CS 4424
Foundations of computer algebra

Éric Schost
eschost@uwo.ca

This course

- Basic objects

  polynomials, matrices

- Basic techniques

  divide-and-conquer, Newton iteration, Hensel lifting

- Goal of the course: what happens when you issue

```
> factor(x^4+x^3-6*x^2+5*x-1);
                                   2                   2
                                 (x  + 3 x - 1) (x - 1)
```

finite fields, lattice reduction

Assignments, project, etc

3 assignments

- due in September and October

Midterm

- November 4th
- open book

Project

- Reading papers
- Coding may be involved, but not required

Office hours

- Monday, 9:30am – 11:30am

Computer algebra

Roughly, studies how to solve mathematical problems on a computer, with an emphasis on "exact solutions".

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

Many aspects

- programming languages for expressing mathematical notions;
- algorithms and complexity;
- implementation;
- ...

Here: emphasis on algorithms and complexity.

Numbers

Basic problem: dealing with numbers properly.

- exactness means that we handle multi-precision (arbitrary length) numbers.

A handful of algorithms

- addition easy

  $-\infty$

- multiplication hard, but satisfactory answers

  1960's

- division well-understood

  1960's

- factorization ultra-hard

  became especially hot after the discovery of the RSA scheme.

Linear equations

A large part of the world's computers are busy solving linear systems

$$
\begin{aligned}
x_1 + x_2 - 3x_3 &= 3 \\
-x_1 + 3x_2 - x_3 &= 0 \\
10x_1 + 3x_2 - x_3 &= 5
\end{aligned}
$$

- google
- simplex for linear programming
- numerical simulations of differential equations

Linear 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,

  NAN or slowdown with ill-posed problems
- in contexts that are not numerical,

  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.

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:

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

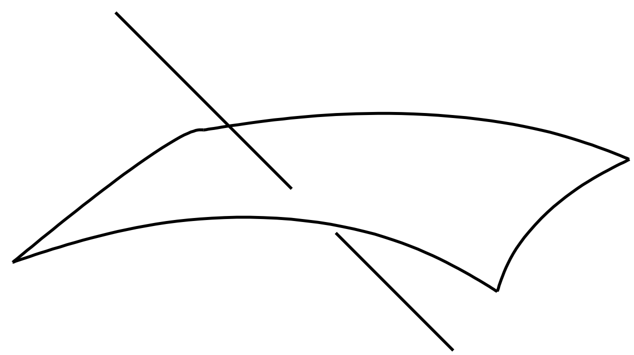
The second case is typical.

Polynomial equations

In general, a system of n polynomial equations of degree d in n unknowns:

- has d^n solutions
- which do not have rational coordinates.

But all sorts of degenerate situations can occur.



Polynomial equations

Purely numerical approaches do not deal well with nasty situations.

- exact treatment reveals all the information;
- mixed symbolic / numeric algorithms.

All such polynomial system solving algorithms are complex and costly.

- in the nicest case (finite number of solutions, nice system), output size is d^n;
- with infinitely many solutions, it's worse:
  - naively, d^{rn} — r is the dimension of the solution set
  - better encodings: d^n
- worst case: 2^{2^n}.

Computing with sequences

Problem: find the next term.

$U: \; 1, 1, 1, 1, 1, 1, 1, 1$

$V: \; 0, 1, 1, 2, 3, 5, 8, 13$

$W: \; 12, 134, 222, 21, -3898, -40039, -347154, -2929918, -24657854$

Answer: 1, 21 and -207605083.

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.

Computing with sequences

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

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

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

$$a_n = \sum_{k=0}^{n} \binom{n}{k}^2 \binom{n+k}{k}^2 c_{n,k}.$$

Then each sequence a_n and b_n satisfies the recurrence

$$(n+2)^3 u_{n+2} + (\cdots) u_{n+1} + (\cdots) u_n = 0.$$

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

Polynomial (and integer) multiplication

Problem statement

Input

- two polynomials

$$F = f_0 + f_1 x + \cdots + f_{n-1} x^{n-1} \qquad G = g_0 + g_1 x + \cdots + g_{n-1} x^{n-1}$$

Output

- the product

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

with

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

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.

Results to remember

Prop. One can multiply polynomials with n terms using ...

- the naive algorithm

  $O(n^2)$ operations.

- Karatsuba's algorithm

  $O(n^{1.59})$ operations $\qquad 1.59 = \log_2(3)$

- Toom's algorithm(s)

  $O(n^{1.47})$ operations $\qquad 1.47 = \log_3(5)$

- Fast Fourier Transform

  $O(n\log(n))$ operations $\qquad$ nice cases

  $O(n\log(n)\log(\log(n)))$ operations $\qquad$ in general

It's still unknown with the optimal is.

Thresholds

Practical aspects: don't neglect ...

- the constants in the $O(\dots)$ (usually better for the simpler (slower) algorithms)
- lower-level aspects (data representation, architecture)

In the best current implementations (over nice coefficient rings)

- Karatsuba beats the naive algorithm for degrees about 20.
- FFT wins for degrees about 100.

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).

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

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

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

Conclusion: similarities, but carries make the integer case harder.

Results to remember

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

  $O(n \log(n) 2^{\log^*(n)})$ bit operations $\log^*(n) =$ number of logs to reach 1

It's still unknown with the optimal is.

Thresholds

Practical aspects: don't neglect ...

- the constants in the $O(\dots)$ (usually better for the simpler (slower) algorithms)

In the best current implementations (over nice coefficient rings)

- Karatsuba beats the naive algorithm for about 100 words.
- FFT wins for about 10000 words.

Some problems require to handle integer with about 800000000 words (100 MB storage).

Coefficient rings

Coefficients

Most algorithms 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,
- with some obvious good-behaviour rules.

Rings

A ring is a set with a $+$ and a $\times$ where everything we expect holds.

Addition and subtraction

- $a - a = 0$
- $a + b = b + a$
- $a + (b + c) = (a + b) + c$

Multiplication

- $a(bc) = (ab)c$

Addition and multiplication

- $a(b + c) = ab + ac$

Examples and non-examples

Examples

- integers, rationals, complex numbers, ...

Counterexamples

- machine floats

```
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 form a ring with the operations

xor	0	1
0	0	1
1	1	0

and

and	0	1
0	0	0
1	0	1

that we prefer to write

$+$	0	1
0	0	1
1	1	0

and

$\times$	0	1
0	0	0
1	0	1

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

Notation: $\{0, 1\} = \mathbb{F}_2$.

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

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

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: $f_0 g_0, f_0 g_1, f_1 g_0, f_1 g_1$.

Better:

1. compute $f_0 g_0$ and $f_1 g_1$
2. 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

1. If $n = 1$, return $h = f_0 g_0$. Else:
2. Compute $f_0 g_0$ and $f_1 g_1$.
3. Deduce $f_0 g_1 + f_1 g_0 = (f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1$.
4. Deduce h.

3 recursive calls and some additions.

Simplified analysis

We count only multiplications:

- $M(n)$ is the number of multiplications with inputs of size n, $n = 2^k$.

Recurrence:

- $M(1) = 1$
- $M(n) = 3M(n/2)$

Unrolling the recurrence:

$$M(n) = M(2^k) = 3M(2^{k-1}) = 3^2M(2^{k-2}) = \cdots = 3^kM(1) = 3^k.$$

Simplification: $M(n) = 3^k = 3^{\log_2(n)} = n^{\log_2(3)}$.

Generalization: for any degree, $O(n^{\log_2(3)})$ multiplications.

Counting all operations

Total complexity

- $K(n)$ is the number of operations with inputs of size n, $n = 2^k$.

Recurrence:

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

Here, ℓ is a constant that I don't want to estimate ℓ is about 4.

Unrolling the recurrence:

$$K(n) = O(n^{\log_2(3)}).$$

Master theorem, first version

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

$$T(n) \;\leq\; aT(\frac{n}{b}) + cn^k,$$

with

- n a power of b — not really needed, just for simplicity
- $b > 1$,
- $a > b$,
- $\log_b(a) > k$.

Conclusion: then

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

Consequence: the cost of Karatsuba's algorithm is $T(n) = O(n^{\log_b(a)})$.

Toom's algorithm

The idea behind the trick

Evaluation.

$$\begin{aligned} f_0 &= f(0) & g_0 &= g(0) \\ f_0 + f_1 &= f(1) & g_0 + g_1 &= g(1) \\ f_1 &= f(\infty) & g_1 &= g(\infty) \end{aligned}$$

Multiplication. After the products, we know

$$\begin{aligned} h(0) &= f(0)g(0) \\ h(1) &= f(1)g(1) \\ h(\infty) &= f(\infty)g(\infty) \end{aligned}$$

Interpolation.

$$h = h(0) + (h(1) - h(0) - h(\infty))x + h(\infty)x^2.$$

Toom's algorithm

Let

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

and

$$H = FG = h_0 + h_1 x + h_2 x^2 + h_3 x^3 + h_4 x^4.$$

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.

$$\begin{array}{lcl}
f(0) & = & f_0 \\
f(1) & = & f_0 + f_1 + f_2 \\
f(-1) & = & f_0 - f_1 + f_2 \\
f(2) & = & f_0 + 2f_1 + 4f_2 \\
f(\infty) & = & f_2
\end{array}
\qquad
\begin{array}{lcl}
g(0) & = & g_0 \\
g(1) & = & g_0 + g_1 + g_2 \\
g(-1) & = & g_0 - g_1 + g_2 \\
g(2) & = & g_0 + 2g_1 + 4g_2 \\
g(\infty) & = & g_2
\end{array}$$

Multiplication: the products give us

$$h(0) = f(0)g(0), \quad \ldots, \quad h(\infty) = f(\infty)g(\infty)$$

Interpolation: recover H from its values.

The Toom recursion

Analysis: at each step,

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

Recurrence:

$$T(n) \leq 5T(\frac{n}{3}) + \ell n.$$

Master theorem:

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

Generalization of Toom

Let

$$F = f_0 + f_1 x + \cdots + f_{k-1} x^{k-1}, \quad G = g_0 + g_1 x + \cdots + g_{k-1} x^{k-1}$$

and

$$H = FG = h_0 + h_1 x + \cdots + h_{2k-2} x^{2k-2}.$$

Analysis: at each step,

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

Master theorem:

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

Examples:

$$k = 100 \implies O(n^{1.15}), \quad k = 1000 \implies O(n^{1.1}), \quad k = 10000 \implies O(n^{1.07})$$

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 $O(n)$

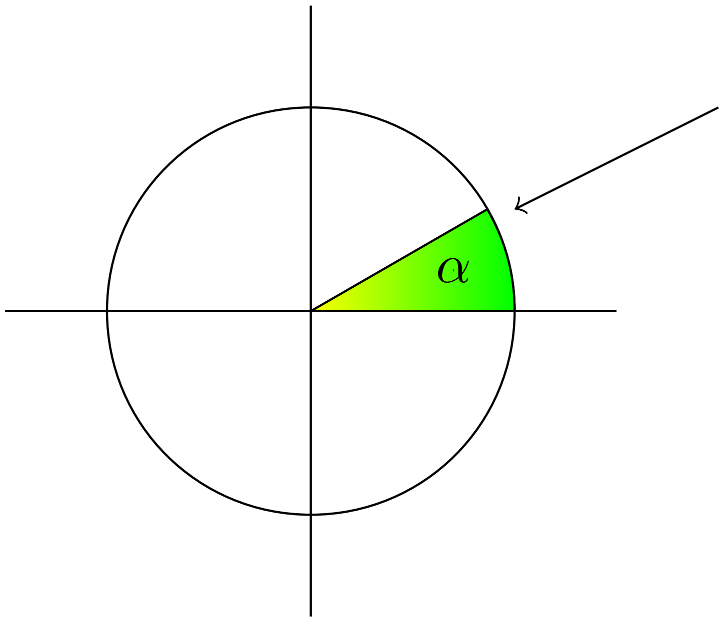
Total: $O(n)$.

In real life

- evaluation and interpolation are expensive in general;
- FFT provides with a $O(n\log(n))$ evaluation and interpolation;
- and so a $O(n\log(n))$ multiplication.

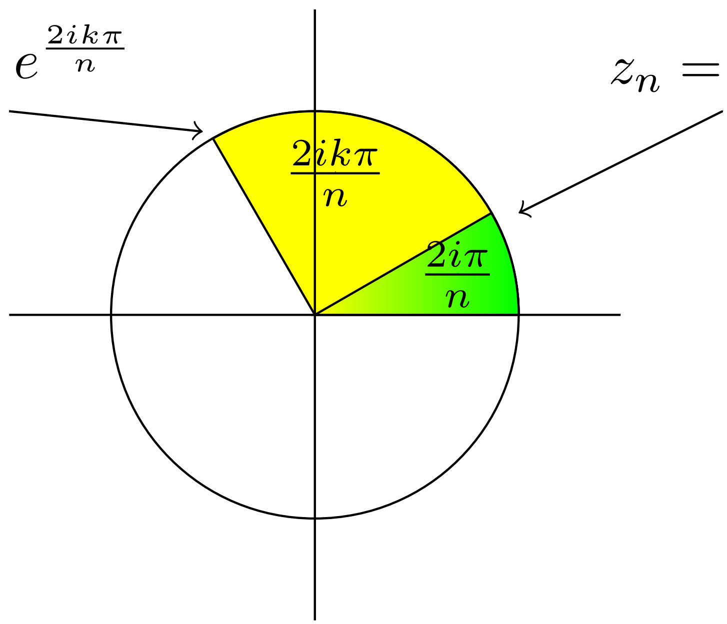
Complex numbers

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



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

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



Roots of unity

Def.

- A nth root of unity is a complex number z such that $z^n = 1$.
- The primitive nth root of unity is

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

Prop.

- The nth roots of unity are the powers

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

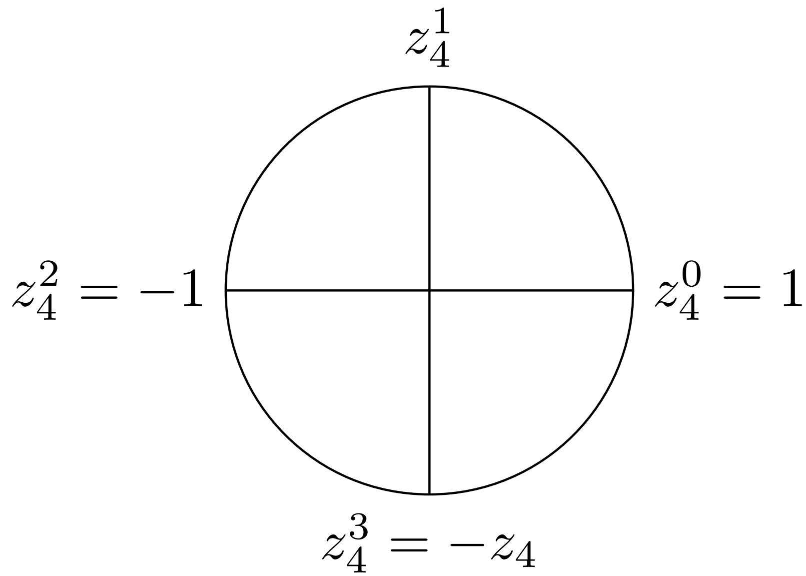
Prop

- If $n = 2m$, then

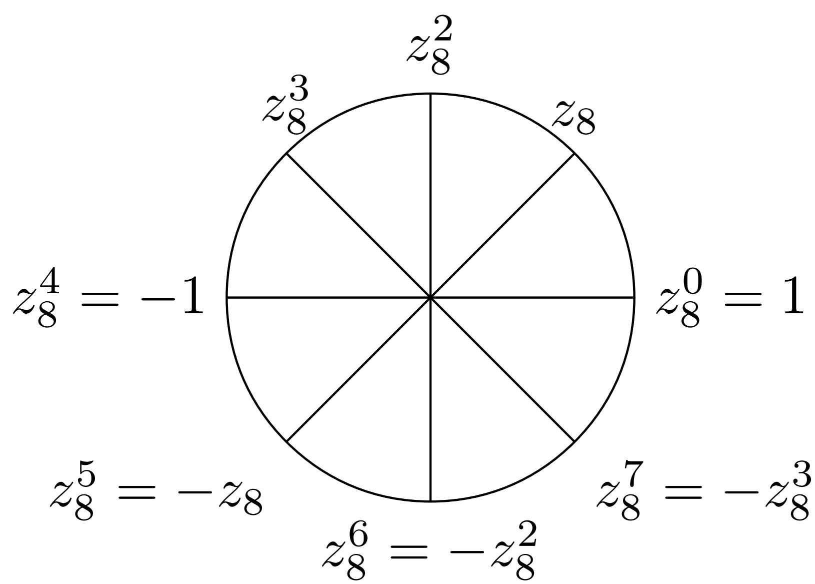
$$z_m = z_n^2.$$

Examples

$n = 4$



$n = 8$



Discrete Fourier Transform

Consider the nth roots of unity:

$$z_n^0, \quad \ldots, \quad z_n^{n-1},$$

Then the operation

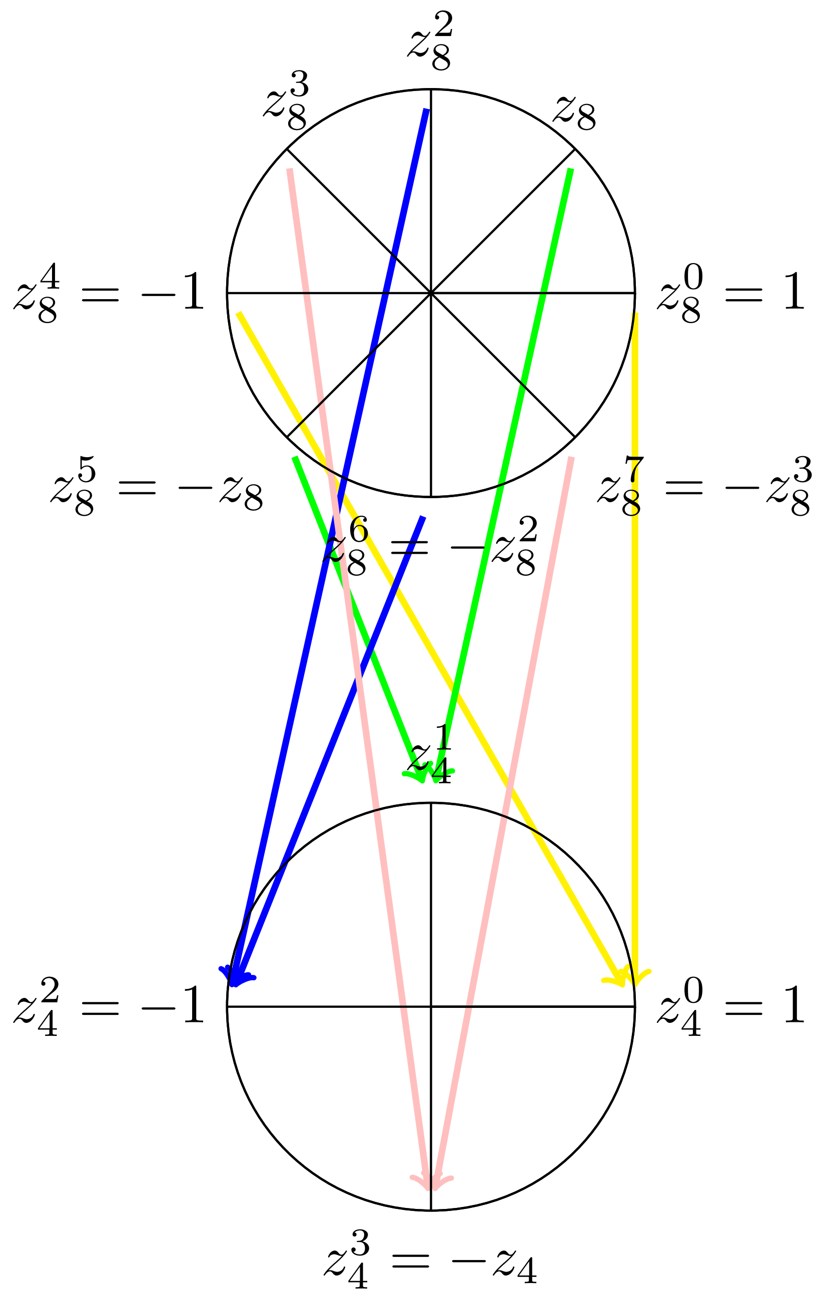
$$F = f_0 + \cdots + f_{n-1}x^{n-1} \mapsto (F(z_n^0), \quad \cdots, F(z_n^{n-1}))$$

is called the Discrete Fourier Transform.

Costs:

- Naive algorithm: $O(n^2)$ operations.
- FFT: $O(n\log(n))$ operations.

Squaring for n even



Squaring for n even

With $n = 2m$, squaring

- sends all nth roots of unity to mth roots;
- z_n^i and $z_n^{i+m} = -z_n^i$ have the same square.

We are setting up a divide-and-conquer for roots of unity.

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.

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

We are setting up a divide-and-conquer for polynomials.

Decomposition and evaluation

To evaluate $F(x)$:

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

To evaluate all $F(x_0), \ldots, F(x_{n-1})$:

- evaluate all $v_i = F_{\text{even}}(x_i^2)$
- evaluate all $v_i' = F_{\text{odd}}(x_i^2)$
- deduce $F(x_i) = v_i + x_i v_i'$.

Fast Fourier Transform

Suppose that the points x_i are nth roots of unity:

$$z_n^0, \quad \ldots, \quad z_n^{n-1},$$

with $n = 2m$. Then, their squares are

$$z_m^0, \quad \ldots, \quad z_m^{m-1}$$

FFT(F, n) $\qquad n = 2^k$

- if $n = 1$, return f_0.
- let $V = \mathrm{FFT}(F_{\mathrm{even}}, n/2)$
- let $V' = \mathrm{FFT}(F_{\mathrm{odd}}, n/2)$
- return $(V[i \bmod n/2] + z_n^i V'[i \bmod n/2] \ : \ 0 \le i < n)$

Master theorem, second version

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

$$T(n) \ \leq \ 2T(\frac{n}{2}) + cn,$$

for n a power of 2.

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

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

- $F(1) = 0$
- $F(n) = 2F(n/2) + 2n$,

so $F(n) = O(n \log(n))$.

Inverse DFT

Prop.

- Performing the inverse DFT in size n is the same thing as
  - performing a DFT at

$$\frac{1}{z_n^0}, \quad \frac{1}{z_n^1}, \quad \dots, \quad \frac{1}{z_n^{n-1}}$$

  - dividing the results by n.

- this new DFT is the same as before:

$$\frac{1}{z_n^i} = z_n^{n-i},$$

  so the outputs are just shuffled.

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

FFT multiplication

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 $\qquad n \leq 2m$
- compute $\mathrm{DFT}(F, n)$ and $\mathrm{DFT}(G, n)$ $\qquad O(n \log(n))$
- multiply the values to get $\mathrm{DFT}(H, n)$ $\qquad O(n)$
- recover H by inverse DFT. $\qquad O(n \log(n))$

Cost: $O(n \log(n)) = O(m \log(m))$.

Why "Fourier Transform"?

In analysis, one uses the continuous Fourier Transform

$$k \mapsto \widehat{f}(k) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i k t} dt.$$

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

$$\begin{aligned} k \mapsto \widehat{f}(k) &= \textstyle\sum_{j=0}^{n-1} f(\frac{j}{n}) e^{\frac{-2\pi i j k}{n}} \\ &= \textstyle\sum_{j=0}^{n-1} f(\frac{j}{n}) \left(e^{\frac{-2\pi i j}{n}}\right)^k \\ &= \textstyle\sum_{j=0}^{n-1} f(\frac{j}{n}) \left(z_n^k\right)^j \\ &= F(z_n^k) \end{aligned}$$

with

$$F(z) = f(0) + f(\frac{1}{n}) z + \cdots + f(\frac{n-1}{n}) z^{n-1}.$$

Multivariate polynomials

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

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

- reduces to univariate polynomial multiplication.

Kronecker's substitution on an example

$$\begin{aligned}
F &= {\color{red}(1+3x_1+4x_1^2)} \;+\; {\color{blue}(22+x_1-x_1^2)x_2} \;+\; {\color{red}(-3-3x_1+2x_1^2)x_2^2} \\
&= {\color{red}F_0(x_1)} \;+\; {\color{blue}F_1(x_1)x_2} \;+\; {\color{red}F_2(x_1)x_2}
\end{aligned}$$

$$\begin{aligned}
G &= {\color{red}(-2+x_1+x_1^2)} \;+\; {\color{blue}(4+x_1+3x_1^2)x_2} \;+\; {\color{red}(3-x_1+x_1^2)x_2^2} \\
&= {\color{red}G_0(x_1)} \;+\; {\color{blue}G_1(x_1)x_2} \;+\; {\color{red}G_2(x_1)x_2}
\end{aligned}$$

Then $H = FG$ is

$$\begin{aligned}
H &= F_0G_0 \\
&+ (F_0G_1 + F_1G_0)x_2 \\
&+ (F_0G_2 + F_1G_1 + F_2G_0)x_2^2 \\
&+ (F_1G_2 + F_2G_1)x_2^2 \\
&+ F_2G_2x_2^2
\end{aligned}$$

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 $\quad 5 = 4 + 1$

$$
\begin{aligned}
F^\star &= (1 + 3x_1 + 4x_1^2) + (22 + x_1 - x_1^2)x_1^5 + (-3 - 3x_1 + 2x_1^2)x_1^{10} \\
&= F_0(x_1) + F_1(x_1)x_1^5 + F_2(x_1)x_1^{10}
\end{aligned}
$$

$$
\begin{aligned}
G^\star &= (-2 + x_1 + x_1^2) + (4 + x_1 + 3x_1^2)x_1^5 + (3 - x_1 + x_1^2)x_1^{10} \\
&= G_0(x_1) + G_1(x_1)x_1^5 + G_2(x_1)x_1^{10}
\end{aligned}
$$

Kronecker's substitution on an example

After multiplying $F^\star$ and $G^\star$:

$$\begin{aligned} H^\star &= F_0G_0 \\ &+ (F_0G_1 + F_1G_0)x_1^5 \\ &+ (F_0G_2 + F_1G_1 + F_2G_0)x_1^{10} \\ &+ (F_1G_2 + F_2G_1)x_1^{15} \\ &+ F_2G_2x_1^{20} \end{aligned}$$

Because $\deg(F_iG_j) \leq 4$, there is no overlap.

So we can directly read off the result.