CS 487 / · · ·

Introduction to Symbolic Computation

University of Waterloo
Éric Schost
eschost@uwaterloo.ca
Staff

Instructor

• Éric Schost, DC 3627, eschost@uwaterloo.ca

TA

• Zeming Liu, DC 2551C, z564liu@uwaterloo.ca

Lectures

• Tuesday and Thursday, 8:30 - 9:50 in MC4063

Office hours

• Tuesday, 10:30 - 12:30
Assignments, exams, project, etc

- **4 assignments** (35% undergrad / 30% grad)
  - submission by email
  - due on Friday at 5pm

- **Midterm** (25% / 20%)
  - March 1st in class

- **Final** (40% / 30%)
  - TBD

- **Project** (+5% / 20%)
  - mandatory for grad students
  - optional for undergrads, +5 marks available
  - reading papers and writing a report, coding may be involved but not required
Electronic communication

Piazza

• sign up using your uwaterloo email address
• https://piazza.com/uwaterloo.ca/winter2018/cs487
• posting solutions to assignments is forbidden

email

• course account cs487@student.cs.uwaterloo.ca
• use your uwaterloo address
• CS487 / · · · in the subject line
This course

What you should know

• CS240-level algorithms
• big-O notation, master theorem
• a few things about matrices

What we will do

• a lot of algorithms
• a bit of math (algebra – rings, fields, …)

What we will not do

• difficult math (analysis, SVD / QR, probabilities, …)
• spend a lot of time learning Maple, Julia, python, …
Roughly, studies how to solve mathematical problems on a computer, with an emphasis on “exact solutions”.

\[
\text{solve}(2x + 1 = 0) \implies x = -\frac{1}{2}, \quad \text{not} \quad x = -0.499999999999.
\]

Many aspects

- programming languages for expressing mathematical notions;
- algorithms and complexity;
- mixing symbolics and numerics
- ...

This course: emphasis on algorithms and complexity.
Basic problem: dealing with numbers properly.

- **exactness** means that we handle multi-precision (arbitrary length) numbers
- “efficient” algorithm = polynomial in the size of the input / output

A handful of algorithms

- addition easy
  
- multiplication hard, but satisfactory answers
  
- division well-understood
  
- factorization ultra-hard

became especially hot after the discovery of the RSA scheme.
A large fraction of the world’s computers are busy solving **linear systems** (or mining Bitcoins)

\[
\begin{align*}
  x_1 + x_2 - 3x_3 &= 3 \\
  -x_1 + 3x_2 - x_3 &= 0 \\
  10x_1 + 3x_2 - x_3 &= 5
\end{align*}
\]

- google
- simplex for linear programming
- numerical simulations of differential equations
In many cases, **floating-point** computations are used. **Exact solutions** are still useful:

- when exact answers are wanted, mathematicians sometimes expect exact solutions
- handling degenerate problems, \texttt{NAN} or slowdown with ill-posed problems
- in contexts that are not numerical, \texttt{crypto}: RSA, ECC
- as sub-routines of higher-level algorithms. like polynomial system solving

Fortunately for us, solving systems in an exact manner, we mostly forget about numerical instability.
Matrix multiplication timeline

- naive algorithm
  \(n^3\) mults
- Pan - Winograd
  1966-68
  \(n^3/2\) mults
- Strassen
  1969, divide-and-conquer
  \(O(n^{\log_2(7)}) \approx O(n^{2.81})\) ops
- Bini et al.
  1980, approximate algorithms
  \(O(n^{\log_{12}(1000)+\varepsilon}) \approx O(n^{2.78})\) ops
- Schönhage
  1981, \(\tau\)-theorem
  \(O(n^{\log_{10}(140608)+\varepsilon}) \approx O(n^{2.52})\) ops
- Coppersmith-Winograd
  1990, sets of integers without arithmetic progressions
  \(O(n^{2.37})\) ops
Polynomial equations

This is where properly understanding the output you expect becomes important.

System:

\[ F_1 = -3x_2^2 - 3x_2 + x_1^2 - 1, \quad F_2 = -x_2^2 + x_1^2. \]

Solutions:

\((-1, -1), \quad (1, -1), \quad (-1/2, -1/2), \quad (1/2, -1/2).\)
Polynomial equations

This is where properly understanding the **output you expect** becomes important.

**System:**

\[
F_1 = -3x_2^2 - 3x_2 + x_1^2 - 1, \quad F_2 = -x_2^2 + x_1^2.
\]

**Solutions:**

\((-1, -1), \quad (1, -1), \quad (-1/2, -1/2), \quad (1/2, -1/2).\)

**System:**

\[
F_1 = -3x_2^2 - 3x_2 + x_1^2 - 1, \quad F_2 = -x_2^2 + x_1^2 + 1.
\]

**Solutions:**

\[
x_1^4 + \frac{7}{4}x_1^2 + \frac{7}{4} = 0, \quad x_2 = -\frac{2}{3}x_1^2 - \frac{4}{3}.
\]

The **second case** is typical.
A brief timeline

• Consistency is decidable (but doubly exponential)
  Hermann, 1926

• Practical algorithms to solve polynomial systems
  Buchberger, 1965

• Nowadays:
Problem: find the next term.

\[ U : 1, 1, 1, 1, 1, 1, 1 \]
\[ V : 0, 1, 1, 2, 3, 5, 8 \]
\[ W : 12, 134, 222, 21, -3898, 40039, -347154, -2929918 \]
Problem: find the next term.

\( U : 1, 1, 1, 1, 1, 1, 1 \)

\( V : 0, 1, 1, 2, 3, 5, 8 \)

\( W : 12, 134, 222, 21, -3898, 40039, -347154, -2929918 \)

Answer: 1, 13 and \(-24657854\).
Computing with sequences

Problem: find the next term.

\[ U : 1, 1, 1, 1, 1, 1, 1 \]

\[ V : 0, 1, 1, 2, 3, 5, 8 \]

\[ W : 12, 134, 222, 21, -3898, 40039, -347154, -2929918 \]

Answer: 1, 13 and \(-24657854\).

How? The sequences \( U, V, W \) satisfy linear recurrences with constant coefficients:

\[ U_{n+1} = U_n, \]

\[ V_{n+2} = V_{n+1} + V_n, \]

\[ W_{n+4} = 12W_{n+3} - 33W_{n+2} + 22W_{n+1} + 19W_n. \]

Euclid’s algorithm provides a way to find the recurrence.
1978: Apéry proves that $\sum_{n \geq 1} \frac{1}{n^3}$ is irrational.

To convince ourselves of the validity of Apéry’s method we need only complete the following exercise. Let

$$b_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2$$

$$c_{n,k} = \sum_{m=1}^{n} \frac{1}{m^3} + \sum_{m=1}^{k} \frac{(-1)^{m-1}}{2m^3 \binom{n}{m} \binom{n+m}{m}}$$

$$a_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2 c_{n,k}.$$
1978: Apéry proves that $\sum_{n\geq 1} \frac{1}{n^3}$ is irrational.

Then each sequence $a_n$ and $b_n$ satisfies the recurrence

$$n^3 u_n - (34n^3 - 51n^2 + 27n - 5)u_{n-1} + (n - 1)^3 u_{n-2} = 0.$$ 

Neither Cohen or I (van der Poorten) had been able to prove this in two months.
A brief history

- AI at MIT: sums and integrals  
  Minski, Moses, . . . , 1960’s

- a small project at UWaterloo  
  Maple, 1980

- So let’s look forward to that near future, where all the “real” math would be done by computers
This course

• Basic objects
  polynomials, matrices

• Basic techniques
  divide-and-conquer, Newton iteration, Euclid’s algorithm

• Vague goal of the course: understanding some applications to coding theory (Reed Solomon codes – CD, DVD, QR, …)
  finite fields, algorithms on polynomials
Polynomial (and integer) multiplication
Problem statement

Input

• two polynomials

\[ F = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1} \quad G = g_0 + g_1 x + \cdots + g_{n-1} x^{n-1} \]

Output

• the product

\[ H = FG = h_0 + \cdots + h_{2n-2} x^{2n-2} \]

with

\[ h_0 = f_0 g_0 \quad \cdots \quad h_i = \sum_{j+k=i} f_j g_k \quad \cdots \quad h_{2n-2} = f_{n-1} g_{n-1} \]
Motivation

Multiplication is a **central problem**.

Algorithms for

- gcd
- factorization
- root-finding
- evaluation, interpolation
- Chinese remaindering
- linear algebra (a little bit)
- polynomial system solving (a little bit)

rely on polynomial multiplication, and their complexity can be expressed using that of multiplication.
Prop. One can multiply polynomials with \( n \) terms using \ldots

- the naive algorithm
  \( O(n^2) \) operations.

- Karatsuba’s algorithm
  \( O(n^{1.59}) \) operations

- Toom’s algorithm(s)
  \( O(n^{1.47}) \), \ldots \ operations

- Fast Fourier Transform
  \( O(n \log(n)) \) operations
  \( O(n \log(n) \log(\log(n))) \) operations

It’s still unknown with the optimal is.
Practical aspects: don’t neglect . . .

- the constants in the $O(\ldots)$: usually better for the simpler (slower) algorithms
- lower-level aspects (data representation, memory access, . . .)

In the best current implementations

- Karatsuba beats the naive algorithm for degrees about 32.
- FFT wins for degrees about 128.

Some problems (crypto, number theory) require to handle polynomials of degree about 1000000.
Polynomials and integers

**Polynomials.** You want to multiply $3x^2 + 2x + 1$ and $6x^2 + 5x + 4$.

$$(3x^2 + 2x + 1) \times (6x^2 + 5x + 4)$$

$$= (3 \cdot 6)x^4 + (3 \cdot 5 + 2 \cdot 6)x^3 + (3 \cdot 4 + 2 \cdot 5 + 1 \cdot 6)x^2 + (2 \cdot 4 + 1 \cdot 5)x + (1 \cdot 4)$$

$$= 18x^4 + 27x^3 + 28x^2 + 13x + 4.$$  

**Integers.** You want to multiply 321 and 654 (base 10).

$$(3 \cdot 10^2 + 2 \cdot 10 + 1) \times (6 \cdot 10^2 + 5 \cdot 10 + 4)$$

$$= 18 \cdot 10^4 + 27 \cdot 10^3 + 28 \cdot 10^2 + 13 \cdot 10 + 4$$

$$= 2 \cdot 10^5 + 9 \cdot 10^3 + 9 \cdot 10^2 + 3 \cdot 10 + 4 = 209934.$$  

**Conclusion:** similarities, but carries make the integer case seemingly harder.
The algorithms work almost the same, but are more complicated.

**Prop.** One can multiply integer with $n$ bits using …

- the naive algorithm
  $O(n^2)$ bit operations.

- Karatsuba’s algorithm
  $O(n^{1.59})$ bit operations
  $1.59 = \log_2(3)$

- Toom’s algorithm(s)
  $O(n^{1.47})$ bit operations
  $1.47 = \log_3(5)$

- Fast Fourier Transform
  $O(n \log(n)2^{O(\log^*(n))})$ bit operations $\log^* = \text{nbr of logs to reach 1}$

It’s still unknown with the optimal is.
In practice

Some people work very hard on integer multiplication

**GMP (GNU Multiple Precision)**

- started in 1991, written in C, 15+ MB source code
- store integers as arrays of **unsigned long** + some size info
- a lot of assembly

**Why?**

- integers → rationals, long floats, …
- compiling gcc:

```
./contrib/download_prerequisites
  gmp='gmp-6.1.0.tar.bz2'
  mpfr='mpfr-3.1.4.tar.bz2'
```
In practice

Some people work very hard on integer multiplication

from gmplib.org
What are our coefficients

Most algorithms for polynomials, matrices, ... are insensitive to the nature of the coefficients:

- integers,
- rational numbers,
- complex numbers,
- others.

All that is needed is that . . .

- you can add coefficients,
- and multiply them,
- and sometimes divide them
- with some obvious good-behaviour rules.
Rings

A ring is a set with an operation $+$ and an operation $\times$ where everything we would expect to hold, holds.
A ring is a set with an operation $+$ and an operation $\times$ where everything we would expect to hold, holds.

**Addition and subtraction**
- $a - a = 0$
- $a + b = b + a$
- $a + (b + c) = (a + b) + c$  
  (so we can define $a + b + c$)

**Multiplication** (we often omit the symbol $\times$)
- $a(bc) = (ab)c$  
  (so we can define $abc$)

**Addition and multiplication**
- $a(b + c) = ab + ac$
Examples and non-examples

Examples

• integers, rationals, complex numbers, ...

Counterexamples

• machine floats

```c
void main()
{
    float a, b, c;    a = 3432.675;
    b = 0.03232;
    c = 24.535;
    printf("%f\n", ((a+b)+c) - (a+(b+c)));
}

-0.000244
```
Further examples

Bits = \{0, 1\} form a ring with the operations

<table>
<thead>
<tr>
<th>xor</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>and</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

that we prefer to write

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>×</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Rule:** do the operation as if you had integers, and reduce modulo 2.

**Notation:** \{0, 1\} = \mathbb{F}_2 = \text{GF}(2) = \mathbb{Z}/2\mathbb{Z}.
Naive algorithm
Naive multiplication

You have to multiply

\[ F = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1}, \quad G = g_0 + g_1 x + \cdots + g_{n-1} x^{n-1}; \]

the result is

\[ H = FG = h_0 + \cdots + h_{2n-2} x^{2n-2} \]

with

\[ h_0 = f_0 g_0 \quad \ldots \quad h_i = \sum_{j+k=i} f_j g_k \quad \ldots \quad h_{2n-2} = f_{n-1} g_{n-1}. \]

Looking at the formula, computing all \( h_i \) takes \( n^2 \) multiplications and \( (n - 1)^2 \) additions.

Total: \( O(n^2) \).
Karatsuba’s algorithm
Karatsuba’s algorithm

Two ingredients

- a trick for low degree
- divide-and-conquer

The trick. You have to multiply

\[ F = f_0 + f_1 x, \quad G = g_0 + g_1 x, \]

so the product is

\[ H = f_0 g_0 + (f_0 g_1 + f_1 g_0) x + f_1 g_1 x^2. \]

Slow algorithm: compute \( f_0 g_0, f_0 g_1, f_1 g_0, f_1 g_1. \)
Karatsuba’s algorithm

Two ingredients

• a trick for low degree
• divide-and-conquer

The trick. You have to multiply

\[ F = f_0 + f_1 x, \quad G = g_0 + g_1 x, \]

so the product is

\[ H = f_0 g_0 + (f_0 g_1 + f_1 g_0) x + f_1 g_1 x^2. \]

Better:

• compute \( f_0 g_0 \) and \( f_1 g_1 \)
• deduce \( f_0 g_1 + f_1 g_0 = (f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1 \)

3 multiplications and 4 additions.
Divide and conquer

Suppose now that \( F, G \) have \( n \) terms, with \( n = 2^k \), and let

\[
F = F_0 + F_1 x^{n/2}, \quad G = G_0 + G_1 x^{n/2};
\]

so \( F_0, F_1, G_0, G_1 \) have \( n/2 \) terms. As before, \( H = FG \) is

\[
H = F_0 G_0 + (F_0 G_1 + F_1 G_0) x^{n/2} + F_1 G_1 x^n.
\]

Algorithm

- \textbf{if} \( n = 1 \), \textbf{return} \( h = f_0 g_0 \)
- compute \textbf{recursively} \( F_0 G_0, F_1 G_1, (F_0 + F_1)(G_0 + G_1) \)
- deduce \( F_0 G_1 + F_1 G_0 = (F_0 + F_1)(G_0 + G_1) - F_0 G_0 - F_1 G_1. \)
- \textbf{return} \( H \).

3 recursive calls and some additions.
Warmup: simplified analysis

We count only **multiplications**:

- \( k(n) \) is the number of multiplications with inputs of size \( n = 2^s \).

**Recurrence:**

- \( k(1) = 1 \)
- \( k(n) = 3k(n/2) \)

**Unrolling the recurrence:**

\[
    k(n) = k(2^s) = 3k(2^{s-1}) = 3^2k(2^{s-2}) = \cdots = 3^s k(1) = 3^s.
\]

**Simplification:** \( k(n) = 3^s = 3^{\log_2(n)} = n^{\log_2(3)} \).
Total complexity

- \( K(n) \) is the number of operations with inputs of size \( n = 2^s \).

Recurrence:

- \( K(1) = 1 \)
- \( K(n) = 3K(n/2) + \ell n \)

Here, \( \ell \) is a constant that I don’t want to estimate (\( \ell \) is about 4)

Unrolling the recurrence:

\[
K(n) = O(n \log_2(3)).
\]
Any $n$

Easy solution
- replace $n$ by the next power of 2.

To do better
- adjust the size of the recursive calls: $2 \times \lceil n/2 \rceil$ and $1 \times \lfloor n/2 \rfloor$.

(number of mults in size $2^s$)/$3^s$:

http://pauillac.inria.fr/~quercia/
Master theorem, first version

**Assumption:** suppose that a function $T(n)$ satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k,$$

for $n$ a power of $b$, with

- $b > 1$ \hspace{1cm} \text{divide problem size by } b$
- $a > b$ \hspace{1cm} \text{do } a \text{ recursive calls}$
- $\log_b(a) > k$ \hspace{1cm} \text{runtime } O(n^{\log_b(a)}) \text{ if no second term}$

**Conclusion:** for $n$ a power of $b$,

$$T(n) = \Theta(n^{\log_b(a)}).$$

**Example:** Karatsuba with $a = 3$, $b = 2$. 

37 / 65
Toom’s algorithm(s)
The idea behind Karatsuba’s trick

**Evaluation.**

\[
\begin{align*}
  f_0 &= F(0) & g_0 &= G(0) \\
  f_0 + f_1 &= F(1) & g_0 + g_1 &= G(1) \\
  f_1 &= F(\infty) & g_1 &= G(\infty)
\end{align*}
\]

**Multiplication.** After the products, we know

\[
\begin{align*}
  H(0) &= F(0)G(0) \\
  H(1) &= F(1)G(1) \\
  H(\infty) &= F(\infty)G(\infty)
\end{align*}
\]

**Interpolation.**

\[
H = H(0) + (H(1) - H(0) - H(\infty))x + H(\infty)x^2.
\]
Toom’s 3-algorithm

Let

\[ F = f_0 + f_1 x + f_2 x^2, \quad G = g_0 + g_1 x + g_2 x^2 \]

and

\[ H = F G = h_0 + h_1 x + h_2 x^2 + h_3 x^3 + h_4 x^4. \]

To get \( H \), we still do

- evaluation,
- multiplication,
- interpolation.

Now, we need 5 values because \( H \) has 5 unknown coefficients:

- 0, 1, −1, 2, \( \infty \)

Other choices are possible

- would not work with coefficients in \( \mathbb{F}_2 \).
The evaluation / interpolation phase

Evaluation.

\[ F(0) = f_0 \quad G(0) = g_0 \]
\[ F(1) = f_0 + f_1 + f_2 \quad G(1) = g_0 + g_1 + g_2 \]
\[ F(-1) = f_0 - f_1 + f_2 \quad G(-1) = g_0 - g_1 + g_2 \]
\[ F(2) = f_0 + 2f_1 + 4f_2 \quad G(2) = g_0 + 2g_1 + 4g_2 \]
\[ F(\infty) = f_2 \quad G(\infty) = g_2 \]

Multiplication: the products give us

\[ H(0) = F(0)G(0), \quad \ldots, \quad H(\infty) = F(\infty)G(\infty) \]
Interpolation.

Recover \( h_0, h_1, h_2, h_3, h_4 \) knowing

\[
\begin{align*}
H(0) &= h_0 \\
H(-1) &= h_0 - h_1 + h_2 - h_3 + h_4 \\
H(1) &= h_0 + h_1 + h_2 + h_3 + h_4 \\
H(2) &= h_0 + 2h_1 + 4h_2 + 8h_3 + 16h_4 \\
H(\infty) &= h_4
\end{align*}
\]

Linear system of 5 equations in 5 unknowns.

Remark: rather sophisticated algorithms used to optimize the resolution (quasi-exhaustive search, processor dependent, \ldots)
The evaluation / interpolation phase

\[ A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{4=2} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 3 & 3 & 9 & 15 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{2=3} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & -2 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 3 & 3 & 9 & 15 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{3=1} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & -2 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 3 & 3 & 9 & 15 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{2/=(-1)} \]

\[ \tilde{A}_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 3 & 3 & 9 & 15 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{4=3} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 6 & 12 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{4/=(6)} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{3=5} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \xrightarrow{I_5} \]

http://marco.bodrato.it/
The Toom recursion

Analysis: at each step,

- we divide $n$ by 3;
- and we do 5 recursive calls;
- the extra operations count is $\ell n$, for some $\ell$.

Recurrence:

$$T(n) = 5T\left(\frac{n}{3}\right) + \ell n.$$  

Master theorem:

$$T(n) = \Theta(n^{\log_3(5)}) = O(n^{1.47}).$$

Remark: the constant in the $O(\ )$ is $\approx \ell$. 
Generalizations of Toom’s algorithm

**Algorithm:** Write the input $F, G$ as

$$F = F_0 + F_1 x^{n/k} + \cdots + F_{k-1} x^{(k-1)n/k}, \quad G = G_0 + G_1 x^{n/k} + \cdots + G_{k-1} x^{(k-1)n/k},$$

and the output as $H = FG = H_0 + H_1 x^{n/k} + \cdots + H_{2k-2} x^{(2k-2)n/k}$.

**Analysis:** at each step,

- we divide $n$ by $k$;  
  number of terms in $F, G$
- we do $2k - 1$ recursive calls;  
  number of terms in $H$
- the extra operations count is $ln$, for some $l$.

**Master theorem:**

$$T(n) = \Theta(n^{\log_k (2k-1)}).$$

**Examples:**

$$k = 100 \implies O(n^{1.15})$$
Generalizations of Toom’s algorithm

Algorithm: Write the input $F, G$ as

$$F = F_0 + F_1 x^{n/k} + \cdots + F_{k-1} x^{(k-1)n/k}, \quad G = G_0 + G_1 x^{n/k} + \cdots + G_{k-1} x^{(k-1)n/k},$$

and the output as $H = FG = H_0 + H_1 x^{n/k} + \cdots + H_{2k-2} x^{(2k-2)n/k}$.

Analysis: at each step,
- we divide $n$ by $k$; \hspace{2cm} \text{number of terms in } F, G
- we do $2k - 1$ recursive calls; \hspace{2cm} \text{number of terms in } H
- the extra operations count is $\ell n$, for some $\ell$.

Master theorem:

$$T(n) = \Theta(n^{\log_k(2k-1)}).$$

Examples:

$$k = 1000 \implies O(n^{1.1})$$
Generalizations of Toom’s algorithm

Algorithm: Write the input $F, G$ as

$$ F = F_0 + F_1 x^{n/k} + \cdots + F_{k-1} x^{(k-1)n/k}, \quad G = G_0 + G_1 x^{n/k} + \cdots + G_{k-1} x^{(k-1)n/k}, $$

and the output as $H = FG = H_0 + H_1 x^{n/k} + \cdots + H_{2k-2} x^{(2k-2)n/k}$.

Analysis: at each step,

- we divide $n$ by $k$; \hspace{2cm} \text{number of terms in } F, G
- we do $2k - 1$ recursive calls; \hspace{2cm} \text{number of terms in } H
- the extra operations count is $\ell n$, for some $\ell$.

Master theorem:

$$ T(n) = \Theta(n^{\log_k (2k-1)}). $$

Examples:

$$ k = 10000 \implies O(n^{1.07}) $$
Fast Fourier Transform
(over $\mathbb{C}$)
The idea behind FFT

Suppose that (e.g. in Toom’s algorithm), evaluation and interpolation were *almost free*, say *linear time*.

**Multiplication algorithm:**

- evaluate $F$ and $G$ at $2n - 1$ points $O(n)$
- multiply the values $O(n)$
- interpolate $H$ $O(n)$

**Total:** $O(n)$. 
The idea behind FFT

Suppose that (e.g. in Toom’s algorithm), evaluation and interpolation were almost free, say linear time.

**Multiplication algorithm:**

- evaluate \( F \) and \( G \) at \( 2n - 1 \) points \( O(n) \)
- multiply the values \( O(n) \)
- interpolate \( H \) \( O(n) \)

**Total:** \( O(n) \).

**In real life**

- evaluation and interpolation are expensive in general;
- FFT gives an \( O(n \log(n)) \) evaluation and interpolation;
- and so an \( O(n \log(n)) \) multiplication.
Complex numbers

\[ z = e^{i\alpha} = \cos(\alpha) + i \sin(\alpha) \]

\[ z_n^k = e^{\frac{2ik\pi}{n}} \]

\[ z_n = e^{\frac{2i\pi}{n}} \]
## Roots of unity

### Definition

- **A \( n \)th root of unity** is a complex number \( z \) such that \( z^n = 1 \).
- The **primitive \( n \)th root of unity** is

\[
z_n = e^{\frac{2i\pi}{n}}
\]

### Prop.

- The \( n \)th roots of unity are the powers

\[
z_n^0 = 1, \quad z_n, \quad z_n^2, \quad \ldots, \quad z_n^{n-1}
\]

- If \( n = 2m \), then

\[
z_m = z_n^2.
\]
Examples

\[ n = 4 \quad z_4^2 = -1 \quad z_4^0 = 1 \]

\[ z_4^3 = -z_4 \]
Examples

\[ n = 8 \quad z_8^4 = -1 \quad z_8^5 = -z_8 \quad z_8^6 = -z_8^2 \quad z_8^7 = -z_8^3 \quad z_8^0 = 1 \quad z_8^2 = \]

\[ n = 8 \quad z_8^4 = -1 \quad z_8^5 = -z_8 \quad z_8^6 = -z_8^2 \quad z_8^7 = -z_8^3 \quad z_8^0 = 1 \quad z_8^2 = \]
Consider the \( n \)th roots of unity:

\[ z_0^n, \ldots, z_{n-1}^n, \]

Then the operation

\[ F = f_0 + \cdots + f_{n-1}x^{n-1} \mapsto (F(z_0^n), \ldots, F(z_{n-1}^n)) \]

is called the \textbf{Discrete Fourier Transform}.

\textbf{Costs:}

- \textbf{naive algorithm:} \( O(n^2) \) operations.
- \textbf{FFT:} \( O(n \log(n)) \) operations.
Squaring for $n$ even

**Goal:** write a divide-and-conquer algorithm

- reduce an instance of size $n$ to two instances of size $n/2$
- need to divide the number of points by 2
- same with the degree of the polynomials.

With $n = 2m$, squaring

- sends all $n$th roots of unity to $m$th roots;
- $z^n_i$ and $z^{i+m}_n = -z^n_i$ have the same square.
Squaring for $n$ even

- $z_{8}^{0} = 1$
- $z_{8}^{1} = z_{8}$
- $z_{8}^{2} = z_{8}^{2}$
- $z_{8}^{3} = z_{8}^{3}$
- $z_{8}^{4} = -1$
- $z_{8}^{5} = -z_{8}$
- $z_{8}^{6} = z_{8}$
- $z_{8}^{7} = -z_{8}^{3}$

- $z_{4}^{0} = 1$
- $z_{4}^{1} = z_{4}$
- $z_{4}^{2} = -1$
- $z_{4}^{3} = -z_{4}$
- $z_{4}^{4} = z_{4}^{2}$

Even and odd decomposition

Any polynomial

\[ F = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1} \]

can be written

\[ F = F_{\text{even}}(x^2) + x F_{\text{odd}}(x^2), \]

with

\[ \deg(F_{\text{even}}) < n/2, \quad \deg(F_{\text{odd}}) < n/2. \]

**Example.**

- \( F = 28 + 11x + 34x^2 - 55x^3 \)
- \( F_{\text{even}} = 28 + 34x \)
- \( F_{\text{odd}} = 11 - 55x \)

This is the divide-and-conquer process for polynomials.
To evaluate $F(u)$:

- evaluate $v = F_{\text{even}}(u^2)$
- evaluate $v' = F_{\text{odd}}(u^2)$
- deduce $F(u) = v + uv'$. 

Danger: if we choose $u_1, \ldots, u_{n-1}$ poorly, we have to evaluate two polynomials of degree $< n/2$ at $n$ points.
To evaluate $F(u)$:

- evaluate $v = F_{\text{even}}(u^2)$
- evaluate $v' = F_{\text{odd}}(u^2)$
- deduce $F(u) = v + uv'$.

To evaluate all $F(u_0), \ldots, F(u_{n-1})$:

- evaluate all $v_i = F_{\text{even}}(u_i^2)$
- evaluate all $v'_i = F_{\text{odd}}(u_i^2)$
- deduce $F(u_i) = v_i + u_i v'_i$. 
Decomposition and evaluation

To evaluate $F(u)$:
- evaluate $v = F_{\text{even}}(u^2)$
- evaluate $v' = F_{\text{odd}}(u^2)$
- deduce $F(u) = v + uv'$.

To evaluate all $F(u_0), \ldots, F(u_{n-1})$:
- evaluate all $v_i = F_{\text{even}}(u_i^2)$
- evaluate all $v'_i = F_{\text{odd}}(u_i^2)$
- deduce $F(u_i) = v_i + u_iv'_i$.

Danger: if we choose $u_1, \ldots, u_{n-1}$ poorly, we have to evaluate two polynomials of degree $< n/2$ at $n$ points.
Suppose that the points \( u_i \) are \( n \)th roots of unity:

\[
z_0^n, \ldots, z_{n-1}^n,
\]

with \( n = 2m \). Then, their squares are

\[
z_0^m, \ldots, z_{m-1}^m
\]

**FFT** \((F, n)\)

- **if** \( n = 1 \), return \( f_0 \).
- let \( V = \text{FFT}(F_{\text{even}}, n/2) \)
- let \( V' = \text{FFT}(F_{\text{odd}}, n/2) \)
- **return** \([V[i \mod n/2] + z_n^i V'[i \mod n/2] : 0 \leq i < n]\)
Master theorem, second version

Assumption: suppose that a function $T(n)$ satisfies

$$ T(n) = 2T\left(\frac{n}{2}\right) + cn,$$

for $n$ a power of 2.

Conclusion: $T(n) = \Theta(n \log(n))$, for $n$ a power of 2.

Application: the cost $F(n)$ of the FFT algorithm satisfies

- $F(1) = 0$
- $F(n) = 2F(n/2) + \frac{3}{2}n$,

so $F(n) = \Theta(n \log(n))$. 
Prop. Performing the inverse DFT in size $n$ is done by

- performing a DFT at

$$z_0^n, \ z_{n-1}^{-}, \ \cdots, \ z_{-(n-1)}^{-}$$

- dividing the results by $n$.

This new DFT is the same as before:

$$z_n^{-i} = z_{n-i}^{-}$$

so the outputs are just shuffled.

Consequence: the cost of the inverse DFT is $\Theta(n \log(n))$. 

To multiply two polynomials $F, G$ in $\mathbb{C}[x]$, of degrees $< m$:

- Find $n = 2^k$ such that $H = FG$ has degree less than $n$ \[ n \leq 2m \]
- Compute $\text{DFT}(F, n)$ and $\text{DFT}(G, n)$ \[ O(n \log(n)) \]
- Multiply the values to get $\text{DFT}(H, n)$ \[ O(n) \]
- Recover $H$ by inverse DFT. \[ O(n \log(n)) \]

**Cost:** $O(n \log(n)) = O(m \log(m))$. 
Why “Fourier Transform”?

In analysis, one uses the continuous Fourier Transform

\[ k \mapsto \hat{f}(k) = \int_{-\infty}^{\infty} f(t)e^{-2\pi ikt} \, dt. \]

In signal processing, discrete Fourier Transform, for discrete signals:

\[ k \mapsto \hat{\varphi}(k) = \sum_{j=0}^{n-1} \varphi\left(\frac{j}{n}\right)e^{-2\pi ijk} = \sum_{j=0}^{n-1} \varphi\left(\frac{j}{n}\right) \left( e^{-\frac{2\pi ik}{n}} \right)^j = \sum_{j=0}^{n-1} \varphi\left(\frac{j}{n}\right) (z_n^k)^j = F(z_n^k) \]

with

\[ F(z) = \varphi(0) + \varphi\left(\frac{1}{n}\right)z + \cdots + \varphi\left(\frac{n-1}{n}\right)z^{n-1}. \]
Gauss or Fourier Transform?

<table>
<thead>
<tr>
<th>$x$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0^0$</td>
<td>$6^0 48' \text{ Bor.} = + 408'$</td>
</tr>
<tr>
<td>$30$</td>
<td>$1 29 \ldots \ldots \ldots + 89$</td>
</tr>
<tr>
<td>$60$</td>
<td>$1 6 \text{ Austr.} \ldots - 66$</td>
</tr>
<tr>
<td>$90$</td>
<td>$0 10 \text{ Bor.} \ldots + 10$</td>
</tr>
<tr>
<td>$120$</td>
<td>$5 38 \ldots \ldots \ldots + 338$</td>
</tr>
<tr>
<td>$150$</td>
<td>$13 27 \ldots \ldots \ldots + 807$</td>
</tr>
<tr>
<td>$180$</td>
<td>$20 38 \ldots \ldots \ldots + 1238$</td>
</tr>
<tr>
<td>$210$</td>
<td>$25 11 \ldots \ldots \ldots + 1511$</td>
</tr>
<tr>
<td>$240$</td>
<td>$26 23 \ldots \ldots \ldots + 1583$</td>
</tr>
<tr>
<td>$270$</td>
<td>$24 22 \ldots \ldots \ldots + 1462$</td>
</tr>
<tr>
<td>$300$</td>
<td>$19 43 \ldots \ldots \ldots + 1183$</td>
</tr>
<tr>
<td>$330$</td>
<td>$13 24 \ldots \ldots \ldots + 804$</td>
</tr>
</tbody>
</table>

Distribuamus hanc periodum primo in tres periodos quaternorum terminorum

<table>
<thead>
<tr>
<th>$a = 0^0$</th>
<th>$A = + 408$</th>
<th>$a' = 30^0$</th>
<th>$A' = + 89$</th>
<th>$a'' = 60^0$</th>
<th>$A'' = - 66$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b = 90^0$</td>
<td>$B = + 10$</td>
<td>$b' = 120^0$</td>
<td>$B' = + 338$</td>
<td>$b'' = 150^0$</td>
<td>$B'' = + 807$</td>
</tr>
<tr>
<td>$c = 180^0$</td>
<td>$C = + 1238$</td>
<td>$c' = 210^0$</td>
<td>$C' = + 1511$</td>
<td>$c'' = 240^0$</td>
<td>$C'' = + 1583$</td>
</tr>
<tr>
<td>$d = 270^0$</td>
<td>$D = + 1462$</td>
<td>$d' = 300^0$</td>
<td>$D' = + 1183$</td>
<td>$d'' = 330^0$</td>
<td>$D'' = + 804$</td>
</tr>
</tbody>
</table>
Multivariate polynomials
Things are usually more complicated

- the **degree** is not the proper measure anymore;
- the **shape** of the set monomials becomes more important.

**Empirically**, many problems in several variables are **sparse**

- in the sparsest possible case, the naive algorithm is optimal.
Multivariate polynomials

One useful trick, **Kronecker substitution:**

- works for **any** multivariate polynomials;
- good for polynomials $F(x_1, \ldots, x_n)$ with

  $$\deg(F, x_1) < d_1, \ldots, \deg(F, x_n) < d_n;$$

- reduces to **univariate** polynomial multiplication.
Kronecker’s substitution on an example

\[ F = (1 + 3x_1 + 4x_1^2) + (22 + x_1 - x_1^2)x_2 + (-3 - 3x_1 + 2x_1^2)x_2^2 \]
\[ = F_0(x_1) + F_1(x_1)x_2 + F_2(x_1)x_2^2 \]

\[ G = (-2 + x_1 + x_1^2) + (4 + x_1 + 3x_1^2)x_2 + (3 - x_1 + x_1^2)x_2^2 \]
\[ = G_0(x_1) + G_1(x_1)x_2 + G_2(x_1)x_2^2 \]

Then \( H = FG \) is

\[ H = F_0G_0 + (F_0G_1 + F_1G_0)x_2 + (F_0G_2 + F_1G_1 + F_2G_0)x_2^2 + (F_1G_2 + F_2G_1)x_2^3 + F_2G_2x_2^4 \]
Kronecker’s substitution on an example

• Remark that all $F_i(x_1)G_j(x_1)$ have degree at most 4
• So we replace $x_2$ by $x_1^5$

\[ F^* = (1 + 3x_1 + 4x_1^2) + (22 + x_1 - x_1^2)x_1^5 + (-3 - 3x_1 + 2x_1^2)x_1^{10} \]
\[ = F_0(x_1) + F_1(x_1)x_1^5 + F_2(x_1)x_1^{10} \]

\[ G^* = (-2 + x_1 + x_1^2) + (4 + x_1 + 3x_1^2)x_1^5 + (3 - x_1 + x_1^2)x_1^{10} \]
\[ = G_0(x_1) + G_1(x_1)x_1^5 + G_2(x_1)x_1^{10} \]

\[ 5 = 4 + 1 \]
Kronecker’s substitution on an example

After multiplying $F^*$ and $G^*$:

\[
H^* = F_0G_0 \\
+ (F_0G_1 + F_1G_0)x_1^5 \\
+ (F_0G_2 + F_1G_1 + F_2G_0)x_1^{10} \\
+ (F_1G_2 + F_2G_1)x_1^{15} \\
+ F_2G_2x_1^{20}
\]

Because $\text{deg}(F_iG_j) \leq 4$, there is no overlap.
So we can directly read off the result.