Year	Runtime	How?
[everyone]	<1960	$O(n^2)$	All human ways
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT

Unfortunately, recombining also scales with k^2. By k=4, you're better off running Schönhage–Strassen.

Some practical details.

Name(s)	Year	Runtime	How?	Good n
[everyone]	<1960	$O(n^2)$	All human ways	n < 30ish
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer	n < 1000s
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2	
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)	n ~ 1000s
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT	
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT	

Some practical details.

Name(s)	Year	Runtime	How?	Good n
[everyone]	<1960	$O(n^2)$	All human ways	n < 30ish
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer	n < 1000s
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2	
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)	n ~ 1000s
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT	
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT	1729^{12}

Some practical details.

Name(s)	Year	Runtime	How?	Good n
[everyone]	<1960	$O(n^2)$	All human ways	n < 30ish
Karatsuba	1960	$O(n^{1.59})$	Divide and Conquer	n < 1000s
Toom and Cook	1963	$O(n^{(\log 2k-1)/(\log k)})$	Split into k pieces instead of 2	
Schönhage–Strassen	1971	$O(n \cdot \log n \cdot \log \log n)$	Fast Fourier Transform (FFT)	n ~ 1000s
Fürer	2007	$O(n \cdot \log n \cdot 2^{\log^* n})$	Recursive FFT	
Harvey and van der Hoeven	2019	$O(n \cdot \log n)$	Multidimensional FFT	**2 to the power of 1729^{12}**

Then for all $n \geqslant n_0 = 2^{1729^{12}}$

This is a *galactic algorithm* – asymptotically great, practically useless.

Summary

Today we saw one of the most fundamental algorithms in computing, Karatsuba multiplication, which is also the first divide-and-conquer algorithm in recorded history.

We also learned about the Master Theorem, a convenient and useful way to determine the runtime of a divide-and-conquer algorithm from its recurrence.

Finally, through different multiplication algorithms, we saw why asymptotic analysis is a good but imperfect measurement of efficiency.