# Fast Long Integer Multiplication in an Pre-FFT Era

Lars Quentin

University of Göttingen

November 18, 2021

# “Be fruitful and multiply”

Genesis 1:28

# Overview

# Where is Karatsuba's Algorithm used nowadays?

- CPython uses Karatsuba's multiplication
  - Also PyPy3
- The Glassgow Haskell Compiler uses Karatsuba/Toom-3 for Multiplication.
- The GMP Library is used in all of HPC.
  - It automatically uses Karatsuba/Toom Cook for medium sized numbers.
  - See: "High-Precision Arithmetic in Mathematical Physics" for applications.

# Integer Represenation 1

- Processors are computating data in chunks of a certain size, so called *words*.
  - Since the word size is constant, all elementary operations can be (and are) implemented in $O(1)$
- In modern CISC architecture the default word size is 64 bits
  - i.e for Integers: $\{0, \ldots, 2^{64} - 1\}$
- Often, this is not enough. Thus we now define *multiprecision integers*:

# Integer Represenation 2

## Definition 1: Multiprecision Integer

The *multiprecision integer* $a \in \mathbb{N}$ is represented as a vector of *words* $a_i$ such that

$$a = \sum_{0 \leq i \leq n} a_i \cdot 2^{64i}$$

where $n \in \mathbb{N}$, $a_i \in \{0, \ldots, 2^{64} - 1\}$ for all digits $i$.

# Some more formality...

## Definition 2: Ring

A *ring* $(R, +, \cdot)$ is a set $R$ with 2 binary operations $+$ and $\cdot$ which satify the following axioms.

1. $(R, +)$ is an abelian group under addition, meaning that
   - $+$ is associative
   - $+$ is commutative
   - There is a neutral element $0 \in R$
   - Every element has an additive inverse
2. $(R, \cdot)$ is a semigroup under multiplication, meaning that $\cdot$ is associative
3. Multiplication is distributiove with respect to addition, i.e. $\forall a, b, c \in R$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

## Definition 3: Commutative ring with 1

An *commutative ring with* 1 $(R, +, \cdot)$ is a ring that satisfies the following:

1. $(R, \cdot)$ is not only a semigroup but also a monoid, meaning that:
   - $(R, \cdot)$ is associative (i.e. a semigroup)
   - There exists a neutral element $1 \in R$ over multiplication (the *multiplicative identity*)
2. $(R, \cdot)$ is commutative

A ring with 1 is also called a *unitary ring*.

## Definition 3: Commutative ring with 1

An *commutative ring with* 1 $(R, +, \cdot)$ is a ring that satisfies the following:

1. $(R, \cdot)$ is not only a semigroup but also a monoid, meaning that:
   - $(R, \cdot)$ is associative (i.e. a semigroup)
   - There exists a neutral element $1 \in R$ over multiplication (the *multiplicative identity*)
2. $(R, \cdot)$ is commutative

A ring with 1 is also called a *unitary ring*.

## Note:

From now on, every ring is a commutative ring with 1.

# We are almost ready I swear

## Definition 4: Discrete Convolution

Let $D \subseteq \mathbb{Z}, f, g : D \to \mathbb{C}$.
A *discrete convolution* of $f$ and $g$ is defined as

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(n-m)g(m)$$

# We are almost ready I swear

## Definition 4: Discrete Convolution

Let $D \subseteq \mathbb{Z}, f, g : D \to \mathbb{C}$.
A *discrete convolution* of $f$ and $g$ is defined as

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(n-m)g(m)$$

When $g$ has finite support over $\{-M, -M+1, \ldots, M\}$, then it can be simplified to

$$(f * g)(n) = \sum_{m=-M}^{M} f(n-m)g(m)$$

# Definition 5: Polynomial Ring

A *polynomial ring* $R[X]$ is a commutative ring with 1 $(R^{(\mathbb{N}_0)}, +, \cdot)$ defined as

- $R^{(\mathbb{N}_0)}$ is the set of sequences

$$R^{(\mathbb{N}_0)} := \{(a_i)_{i \in \mathbb{N}_0} : a_i \in R, a_i = 0 \text{ for almost all } i\}$$

- $+$ is defined as the componentwise addition, meaning that

$$(a_i)_{i \in \mathbb{N}_0} + (b_i)_{i \in \mathbb{N}_0} := (a_i + b_i)_{i \in \mathbb{N}_0}$$

- $\cdot$ is defined as the discrete convolution, meaning that

$$(a_i)_{i \in \mathbb{N}_0} \cdot (b_i)_{i \in \mathbb{N}_0} := \left( \sum_{0 \leq i \leq k} a_i b_{k-i} \right)_{k \in \mathbb{N}_0} = \left( \sum_{i+j=k} a_i b_j \right)_{k \in \mathbb{N}_0}$$

# Cont.

- Let $X$ be defined such that the following holds:
  - $X \in R^{(\mathbb{N}_0)}$ is defined as
$$X = X^1 := (0, 1, 0, \dots)$$
  - $1 \in R^{(\mathbb{N}_0)}$ is defined as
$$1 := X^0 = (1, 0, 0, \dots)$$
  - Every power $X^k \in R^{(\mathbb{N}_0)}, k \in \mathbb{N}_0$ is defined as
$$X^k := \underbrace{X \cdot X \cdot \dots \cdot X}_{k \text{ times}} = (\underbrace{0, \dots, 0}_{k \text{ zeros}}, 1, 0, 0, \dots)$$
  - With $X$ defined we can write any polynomial $f \in R[X]$ as
$$f = \sum_{i=0}^{n} a_i X_i$$

Let $R[X]$ be a polynomial ring and $p, q \in R[X]$ be two polynomials.

## Note:

Since we have finite sequences, the discrete convolution is also called the *Cauchy Product*.

Let $R[X]$ be a polynomial ring and $p, q \in R[X]$ be two polynomials.

## Note:

Since we have finite sequences, the discrete convolution is also called the *Cauchy Product*.

## Note:

When written in polynomial form, the cauchy product is equivalent to the intuitive multiplication of two polynomials, i.e.

$$p \cdot q = s_0 + s_1 \cdot X + \cdots + s_{\deg(p)+\deg(q)} \cdot X^{\deg(p)+\deg(q)}$$

where $s_i = p_0 q_i + p_1 q_{i-1} + \cdots + p_i q_0$

# Why do we want to use polynomials?

- More generic: If we define $X = 10$, we have normal numbers. $X = 2$ for binary.

# Why do we want to use polynomials?

- More generic: If we define $X = 10$, we have normal numbers. $X = 2$ for binary.
- It makes the time complexity analysis easier: We do not need to worry about carry-overs.

# Why do we want to use polynomials?

- More generic: If we define $X = 10$, we have normal numbers. $X = 2$ for binary.
- It makes the time complexity analysis easier: We do not need to worry about carry-overs.
- We can analyze real algorithm runtime: Set $X = 2^{64}$.

# Naive Multiplication: An Example

$$
\begin{array}{ccccc}
 & & ax & + & b \\
 & & cx & + & d \\
\hline
 & & adx & & bd \\
 & & bcx & & \\
acx^2 & & & & \\
\hline
acx^2 & + & (ad+bc)x & + & bd
\end{array}
$$

# Naive Multiplication: An Example

$$
\begin{array}{ccccc}
 & & ax & + & b \\
 & & cx & + & d \\
\hline
 & & adx & & bd \\
 & & bcx & & \\
acx^2 & & & & \\
\hline
acx^2 & + & (ad + bc)x & + & bd
\end{array}
$$

- Time Complexity: $n^2$ Multiplications, $\Theta(n)$ Additions $\Rightarrow \Theta(n^2)$

# Algorithm 1: Naive Multiplication of two polynomials

Let $R[x]$ be a ring.

*Input*: The coefficients of $a = \sum_{0 \leq i \leq n} a_i x^i$ and $b = \sum_{0 \leq i \leq n} b_i x^i$ with $a, b \in R[x]$
*Output*: The coefficients of $c = a \cdot b \in R[x]$

1. **for** $i = 0, \ldots, n$ **do** $d_i = a_i x^i \cdot b$
2. **return** $c = \sum_{0 \leq i \leq n} d_i$

- Time Complexity: $\Theta(n^2)$

# Karatsuba's Idea

$$
\begin{array}{ccccc}
 & & ax & + & b \\
 & & cx & + & d \\
\hline
 & & adx & & bd \\
 & & bcx & & \\
acx^2 & & & & \\
\hline
acx^2 & + & (ad+bc)x & + & bd
\end{array}
$$

# Karatsuba's Idea 2

The formula:

$$(ax + b) \cdot (cx + d) = acx^2 + (ad + bc)x + bd$$

# Karatsuba's Idea 2

The formula:

$$(ax + b) \cdot (cx + d) = acx^2 + (ad + bc)x + bd$$

- We obviously need 4 multiplications.
  - $ac$ and $bd$ for the upper and lower digit
  - $ad$ and $bc$ for the middle digit

# Karatsuba's Idea 2

The formula:

$$(ax + b) \cdot (cx + d) = acx^2 + (ad + bc)x + bd$$

- We obviously need 4 multiplications.
  - $ac$ and $bd$ for the upper and lower digit
  - $ad$ and $bc$ for the middle digit
- Karatsuba's idea was to first add the digits up, then multiply them.

$$(a + b) \cdot (c + d) = ac + ad + bc + bd$$
$$\Leftrightarrow ad + bc = (a + b) \cdot (c + d) - ac - bd$$

# Karatsuba's Idea 2

The formula:

$$(ax + b) \cdot (cx + d) = acx^2 + (ad + bc)x + bd$$

- We obviously need 4 multiplications.
  - $ac$ and $bd$ for the upper and lower digit
  - $ad$ and $bc$ for the middle digit
- Karatsuba's idea was to first add the digits up, then multiply them.

$$\begin{aligned}(a + b) \cdot (c + d) &= ac + ad + bc + bd \\ \Leftrightarrow ad + bc &= (a + b) \cdot (c + d) - ac - bd\end{aligned}$$

- We already have $ac$ and $bd$.
  Thus only 3 Multiplications!

# Karatsuba's Multiplication: An Example

$$
\begin{array}{ccccc}
 & & ax & + & b \\
 & & cx & + & d \\
\hline
acx^2 & & & & bd \\
 & & (a+b)(c+d)x & & \\
\hline
acx^2 & + & ((a+b)(c+d) - ac - bd)x & + & bd
\end{array}
$$

# But wait....

- How does this generalize?

# But wait....

- How does this generalize?
- Let $R[x]$ be a polynomial ring and $p, q \in R$ polynomials of degree $n-1$.
  We can now divide them into

  $$p(x) = p_1 x^{n/2} + p_0$$
  $$q(x) = q_1 x^{x/2} + q_0$$

  and apply Karatsuba's theorem.

# But wait....

- How does this generalize?
- Let $R[x]$ be a polynomial ring and $p, q \in R$ polynomials of degree $n-1$.
  We can now divide them into

$$p(x) = p_1 x^{n/2} + p_0$$
$$q(x) = q_1 x^{x/2} + q_0$$

  and apply Karatsuba's theorem.
- But this leaves us with three $\Theta(n/2)$ Multiplications...

# But wait....

- How does this generalize?
- Let $R[x]$ be a polynomial ring and $p, q \in R$ polynomials of degree $n-1$.
  We can now divide them into

  $$p(x) = p_1 x^{n/2} + p_0$$
  $$q(x) = q_1 x^{x/2} + q_0$$

  and apply Karatsuba's theorem.
- But this leaves us with three $\Theta(n/2)$ Multiplications...
  - We can use recursion!

## Algorithm 2: Karatsuba's Multiplication

Let $R[x]$ be a ring.

*Input*: $f, g \in R[x]$ of degree of $n$, where n is a power of 2.
*Output*: $f \cdot g \in R[x]$

1. **if** $n = 1$ **then return** $f \cdot g \in R$ (base case)
2. let $f = f_1 x^{n/2} + f_0$ and $g = g_1 x^{n/2} + G_0$ where $f_0, f_1, g_0, g_1 \in R[x]$ with degree $< n/2$
3. compute $f_0 g_0$, $f_1, g_1$ and $(f_0 + f_1)(g_0 + g_1)$ recursively
4. **return** $f_1 g_1 x^n + ((f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1) x^{n/2} + f_0 g_0$

# First Analysis of the time complexity

- Let $T(n)$ denote the time complexity of *Algorithm 2* with input size $n$
- We can see that
  - We split the problem into 2 problems of half size.
  - We do 3 smaller multiplications
  - Then we add it up in some linear time.
- We can write it as a recursion. Any ideas?

# First Analysis of the time complexity

- Let $T(n)$ denote the time complexity of *Algorithm 2* with input size $n$
- We can see that
  - We split the problem into 2 problems of half size.
  - We do 3 smaller multiplications
  - Then we add it up in some linear time.
- We can write it as a recursion. Any ideas?

$$T(n) = 3T(n/2) + \Theta(n)$$

# Let's look at some trees

- In "Algorithms on Sequences", I learned that it is sometimes easier to solve a harder problem first.

# Let's look at some trees

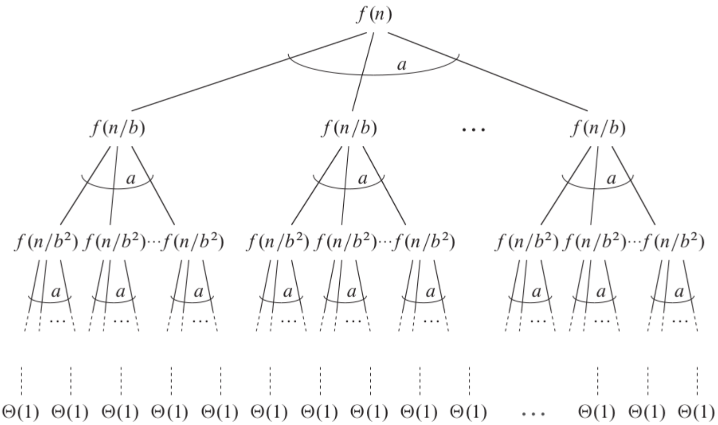
- In "Algorithms on Sequences", I learned that it is sometimes easier to solve a harder problem first.
- Let's generalize our recursion:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(n/b) + f(x) & \text{else} \end{cases}$$

where

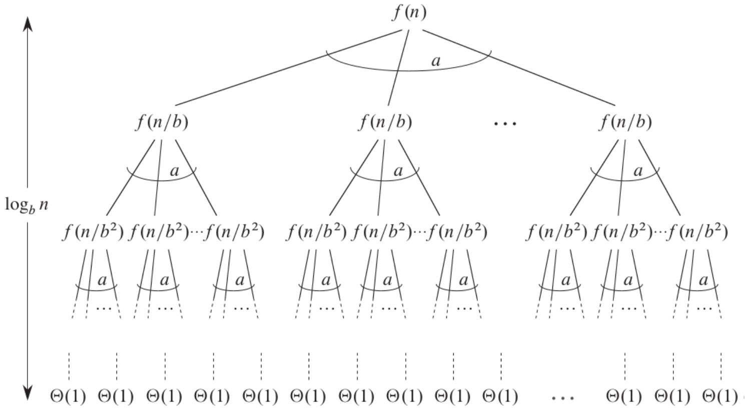
- $a$ is our branching factor
- $n/b$ is the new input size
- $f(x)$ is some work done in each call
- The base case is a subproblem of size 1.

# Trees!!!



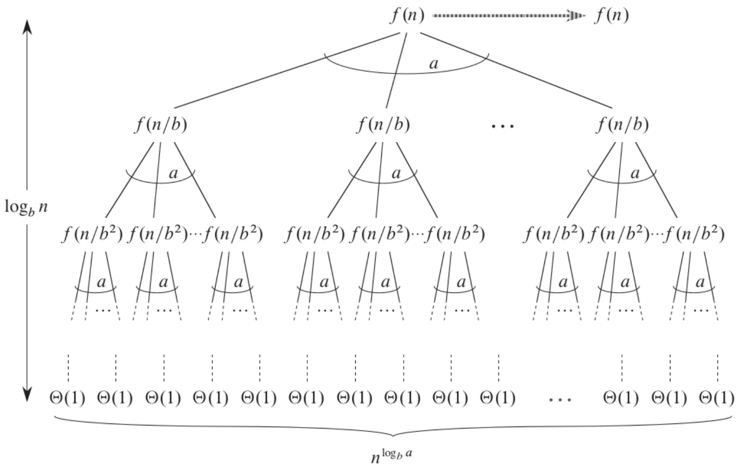

Question: What is the height of the tree?

# Now a bit harder




Question: Any idea how many leafes the tree could have?

# Let's sum it up depthwise.




Question: Depth 0 has cost $f(n)$. What about the other depths?

# I think we can divide it into 3 cases.

So, the total cost is

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$$

# I think we can divide it into 3 cases.

So, the total cost is

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$$

This can be divided into 3 cases:

1. The cost is dominated by the cost in the leafes, i.e. $O(n^{\log_b(a)}) > \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$

# I think we can divide it into 3 cases.

So, the total cost is

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$$

This can be divided into 3 cases:

1. The cost is dominated by the cost in the leafes, i.e. $O(n^{\log_b(a)}) > \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$
2. The cost is evenly distributed, i.e. $O(n^{\log_b(a)}) \ni \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$

# I think we can divide it into 3 cases.

So, the total cost is

$$T(n) = \Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$$

This can be divided into 3 cases:

1. The cost is dominated by the cost in the leafes, i.e. $O(n^{\log_b(a)}) > \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$
2. The cost is evenly distributed, i.e. $O(n^{\log_b(a)}) \ni \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$
3. The cost is dominated by the cost in the root i.e. $O(n^{\log_b(a)}) < \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$

I have see that before...

## Theorem 1: The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n) : \mathbb{N}_0 \to \mathbb{N}_0$

$$T(n) = aT(n/b) + f(n)$$

for powers of 2. Then

1. If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$
2. If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n) = \Theta(n^{\log_b(a)} \log(n))$
3. If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$

# Theorem 2: Runtime Karatsuba's Multiplication

The runtime of Karatsuba's Algorithm is $\Theta(n^{\log_2(3)}) \approx \Theta(n^{1.5849})$

## Theorem 2: Runtime Karatsuba's Multiplication

The runtime of Karatsuba's Algorithm is $\Theta(n^{\log_2(3)}) \approx \Theta(n^{1.5849})$

## Proof.

- We did 3 Multiplication with a problem of size $n/2$ and did linear work adding it up

$$T(n) = 3T(n/2) + \Theta(n)$$

# Theorem 2: Runtime Karatsuba's Multiplication

The runtime of Karatsuba's Algorithm is $\Theta(n^{\log_2(3)}) \approx \Theta(n^{1.5849})$

## Proof.

- We did 3 Multiplication with a problem of size $n/2$ and did linear work adding it up

$$T(n) = 3T(n/2) + \Theta(n)$$

- We know that $\Theta(n) \in O(n^{\log_2(3)-\epsilon})$ for some $\epsilon > 0$

## Theorem 2: Runtime Karatsuba's Multiplication

The runtime of Karatsuba's Algorithm is $\Theta(n^{\log_2(3)}) \approx \Theta(n^{1.5849})$

## Proof.

- We did 3 Multiplication with a problem of size $n/2$ and did linear work adding it up

$$T(n) = 3T(n/2) + \Theta(n)$$

- We know that $\Theta(n) \in O(n^{\log_2(3)-\epsilon})$ for some $\epsilon > 0$
- Proven via master theorem, case 1.

$\square$

# The Toom-Cook Algorithm

- The Toom-Cook Algorithm needs is kind of a weird algorithm

# The Toom-Cook Algorithm

- The Toom-Cook Algorithm needs is kind of a weird algorithm
  - It has a way bigger constant time

# The Toom-Cook Algorithm

- The Toom-Cook Algorithm needs is kind of a weird algorithm
  - It has a way bigger constant time
  - Schönhage-Straßen is near linear with not that much more overhead

# The Toom-Cook Algorithm

- The Toom-Cook Algorithm needs is kind of a weird algorithm
  - It has a way bigger constant time
  - Schönhage-Straßen is near linear with not that much more overhead
  - It requires way more rigour to formalize;
    it needs to be formalized on a case-by-case basis
- I'll just sketch it out and show it's time complexity

# The general idea

- We can split up into more than 2 parts!

# The general idea

- We can split up into more than 2 parts!
- The most used case is Toom-3, i.e. splitting up into 3 parts.
  The recursion would look like

$$T(n) = aT(n/3) + \Theta(n)$$

# The general idea

- We can split up into more than 2 parts!
- The most used case is Toom-3, i.e. splitting up into 3 parts.
  The recursion would look like

$$T(n) = aT(n/3) + \Theta(n)$$

- If we get $a < 9$, we get $\Theta(n^{\log_3(a)}) < \Theta(n^2)$, thus beating normal multiplication.

# The general idea

- We can split up into more than 2 parts!
- The most used case is Toom-3, i.e. splitting up into 3 parts.
  The recursion would look like

$$T(n) = aT(n/3) + \Theta(n)$$

- If we get $a < 9$, we get $\Theta(n^{\log_3(a)}) < \Theta(n^2)$, thus beating normal multiplication.
- Let's see how they try to beat it.

# The steps

Toom-3 is divided in 5 steps:

# The steps

Toom-3 is divided in 5 steps:

1. Splitting the polynomials into 3 even parts, i.e.

$$p = p' := p'_2 x^2 + p'_1 x + p'_0$$

# The steps

Toom-3 is divided in 5 steps:

1. Splitting the polynomials into 3 even parts, i.e.

$$p = p' := p'_2 x^2 + p'_1 x + p'_0$$

2. Choose $\deg pq + 1 = \deg p + \deg q + 1$ points to interpolate it later

# The steps

Toom-3 is divided in 5 steps:

1. Splitting the polynomials into 3 even parts, i.e.

$$p = p' := p'_2 x^2 + p'_1 x + p'_0$$

2. Choose $\deg pq + 1 = \deg p + \deg q + 1$ points to interpolate it later
3. Evaluate these 5 big multiplications for $pq$ recursively

# The steps

Toom-3 is divided in 5 steps:

1. Splitting the polynomials into 3 even parts, i.e.

$$p = p' := p'_2 x^2 + p'_1 x + p'_0$$

2. Choose $\deg pq + 1 = \deg p + \deg q + 1$ points to interpolate it later
3. Evaluate these 5 big multiplications for $pq$ recursively
4. Interpolate with the 5 evaluated points

# The steps

Toom-3 is divided in 5 steps:

1. Splitting the polynomials into 3 even parts, i.e.

$$p = p' := p'_2 x^2 + p'_1 x + p'_0$$

2. Choose $\deg pq + 1 = \deg p + \deg q + 1$ points to interpolate it later
3. Evaluate these 5 big multiplications for $pq$ recursively
4. Interpolate with the 5 evaluated points
5. Add up the result

# The steps

Toom-3 is divided in 5 steps:

1. Splitting the polynomials into 3 even parts, i.e.

$$p = p' := p'_2 x^2 + p'_1 x + p'_0$$

2. Choose $\deg pq + 1 = \deg p + \deg q + 1$ points to interpolate it later
3. Evaluate these 5 big multiplications for $pq$ recursively
4. Interpolate with the 5 evaluated points
5. Add up the result

When done in a smart way, we just need 5 multipliations.

## Theorem 3: Time Complexity Toom-3

The time complexity of Toom-3 is $\Theta(n^{\log_3(5)}) \approx \Theta(n^{1.46})$

## Theorem 3: Time Complexity Toom-3

The time complexity of Toom-3 is $\Theta(n^{\log_3(5)}) \approx \Theta(n^{1.46})$

## Proof.

We need 5 multiplications and divide the problem into 3 even parts.
All other work in linear.
Thus

$$T(n) = 5T(n/5) + \Theta(n)$$

It follows from the master theorem, case 1. □

# What happened after Toom-Cook

# What happened after Toom-Cook

- 1965: The FFT became popular

# What happened after Toom-Cook

- 1965: The FFT became popular
- 1971: Schönhage Straßen:
  - $O(N \log(N) \log(\log(N)))$

# What happened after Toom-Cook

- 1965: The FFT became popular
- 1971: Schönhage Straßen:
  - $O(N \log(N) \log(\log(N)))$
- 2007: Martin Fürer:
  - $O(N \cdot \log(N) \cdot K^{\log^*(N)})$

# What happened after Toom-Cook

- 1965: The FFT became popular
- 1971: Schönhage Straßen:
  - $O(N \log(N) \log(\log(N)))$
- 2007: Martin Fürer:
  - $O(N \cdot \log(N) \cdot K^{\log^*(N)})$
- 2019: Harvey & van der Hoeven:
  - $O(N \log(N))$
  - To be worth it: $n$ has to have $2^{1729^{12}}$ digits long.

# Sources: Books

Books

- Modern Computer Algebra
- Introduction to Algorithms
- The Art of Computer Programming: Volume 2

Videos

- MIT OCW: Karatsuba, Master Method
- Neman: How Karatsuba's algorithm gave us new ways to multiply

Other:

- The GMP manual: 15.1.3 Toom 3-Way Multiplication


THE END
disneyscreencaps.com