From univariate polynomials to probabilistically checkable and error-tolerant proofs

Computer Science Club, St Petersburg
17–18 November 2018

Petteri Kaski
Department of Computer Science
Aalto University

- What?
- Why?
- How?

What?

Short synopsis of lectures

- **Polynomials in one variable** are among the most elementary and most useful mathematical objects, with broad-ranging applications from **signal processing** to **error-correcting codes** and advanced applications such as **probabilistically checkable proofs** and **error-tolerant computation**
- One of the main reasons why polynomials are useful in a myriad of applications is that highly efficient algorithms are known for computing with polynomials
- These lectures introduce you to this near-linear-time toolbox and its select applications, with some algorithmic ideas dating back millennia, and some introduced only in the last few years

Lecture 1:
Polynomials in one variable

- We start with elementary computational tasks involving polynomials, such as polynomial addition, multiplication, division (quotient and remainder), greatest common divisor, evaluation, and interpolation
- We observe that polynomials admit two natural representations: coefficient representation and evaluation representation
- We encounter the more-than-2000-year-old algorithm of Euclid for computing a greatest common divisor

Lecture 2:
The fast Fourier transform and fast multiplication

- We derive one of the most fundamental and widely deployed algorithms in all of computing, namely the fast Fourier transform and its inverse
- We explore the consequences of this near-linear-time-computable duality between the coefficient and evaluation representations of a polynomial
- A key consequence is that we can multiply two polynomials in near-linear-time

Lecture 3:
Quotient and remainder; evaluation and interpolation

- We continue the development of the fast polynomial toolbox with near-linear-time polynomial division (quotient and remainder)
- We encounter Newton iteration as the key tool for fast division
- We derive near-linear-time algorithms for batch evaluation and interpolation of polynomials using recursive remaindering along a subproduct tree

Lecture 4: Extended Euclidean algorithm and interpolation from erroneous data

- This lecture culminates our development of the near-linear-time toolbox for univariate polynomials
- First, we develop a divide-and-conquer version of the extended Euclidean algorithm for polynomials that recursively truncates the inputs to achieve near-linear running time
- Second, we present a near-linear-time polynomial interpolation algorithm that is robust to errors in the input data up to the information-theoretic maximum number of errors for correct recovery
- As an application, we encounter Reed–Solomon error-correcting codes together with near-linear-time encoding and decoding algorithms

Lecture 5:
Identity testing and probabilistically checkable proofs

- We investigate some further applications of the near-linear-time toolbox involving randomization in algorithm design and proof systems with probabilistic soundness
- We find that the elementary fact that a low-degree nonzero polynomial has only a small number of roots enables us to (probabilistically) verify the correctness of intricate computations substantially faster than running the computation from scratch
- Furthermore, we observe that proof preparation intrinsically tolerates errors by virtue of Reed–Solomon coding

Why?

Motivation (1/3): A showcase of algorithm design techniques

- The toolbox of near-linear-time algorithms for univariate polynomials and large integers provides a practical showcase of recurrent mathematical ideas in algorithm design such as
  - linearity
  - duality
  - divide-and-conquer
  - dynamic programming
  - iteration and invariants
  - parameterization
  - randomization

Motivation (2/3): Applications

- We gain exposure to a number of classical and recent applications, such as
  - secret-sharing
  - error-correcting codes
  - probabilistically checkable proofs
  - error-tolerant computation

Motivation (3/3): Delegating computation

Client

modest resources
reliable

Problem instance

Solution

Service-provider

massively SIMD-parallel resources
error-prone

- How to verify that the solution is correct ?

- How to design an algorithm to tolerate (a small number of) errors *during computation* ?
- How to convince the client or a third party that the solution is correct ?

How?

Five lectures and problem sets

- Each **lecture** (45 minutes + 45 minutes) reviews the key ideas
- Also learning by doing — a **problem set** of four problems is associated with each lecture; solving the problems is recommended to reach a detailed understanding
- **Lecture slides** available online to accompany the lectures
- **Model solutions** to each problem set available upon request

Lecture schedule

- Today (Saturday):
  - 1. Polynomials in one variable
  - 2. The fast Fourier transform and fast multiplication
- Tomorrow (Sunday):
  - 3. Quotient and remainder; evaluation and interpolation
  - 4. Extended Euclidean algorithm and interpolation from erroneous data
  - 5. Identity testing and probabilistically checkable proofs

1. Polynomials in one variable

Computer Science Club, St Petersburg
17–18 November 2018

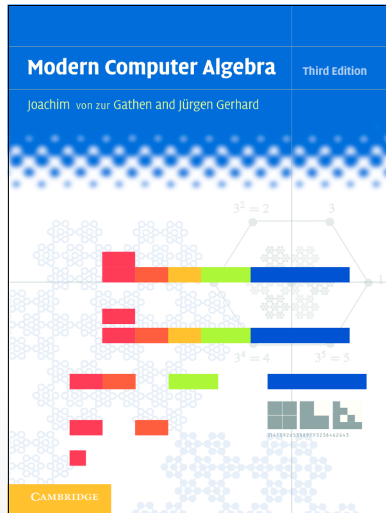
Petteri Kaski
Department of Computer Science
Aalto University

Key content for Lecture 1

- A boot camp of basic concepts and definitions in algebra
- Polynomials in one variable (*univariate* polynomials)
- Basic tasks and first algorithms for univariate polynomials
  - addition
  - multiplication
  - division (quotient and remainder)
  - evaluation
  - interpolation
  - greatest common divisor
- Evaluation–interpolation -duality of polynomials
- The (traditional) extended Euclidean algorithm and its analysis

A boot camp of basic concepts and definitions in algebra

(von zur Gathen and Gerhard [6],
Sections 2.2–3.2, 25.1–4)



Group

- A **group** is a nonempty set G with a binary operation $\cdot : G \times G \to G$ satisfying
  1. for all $a, b, c \in G$ we have $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, (Associativity)
  2. there exists a $1 \in G$ such that $a \cdot 1 = 1 \cdot a = a$ for all $a \in G$, (Identity)
  3. for all $a \in G$ there exists an $a^{-1} \in G$ with $a \cdot a^{-1} = a^{-1} \cdot a = 1$ (Inverses)
- A group G is **commutative** if for all $a, b \in G$ we have $a \cdot b = b \cdot a$
- *Examples*:
  - $(\mathbb{Z}, +, 0)$ and $(\mathbb{Z}_m, +, 0)$ for $m \in \mathbb{Z}_{\geq 2}$ are commutative groups
  - $(\mathbb{Q} \setminus \{0\}, \cdot, 1)$ and $(\mathbb{Z}_m^\times, \cdot, 1)$ for $\mathbb{Z}_m^\times = \{1 \leq a < m : \gcd(a, m) = 1\}$ are commutative groups

(Commutative) ring

- A **ring** R is a set with two binary operations $+ : R \times R \to R$ and $\cdot : R \times R \to R$ satisfying
  1. R together with $+$ is a commutative group with identity 0,
  2. $\cdot$ is associative,
  3. R has an identity element 1 for $\cdot$,
  4. for all $a, b, c \in R$ we have $a(b + c) = (ab) + (ac)$ and $(b + c)a = (ba) + (ca)$
- A ring R is **commutative** if $\cdot$ is commutative
- A ring R is **nontrivial** if $0 \neq 1$
- Unless mentioned otherwise, in what follows we always assume that a ring R is both commutative and nontrivial
- *Examples*:
  $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}_m$ for $m \in \mathbb{Z}_{\geq 2}$

Example: $\mathbb{Z}_5$ (the integers modulo 5)

- One way to represent a (finite) ring is to give the addition and multiplication tables for the operations operations $+$ and $\cdot$
- In the two tables below, the entries at row x column y are $x+y$ and $x \cdot y$, respectively

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

·	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

(1)

Example: $\mathbb{Z}_6$ (the integers modulo 6)

- Below are the addition and multiplication tables for $\mathbb{Z}_6$

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

·	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

(2)

- Compare the *multiplication* tables for $\mathbb{Z}_6$ (above) and $\mathbb{Z}_5$ (see (1)) — what qualitative differences can you spot?

Example: $\mathbb{Z}_{10}$ (the integers modulo 10)

- Here is a yet further example, the integers modulo 10

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

·	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

(3)

- What patterns can you identify from the multiplication table?

Field, unit, associate

- A **unit** in a ring R is an element $u \in R$ for which there exists a multiplicative inverse $v \in R$ with $uv = 1$
- The set $R^\times$ of all units of R is a group under multiplication
- A ring R is a **field** if all nonzero elements of R are units
- *Examples:* (of fields)
  $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}_p$ for p prime
- We say that $a \in R$ is an **associate** of $b \in R$ and write $a \sim b$ if there exists a unit $u \in R$ such that $a = ub$
- $\sim$ is an equivalence relation on R

Examples / work points

- Study the multiplication table for $\mathbb{Z}_5$ in (1)
  — how can you identify which elements are units?
- Based on the units that you identify, conclude that $\mathbb{Z}_5$ is a field
- By studying the multiplication table for $\mathbb{Z}_6$ in (2), conclude that $\mathbb{Z}_6$ is *not* a field by identifying a nonzero element in $\mathbb{Z}_6$ that does not have a multiplicative inverse
- Study (2) and (3). Which elements are units in $\mathbb{Z}_6$? How about in $\mathbb{Z}_{10}$?
- Determine the equivalence classes for the associate relation $\sim$ in $\mathbb{Z}_5$, $\mathbb{Z}_6$, and $\mathbb{Z}_{10}$

Polynomials over a ring (1/2)

- Let R be a ring and let x be a formal indeterminate
- A **polynomial** $a \in R[x]$ in x over R is a finite sequence $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ of elements of R (the **coefficients** of a) which we write as

$$a = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_{n-1} x^{n-1} + \alpha_n x^n = \sum_{i=0}^{n} \alpha_i x^i$$

- A polynomial a is **nonzero** if there exists a $j = 0, 1, \ldots, n$ with $\alpha_j \neq 0$
- For nonzero a, we assume that $\alpha_n \neq 0$ and say that $n = \deg a$ is the **degree** of a; the coefficient $\alpha_n = \mathrm{lc}(a)$ is the **leading coefficient** of a
- For zero a, it is convenient to assume that $a = (0)$ and set $\deg a = -\infty$
- A nonzero polynomial is **monic** if $\mathrm{lc}(a) = 1$

Polynomials over a ring (2/2)

- The set $R[x]$ equipped with the usual notions of addition and multiplication of polynomials (recalled in what follows) is a ring with additive identity (0) and multiplicative identity (1) for $0, 1 \in R$
- As a notational convention when working with polynomials, we use symbols x, y, z, w late in the Roman alphabet for formal indeterminates, and symbols $a, b, c, \ldots, s, t$ early in the Roman alphabet for polynomials
- We use symbols $\alpha, \beta, \gamma, \ldots, \omega$ in the Greek alphabet for elements in R

Complexity of an algorithm

- When studying algorithms that compute with given elements of $R[x]$, we adopt the convention of counting the number of **arithmetic operations** in R as a measure of the "running time" of an algorithm
- Arithmetic operations in R include addition, subtraction, multiplication and taking a multiplicative inverse (of a unit)
- We focus on **worst-case running time** (worst-case number of arithmetic operations in R) as a function of the degree(s) of the input polynomial(s) in $R[x]$
- To avoid degenerate cases, we tacitly assume that all degrees are at least 1 for purposes of running time analysis
- We will work with asymptotic notation $O(\ )$ and $\tilde{O}(\ )$

Addition of polynomials

- Let $a = \sum_i \alpha_i x^i, b = \sum_i \beta_i x^i \in R[x]$ be given as input with $\deg a = n$ and $\deg b = m$
- The sum $c = a + b = \sum_i \gamma_i x^i \in R[x]$ is the polynomial with $\deg c \leq \max(n, m)$ defined for all $i = 0, 1, \ldots, \max(n, m)$ by

$$\gamma_i = \alpha_i + \beta_i \in R$$

- Given a, b as input, it is immediate that we can compute c in $O(\max(n, m))$ operations in R
- Subtraction and multiplication with a given element of R are defined analogously

Multiplication of polynomials

- Let $a = \sum_i \alpha_i x^i, b = \sum_i \beta_i x^i \in R[x]$ be given as input with $\deg a = n$ and $\deg b = m$
- The product $c = ab = \sum_i \gamma_i x^i \in R[x]$ is the polynomial with $\deg c \leq n + m$ defined for all $i = 0, 1, \ldots, n + m$ by

$$\gamma_i = \sum_{j=0}^{i} \alpha_j \beta_{i-j} \in R$$

- Given a, b as input, it is immediate that we can compute c in $O((n + m)^2)$ operations in R
- ... but could we do better? The output consists of only $O(n + m)$ elements of R ...

Polynomial division (quotient and remainder)

- Let $a = \sum_i \alpha_i x^i, b = \sum_i \beta_i x^i \in R[x]$ be given as input with $\deg a = n$, $\deg b = m$, $n \geq m \geq 0$, and suppose that $\beta_m \in R$ is a unit
- We want to compute $q, r \in R[x]$ with $a = qb + r$ and $\deg r < m$
- The classical division algorithm:
  1. $r \leftarrow a, \mu \leftarrow \beta_m^{-1}$
  2. **for** $i = n - m, n - m - 1, \ldots, 0$ **do**
  3. **if** $\deg r = m + i$ **then** $\eta_i \leftarrow \mathrm{lc}(r)\mu, r \leftarrow r - \eta_i x^i b$
     else $\eta_i \leftarrow 0$
  4. **return** $q = \sum_{i=0}^{n-m} \eta_i x^i$ and r
- We leave checking that $a = qb + r$ and $\deg r < m$ as an exercise; given a, b as input, it is immediate that we can compute q, r in $O((n + m)^2)$ operations in R
- ... but could we do better? The output consists of only $O(n + m)$ elements of R ...

Example (quotient and remainder)

- $a = x^4 + x^3 + x^2 + 1 \in \mathbb{Z}_2[x], \quad b = x^2 + 1 \in \mathbb{Z}_2[x]$
- $n = 4, m = 2$
- $\mu = \beta_m^{-1} = 1^{-1} = 1 \in \mathbb{Z}_2$
- Tracing the **for**-loop for $i = n - m, n - m - 1, \ldots, 0$, we have

i	η_i	r
		$x^4 + x^3 + x^2 + 1$
2	1	$x^3 + 1$
1	1	$x + 1$
0	0	$x + 1$

- $q = \eta_2 x^2 + \eta_1 x + \eta_0 = x^2 + x, \quad r = x + 1$

Evaluation (at a single point)

- Let $a = \sum_i \alpha_i x^i \in R[x]$ and $\xi \in R$ be given as input with $\deg a = n$
- We want to compute $a(\xi) = \sum_{i=0}^{n} \alpha_i \xi^i \in R$
- **Horner's rule**:

$$a(\xi) = (\cdots(((\alpha_n \xi + \alpha_{n-1})\xi + \alpha_{n-2})\xi + \alpha_{n-3})\xi + \cdots \alpha_1)\xi + \alpha_0$$

- Using Horner's rule, it takes $O(n)$ operations in R to compute $a(\xi)$

Batch evaluation (at m points)

- Let $a = \sum_i \alpha_i x^i \in R[x]$ and $\xi_1, \xi_2, \ldots, \xi_m \in R$ be given as input with $\deg a = n$
- We want to compute $a(\xi_1), a(\xi_2), \ldots, a(\xi_m) \in R$
- Repeated application of Horner's rule achieves this in $O(mn)$ operations in R
- ... but could we do better yet again? ...

Interpolation

- Let F be a field
- Let distinct $\xi_0, \xi_1, \ldots, \xi_n \in F$ and $\eta_0, \eta_1, \ldots, \eta_n \in F$ be given as input
- We want to compute the unique polynomial $f \in F[x]$ of degree at most n that satisfies

$$f(\xi_0) = \eta_0, \quad f(\xi_1) = \eta_1, \quad \ldots, \quad f(\xi_n) = \eta_n$$

- A classical algorithm (with complexity bounded by a polynomial in n) for this task will be studied in the exercises
- ... but could we do better yet again? ...

Integral domain

- An element $a \in R$ in a ring R is a **zero divisor** if there exists a nonzero $b \in R$ with $ab = 0$
- A ring D is an **integral domain** if there are no nonzero zero divisors
- *Examples*: (of integral domains)
  $\mathbb{Z}$, any field (exercise: units are not zero divisors), $F[x]$ for a field F
- *Work point*:
  Using (1), (2), and (3), determine all zero divisors in $\mathbb{Z}_5$, $\mathbb{Z}_6$, and $\mathbb{Z}_{10}$, respectively

Greatest common divisor

- Let R be a ring and let $a, b \in R$
- We say that a **divides** b and write $a|b$ if there exists a $q \in R$ with $aq = b$
- For $a, b, c \in R$ we say that c is a **greatest common divisor** (or gcd) of a and b if
  1. $c|a$ and $c|b$,
  2. for all $d \in R$ if $d|a$ and $d|b$, then $d|c$
- A greatest common divisor need not exist, and need not be unique
- In an integral domain, any two greatest common divisors are associates

Euclidean domain

- An integral domain E together with a function $d : E \to \mathbb{Z}_{\geq 0} \cup \{-\infty\}$ is a **Euclidean domain** if for all $a, b \in E$ with $b \neq 0$ there exist $q, r \in E$ with $a = qb + r$ and $d(r) < d(b)$
- We say that $q = a \operatorname{quo} b$ is a **quotient** and $r = a \operatorname{rem} b$ a **remainder** in the division of a by b
- We assume that we have available as a subroutine a **division algorithm** that for given $a, b \in E$ with $b \neq 0$ computes $q, r \in E$ with $a = qb + r$ and $d(r) < d(b)$
- *Examples:* (of Euclidean domains)
  - $\mathbb{Z}$ with $d(a) = |a| \in \mathbb{Z}_{\geq 0}$
    - Quotient and remainder can be determined with a division algorithm for integers
  - $F[x]$ for a field F with $d(a) = \deg a$
    - Quotient and remainder can be determined with a division algorithm for polynomials

Traditional Euclidean algorithm

- Let E be an Euclidean domain
- Let $f, g \in E$ be given as input
- We seek to compute a greatest common divisor of f and g
  - Since E is an integral domain, any two greatest common divisors of f and g are related to each other by multiplication with a unit
  - The Euclidean algorithm both (a) shows that greatest common divisors *exist* and (b) gives a way of *computing* a greatest common divisor by iterative remainders
- Traditional Euclidean algorithm:
  1. $r_0 \leftarrow f$, $r_1 \leftarrow g$
  2. $i \leftarrow 1$,
     while $r_i \neq 0$ **do** $r_{i+1} \leftarrow r_{i-1} \operatorname{rem} r_i$, $i \leftarrow i + 1$
  3. **return** r_{i-1} (a greatest common divisor)
- *Why does this algorithm always stop?* (Hint: $d(r_{i+1}) < d(r_i)$)

Traditional extended Euclidean algorithm

- Let $f, g \in E$ be given as input from an Euclidean domain E
- Traditional extended Euclidean algorithm:
  1. $r_0 \leftarrow f$, $s_0 \leftarrow 1$, $t_0 \leftarrow 0$,
     $r_1 \leftarrow g$, $s_1 \leftarrow 0$, $t_1 \leftarrow 1$
  2. $i \leftarrow 1$,
     while $r_i \neq 0$ **do**
        $q_i \leftarrow r_{i-1} \text{ quo } r_i$
        $r_{i+1} \leftarrow r_{i-1} - q_i r_i$
        $s_{i+1} \leftarrow s_{i-1} - q_i s_i$
        $t_{i+1} \leftarrow t_{i-1} - q_i t_i$
        $i \leftarrow i + 1$
  3. $\ell \leftarrow i - 1$
     return ℓ, r_i, s_i, t_i for $i = 0, 1, \ldots, \ell + 1$, and q_i for $i = 1, 2, \ldots, \ell$

Example (over $\mathbb{Z}_2[x]$)

- Let $f = x^5 + x^4 + x^3 + x^2 + x + 1 \in \mathbb{Z}_2[x]$ and $g = x^5 + x^4 + 1 \in \mathbb{Z}_2[x]$
- We obtain

i	r_i	s_i	t_i	q_i
0	$x^5 + x^4 + x^3 + x^2 + x + 1$	1	0	
1	$x^5 + x^4 + 1$	0	1	1
2	$x^3 + x^2 + x$	1	1	$x^2 + 1$
3	$x^2 + x + 1$	$x^2 + 1$	x^2	x
4	0	$x^3 + x + 1$	$x^3 + 1$	

- In particular $\ell = 3$ and $r_\ell = x^2 + x + 1$ is a greatest common divisor of $x^5 + x^4 + x^3 + x^2 + x + 1$ and $x^5 + x^4 + 1$

Analysis using invariants (in the problem set)

- Suppose on input $f, g \in E$ we obtain the output ℓ, r_i, s_i, t_i for $i = 0, 1, \ldots, \ell + 1$, and q_i for $i = 1, 2, \ldots, \ell$
- Introduce the matrices

$$R_0 = \begin{bmatrix} s_0 & t_0 \\ s_1 & t_1 \end{bmatrix} \in E^{2\times 2}, \qquad Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \in E^{2\times 2} \quad \text{for } i = 1, 2, \ldots, \ell,$$

  and $R_i = Q_i Q_{i-1} \cdots Q_1 R_0 \in E^{2\times 2}$ for $i = 0, 1, \ldots, \ell$
- The following invariants hold for all $i = 0, 1, \ldots, \ell$:
  1. $R_i \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix}$.
  2. $R_i = \begin{bmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{bmatrix}$.
  3. r_ℓ is a greatest common divisor of r_i and r_{i+1}.
  4. $s_i f + t_i g = r_i$.

Recap of key content in Lecture 1

- A boot camp of basic concepts and definitions in algebra
- Polynomials in one variable (univariate polynomials)
- Basic tasks and first algorithms for univariate polynomials
  - addition
  - multiplication
  - division (quotient and remainder)
  - evaluation
  - interpolation (exercise)
  - greatest common divisor
- Evaluation–interpolation -duality of polynomials (exercise)
- Analysis of the extended Euclidean algorithm via invariants (exercise)

Problem Set 1 – I

1. Warmup with univariate polynomials over $\mathbb{Z}_2 = \{0, 1\}$.

   (a) Multiply $x + x^2 \in \mathbb{Z}_2[x]$ and $1 + x + x^3 \in \mathbb{Z}_2[x]$.

   (b) Divide $a = 1 + x^2 + x^3 + x^4 + x^6 \in \mathbb{Z}_2[x]$ by $b = 1 + x^3 + x^4 \in \mathbb{Z}_2[x]$. Present a quotient $q \in \mathbb{Z}_2[x]$ and a remainder $r \in \mathbb{Z}_2[x]$ such that $a = qb + r$ and $\deg r < \deg b$.

   Hints: Recall that arithmetic in $\mathbb{Z}_2$ is super-easy. We have $0 + 0 = 1 + 1 = 0$, $0 + 1 = 1 + 0 = 1$, $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0$, $1 \cdot 1 = 1$, and $1^{-1} = 1$. If you want, you can rely on a computer algebra system, or perhaps implement the algorithms from the lecture slides yourself. Make sure that your solutions are correct and the coefficients are reduced to $\{0, 1\}$.

Problem Set 1 – II

2. The traditional extended Euclidean algorithm. Present the complete output of the algorithm (as defined in the lecture slides) in the following two cases.
   (a) Find a greatest common divisor of $f = 1234567$ and $g = 123$ in $\mathbb{Z}$. Using the output of the algorithm, find $g^{-1} \in \mathbb{Z}_f$.
   (b) Find a greatest common divisor of $f = 1 + x + x^3 + x^4$ and $g = 1 + x^4$ in $\mathbb{Z}_2[x]$.

   Hints: You may want to use a computer algebra system to avoid error-prone manual calculations. Make sure to present the complete output of the algorithm. Refer to Problem 4(d) for the second part of 2(a).

Problem Set 1 – III

3. Let $\xi_0, \xi_1, \ldots, \xi_d \in F$ be distinct elements in a field F. Show that the Vandermonde matrix

$$\Xi = \begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix} \in F^{(d+1)\times(d+1)}$$

is invertible.

Hints: For $i = 0, 1, \ldots, d$, define the *Lagrange polynomial* $\ell_i \in F[x]$ by

$$\ell_i = \prod_{\substack{j=0 \\ j\neq i}}^{d} \frac{x - \xi_j}{\xi_i - \xi_j} = \sum_{k=0}^{d} \lambda_{ik} x^k \, .$$

Observe that the polynomial ℓ_i has degree d and is well defined because the values $\xi_0, \xi_1, \ldots, \xi_d$ are distinct. What happens if you evaluate ℓ_i at $x = \xi_j$? Arrange the coefficients λ_{ik} into a matrix. Show that this matrix is the inverse of Ξ.

Problem Set 1 – IV

4. Analysis of the traditional extended Euclidean algorithm. Suppose we run the algorithm on input $f, g \in E$ in an Euclidean domain E, and obtain the output ℓ, r_i, s_i, t_i for $i = 0, 1, \ldots, \ell + 1$, and q_i for $i = 1, 2, \ldots, \ell$. Introduce the matrices

$$R_0 = \begin{bmatrix} s_0 & t_0 \\ s_1 & t_1 \end{bmatrix} \in E^{2\times 2}, \qquad Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \in E^{2\times 2} \quad \text{for } i = 1, 2, \ldots, \ell,$$

and $R_i = Q_i Q_{i-1} \cdots Q_1 R_0 \in E^{2\times 2}$ for $i = 0, 1, \ldots, \ell$.
Show that each of the following invariants holds for all $i = 0, 1, \ldots, \ell$:

(a) $R_i \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix}$.

(b) $R_i = \begin{bmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{bmatrix}$.

(c) r_ℓ is a greatest common divisor of r_i and r_{i+1}.

(d) $s_i f + t_i g = r_i$.

Problem Set 1 – V

Hints: Study the steps of the algorithm as presented in the lecture slides. For (a) and (b), use induction on i. Do not forget to verify the base case. For (c), use (a), $r_{\ell+1} = 0$, and the fact that Q_i is invertible with $Q_i^{-1} = \begin{bmatrix} q_i & 1 \\ 1 & 0 \end{bmatrix}$. For (d), study (a) and (b). It is a good idea to solve Problem 2 first and review that the invariants hold in practice.

2. The fast Fourier transform and fast multiplication

Computer Science Club, St Petersburg
17–18 November 2018

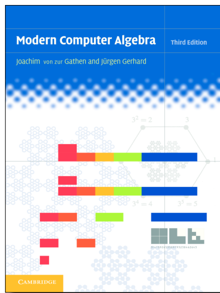
Petteri Kaski
Department of Computer Science
Aalto University

Recap of last lecture

- A boot camp of basic concepts and definitions in algebra
- Polynomials in one variable (univariate polynomials)
- Basic tasks and first algorithms for univariate polynomials
  - addition
  - multiplication
  - division (quotient and remainder)
  - evaluation
  - interpolation
  - greatest common divisor
- Evaluation–interpolation -duality of polynomials
- Analysis of the extended Euclidean algorithm via invariants

Goal: Near-linear-time toolbox for univariate polynomials

- Multiplication (this lecture)
- Division (quotient and remainder)
- Batch evaluation
- Interpolation
- Extended Euclidean algorithm (gcd)
- Interpolation from partly erroneous data



Chapter 5

A NEW ALGORITHM FOR DECODING REED-SOLOMON CODES

Shuhong Gao
Department of Mathematical Sciences
Clemson University,
Clemson, SC 29634-0975, USA.

Abstract A new algorithm is developed for decoding Reed-Solomon codes. It uses fast Fourier transforms and computes the message symbols directly without explicitly finding error locations or error magnitudes. In the decoding radius (up to half of the minimum distance), the new method is easily adapted for error and erasure decoding. It can also detect all errors outside the decoding radius. Compared with the Berlekamp-Massey algorithm, discovered in the late 1960's, the new method seems simpler and more natural yet it has a similar time complexity.

1. Introduction

Reed-Solomon codes are the most popular codes in practical use today with applications ranging from CD players in our living rooms to spacecrafts in deep space exploration. Their main advantage lies in two facts: high capability of correcting both random and burst errors; and existence of efficient decoding algorithm for them, namely the Berlekamp-Massey algorithm, discovered in the late 1960's [1, 9]. The Berlekamp-Massey

Further motivation for this lecture

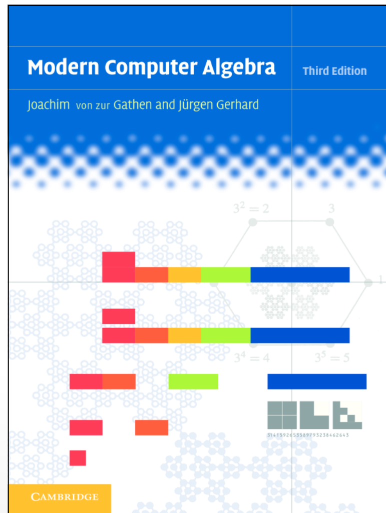
- The fast Fourier transform (FFT) is one of the most widely deployed and useful algorithms in all of computing
- Quick demo (offline):
  FFT and fast polynomial multiplication (fast convolution) over $\mathbb{C}$ in signal processing
- In the exercises we will
  (a) derive an FFT over a ring R endowed with a primitive root of unity ω of order a power of 2, and
  (b) prove a version of the convolution theorem

Key content for Lecture 2

- Evaluation–interpolation duality of polynomials
- Multiplication is a pointwise product in the dual
- Transforming between the primal and a (carefully chosen) dual —**roots of unity** and the **discrete Fourier transform** (DFT)
- Factoring a composite-order DFT to obtain a **fast Fourier transform** (FFT)
- Fast cyclic convolution (assuming a suitable root of unity exists)
- Fast negative-wrapping cyclic convolution (**Schönhage–Strassen** algorithm)

Fast multiplication

(von zur Gathen and Gerhard [6],
Sections 8.2 and 8.3)



Coefficient and evaluation representations

- Let F be a field and let $\xi_0, \xi_1, \ldots, \xi_d \in F$ be distinct
- Let $a = \alpha_0 + \alpha_1 x + \ldots + \alpha_{d-1} x^{d-1} + \alpha_d x^d \in F[x]$ be a polynomial of degree at most d
- We can represent a as a list $(\alpha_0, \alpha_1, \ldots, \alpha_d) \in F^{d+1}$ of $d+1$ **coefficients**
- Alternatively, we can represent a as a list of $d+1$ **values** $(a(\xi_0), a(\xi_1), \ldots, a(\xi_d)) \in F^{d+1}$
- Indeed, we have

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{bmatrix} = \begin{bmatrix} a(\xi_0) \\ a(\xi_1) \\ \vdots \\ a(\xi_d) \end{bmatrix}$$

  and the left-hand side Vandermonde matrix is invertible over F (recall exercise from last problem set)

Evaluation and interpolation

- To **evaluate** a polynomial $(\alpha_0, \alpha_1, \ldots, \alpha_d) \in F^{d+1}$ at distinct points $\xi_0, \xi_1, \ldots, \xi_d \in F$, we multiply from the left with the Vandermonde matrix:

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{bmatrix} = \begin{bmatrix} a(\xi_0) \\ a(\xi_1) \\ \vdots \\ a(\xi_d) \end{bmatrix}$$

- To **interpolate** the coefficients of a polynomial with values $(a(\xi_0), a(\xi_1), \ldots, a(\xi_d)) \in F^{d+1}$ at distinct $\xi_0, \xi_1, \ldots, \xi_d \in F$, we multiply from the left with the inverse of the Vandermonde matrix:

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix}^{-1} \begin{bmatrix} a(\xi_0) \\ a(\xi_1) \\ \vdots \\ a(\xi_d) \end{bmatrix} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{bmatrix}$$

Example (evaluation)

- Let us evaluate $a = 1 + 2x + 3x^2 + 4x^3 + 5x^4 \in \mathbb{Z}_{13}$ at $\xi_0 = 0$, $\xi_1 = 1$, $\xi_2 = 2$, $\xi_3 = 3$, $\xi_4 = 4$ in $\mathbb{Z}_{13}$
- We have

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 3 \\ 1 & 3 & 9 & 1 & 3 \\ 1 & 4 & 3 & 12 & 9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 12 \\ 1 \\ 7 \end{bmatrix}$$

  and hence $a(0) = 1$, $a(1) = 2$, $a(3) = 12$, $a(4) = 1$, $a(5) = 7$

Example (interpolation)

- Let us interpolate the coefficients of the unique polynomial $a \in \mathbb{Z}_{13}$ of degree at most 4 with values $a(\xi_0) = 1$, $a(\xi_1) = 2$, $a(\xi_2) = 12$, $a(\xi_3) = 1$, $a(\xi_4) = 7$ at $\xi_0 = 0$, $\xi_1 = 1$, $\xi_2 = 2$, $\xi_3 = 3$, $\xi_4 = 4$ in $\mathbb{Z}_{13}$
- We have

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 3 \\ 1 & 3 & 9 & 1 & 3 \\ 1 & 4 & 3 & 12 & 9 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 2 \\ 12 \\ 1 \\ 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 12 & 4 & 10 & 10 & 3 \\ 2 & 0 & 8 & 2 & 1 \\ 5 & 8 & 11 & 12 & 3 \\ 6 & 2 & 10 & 2 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 12 \\ 1 \\ 7 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

  and hence $\alpha_0 = 1$, $\alpha_1 = 2$, $\alpha_2 = 3$, $\alpha_3 = 4$, $\alpha_4 = 5$
- The inverse of the Vandermonde matrix can be computed e.g. by Gaussian elimination or by using Lagrange polynomials (recall exercise in last problem set)

Evaluation–interpolation duality

- Evaluation–interpolation constitutes and example of two dual representations (coefficient and value representations of a polynomial)
- Both representations uniquely identify the object (the polynomial) under consideration
- In many cases one can make use of **duality** in algorithm design. Often a problem has a corresponding **dual** problem that is obtainable from the original (the **primal**) problem by means of an easy transformation. The primal and dual control each other, enabling an algorithm designer to use the interplay between the two representations

Evaluation–interpolation duality

- *Often a* **problem** *has a corresponding* **dual** *problem that is obtainable from the original (the* **primal***) problem by means of an easy* **transformation**
  - Polynomial multiplication (primal):
    Given coefficients of a and b as input, output coefficients of ab
  - Polynomial multiplication (dual):
    Given evaluations of a and b as input, output evaluations of ab
  - Transformation:
    Evaluation (primal $\rightarrow$ dual), interpolation (dual $\rightarrow$ primal)
- *The primal and dual control each other, enabling an algorithm designer to use the interplay between the two representations ...*

Multiplication is easy in the dual

- Polynomial multiplication (dual):
  Given evaluations of a and b as input, output evaluations of ab
- Suppose $\deg a \leq n$ and $\deg b \leq m$
- Then $\deg ab \leq n + m$ and $n + m + 1$ evaluations of ab suffice to uniquely determine ab
- So suppose $\xi_0, \xi_1, \ldots, \xi_{n+m} \in F$ are distinct and we have the evaluations $a(\xi_0), a(\xi_1), \ldots, a(\xi_{n+m}) \in F$ and $b(\xi_0), b(\xi_1), \ldots, b(\xi_{n+m}) \in F$
- Then,

$$
\begin{aligned}
ab(\xi_0) &= a(\xi_0)b(\xi_0), \\
ab(\xi_1) &= a(\xi_1)b(\xi_1), \\
&\vdots \\
ab(\xi_{n+m}) &= a(\xi_{n+m})b(\xi_{n+m}) \in F
\end{aligned}
$$

- Thus, $O(n + m)$ multiplications suffice to determine ab, assuming we are in the dual

Multiplication in primal representation

- Polynomial multiplication (primal):
  Given coefficients of a and b as input, output coefficients of ab
- For $\deg a \le d$ and $\deg b \le d$, the classical algorithm uses $O(d^2)$ operations
- *The primal and dual control each other, enabling and algorithm designer to use the interplay between the two representations ...*
- Idea:
  1. Transform the inputs a and b into dual representation
  2. Multiply in the dual
  3. Transform the product ab back to primal representation
- Needed:
  Fast transformation between primal and dual representations

Fast evaluation and interpolation?

- To **evaluate** a polynomial $(\alpha_0, \alpha_1, \ldots, \alpha_d) \in F^{d+1}$ at distinct points $\xi_0, \xi_1, \ldots, \xi_d \in F$, we multiply from the left with the Vandermonde matrix:

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{bmatrix} = \begin{bmatrix} a(\xi_0) \\ a(\xi_1) \\ \vdots \\ a(\xi_d) \end{bmatrix}$$

- To **interpolate** the coefficients of a polynomial with values $(a(\xi_0), a(\xi_1), \ldots, a(\xi_d)) \in F^{d+1}$ at distinct $\xi_0, \xi_1, \ldots, \xi_d \in F$, we multiply from the left with the inverse of the Vandermonde matrix:

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix}^{-1} \begin{bmatrix} a(\xi_0) \\ a(\xi_1) \\ \vdots \\ a(\xi_d) \end{bmatrix} = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_d \end{bmatrix}$$

Fast evaluation and interpolation?

- It is too expensive to construct the Vandermonde matrix (or its inverse) in explicit form
- Indeed, the matrix has $(d+1)^2$ elements in F, so working with the matrix in explicit form yields no better algorithms than classical multiplication in the primal representation
- For multiplication, both the input and the output use only $O(d)$ elements of F
- We have the freedom to choose any distinct $\xi_0, \xi_1, \ldots, \xi_d \in F$
- Perhaps a *good choice* enables evaluation and interpolation in $\tilde{O}(d)$ operations without constructing the Vandermonde matrix explicitly ...

Fast evaluation and interpolation?

- Idea:
  Choose
  $$\xi_0 = \omega^0, \quad \xi_1 = \omega^1, \quad \xi_2 = \omega^2, \quad \ldots, \quad \xi_d = \omega^d$$
  for a carefully chosen element $\omega \in F$ (whose existence depends on F)

- Intuition:
  With such a choice, the Vandermonde matrix should have a great deal of useful algebraic structure that maybe enables the use of, say, divide-and-conquer for matrix–vector multiplication, without even explicitly constructing the matrix ...

Key content (revisited)

- Evaluation–interpolation duality of polynomials
- Multiplication is a pointwise product in the dual
- Transforming between the primal and a (carefully chosen) dual —**roots of unity** and the **discrete Fourier transform** (DFT)
- Factoring a composite-order DFT to obtain a **fast Fourier transform** (FFT)
- Fast cyclic convolution (assuming a suitable root of unity exists)
- Fast negative-wrapping cyclic convolution (**Schönhage–Strassen** algorithm)

Roots of unity

- Let R be a ring (recall that we tacitly assume that R is commutative with $0 \neq 1$)
- For $n \in \mathbb{Z}_{\geq 1}$ and $\omega \in R$, we say that ω is a **root of unity** of **order** n in R if $\omega^n = 1$

The discrete Fourier transform (DFT)

- Let ω be a root of unity of order n in R and let

$$f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_{n-1} x^{n-1} \in R[x]$$

- The n-**point discrete Fourier transform of** f **at** ω is the vector of evaluations

$$\mathrm{DFT}_\omega(f) = \hat{f} = (f(\omega^0), f(\omega^1), \ldots, f(\omega^{n-1})) \in R^n .$$

- Equivalently, we may view $\mathrm{DFT}_\omega : f \mapsto \hat{f}$ as the R-linear map that takes the vector of coefficients $f = (\varphi_0, \varphi_1, \ldots, \varphi_{n-1}) \in R^n$ to the vector $\hat{f} = (\hat{\varphi}_0, \hat{\varphi}_1, \ldots, \hat{\varphi}_{n-1}) \in R^n$ defined for all $i = 0, 1, \ldots, n-1$ by

$$\hat{\varphi}_i = \sum_{j=0}^{n-1} \varphi_j \omega^{ij}$$

Towards a first FFT: Splitting into even and odd parts

- Suppose that $n \in \mathbb{Z}_{\geq 2}$ is even and let $f = \sum_{i=0}^{n-1} \varphi_i x^i \in R[x]$
- Introduce the two polynomials

$$f_{\text{even}} = \sum_{i=0}^{n/2-2} \varphi_{2i} x^i \in R[x], \qquad f_{\text{odd}} = \sum_{i=0}^{n/2-2} \varphi_{2i+1} x^i \in R[x]$$

- We observe that

$$f(x) = f_{\text{even}}(x^2) + x \cdot f_{\text{odd}}(x^2)$$

- Here f has degree at most $n-1$,
  whereas f_{even} and f_{odd} have degree at most $n/2-1$

Towards a first FFT: Evaluating at a root of unity of order n

- Let $n \in \mathbb{Z}_{\geq 2}$ be even and $f = \sum_{i=0}^{n-1} \varphi_i x^i,\ f_{\text{even}} = \sum_{i=0}^{n/2-2} \varphi_{2i} x^i,\ f_{\text{odd}} = \sum_{i=0}^{n/2-2} \varphi_{2i+1} x^i$
- We recall that

$$f(x) = f_{\text{even}}(x^2) + x \cdot f_{\text{odd}}(x^2) \tag{4}$$

- Let $\omega \in R$ be a root of unity of order n; that is, $\omega^n = 1$
- We want to compute $f(\omega^0), f(\omega^1), \ldots, f(\omega^{n-1})$; that is, $\mathrm{DFT}_\omega(f)$
- From (4) and $\omega^n = 1$ we have that it suffices to first compute

$$\begin{array}{ccc} f_{\text{even}}(\omega^0), f_{\text{even}}(\omega^2), \ldots, f_{\text{even}}(\omega^{2n-2}) & \sim & f_{\text{even}}(\omega^0), f_{\text{even}}(\omega^2), \ldots, f_{\text{even}}(\omega^{n-2}) \\ f_{\text{odd}}(\omega^0), f_{\text{odd}}(\omega^2), \ldots, f_{\text{odd}}(\omega^{2n-2}) & \sim & \underbrace{f_{\text{odd}}(\omega^0), f_{\text{odd}}(\omega^2), \ldots, f_{\text{odd}}(\omega^{n-2})}_{\text{That is, } \mathrm{DFT}_{\omega^2}(f_{\text{even}}) \text{ and } \mathrm{DFT}_{\omega^2}(f_{\text{odd}})} \end{array}$$

and then do $O(n)$ arithmetic operations in R

A first FFT: Recursion and analysis

- We just saw that to compute the n-point $\mathrm{DFT}_\omega(f)$, it suffices to
  1. split f into the even part f_{even} and the odd part f_{odd}
  2. compute the $n/2$-point $\mathrm{DFT}_{\omega^2}(f_{\mathrm{even}})$,
  3. compute the $n/2$-point $\mathrm{DFT}_{\omega^2}(f_{\mathrm{odd}})$, and
  4. do $O(n)$ further arithmetic operations in R to recover $\mathrm{DFT}_\omega(f)$
- That is, the total number of arithmetic operations is $T(n) \le 2 \cdot T(n/2) + O(n)$
- This is $T(n) = O(n \log_2 n)$ when $n = 2^k$ for $k \in \mathbb{Z}_{\ge 1}$ and we apply recursion

Primitive root of unity

- A root of unity $\omega \in R$ of order n is **primitive** if for any prime divisor t of n it holds that $\omega^{n/t} - 1$ is not a zero divisor in R
- *Examples:*
  - $\omega_n = \exp(2\pi i/n)$ is a primitive root of unity of order n in $\mathbb{C}$
  - 2 is a primitive root of unity of order 12 in $\mathbb{Z}_{13}$
  - For

k	29	71	75	95	108	123
ω	21	287	149	55	64	493

we have that $p = k \cdot 2^{57} + 1$ is a prime and $\omega \in \mathbb{Z}_p$ is the least primitive root of unity of order 2^{57} in $\mathbb{Z}_p$

Properties of primitive roots of unity

Lemma 1

Let ω be a primitive root of unity of order n in R.
Then, for all integers s not divisible by n it holds that

(i) $\omega^s - 1$ *is not a zero divisor in* R, *and*

(ii) $\sum_{i=0}^{n-1} \omega^{is} = 0$

Lemma 2

Let ω be a primitive root of unity of order n in R.
Then, ω^a is a primitive root of unity of order $|n/a|$ in R for all divisors a of n

Proof of Lemma 1 I

- For any $\rho \in R$ and any $k \in \mathbb{Z}_{\geq 0}$, we have

$$(\rho - 1) \sum_{j=0}^{k-1} \rho^j = \rho^k - 1 \tag{5}$$

- Select any $s \in \mathbb{Z}$ that is not divisible by n. Since $\omega^n = 1$, we may assume $s = 1, 2, \ldots, n - 1$
- Let $1 \leq g \leq s$ be the gcd of s and n with $us + vn = g$ for $u \in \mathbb{Z}_{\geq 0}$ and $v \in \mathbb{Z}_{\leq 0}$
- Since $s < n$, we can choose a prime divisor t of n so that g divides n/t
- Take $\rho = \omega^g$ and $k = n/(gt)$ in (5) to obtain that $\omega^g - 1$ divides $\omega^{n/t} - 1$. That is, $(\omega^g - 1)\gamma = \omega^{n/t} - 1$ for some $\gamma \in R$

Proof of Lemma 1 II

- Thus $\omega^g - 1$ cannot be a zero divisor since if it were, we could conclude that $0 = 0 \cdot \gamma = (\omega^g - 1)\beta\gamma = (\omega^{n/t} - 1)\beta$ for a nonzero $\beta \in R$ and hence $\omega^{n/t} - 1$ would be a zero divisor, a contradiction
- Take $\rho = \omega^s$ and $k = u$ in (5) to obtain that $\omega^s - 1$ divides $\omega^{us} - 1 = \omega^{us+vn} - 1 = \omega^g - 1$
- Thus, $\omega^s - 1$ cannot be a zero divisor since if it were, we could conclude that $\omega^g - 1$ is a zero divisor, a contradiction
- Take $\rho = \omega^s$ and $k = n$ in (5) to obtain $(\omega^s - 1)\sum_{i=0}^{n-1} \omega^{is} = \omega^{ns} - 1 = 1 - 1 = 0$
- Since $\omega^s - 1$ is not a zero divisor, we conclude that $\sum_{i=0}^{n-1} \omega^{is} = 0$ $\square$

Proof of Lemma 2

- Let a be a divisor of n
- For $|a| = n$ we observe that $\omega^n = \omega^{-n} = 1$ and hence $\omega^a = 1$ is trivially a primitive root of unity of order 1
- Suppose that $|a| < n$ and let t be any prime divisor of $|n/a| > 1$
- Then, $s = a|n/a|/t$ is not divisible by n and hence Lemma 1 implies that $\omega^s - 1 = (\omega^a)^{|n/a|/t} - 1$ is not a zero divisor
- Since t was arbitrary, ω^a is a primitive root of unity of order $|n/a|$ in R □

The inverse discrete Fourier transform (inverse DFT)

Lemma 3

Suppose that n *is a unit in* R *and let* $\omega \in R$ *be a primitive root of order* n*. Then,*
$\mathrm{DFT}_{\omega}^{-1} = \frac{1}{n} \cdot \mathrm{DFT}_{\omega^{-1}}$

Proof.

It suffices to show that for all $k = 0, 1, \ldots, n-1$ we have

$$\varphi_k = \frac{1}{n} \sum_{i=0}^{n-1} \hat{\varphi}_i \omega^{-ik}$$

Take $s = j - k$ in Lemma 1 to conclude that

$$\frac{1}{n} \sum_{i=0}^{n-1} \hat{\varphi}_i \omega^{-ik} = \frac{1}{n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \varphi_j \omega^{ij} \omega^{-ik} = \sum_{j=0}^{n-1} \varphi_j \cdot \frac{1}{n} \sum_{i=0}^{n-1} \omega^{i(j-k)} = \varphi_k$$

□

Beyond the first (radix-2) FFT

- The rest of the lecture goes beyond the first recursive derivation of a (radix-2) FFT where we assumed that
  1. the ring R has a (primitive) root of unity ω of order $n = 2^k$, and
  2. we are content with a recursive implementation
- The rest of the lecture contains more advanced material that shows how to (a) unfold the recursion into a sequence of linear transformations suitable e.g. for parallel implementation; and (b) work with rings that do not have a suitable root of unity
- This more advanced material is not necessary for successfully following the rest of the course

The positional number system (base B)

- Let $B \in \mathbb{Z}_{\geq 2}$
- Suppose that $\alpha \in \mathbb{Z}$ with $0 \leq \alpha \leq B^d - 1$ for some $d \in \mathbb{Z}_{\geq 0}$
- Then, there is a unique finite sequence

$$(\alpha_{d-1}, \alpha_{d-2}, \ldots, \alpha_1, \alpha_0) \in \mathbb{Z}_{\geq 0}^{d} \tag{6}$$

with $0 \leq \alpha_i \leq B - 1$ for all $i = 0, 1, \ldots, d - 1$ such that

$$\alpha = \sum_{i=0}^{d-1} \alpha_i B^i = \alpha_{d-1} B^{d-1} + \alpha_{d-2} B^{d-2} + \ldots + \alpha_2 B^2 + \alpha_1 B + \alpha_0 \tag{7}$$

- We say that the sequence (6) is the (d-digit) representation of the integer α in the positional number system with **base** B (or **radix** B)
- The elements α_i are the **digits** of α
- We say that α_{d-1} is the **most significant** digit and α_0 is the **least significant** digit

Example (base 10)

- Let us represent $123 \in \mathbb{Z}$ in base $B = 10$
- We have

$$123 = 1 \cdot 10^2 + 2 \cdot 10 + 3 \cdot 1$$

- Hence, the sequence $(1, 2, 3)$ represents 123 in base 10
- *Question/work point:*
  Given a representation in base B as input, how do you compute a representation in base C? Hint: quotient and remainder.

The positional number system (varying base)

- Let $B_{d-1}, B_{d-2}, \ldots, B_1, B_0 \in \mathbb{Z}_{\geq 2}$
- Suppose that $\alpha \in \mathbb{Z}$ with $0 \leq \alpha \leq B_{d-1}B_{d-2}\cdots B_1B_0 - 1$
- Then, there is a unique finite sequence

$$(\alpha_{d-1}, \alpha_{d-2}, \ldots, \alpha_1, \alpha_0) \in \mathbb{Z}^d_{\geq 0} \tag{8}$$

  with $0 \leq \alpha_i \leq B_i - 1$ for all $i = 0, 1, \ldots, d-1$ such that

$$\begin{aligned} \alpha &= \sum_{i=0}^{d-1} \alpha_i B_{i-1}B_{i-2}\cdots B_0 \\ &= \alpha_{d-1}B_{d-2}B_{d-3}\cdots B_0 + \ldots + \alpha_2B_1B_0 + \alpha_1B_0 + \alpha_0 \end{aligned} \tag{9}$$

- We say that the sequence (8) is the representation of the integer α in the positional number system with (varying) **base** $(B_{d-2}, B_{d-1}, \ldots, B_1, B_0)$

Example (varying base)

- Let us represent $123 \in \mathbb{Z}$ in base $(9, 8, 7)$. We have

$$123 = 2 \cdot 8 \cdot 7 + 1 \cdot 7 + 4 \cdot 1$$

- Thus, the representation of 123 in base $(9, 8, 7)$ is $(2, 1, 4)$
- *Question/work point:*
  Given a representation in base $(B_{d-1}, B_{d-2}, \ldots, B_1, B_0)$ as input, how do you compute a representation in base $(C_{e-1}, C_{e-2}, \ldots, C_1, C_0)$? Hint: quotient and remainder.

Factoring a composite-order DFT (1/3)

- Let ω be a primitive root of unity of composite order $n = st$ in R for integers $s, t \geq 2$
- We can view an index $k \in \{0, 1, \ldots, st - 1\}$ as a varying-base integer $k = k_s t + k_t$ with $k_s \in \{0, 1, \ldots, s - 1\}$ and $k_t \in \{0, 1, \ldots, t - 1\}$
- That is, k_s and k_t are the digits of k in base (s, t) so that k_s is the most significant digit and k_t is the least significant digit
- Recall that for all $i = 0, 1, \ldots, st - 1$ we have

$$\hat{\varphi}_i = \sum_{j=0}^{st-1} \varphi_j \omega^{ij}$$

- Let us expand the output index i in base (s, t) and the input index j in base (t, s)
- We have

$$\hat{\varphi}_{i_s t + i_t} = \sum_{j_s=0}^{s-1} \sum_{j_t=0}^{t-1} \varphi_{j_t s + j_s} \omega^{(i_s t + i_t)(j_t s + j_s)}$$

Factoring a composite-order DFT (2/3)

- Expand and use the fact that $\omega^{st} = 1$ to obtain

$$
\begin{aligned}
\hat{\varphi}_{i_s t + i_t} &= \sum_{j_s=0}^{s-1} \sum_{j_t=0}^{t-1} \varphi_{j_t s + j_s} \omega^{(i_s t + i_t)(j_t s + j_s)} \\
&= \sum_{j_s=0}^{s-1} \sum_{j_t=0}^{t-1} \varphi_{j_t s + j_s} \omega^{i_s j_t s t + i_s j_s t + i_t j_t s + i_t j_s} \\
&= \sum_{j_s=0}^{s-1} \sum_{j_t=0}^{t-1} \varphi_{j_t s + j_s} \omega^{i_s j_s t} \omega^{i_t j_t s} \omega^{i_t j_s} \\
&= \underbrace{\sum_{j_s=0}^{s-1} \left(\omega^{t}\right)^{i_s j_s} \underbrace{\omega^{i_t j_s} \underbrace{\sum_{j_t=0}^{t-1} \varphi_{j_t s + j_s} \left(\omega^{s}\right)^{i_t j_t}}_{\text{(i)}}}_{\text{(ii)}}}_{\text{(iii)}} \,.
\end{aligned}
$$

Factoring a composite-order DFT (3/3)

- Let us study (i), (ii), and (iii) as the indices i_s, i_t, j_s, j_t range over their domains:

$$\hat{\varphi}_{i_s t + i_t} = \underbrace{\sum_{j_s=0}^{s-1} \left(\omega^t\right)^{i_s j_s} \underbrace{\omega^{i_t j_s} \underbrace{\sum_{j_t=0}^{t-1} \varphi_{j_t s + j_s} \left(\omega^s\right)^{i_t j_t}}_{\text{(i)}}}_{\text{(ii)}}}_{\text{(iii)}}$$

- Part (i) takes the $t \times s$ input f and outputs the $t \times s$ array obtained by taking the t-point discrete Fourier transform at ω^s for each of the s columns of f
- Part (ii) multiplies the resulting $t \times s$ array entrywise (Hadamard product) with the $t \times s$ Vandermonde matrix with entries $\omega^{i_t j_s}$
- Part (iii) takes as input the $t \times s$ array output by (ii) and outputs the $t \times s$ array obtained by taking the s-point discrete Fourier transform at ω^t for each of the t rows of the array
- Finally, transpose the $t \times s$ array to obtain the $s \times t$ output $\hat{f}$

Fast Fourier transform (FFT)

- Idea:
  Apply the previous factorization recursively for smooth n
- For example, suppose that $n = 2^k$ for $k \in \mathbb{Z}_{\geq 1}$
- Take $s = 2^{k-1}$ and $t = 2$
- Compute Parts (i) and (ii) explicitly, and apply the factorization recursively in Part (iii)
- Thus, the DFT at a primitive root of unity ω of order 2^k in R can be computed in $T(2^k) \leq 2T(2^{k-1}) + O(2^k)$ operations in R (exercise)
- In particular, $T(n) = O(n \log n)$

Factors in an FFT (1/4)

- Let us now look at a possible implementation of Parts (i), (ii), and (iii) in more detail
- Let ω be a primitive root of order $n \in \mathbb{Z}_{\geq 1}$ in R and let $n = pqr$ for $p, q, r \in \mathbb{Z}_{\geq 1}$
- We will study two types of transformations that take as input an array $a \in R^n$ and produce as output an array $b \in R^n$
- We assume that the entries $a[i] \in R$ of an array are indexed with $i = 0, 1, \ldots, n-1$
- Let $w \in R^n$ be an array with $w[i] = \omega^i$ for all $i = 0, 1, \ldots, n-1$ (in an implementation this array can be precomputed with $O(n)$ operations in R)

Factors in an FFT (2/4)

- The first transformation $\Phi_{(p,q,r)} : R^n \to R^n$ sets

$$b[iqr + jr + k] = \sum_{\ell=0}^{q-1} w[(j\ell pr) \text{ rem } n] a[iqr + \ell r + k] \tag{10}$$

  for all $i \in \{0, 1, \ldots, p-1\}$, $j \in \{0, 1, \ldots, q-1\}$, $k \in \{0, 1, \ldots, r-1\}$
- Observe that the transformation relies on integers in base (p, q, r) for indexing the input a and the output b
- Also observe that the transformation implements pr disjoint copies of a q-point DFT, using in total $O(pq^2 r) = O(nq)$ operations in R
- This transformation can be used to implement Parts (i) and (iii)

Factors in an FFT (3/4)

- The second transformation $\Theta_{(p,q,r)} : R^n \to R^n$ sets
$$b[iqr + jr + k] = w[jkp]a[iqr + jr + k] \tag{11}$$
for all $i \in \{0, 1, \ldots, p-1\}$, $j \in \{0, 1, \ldots, q-1\}$, $k \in \{0, 1, \ldots, r-1\}$
- Again we observe that we work in base (p, q, r)
- This transformation runs in $O(pqr) = O(n)$ operations in R
- This transformation can be used to implement Part (ii)

Factors in an FFT (4/4)

- A naïve implementation of Parts (iii), (ii), and (i) would now implement an n-point DFT for $n = st$ and input $f \in R^n$ as a sequence of three transformations, read from right to left, $\Phi_{(1,t,s)}\Theta_{(1,s,t)}\Phi_{(1,s,t)}(f)$
- This must be followed by transposition of the resulting array from $t \times s$ to $s \times t$ to obtain the output $\hat{f}$ (*Why?*)
- However, this does not yet reduce the number of operations to $O(n \log n)$ (*Why?*)
- In an implementation with $n = 2^k$, one can fix $q = 2$ and proceed with a sequence of $2k - 1$ transformations, with $p = 2^j$ and $r = 2^{k-1-j}$ for $j = 0, 1, \ldots k - 1$ (completing the details are an exercise), followed by final permutation of the resulting array

Remarks

- The previous example gave one possibility to implement an FFT
- In general the term "fast Fourier transform" refers to a family of algorithms that rely on factoring an $n \times n$ Vandermonde matrix $\Omega = (\omega^{ij} : i, j = 0, 1, \ldots, n-1)$ for a composite n into a sequence of simpler (sparse) matrices such that matrix–vector multiplication with each matrix in the sequence is cheap to execute
- For example, you may want to view the transformations (10) and (11) as obtaining the vector b by multiplying a matrix with the vector a
- Van Loan [12] gives an extensive treatment of computational frameworks for the FFT

Key content (revisited)

- Evaluation–interpolation duality of polynomials
- Multiplication is a pointwise product in the dual
- Transforming between the primal and a (carefully chosen) dual —**roots of unity** and the **discrete Fourier transform** (DFT)
- Factoring a composite-order DFT to obtain a **fast Fourier transform** (FFT)
- Fast cyclic convolution (assuming a suitable root of unity exists)
- Fast negative-wrapping cyclic convolution (**Schönhage–Strassen** algorithm)

Ideal, principal ideal

- Let R be a ring
- A nonempty subset I of R is an **ideal** if
  1. for all $a, b \in I$ we have $a + b \in I$, and
  2. for all $a \in I$ and $r \in R$ we have $ar \in I$
- *Examples:*
  - For any $a \in R$ we have that $\langle a \rangle = aR = \{ar : r \in R\}$ is an ideal; we say that $\langle a \rangle$ is the **principal ideal** generated by $a \in R$
  - For any $n \in \mathbb{Z}$, we observe that $n\mathbb{Z} = \{\ldots, -2n, -n, 0, n, 2n, \ldots\}$ is a (principal) ideal of $\mathbb{Z}$

Congruence modulo an ideal, residue class

- Let I be an ideal of R
- We say that $r, s \in R$ are **congruent modulo** I and write $r \equiv s \pmod{I}$ if $r - s \in I$
- For $r \in R$ we say that the set $r + I = \{r + a : a \in I\}$ is a **residue class modulo** I
- For all $r, s \in R$ we have

$$r + I = s + I \quad \Leftrightarrow \quad r - s \in I \quad \Leftrightarrow \quad r \equiv s \pmod{I}$$

Residue class ring (factor ring)

- Let I be an ideal of R
- The set $R/I = \{r + I : r \in R\}$ of all residue classes modulo I is a ring (the **factor ring** or **residue class ring** of R modulo I) if we define the ring operations for all $r, s \in R$ by

$$(r + I) + (s + I) = (r + s) + I$$

  and

$$(r + I)(s + I) = (rs) + I$$

- Observe in particular that the aforementioned operations are well-defined in the sense that they do not depend on the choices of representatives r, s for residue classes modulo I (exercise)
- *Example*:
  For $R = \mathbb{Z}$ and $I = n\mathbb{Z}$ with $n \in \mathbb{Z}_{\geq 1}$, we have $R/I = \mathbb{Z}/n\mathbb{Z} \cong \mathbb{Z}_n$

Example: Cyclic convolution

- Let R be a ring and let $n \in \mathbb{Z}_{\geq 1}$
- Consider the factor ring $R[x]/\langle x^n - 1 \rangle$
- We may view the elements of $R[x]/\langle x^n - 1 \rangle$ as polynomials of degree at most $n - 1$ in $R[x]$
- Addition and multiplication in $R[x]/\langle x^n - 1 \rangle$ are as in $R[x]$, with the exception that after multiplication we simplify the result with the substitution $x^n = 1$

Example: Cyclic convolution

- Suppose that $n = 8$ and that $R = \mathbb{Z}_{17}$
- Let us multiply the following two polynomials in $R[x]/\langle x^n - 1\rangle$

$$f = 1 + 8x + 13x^2 + 16x^3 + 15x^4 + 6x^5 + 7x^6 + 10x^7$$
$$g = 4 + 3x + 16x^2 + 7x^3 + 6x^4 + 11x^5 + 9x^6 + 15x^7$$

- In $R[x]$, the product is

$$fg = 4 + x + 7x^2 + 4x^4 + 16x^5 + 12x^6 + 10x^7 + 7x^8 + x^9 + 9x^{10} + 8x^{11} + 8x^{12} + 8x^{13} + 14x^{14}$$

- In $R[x]/\langle x^n - 1\rangle$, we can first compute fg in $R[x]$ as above, reduce the result with the substitution $x^n = 1$, and then simplify to obtain the result in $R[x]/\langle x^n - 1\rangle$ (or, what is the same, first multiply in $R[x]$ and take the remainder in the division with $x^n - 1$):

$$\begin{aligned} fg &= 4 + x + 7x^2 + 4x^4 + 16x^5 + 12x^6 + 10x^7 + 7 + x + 9x^2 + 8x^3 + 8x^4 + 8x^5 + 14x^6 \\ &= 11 + 2x + 16x^2 + 8x^3 + 12x^4 + 7x^5 + 9x^6 + 10x^7 \end{aligned}$$

Cyclic convolution via the DFT

- Let R be a ring, let $n \in \mathbb{Z}_{\geq 1}$, and let $\omega \in R$ be a primitive root of unity of order n
- For two vectors $a = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) \in R^n$ and $b = (\beta_0, \beta_1, \ldots, \beta_{n-1}) \in R^n$, let us write $a \cdot b$ for the pointwise product $a \cdot b = (\alpha_0\beta_0, \alpha_1\beta_1, \ldots, \alpha_{n-1}\beta_{n-1}) \in R^n$

Theorem 4 (Convolution Theorem)

For all $f, g \in R[x]/\langle x^n - 1\rangle$ *we have* $\mathrm{DFT}_\omega(fg) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$

Proof.

Exercise. □

- Furthermore, if n is a unit in R, we have $fg = \frac{1}{n}\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g))$
- This enables fast algorithms for computing fg assuming (i) R admits a suitable primitive root of unity, (ii) n is a unit in R, and (iii) n is divisible enough to enable an FFT by divide and conquer

Example: Cyclic convolution via the DFT (1/2)

- Suppose that $n = 8$ and that $R = \mathbb{Z}_{17}$
- We observe that $\omega = 2$ is a primitive root of unity of order $n = 8$ in $R = \mathbb{Z}_{17}$; indeed,

$$\omega^0 = 1,\ \omega^1 = 2,\ \omega^2 = 4,\ \omega^3 = 8,\ \omega^4 = 16,\ \omega^5 = 15,\ \omega^6 = 13,\ \omega^7 = 9,\ \omega^8 = 1$$

- We also observe that $2^{-1} = 9 \in R = \mathbb{Z}_{17}$ and hence $8^{-1} = 2^{-3} = 15 \in R = \mathbb{Z}_{17}$
- Using the DFT, let us multiply the following two polynomials in $R[x]/\langle x^n - 1\rangle$

$$f = 1 + 8x + 13x^2 + 16x^3 + 15x^4 + 6x^5 + 7x^6 + 10x^7$$
$$g = 4 + 3x + 16x^2 + 7x^3 + 6x^4 + 11x^5 + 9x^6 + 15x^7$$

- First we compute the n-point DFTs of f and g at ω to obtain

$$\mathrm{DFT}_\omega(f) = (8, 11, 16, 7, 13, 9, 10, 2)$$
$$\mathrm{DFT}_\omega(g) = (3, 14, 4, 9, 16, 4, 0, 16)$$

Example: Cyclic convolution via the DFT (2/2)

- Next we take the pointwise product of the DFTs

$$\mathrm{DFT}_\omega(f) = (8, 11, 16, 7, 13, 9, 10, 2) \in R^n$$

$$\mathrm{DFT}_\omega(g) = (3, 14, 4, 9, 16, 4, 0, 16) \in R^n$$

to obtain

$$\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g) = (7, 1, 13, 12, 4, 2, 0, 15) \in R^n$$

- Finally take the inverse n-point DFT to obtain the result

$$\frac{1}{n}\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)) = (11, 2, 16, 8, 12, 7, 9, 10) \in R^n$$

or what is the same as a polynomial

$$fg = 11 + 2x + 16x^2 + 8x^3 + 12x^4 + 7x^5 + 9x^6 + 10x^7$$

Remarks (1/2)

- Cyclic convolution via the DFT (when implemented with FFTs) can be used to multiply polynomials in $R[x]$ fast
- Indeed, simply choose a large enough n so that the degree of the product of the two polynomials is less than n
- In this situation we can multiply two polynomials using $O(n \log n)$ operations in R; contrast this with the classical $O(n^2)$ operations
- Caveat:
  This approach needs (i) that R is endowed with a primitive root of unity ω of order n and (ii) that n is a unit in R
- *So what to do when R does not meet (i) and (ii) ?*

Remarks (2/2)

- Next we will look at a multiplication algorithm, *Schönhage–Strassen multiplication* [10], that needs very light assumptions about the coefficient ring
- Our present exposition roughly follows the exposition of a polynomial version of the Schönhage–Strassen algorithm in von zur Gathen and Gerhard [6, Section 8.3]
- For convenience in what follows, let us write S instead of R for the coefficient ring, and y instead of x for the polynomial indeterminate
- Rather than relying on cyclic convolution, the algorithm will rely on the following notion of *negative-wrapping* cyclic convolution ...

Example: Negative-wrapping cyclic convolution

- Let S be a ring and let $n \in \mathbb{Z}_{\geq 1}$
- Consider the factor ring $S[y]/\langle y^n + 1 \rangle$
- We may view the elements of $S[y]/\langle y^n + 1 \rangle$ as polynomials of degree at most $n - 1$ in $S[y]$
- Addition and multiplication in $S[y]/\langle y^n + 1 \rangle$ are as in $S[y]$, with the exception that after multiplication we simplify the result with the substitution $y^n = -1$

Example: Negative-wrapping cyclic convolution

- Suppose that $n = 8$ and that $S = \mathbb{Z}_5$
- Let us multiply the following two polynomials in $S[y]/\langle y^n + 1\rangle$

$$
\begin{aligned}
f &= 1 + 2y + 2y^2 + 4y^3 + 3y^4 + 4y^5 + 2y^6 + 3y^7 \\
g &= 3 + 2y + 4y^2 + y^4 + 4y^5 + y^6 + 2y^7
\end{aligned}
$$

- In $S[y]$, the product is

$$
fg = 3 + 3y + 4y^2 + 4y^3 + y^4 + 2y^6 + 4y^8 + y^9 + 4y^{10} + y^{11} + 2y^{12} + 2y^{13} + y^{14}
$$

- In $S[y]/\langle y^n + 1\rangle$, we can first compute fg in $S[y]$ as above, reduce the result with the substitution $y^n = -1$, and then simplify to obtain the result in $S[y]/\langle y^n + 1\rangle$ (or, what is the same, first multiply in $S[y]$ and take the remainder in the division with $y^n + 1$):

$$
\begin{aligned}
fg &= 3 + 3y + 4y^2 + 4y^3 + y^4 + 2y^6 - 4 - y - 4y^2 - y^3 - 2y^4 - 2y^5 - y^6 \\
&= 4 + 2y + 3y^3 + 4y^4 + 3y^5 + y^6
\end{aligned}
$$

Schönhage–Strassen multiplication (1/7)

- Let S be a ring
- Suppose that 2 is a unit in S (this is the only assumption we make about S)
- Let $n = 2^k$ for some $k \in \mathbb{Z}_{\geq 3}$
- Let $f, g \in S[y]/\langle y^n + 1\rangle$ be given as input
- We want to compute the product $fg \in S[y]/\langle y^n + 1\rangle$
- With foresight, let $m = 2^{\lfloor k/2 \rfloor}$ and $t = 2^{\lceil k/2 \rceil}$; in particular, we have $n = mt$ and $m \leq t \leq 2m$
- The key idea is to reduce one multiplication in $S[y]/\langle y^{mt} + 1\rangle$ into t multiplications in $S[y]/\langle y^{2m} + 1\rangle$ and then apply recursion

Running example (1/8)

- It will be convenient to illustrate the algorithm design with a running example
- Let us work with $S = \mathbb{Z}_5$; in particular we observe that 2 is a unit with inverse $2^{-1} = 3 \in \mathbb{Z}_5$
- Suppose that $n = 8$ and that our given input in $S[y]/\langle y^n + 1\rangle$ is

$$f = 1 + 2y + 2y^2 + 4y^3 + 3y^4 + 4y^5 + 2y^6 + 3y^7$$
$$g = 3 + 2y + 4y^2 + y^4 + 4y^5 + y^6 + 2y^7$$

- We need to produce the output

$$fg = 4 + 2y + 3y^3 + 4y^4 + 3y^5 + y^6$$

- Since $n = 8$, we have $m = 2$ and $t = 4$

Schönhage–Strassen multiplication (2/7)

- Let us introduce a new indeterminate x and transform f and g so that every monomial y^k is replaced with $x^q y^r$ where q and r are the unique nonnegative integers with $k = qm + r$ and $0 \leq r < m$
- Let us write F and G for the resulting two-variable polynomials in $S[x, y]$
- Let $Q, H \in S[x, y]$ be the unique polynomials such that

$$FG = (x^t + 1)Q + H \tag{12}$$

  and H has x-degree at most $t - 1$
- We observe that Q, H above exist by polynomial division (e.g. recall Lecture 1) since the leading coefficient of $x^t + 1$ is a unit in $S[y]$ with $(S[y])[x] \cong S[x, y]$
- (It should be noted that the actual algorithm never constructs Q in explicit form, here we merely use polynomial division to conclude that Q *exists*.)

Running example (2/8)

- Continuing the running example, we have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$ and the inputs

$$f = 1 + 2y + 2y^2 + 4y^3 + 3y^4 + 4y^5 + 2y^6 + 3y^7$$
$$g = 3 + 2y + 4y^2 + y^4 + 4y^5 + y^6 + 2y^7$$

- Substituting $y^m = x$ to f and g, the polynomials F and G in $S[x, y]$ are

$$F = 1 + 2y + (2 + 4y)x + (3 + 4y)x^2 + (2 + 3y)x^3$$
$$G = 3 + 2y + 4x + (1 + 4y)x^2 + (1 + 2y)x^3$$

- For illustration, let us also display the polynomials FG, Q, and H, but also observe that FG and Q are not computed by the algorithm, and the polynomial H will be obtained only later

$$FG = 3 + 3y + 4y^2 + (4y + 3y^2)x + (3 + y^2)x^2 + (1 + y^2)x^3 + (3 + y + 4y^2)x^4 + yx^5 + (2 + 2y + y^2)x^6$$
$$Q = 3 + y + 4y^2 + yx + (2 + 2y + y^2)x^2$$
$$H = 2y + (3y + 3y^2)x + (1 + 3y)x^2 + (1 + y^2)x^3$$

Schönhage–Strassen multiplication (3/7)

- Substitute $x = y^m$ to both sides of (12) to conclude that
$$F(y^m, y)G(y^m, y) = (y^{mt} + 1)Q(y^m, y) + H(y^m, y)$$
implying
$$F(y^m, y)G(y^m, y) \equiv H(y^m, y) \pmod{y^{mt} + 1}$$
- Since $f = F(y^m, y)$ and $g = G(y^m, y)$, we conclude that it suffices to compute $H(y^m, y)$ to determine the product fg in $S[y]/\langle y^{mt+1} \rangle$
- Indeed, $H(y^m, y)$ is a polynomial in y with degree less than $2mt$, which is easily reduced with the substitution $y^{mt} = -1$ to yield the result fg
- We observe that (12) implies $FG \equiv H \pmod{x^t + 1}$, so our goal in what follows will be to multiply given F and G modulo $x^t + 1$

Running example (3/8)

- Continuing the running example, we have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$ and

$$F = 1 + 2y + (2 + 4y)x + (3 + 4y)x^2 + (2 + 3y)x^3$$
$$G = 3 + 2y + \qquad 4x + (1 + 4y)x^2 + (1 + 2y)x^3$$

- Let us also recall that

$$H = 2y + (3y + 3y^2)x + (1 + 3y)x^2 + (1 + y^2)x^3$$

- Thus, substituting $y^m = x$ into H, we obtain

$$H(y^m, y) = 2y + 3y^3 + 4y^4 + 3y^5 + y^6 + y^8$$

- Substituting $y^{mt} = -1$ into $H(y^m, y)$, we obtain the desired output

$$fg = 4 + 2y + 3y^3 + 4y^4 + 3y^5 + y^6$$

Schönhage–Strassen multiplication (4/7)

- By construction, F and G both have y-degree less than m, so FG has y-degree less than $2m$
- We may thus work with $(S[y]/\langle y^{2m}+1\rangle)[x]$ in place of $S[x,y]$ when computing FG from given F and G
- Accordingly, let $R = S[y]/\langle y^{2m}+1\rangle$
- Restating our goal from the previous slide, given $F, G \in R[x]$ as input, we seek to compute a $H \in R[x]$ of x-degree at most $t-1$ such that there is a $Q \in R[x]$ with $FG = (x^t+1)Q + H$

Schönhage–Strassen multiplication (5/7)

- Next we want to reduce our goal from multiplying modulo $x^t + 1$ to multiplying modulo $x^t - 1$, since the latter can be implemented with cyclic convolution
- Toward this end, it will be useful to have a primitive root of unity of order $2t$ in R; here is where our foresight in the choice of the parameters m and t will pay off
- First, observe that y is a primitive root of unity of order $4m$ in $R = S[y]/\langle y^{2m} + 1\rangle$: indeed, since $y^{2m} \equiv -1 \pmod{y^{2m} + 1}$ holds and 2 is a unit in S by assumption, we observe that $y^{2m} - 1 \equiv -2 \pmod{y^{2m} + 1}$ is a unit in R and hence cannot be a zero divisor in R
- Since m and t are positive integer powers of 2 with $t \leq 2m$, we have that

  $$\eta = y^{2m/t}$$

  is a primitive root of order $2t$ in R by Lemma 2

Running example (4/8)

- Continuing the running example, we have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$
- Accordingly, in $R = S[y]/\langle y^{2m} + 1 \rangle$ we have that $\eta = y^{2m/t} = y$ is a primitive root of order $2t$
- Indeed, in R we have

$$\eta^0 = 1,\ \eta^1 = y,\ \eta^2 = y^2,\ \eta^3 = y^3,\ \eta^4 = -1,\ \eta^5 = -y,\ \eta^6 = -y^2,\ \eta^7 = -y^3,\ \eta^8 = 1$$

Schönhage–Strassen multiplication (6/7)

- Given $F, G \in R[x]$ as input, we seek to compute a $H \in R[x]$ of x-degree at most $t-1$ such that there is a $Q \in R[x]$ with
$$FG = (x^t + 1)Q + H \tag{13}$$
- Observing that $\eta^t = -1$ in R and substituting ηx in place of x in (13), we have, in $R[x]$,
$$\begin{aligned} F(\eta x)G(\eta x) &= ((\eta x)^t + 1)Q(\eta x) + H(\eta x) \\ &= (-x^t + 1)Q(\eta x) + H(\eta x) \\ &= (x^t - 1)\tilde{Q}(\eta x) + H(\eta x) \end{aligned}$$
- That is, we have $F(\eta x)G(\eta x) = H(\eta x)$ in $R[x]/\langle x^t - 1\rangle$
- In particular, we can use cyclic convolution and the FFT at the primitive root of unity $\omega = \eta^2$ of order t in R to multiply $F(\eta x)$ and $G(\eta x)$ in $R[x]/\langle x^t - 1\rangle$ to obtain $H(\eta x)$
- Substituting $\eta^{-1}x$ in place of x in $H(\eta x)$ yields our desired result H in $R[x]$

Running example (5/8)

- Continuing the running example, we have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$ and

$$\begin{aligned} F &= 1 + 2y + (2+4y)x + (3+4y)x^2 + (2+3y)x^3 \\ G &= 3 + 2y + 4x + (1+4y)x^2 + (1+2y)x^3 \\ H &= 2y + (3y+3y^2)x + (1+3y)x^2 + (1+y^2)x^3 \end{aligned}$$

- Recalling that $\eta = y$ in $R = S[y]/\langle y^{2m}+1\rangle$ for our chosen parameters, in $R[x]$ we have

$$\begin{aligned} F(\eta x) &= 1 + 2y + (2y+4y^2)x + (3y^2+4y^3)x^2 + (2+2y^3)x^3 \\ G(\eta x) &= 3 + 2y + 4yx + (y^2+4y^3)x^2 + (3+y^3)x^3 \\ H(\eta x) &= 2y + (3y^2+3y^3)x + (y^2+3y^3)x^2 + (4y+y^3)x^3 \end{aligned}$$

- In particular, observe how substituting ηx in place of x in F, G, H cyclically shifts the coefficients (polynomials in y) with negative wrapping because $y^{2m} = -1$ in R

Schönhage–Strassen multiplication (7/7)

- Let us now summarize the algorithm in one slide

1. To multiply $f, g \in S[y]/\langle y^{mt} + 1\rangle$, construct $F, G \in R[x]$ with $R = S[y]/\langle y^{2m} + 1\rangle$ from f and g by introducing a new indeterminate x and substituting $y^m = x$
2. Let $\eta = y^{2m/t} \in R$ and substitute ηx in place of x to obtain $F(\eta x), G(\eta x) \in R[x]$
3. Compute the product $F(\eta x)G(\eta x) = H(\eta x) \in R[x]/\langle x^t - 1\rangle$ via cyclic convolution

   $$H(\eta x) = \frac{1}{t}\,\mathrm{DFT}_{\omega^{-1}}\Big(\mathrm{DFT}_{\omega}(F(\eta x)) \cdot \mathrm{DFT}_{\omega}(G(\eta x))\Big)$$

   using t-point fast Fourier transforms at the primitive root $\omega = \eta^2$ of order t in R

   [[This leads to t recursive multiplications in $R = S[y]/\langle y^{2m} + 1\rangle$ when taking the pointwise product $\cdot$ above.]]
4. Substitute $\eta^{-1}x$ in place of x in $H(\eta x)$ to obtain H
5. Substitute $x = y^m$ and $y^{mt} = -1$ in H to obtain the output $fg \in S[y]/\langle y^{mt} + 1\rangle$

Running example (6/8)

- Let us illustrate the execution of the algorithm in our running example
- We have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$ and the input

$$\begin{aligned} f &= 1 + 2y + 2y^2 + 4y^3 + 3y^4 + 4y^5 + 2y^6 + 3y^7 \\ g &= 3 + 2y + 4y^2 + y^4 + 4y^5 + y^6 + 2y^7 \end{aligned}$$

1. Substituting $y^m = x$, we construct the polynomials

$$\begin{aligned} F &= 1 + 2y + (2 + 4y)x + (3 + 4y)x^2 + (2 + 3y)x^3 \\ G &= 3 + 2y + 4x + (1 + 4y)x^2 + (1 + 2y)x^3 \end{aligned}$$

2. Substituting ηx in place of x, we obtain

$$\begin{aligned} F(\eta x) &= 1 + 2y + (2y + 4y^2)x + (3y^2 + 4y^3)x^2 + (2 + 2y^3)x^3 \\ G(\eta x) &= 3 + 2y + 4yx + (y^2 + 4y^3)x^2 + (3 + y^3)x^3 \end{aligned}$$

Running example (7/8)

- We have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$

3. Taking the t-point fast Fourier transforms at $\omega = \eta^2$, we obtain

$$\mathrm{DFT}_\omega(F(\eta x)) = (3 + 4y + 2y^2 + y^3, 2 + 4y + 3y^3, 4 + 4y^2 + 2y^3, 4y^2 + 4y^3)$$
$$\mathrm{DFT}_\omega(G(\eta x)) = (1 + y + y^2, 3 + 3y + y^2, 3y + y^2 + 3y^3, 3 + y + 2y^2 + 2y^3)$$

This leads to t recursive multiplications in $R = S/\langle y^{2m} + 1\rangle$ as follows

$$(3 + 4y + 2y^2 + y^3)(1 + y + y^2) = y + 4y^2 + 2y^3$$
$$(2 + 4y + 3y^3)(3 + 3y + y^2) = 2 + 4y^2 + 3y^3$$
$$(4 + 4y^2 + 2y^3)(3y + y^2 + 3y^3) = 3y + 3y^2 + 4y^3$$
$$(4y^2 + 4y^3)(3 + y + 2y^2 + 2y^3) = 3 + 4y + 4y^2 + y^3$$

That is, we obtain the pointwise product

$$\mathrm{DFT}_\omega(F(\eta x))\cdot\mathrm{DFT}_\omega(G(\eta x)) = (y+4y^2+2y^3, 2+4y^2+3y^3, 3y+3y^2+4y^3, 3+4y+4y^2+y^3)$$

Running example (8/8)

- We have $S = \mathbb{Z}_5$, $n = 8$, $m = 2$, $t = 4$

3. (continued)
Taking the t-point inverse FFT, we obtain
$$\frac{1}{t}\,\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_{\omega}(F(\eta x)) \cdot \mathrm{DFT}_{\omega}(G(\eta x))) = (2y, 3y^2 + 3y^3, y^2 + 3y^3, 4y + y^3)$$
or what is the same as a polynomial in two variables
$$H(\eta x) = 2y + (3y^2 + 3y^3)x + (y^2 + 3y^3)x^2 + (4y + y^3)x^3$$

4. Substituting $\eta^{-1}x$ in place of x in $H(\eta x)$, we obtain
$$H = 2y + (3y + 3y^2)x + (1 + 3y)x^2 + (1 + y^2)x^3$$

5. Finally, we substitute $x = y^m$ and $y^{mt} - 1$ in H to obtain the output
$$fg = 4 + 2y + 3y^3 + 4y^4 + 3y^5 + y^6$$

Implementation remarks (1/3)

- In an implementation, we can represent a polynomial $f \in S[y]/\langle y^{mt} + 1 \rangle$ as an array consisting of mt elements of S
- Accordingly, we can represent a polynomial $F \in (S[y]/\langle y^{2m} + 1 \rangle)[x]/\langle x^t - 1 \rangle$ as an array of length $2mt$ that has been (tacitly) partitioned into t segments, with each segment consisting of $2m$ elements of S
- That is, each segment represents a coefficient in $R = S[y]/\langle y^{2m} + 1 \rangle$ and the t segments together represent a polynomial in $R[x]/\langle x^t - 1 \rangle$

Implementation remarks (2/3)

- Multiplication with powers of y in $R = S[y]/\langle y^{2m} + 1 \rangle$ is easy: we just cyclically shift the list of coefficients of a polynomial in y by as many places as is indicated by the power of y, taking care to adjust the sign of the coefficient in case of wrap-arounds
- Accordingly, multiplication and substitution with powers of η are similarly negative-wrapping cyclic shifts
- In particular, fast Fourier transforms at $\omega = \eta^2$ over $R = S[y]/\langle y^{2m} + 1 \rangle$ similarly amount to additions and negative-wrapping cyclic shifts

Implementation remarks (3/3)

- To build F from f, we (i) view f as a collection of t segments of length m each, and (ii) pad each segment with m zeros of S so that each segment has length $2m$
- Multiplication and substitution with a power of η rotates each segment cyclically (with negative wrapping since $y^{2m} = -1$)
- Recursive multiplications in R operate on pairs of segments
- To build fg from H, we compress back from length $2mt$ to length mt so that each of the mt elements of fg becomes a (signed) sum of 2 elements of H as determined by the substitutions $x = y^m$ and $y^{tm} = -1$

Analysis (1/3)

- For $n = 2^k$ with $k \in \mathbb{Z}_{\geq 0}$, we claim that Schönhage-Strassen multiplication runs in $O(n \log n \log \log n)$ operations in S for two inputs $f, g \in S[y]/\langle y^n + 1 \rangle$ given in coefficient representation
- Recalling that $t = 2^{\lceil k/2 \rceil}$ and $m = 2^{\lfloor k/2 \rfloor}$ with $n = mt \geq 8$, it suffices to analyse the recurrence

$$T(n) \leq tT(2m) + Cn \log_2 n \tag{14}$$

  with $T(1), T(2), T(4) \leq D$ where C and D are constants independent of n
- Indeed, for an input of size $n \geq 8$, the algorithm makes t recursive calls on inputs of size $2m < n$ and does at most $Cn \log_2 n$ work (operations in S) to prepare the recursive calls and to prepare the result based on the return values of the calls

Analysis (2/3)

- Let us reparameterize (14) in terms of k to obtain, for all $k \geq 3$,

$$T(k) \leq 2^{\lceil k/2 \rceil} T(\lfloor k/2 \rfloor + 1) + C \cdot 2^k k \tag{15}$$

- For all nonnegative integers k we have

$$\lfloor k/2 \rfloor + \lceil k/2 \rceil = k, \quad \lfloor (k+1)/2 \rfloor = \lceil k/2 \rceil, \quad \text{and} \quad \lceil (k+1)/2 \rceil = \lfloor k/2 \rfloor + 1 \tag{16}$$

- From (16) we have that (15) is equivalent to, for all $k \geq 2$,

$$T(k+1) \leq 2^{\lfloor k/2 \rfloor + 1} T(\lceil k/2 \rceil + 1) + C \cdot 2^{k+1}(k+1) \tag{17}$$

- For convenience, let us substitute $T(k+1) = 2^k(k-1)L(k)$ to (17) and divide by $2^k(k-1)$ on both sides to obtain the equivalent form, for all $k \geq 2$,

$$L(k) \leq \frac{2(\lceil k/2 \rceil - 1)}{k-1} L(\lceil k/2 \rceil) + \frac{2C(k+1)}{k-1} \tag{18}$$

Analysis (3/3)

- From (18) we obtain that for all $k \geq 2$ we have
$$L(k) \leq L(\lceil k/2 \rceil) + 6C \tag{19}$$
- Now let us observe that at most $\log_2 3k$ iterations of the map $k \mapsto \lceil k/2 \rceil$ suffice to reach the value 1 starting from any positive integer k
- Indeed, the map $k \mapsto \lceil k/2 \rceil$ is dominated by the map $k \mapsto \lfloor k/2 \rfloor + 1$, which can be viewed as right-shifting k (viewed as an integer in base 2 representation) by one bit position and then incrementing the result
- When iterating $k \mapsto \lfloor k/2 \rfloor + 1$, the increments in total contribute at most the least power of 2 at least k (which is at most $2k$), so at most $\log_2 3k$ right-shifts and increments suffice to reach the value 1
- In particular, iterating (19), we obtain, for all $k \geq 2$,
$$L(k) \leq L(\lceil k/2 \rceil) + 6C \leq L(\lceil \lceil k/2 \rceil /2 \rceil) + 12C \leq \cdots \leq L(1) + 6C \log_2 3k = O(\log k)$$
and for $k \geq 3$ thus $T(k) = 2^{k-1}(k-2)L(k-1) = O(2^k k \log k)$

Further remarks

- Using Schönhage–Strassen multiplication, we can multiply two polynomials of degree at most n in $O(n \log n \log \log n)$ operations in the coefficient ring S, provided that 2 is a unit in S
- With some extra work, the assumption that 2 is a unit in S can be lifted to obtain a multiplication algorithm that works over any coefficient ring S in $O(n \log n \log \log n)$ operations; cf. von zur Gathen and Gerhard [6, Section 8.3, Exercises 8.29 and 8.30] and Schönhage [9]
- Cf. also the "three-primes" FFT algorithm for integer multiplication on 64-bit hardware [6, Section 8.3]

Recap of key content for Lecture 2

- Evaluation–interpolation duality of polynomials
- Multiplication is a pointwise product in the dual
- Transforming between the primal and a (carefully chosen) dual —**roots of unity** and the **discrete Fourier transform** (DFT)
- Factoring a composite-order DFT to obtain a **fast Fourier transform** (FFT)
- Fast cyclic convolution (assuming a suitable root of unity exists)
- Fast negative-wrapping cyclic convolution (**Schönhage–Strassen** algorithm)

Problem Set 2 – I

1. Multiplication with the discrete Fourier transform. Let us multiply $f = 1 + x + x^2 \in \mathbb{Z}_{13}[x]$ and $g = 2 + 12x^3 \in \mathbb{Z}_{13}[x]$ using the discrete Fourier transform.
   (a) Compute $\mathrm{DFT}_\omega(f)$ and $\mathrm{DFT}_\omega(g)$ in $\mathbb{Z}_{13}^6$ utilizing the fact that $\omega = 4$ is a primitive root of unity of order 6 in $\mathbb{Z}_{13}$.
   (b) Compute the pointwise product $\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g) \in \mathbb{Z}_{13}^6$.
   (c) Compute the inverse $\frac{1}{6}\,\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)) \in \mathbb{Z}_{13}^6$.

   Hints: To ease your computations, we have $4^{-1} = 10 \in \mathbb{Z}_{13}$ and $6^{-1} = 11 \in \mathbb{Z}_{13}$. Check that your result for (c) agrees with the sequence of coefficients of $fg \in \mathbb{Z}_{13}[x]$.

Problem Set 2 – II

2. The convolution identity. Let $\omega \in R$ be a primitive root of unity of order n in a ring R. Show that for all $f, g \in R[x]/\langle x^n - 1\rangle$ we have $\mathrm{DFT}_\omega(fg) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$.

   Hints: Recall that we may view f and g as elements of $R[x]$ of degree at most $n-1$, in which case we obtain $fg \in R[x]/\langle x^n - 1\rangle$ by multiplying f and g in $R[x]$ and then substituting $x^n = 1$ until the result has degree at most $n-1$. Recalling that $\omega^n = 1$, show that the vectors on the left-hand side and the right-hand side of the identity agree in each position.

Problem Set 2 – III

3. The fast Fourier transform. Let $\omega \in R$ be a primitive root of unity of order $n = 2^k$ in a ring R with $k \in \mathbb{Z}_{\geq 0}$. Present detailed pseudocode for an algorithm that given

$$f = (\varphi_0, \varphi_1, \ldots, \varphi_{n-1}) \in R^n$$

as input computes the discrete Fourier transform

$$\mathrm{DFT}_\omega(f) = (\hat{\varphi}_0, \hat{\varphi}_1, \ldots, \hat{\varphi}_{n-1}) \in R^n$$

in $O(n \log_2 n)$ arithmetic operations in R. Carefully analyse the number of arithmetic operations in R that your algorithm uses.

Hints: You may want to consider a recursive design that relies on the fact that $n = 2^k$ for some $k \in \mathbb{Z}_{\geq 0}$. Consult the factorization of a composite-order DFT into parts (i), (ii), and (iii) in the lecture slides. Remember to set up base cases for the recursion. You may set up the recursion, for example, by factoring the order as $n = st \geq 4$ with s and t equal to powers of 2. Also, you may want to precompute powers of ω into a look-up table so that they are immediately available. If you want to test your design, you can

Problem Set 2 – IV

make use of the fact that $\omega = 19$ is a primitive root of unity of order 32 in $\mathbb{Z}_{97}$.
Compare the output of your algorithm with a reference output obtained by multiplying the input with an appropriate Vandermonde matrix.

Problem Set 2 – V

4. Fast integer multiplication by reduction to polynomial multiplication. Let $\alpha, \beta \in \mathbb{Z}_{\geq 1}$ with $\lfloor \log_2 \alpha \rfloor + 1 \leq m$ and $\lfloor \log_2 \beta \rfloor + 1 \leq m$ be given as input. Furthermore, let us assume that α and β are represented in binary as sequences of 64-bit words. That is, we have $\alpha = \sum_{i=0}^{\lfloor m/64 \rfloor} \alpha_i \cdot 2^{64i}$ and $\beta = \sum_{i=0}^{\lfloor m/64 \rfloor} \beta_i \cdot 2^{64i}$ with $\alpha_i, \beta_i \in \mathbb{Z}$ and $0 \leq \alpha_i, \beta_i \leq 2^{64} - 1$. Design an algorithm that computes the product $\gamma = \alpha\beta$ represented as a sequence of 64-bit words using $\tilde{O}(m)$ operations in $\mathbb{Z}_{2^{128}}$.

   Hints: You may want to apply the Schönhage–Strassen algorithm from the lecture slides. View α and β as polynomials $a = \sum_{i=0}^{\lfloor m/64 \rfloor} \alpha_i y^i$ and $b = \sum_{i=0}^{\lfloor m/64 \rfloor} \beta_i y^i$ in a polynomial ring $S[y]$ for a carefully chosen coefficient ring S. Maybe you want to try $S = \mathbb{Z}_u$ for some $u \in \mathbb{Z}_{\geq 2}$. Suppose you have access to the polynomial product $c = ab$. How do you recover from c the sequence of words that represents γ? Be careful with carries in addition. How does the size of S depend on m? Observe also that 2 must be a unit in S if you want to apply Schönhage–Strassen, so this somewhat limits your choice for u. Carefully justify that the number of operations in $\mathbb{Z}_{2^{128}}$ used by your algorithm is $O(m(\log m)^d)$ for some constant d independent of m. You may use

Problem Set 2 – VI

classical arithmetic algorithms for arithmetic in S, but note that each arithmetic operation in S may consume multiple operations in $\mathbb{Z}_{2^{128}}$ and these need to be accounted for in your analysis.

3. Quotient and remainder; evaluation and interpolation

Computer Science Club, St Petersburg
17–18 November 2018

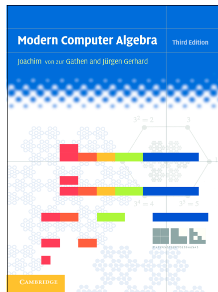
Petteri Kaski
Department of Computer Science
Aalto University

Recap of last lecture

- Evaluation–interpolation duality of polynomials
- Multiplication is a pointwise product in the dual
- Transforming between the primal and a (carefully chosen) dual
—**roots of unity** and the **discrete Fourier transform** (DFT)
- Factoring a composite-order DFT to obtain a **fast Fourier transform** (FFT)
- Fast cyclic convolution (assuming a suitable root of unity exists)
- Fast negative-wrapping cyclic convolution (**Schönhage–Strassen** algorithm)

Goal: Near-linear-time toolbox for univariate polynomials

- Multiplication
- Division (quotient and remainder) (this lecture)
- Batch evaluation (this lecture)
- Interpolation (this lecture)
- Extended Euclidean algorithm (gcd)
- Interpolation from partly erroneous data



Chapter 5

A NEW ALGORITHM FOR DECODING REED-SOLOMON CODES

Shuhong Gao
Department of Mathematical Sciences
Clemson University,
Clemson, SC 29634-0975, USA.

Abstract A new algorithm is developed for decoding Reed-Solomon codes. It uses fast Fourier transforms and computes the message symbols directly without explicitly finding error locations or error magnitudes. In the decoding radius (up to half of the minimum distance), the new method is easily adapted for error and erasure decoding. It can also detect all errors outside the decoding radius. Compared with the Berlekamp-Massey algorithm, discovered in the late 1960's, the new method seems simpler and more natural yet it has a similar time complexity.

1. **Introduction**

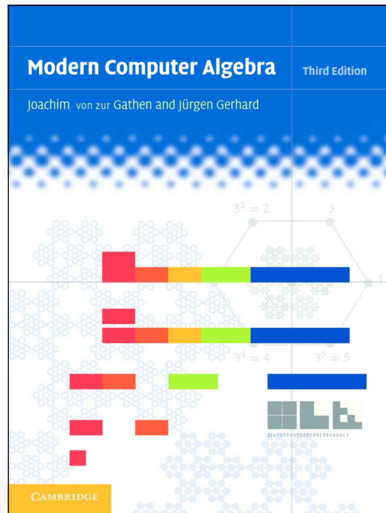
Reed-Solomon codes are the most popular codes in practical use today with applications ranging from CD players in our living rooms to space-crafts in deep space exploration. Their main advantage lies in two facts: high capability of correcting both random and burst errors; and existence of efficient decoding algorithm for them, namely the Berlekamp-Massey algorithm, discovered in the late 1960's [1, 9]. The Berlekamp-Massey

Key content for Lecture 3

- **Division** (**quotient** and **remainder**) for polynomials
- Fast division by **reduction** to fast multiplication
- Polynomial division via **reversal**
- **Newton iteration**
- Newton iteration for the inverse of the reverse of the divisor
- **Convergence analysis** for Newton iteration
- Fast **batch evaluation** and **interpolation** of polynomials
- Reduction to fast quotient and remainder
  —divide-and-conquer recursive remaindering along a **subproduct tree**

Fast quotient and remainder (polynomials)

(von zur Gathen and Gerhard [6], Sections 9.1 and 9.4)



Division (quotient and remainder)

- We start by recalling polynomial division
- We also recall that we can multiply fast
- Our goal for this lecture is to develop division algorithms that are essentially (up to constants) *as fast as our multiplication algorithms*
- The key idea is to proceed by **reduction** to multiplication
- In preparing the reductions, we recall and encounter many useful concepts ...

Polynomial quotient and remainder

- Let R be a ring
- Let $a = \sum_{i=0}^{n} \alpha_i x^i \in R[x]$ and $b = \sum_{i=0}^{m} \beta_i x^i \in R[x]$ such that $\alpha_n \neq 0$ and $\beta_m = 1$
- That is, $\deg a = n$ and b is monic with $\deg b = m$
- Then, there exist polynomials $q, r \in R[x]$ that satisfy $a = qb + r$ with $\deg r < \deg b$
- We write $a \operatorname{quo} b$ for such a **quotient** q and $a \operatorname{rem} b$ for such a **remainder** r in the division of a by b
- In fact, such q and r are unique (exercise)

The classical division algorithm (for polynomials)

- Let $a = \sum_i \alpha_i x^i, b = \sum_i \beta_i x^i \in R[x]$ be given as input with $\deg a = n$, $\deg b = m$, $n \geq m \geq 0$, and suppose that $\beta_m \in R$ is a unit
- We want to compute $q, r \in R[x]$ with $a = qb + r$ and $\deg r < m$
- The classical division algorithm:
  1. $r \leftarrow a, \mu \leftarrow \beta_m^{-1}$
  2. **for** $i = n - m, n - m - 1, \ldots, 0$ **do**
  3. **if** $\deg r = m + i$ **then** $\eta_i \leftarrow \mathrm{lc}(r)\mu, r \leftarrow r - \eta_i x^i b$
     else $\eta_i \leftarrow 0$
  4. **return** $q = \sum_{i=0}^{n-m} \eta_i x^i$ and r
- The classical algorithm runs in $O((n+m)^2)$ operations in R
- ... But could we do better? After Lecture 2, we know how to multiply in near-linear-time ...

Fast polynomial multiplication

- Let R be a ring
- Given $f, g \in R[x]$ with $\deg f \leq d$ and $\deg g \leq d$ as input, we can compute the product $fg \in R[x]$ in $O(M(d))$ operations in R
- We can take $M(d) = O(d \log d)$ if R has a primitive root of unity that supports an appropriate FFT
- In general, we can take $M(d) = O(d \log d \log \log d)$
- (In Lecture 2 we explored Schönhage–Strassen multiplication that assumes 2 is a unit in R; this algorithm can be generalized so that R is an arbitrary ring.)

First reduction towards division: the quotient suffices

- Division (viewed from 36,000ft, see earlier slides for details):

  $$\text{Given } a, b \text{ we need to compute } q, r \text{ such that } a = qb + r$$

- Observation:
  It suffices to compute q since then we can recover $r = a - qb$ by fast multiplication

High-level idea: iterate for the quotient

- Our approach will be to recover the quotient **iteratively**
- In essence, we iterate for a multiplicative inverse of the divisor b such that each iteration increases the accuracy of our inverse
- We want the accuracy (e.g. the polynomial degree) to increase **geometrically** from n to $2n$ in one iteration
- Once a sufficiently accurate version of the inverse is available (n is large enough), we proceed to solve for the quotient
- Each iteration will involve a constant number of multiplications, additions, and subtractions on inputs of size $O(n)$

The cost of a geometric iteration

- We say that a function $T : \mathbb{Z}_{\geq n_0} \to \mathbb{Z}_{\geq 0}$ **grows at least linearly** if for all $n, n_1, n_2 \in \mathbb{Z}_{\geq n_0}$ it holds that $n = n_1 + n_2$ implies $T(n) \geq T(n_1) + T(n_2)$
- *Examples:*
  $T(n) = Cn \log_2 n$ for $n_0 = 1$ and any constant $C > 0$
  $T(n) = Cn \log_2 n \log_2 \log_2 n$ for $n_0 = 2$ and any constant $C > 0$

Lemma 5 (Last step dominates—the previous steps are "for free")

Suppose that T grows at least linearly for $n \geq n_0 \geq 1$ and let 2^{k_0} be the least integer power of 2 at least n_0. Then, for all $k \geq k_0$ we have $\sum_{j=k_0}^{k} T(2^j) \leq T(2^{k+1})$

Proof.

By induction (exercise). □

Goal for fast polynomial division

- Let R be a ring
- Let $a, b \in R[x]$ with b monic and $d \geq \deg a \geq \deg b$ for some $d \in \mathbb{Z}_{\geq 0}$
- We want an algorithm that computes the quotient q and the remainder r in the division of a by b in $O(M(d))$ operations in R
- Here $M(d) = O(d \log d)$ or $M(d) = O(d \log d \log \log d)$ depending on R

Reversal to recover the quotient

- For a polynomial
$$f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_n x^n$$
of degree at most $n \in \mathbb{Z}_{\geq 0}$, the n-**reversal** of f is the polynomial
$$\mathrm{rev}_n f = \varphi_n + \varphi_{n-1} x + \varphi_{n-2} x^2 + \ldots + \varphi_0 x^n$$
- For the quotient-and-remainder identity $a = qb + r$ with $\deg a = n \geq m = \deg b$ and $\deg r \leq m - 1$, we observe (exercise) that the reversal operator satisfies
$$\mathrm{rev}_n\, a = (\mathrm{rev}_{n-m}\, q)(\mathrm{rev}_m\, b) + x^{n-m+1}\, \mathrm{rev}_{m-1}\, r$$
- In particular, working in the factor ring relative to the ideal $\langle x^{n-m+1} \rangle$,
$$\mathrm{rev}_n\, a \equiv (\mathrm{rev}_{n-m}\, q)(\mathrm{rev}_m\, b) \pmod{x^{n-m+1}}$$
- Since b is monic, $\mathrm{rev}_m\, b$ has a multiplicative inverse modulo x^{n-m+1}
- Thus, we can compute the quotient q by computing the inverse of $\mathrm{rev}_m\, b$ modulo x^{n-m+1}, multiplying by $\mathrm{rev}_n\, a$, and $(n - m)$-reversing the result

Example: Reversal (1/2)

- Suppose that in $\mathbb{Z}_5[x]$ we have

$$a = 3 + 3x + x^2 + 2x^3 + x^4 + 4x^6 + x^7 + 3x^8 + 4x^9 + 3x^{10} + x^{11} + x^{12}$$
$$b = 2 + x + x^2 + 3x^3 + 3x^4 + 3x^5 + x^6$$

  with $n = \deg a = 12$ and $m = \deg b = 6$; we also observe that b is monic

- We have $a = qb + r$ and $0 \leq \deg r \leq \deg b - 1$ for

$$q = 3 + 3x + 3x^2 + 4x^3 + x^4 + 3x^5 + x^6$$
$$r = 2 + 4x + 4x^2 + 4x^3 + 4x^4 + 2x^5$$

- Taking reverses, we have

$$\operatorname{rev}_n a = 1 + x + 3x^2 + 4x^3 + 3x^4 + x^5 + 4x^6 + x^8 + 2x^9 + x^{10} + 3x^{11} + 3x^{12}$$
$$\operatorname{rev}_m b = 1 + 3x + 3x^2 + 3x^3 + x^4 + x^5 + 2x^6$$
$$\operatorname{rev}_{n-m} q = 1 + 3x + x^2 + 4x^3 + 3x^4 + 3x^5 + 3x^6$$
$$\operatorname{rev}_{m-1} r = 2 + 4x + 4x^2 + 4x^3 + 4x^4 + 2x^5$$

Example: Reversal (2/2)

- Recalling that

$$\operatorname{rev}_n a = 1 + x + 3x^2 + 4x^3 + 3x^4 + x^5 + 4x^6 + x^8 + 2x^9 + x^{10} + 3x^{11} + 3x^{12}$$
$$\operatorname{rev}_m b = 1 + 3x + 3x^2 + 3x^3 + x^4 + x^5 + 2x^6$$
$$\operatorname{rev}_{n-m} q = 1 + 3x + x^2 + 4x^3 + 3x^4 + 3x^5 + 3x^6$$
$$\operatorname{rev}_{m-1} r = 2 + 4x + 4x^2 + 4x^3 + 4x^4 + 2x^5$$

  with $n = 12$ and $m = 6$, we can now verify the reversed division equality

$$\operatorname{rev}_n a = (\operatorname{rev}_{n-m} q)(\operatorname{rev}_m b) + x^{n-m-1} \operatorname{rev}_{m-1} r$$

- Indeed,

$$\operatorname{rev}_n a = 1 + x + 3x^2 + 4x^3 + 3x^4 + x^5 + 4x^6 + x^8 + 2x^9 + x^{10} + 3x^{11} + 3x^{12}$$
$$(\operatorname{rev}_{n-m} q)(\operatorname{rev}_m b) = 1 + x + 3x^2 + 4x^3 + 3x^4 + x^5 + 4x^6 + 3x^7 + 2x^8 + 3x^9 + 2x^{10} + 4x^{11} + x^{12}$$
$$x^{n-m-1} r = 2x^7 + 4x^8 + 4x^9 + 4x^{10} + 4x^{11} + 2x^{12}$$

The inverse modulo x^d by reduction to fast multiplication

- Let $g = \sum_j \psi_j x^j \in R[x]$ with $\psi_0 = 1$ be given as input
- We set up a Newton iteration that doubles d at every step
- Assume inductively that $f \in R[x]$ satisfies $fg \equiv 1 \pmod{x^{2^k}}$ for $k \in \mathbb{Z}_{\geq 0}$
- To set up the base case $k = 0$, take $f = 1$ and observe that the assumption holds
- Compute $\hat{f} \equiv (2 - fg)f \pmod{x^{2^{k+1}}}$ using fast multiplication, truncating both g and $\hat{f}$ using the substitution $x^{2^{k+1}} = 0$
- Since the assumption holds for f with parameter value k, there exists a $h \in R[x]$ with $fg = 1 + x^{2^k} h$
- We observe that $\hat{f} g \equiv (2 - fg)fg \equiv (1 - x^{2^k} h)(1 + x^{2^k} h) \equiv 1 \pmod{x^{2^{k+1}}}$ and thus the assumption holds for $\hat{f}$ with parameter value $k + 1$
- The cost of step k is $O(M(2^k))$ since M grows at most polynomially; by Lemma 5 the total cost is $O(M(d))$ operations in R

Example: Iterating for the inverse modulo x^d

- Let $g = 1 + 3x + 3x^2 + 3x^3 + x^4 + x^5 + 2x^6 \in \mathbb{Z}_5[x]$
- Let us compute the multiplicative inverse of g modulo x^d for $d = 7$
- The least integer k for which $2^k \geq d$ is $k = 3$, so we need three rounds of Newton iteration
- Truncating g and $\hat{f}$ by setting $x^{2^{k+1}} = 0$ and iterating, we have

k	f	g
0	1	$1 + 3x$
1	$1 + 2x$	$1 + 3x + 3x^2 + 3x^3$
2	$1 + 2x + x^2 + 3x^3$	$1 + 3x + 3x^2 + 3x^3 + x^4 + x^5 + 2x^6$
3	$1 + 2x + x^2 + 3x^3 + x^4 + 2x^5 + 2x^6 + 2x^7$	

- Thus, the multiplicative inverse of g modulo x^d is

$$1 + 2x + x^2 + 3x^3 + x^4 + 2x^5 + 2x^6$$

Example: Division with reversal and Newton iteration

- Suppose that in $\mathbb{Z}_5[x]$ we have

$$a = 3 + 3x + x^2 + 2x^3 + x^4 + 4x^6 + x^7 + 3x^8 + 4x^9 + 3x^{10} + x^{11} + x^{12}$$
$$b = 2 + x + x^2 + 3x^3 + 3x^4 + 3x^5 + x^6$$

  with $n = \deg a = 12$ and $m = \deg b = 6$; we also observe that b is monic

- Reverse a and b to obtain

$$\operatorname{rev}_n a = 1 + x + 3x^2 + 4x^3 + 3x^4 + x^5 + 4x^6 + x^8 + 2x^9 + x^{10} + 3x^{11} + 3x^{12}$$
$$\operatorname{rev}_m b = 1 + 3x + 3x^2 + 3x^3 + x^4 + x^5 + 2x^6$$

- Iterate for the inverse f of $\operatorname{rev}_m b$ modulo x^{n-m+1} to obtain

$$f = 1 + 2x + x^2 + 3x^3 + x^4 + 2x^5 + 2x^6$$

- Compute $f \operatorname{rev}_n a$, truncate with $x^{n-m+1} = 0$, and $(n-m)$-reverse the result to obtain the quotient $q = 3 + 3x + 3x^2 + 4x^3 + x^4 + 3x^5 + x^6$

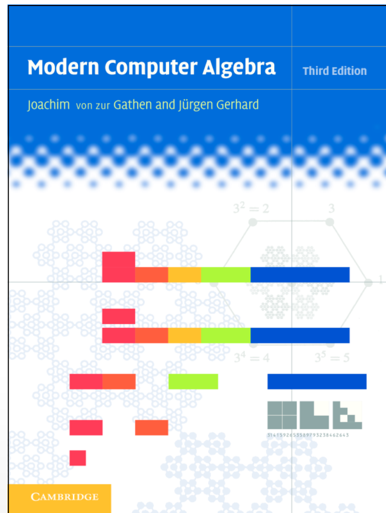
- Compute the remainder $r = a - qb = 2 + 4x + 4x^2 + 4x^3 + 4x^4 + 2x^5$

Summary—fast polynomial division

- Let R be a ring
- Let $a, b \in R[x]$ with b monic and $d \geq \deg a \geq \deg b$ for some $d \in \mathbb{Z}_{\geq 0}$
- We have an algorithm that computes the quotient q and the remainder r in the division of a by b in $O(M(d))$ operations in R
  1. Let $n = \deg a$ and $m = \deg b$
  2. m-reverse b and compute the multiplicative inverse of $\operatorname{rev}_m b$ modulo x^{n-m+1} using Newton iteration, multiply by the result by $\operatorname{rev}_n a$ modulo x^{n-m+1}, and $(n-m)$-reverse the result to obtain the quotient q
  3. Compute remainder r by $r = a - qb$
- Here $M(d) = O(d \log d)$ or $M(d) = O(d \log d \log \log d)$ depending on R

Batch evaluation and interpolation

(von zur Gathen and Gerhard [6], Sections 10.1–10.3 and 5.1–5.4)



Batch evaluation and interpolation

- To **evaluate** a polynomial $(\varphi_0, \varphi_1, \ldots, \varphi_d) \in F^{d+1}$ at ("a batch of") distinct points $\xi_0, \xi_1, \ldots, \xi_d \in F$, we multiply from the left with the Vandermonde matrix:

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix} \begin{bmatrix} \varphi_0 \\ \varphi_1 \\ \vdots \\ \varphi_d \end{bmatrix} = \begin{bmatrix} f(\xi_0) \\ f(\xi_1) \\ \vdots \\ f(\xi_d) \end{bmatrix}$$

- To **interpolate** the coefficients of a polynomial with values $(f(\xi_0), f(\xi_1), \ldots, f(\xi_d)) \in F^{d+1}$ at distinct $\xi_0, \xi_1, \ldots, \xi_d \in F$, we multiply from the left with the inverse of the Vandermonde matrix:

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix}^{-1} \begin{bmatrix} f(\xi_0) \\ f(\xi_1) \\ \vdots \\ f(\xi_d) \end{bmatrix} = \begin{bmatrix} \varphi_0 \\ \varphi_1 \\ \vdots \\ \varphi_d \end{bmatrix}$$

Fast batch evaluation and interpolation?

- Can we go faster than working with the Vandermonde matrix in explicit form?
- Yes, for example, in the case when the points $\xi_0, \xi_1, \ldots, \xi_d$ are powers of a primitive root of unity of composite order $d + 1$ (recall fast Fourier transform from Lecture 2)
- But what about in general?
  That is, when $\xi_0, \xi_1, \ldots, \xi_d$ are arbitrary distinct points in a ring R
- *We now know how to multiply and divide fast, so maybe we could put these algorithms into use ...*

Polynomial division (quotient and remainder) recalled

- Let R be a ring (commutative and nontrivial, as usual)
- Let $a = \sum_i \alpha_i x^i \in R[x]$ and $b = \sum_i \beta_i x^i \in R[x]$ be given as input with $\deg a = n$, $\deg b = m$, and $n \geq m \geq 0$
- Let us also assume that $\beta_m = 1$ (that is, b is **monic**)
- We want to compute $q, r \in R[x]$ with $a = qb + r$ and $\deg r < m$
- That is, $q = a \operatorname{quo} b$ is the **quotient** and $r = a \operatorname{rem} b$ is the **remainder** in the polynomial division with **dividend** a and **divisor** b
- We now have a fast algorithm that divides in $O(M(n))$ operations in R by reduction to fast multiplication
- Let us now develop fast algorithms for batch evaluation and interpolation to by reduction to fast division

Fast batch evaluation by recursive remaindering

- Suppose we have a polynomial $f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_d x^d \in R[x]$ and we want to compute the values $f(\xi_0), f(\xi_1), \ldots, f(\xi_{e-1})$ at e given points $\xi_0, \xi_1, \ldots, \xi_{e-1} \in R$
- **Goal:** $O(M(d) + M(e) \log e)$ operations in R
- We reduce the multi-point (batch) evaluation task to recursive remaindering along a subproduct tree enabled by the following to lemmas (proofs: in the problem set)

Lemma 6 (Evaluation at a point via remainder)

For all $\xi \in R$ and $f \in R[x]$ it holds that $f(\xi) = f \operatorname{rem} (x - \xi)$

Lemma 7 (Recursive remaindering)

Let $a, b, c \in R[x]$, with b and c monic, and suppose that c divides b. Then,
$a \operatorname{rem} c = (a \operatorname{rem} b) \operatorname{rem} c$

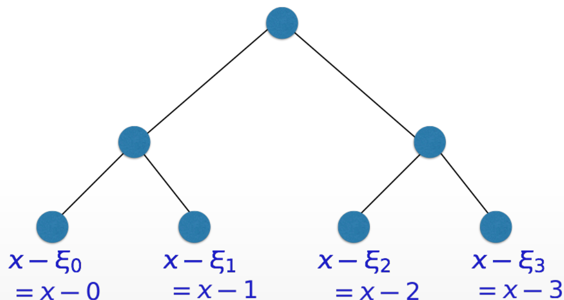
Example: Batch evaluation

- The algorithm for fast batch evaluation is perhaps best illustrated by starting with an example and then proceeding with the details

- Let us work over $R = \mathbb{Z}$ for simplicity
- Let $f = x^5 - x^4 + 2x^3 + 4x - 5 \in \mathbb{Z}[x]$
- Let $\xi_0 = 0$, $\xi_1 = 1$, $\xi_2 = 2$, $\xi_3 = 3$

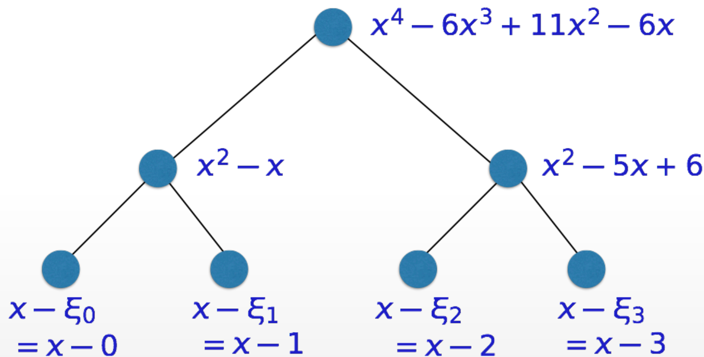
Example: Batch evaluation (0/4)

0. Place the linear polynomials $x - \xi_j$ for $j = 0,1,\ldots,e - 1$ at the leaves of a perfect binary tree (can assume that e is a power of 2)



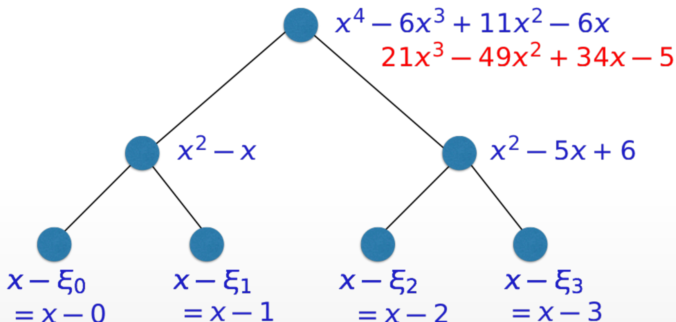
Example: Batch evaluation (1/4)

1. For each internal node in post-order, place the product of the two child nodes at the node



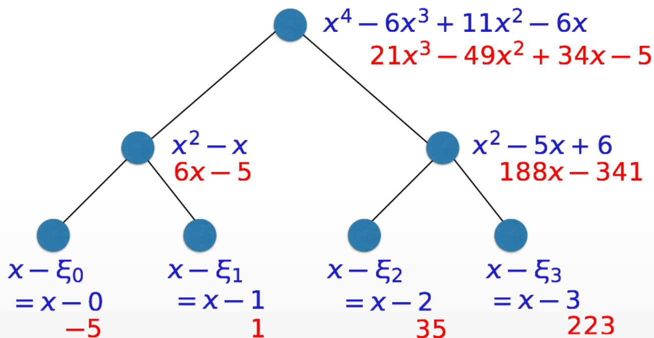
Example: Batch evaluation (2/4)

2. Compute the remainder of $f = x^5 - x^4 + 2x^3 + 4x - 5$ and the root node and place it at the root node



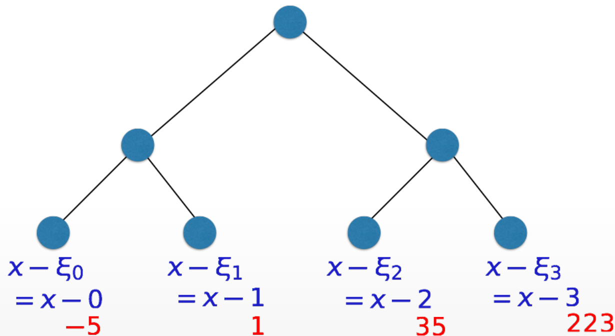
Example: Batch evaluation (3/4)

3. For each nonroot node in preorder, compute the remainder of the parent node and the subproduct at the node



Example: Batch evaluation (4/4)

4. The remainders at the leaf nodes are the evaluations $f(\xi_j)$



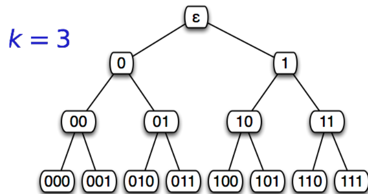
Nodes of a perfect binary tree and binary strings (1/2)

- Let us now present the algorithm in detail
- Without loss of generality we can assume that $e = 2^k$ for some $k \in \mathbb{Z}_{\geq 0}$ (for example, insert new points of evaluation until e is a power of 2)
- We will structure the recursion along a perfect binary tree with 2^k leaves
- Let us write $\{0,1\}^{\underline{k}}$ for the set of all binary strings of length at most k, including the empty string ϵ
- For $u \in \{0,1\}^{\underline{k}}$ let us write $0 \leq |u| \leq k$ for the length of u
- *Example.* For $k = 3$, we have

$$\{0,1\}^{\underline{k}} = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111\}$$

Nodes of a perfect binary tree and binary strings (2/2)

- The $2^{k+1} - 1 = \sum_{j=0}^{k} 2^j$ strings in $\{0, 1\}^{\underline{k}}$ are in a natural one-to-one correspondence with the nodes of a perfect binary tree with 2^k leaves, with the empty string ϵ corresponding to the root and the strings of length k corresponding to the leaves
- Indeed, to navigate from a non-root node to its parent node, simply delete the last bit from the corresponding string
- Dually, to navigate from a non-leaf node to one of its two children, append either the bit 0 (to go the left child) or the bit 1 (to go the right child) to the string



A subproduct tree for batch evaluation

- Let us work with a perfect binary tree with 2^k leaves and nodes indexed by the binary strings in $\{0,1\}^{\underline{k}}$
- Associate with each leaf $v \in \{0,1\}^k$ the linear polynomial

$$s_v = x - \xi_v \tag{20}$$

- Associate with each internal node $u \in \{0,1\}^{\underline{k-1}}$ the product of the children of u by

$$s_u = s_{u0} s_{u1} \tag{21}$$

- We observe that s_u is a monic polynomial of degree $2^{k-|u|}$ for all $u \in \{0,1\}^{\underline{k}}$

Fast batch evaluation using a subproduct tree

- To perform batch evaluation, first compute and store the polynomials s_u for all $u \in \{0,1\}^{\underline{k}}$ using (20) and (21)
- Then, associate the remainder

$$r_\epsilon = f \operatorname{rem} s_\epsilon \tag{22}$$

  with the root ϵ of the binary tree
- For each nonroot $u \in \{0,1\}^{\underline{k}} \setminus \{\epsilon\}$, associate with u the remainder

$$r_u = r_p \operatorname{rem} s_u \tag{23}$$

  where $p \in \{0,1\}^{\underline{k-1}}$ is the parent of u in the binary tree
- For each leaf $v \in \{0,1\}^k$, the remainder r_v satisfies $r_v = f(\xi_v)$

Analysis

- Recall that s_u is a monic polynomial of degree $2^{k-|u|}$ for all $u \in \{0, 1\}^{\underline{k}}$
- From (20) and (21) we have that each s_u can be prepared in $O(M(2^{k-|u|}))$ operations in R using fast multiplication
- There are in total 2^j binary strings $u \in \{0, 1\}^j$, implying that the total cost of level $j = k, k-1, \ldots, 0$ is $O(2^j M(2^{k-j}))$ operations in R, which is $O(M(2^k)) = O(M(e))$ by at-least-linear and at-most-polynomial growth of M
- The root remainder (22) takes $O(M(d) + M(e))$ operations in R using fast division
- Below the root, each level $j = 0, 1, \ldots, k$ similarly takes $O(M(e))$ operations in R using (23) and fast division
- Since there are $k = O(\log e)$ levels, we obtain that that batch evaluation runs in total $O(M(d) + M(e) \log e)$ operations in R

Interpolation

- Let R be a ring
- Let $\xi_0, \xi_1, \ldots, \xi_{e-1} \in R$ and $\eta_0, \eta_1, \ldots, \eta_{e-1} \in R$ such that $\xi_i - \xi_j$ is a unit in R for all $0 \leq i < j \leq e-1$
- We seek to compute the coefficients of the Lagrange interpolation polynomial

$$\ell = \sum_{i=0}^{e-1} \left(\eta_i \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (\xi_i - \xi_j)^{-1} \right) \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (x - \xi_j) \in R[x]$$

that satisfies $\ell(\xi_i) = \eta_i$ for all $i = 0, 1, \ldots, e-1$

Fast interpolation with subproduct trees

- The form

$$\ell = \sum_{i=0}^{e-1} \left(\eta_i \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (\xi_i - \xi_j)^{-1} \right) \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (x - \xi_j) \in R[x]$$

  suggests that one should first seek to construct the coefficients of the polynomial

$$\ell = \sum_{i=0}^{e-1} \lambda_i \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (x - \xi_j) \in R[x]$$

  from e given scalars $\lambda_0, \lambda_1, \ldots, \lambda_{e-1} \in R$

- A strategy based on subproduct-trees works also here and leads to an algorithm that runs in $O(M(e) \log e)$ operations in R (exercise)

Application: How to share a secret

"In this paper we show how to divide data D into n pieces in such a way that D is easily reconstructible from any k pieces, but even complete knowledge of $k-1$ pieces reveals absolutely no information about D. This technique enables the construction of robust key management schemes for cryptographic systems that can function securely and reliably even when misfortunes destroy half the pieces and security breaches expose all but one of the remaining pieces."

(Shamir [11])

Application: How to share a secret (1/5)

- Let us work over a finite field F (for example, $F = \mathbb{Z}_p$ for p prime)
- Let $f = \varphi_0 + \varphi_1 x \in F[x]$ be a line (polynomial of degree at most 1)
- How much do we know about the constant φ_0 of the line f if we know the value $f(\xi)$ for a **nonzero** $\xi \in F$?

Application: How to share a secret (2/5)

- Let us work over a finite field F (for example, $F = \mathbb{Z}_p$ for p prime)
- Let $f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_d x^d \in F[x]$ be a polynomial of degree at most d
- How much do we know about the constant φ_0 of the polynomial f if we know $(\xi_j, f(\xi_j))$ for exactly d **nonzero** distinct values $\xi_j \in F$ for $j = 1, 2, \ldots, d$?

Application: How to share a secret (3/5)

- Let $f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_d x^d \in F[x]$ be a polynomial of degree at most d
- How much do we know about the constant φ_0 of the polynomial f if we know $(\xi_j, f(\xi_j))$ for exactly d **nonzero** distinct values $\xi_j \in F$ for $j = 0, 1, \ldots, d$?
- *We claim that this knowledge reveals no information about* φ_0;
  indeed, let us set $\xi_0 = 0$ and recall the interpolation identity

$$\begin{bmatrix} \xi_0^0 & \xi_0^1 & \cdots & \xi_0^d \\ \xi_1^0 & \xi_1^1 & \cdots & \xi_1^d \\ \vdots & \vdots & & \vdots \\ \xi_d^0 & \xi_d^1 & \cdots & \xi_d^d \end{bmatrix}^{-1} \begin{bmatrix} f(\xi_0) \\ f(\xi_1) \\ \vdots \\ f(\xi_d) \end{bmatrix} = \begin{bmatrix} \varphi_0 \\ \varphi_1 \\ \vdots \\ \varphi_d \end{bmatrix}$$

- Since $f(\xi_0) = f(0) = \varphi_0$, we have that for each choice $\varphi_0 \in F$ the values $f(\xi_1), f(\xi_2), \ldots, f(\xi_d)$ are consistent with exactly one choice $(\varphi_0, \varphi_1, \ldots, \varphi_d) \in F^{d+1}$
- Thus, the values $f(\xi_1), f(\xi_2), \ldots, f(\xi_d)$ reveal no information about φ_0

Application: How to share a secret (4/5)

- Let $f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_d x^d \in F[x]$ be a polynomial of degree at most d
- How much do we know about the constant φ_0 of the polynomial f if we know $(\xi_j, f(\xi_j))$ for exactly e **nonzero** distinct values $\xi_j \in F$ for $j = 1, 2, \ldots, e$?

- For $e \leq d$, we obtain no information about φ_0
- For $e \geq d + 1$, we have full information about φ_0 since we can interpolate all the coefficients of f from any $d + 1$ evaluations at distinct points

Application: How to share a secret (5/5)

- Suppose $\varphi_0 \in F$ is a **secret** that you want to split into s **shares** so that
  - knowledge of any k shares enables recovery of the secret
  - knowledge of any $k-1$ or fewer shares reveals no information about the secret

1. Let $\xi_1, \xi_2, \ldots, \xi_s \in F$ be distinct and **nonzero**
2. Select elements $\varphi_1, \varphi_2, \ldots, \varphi_{k-1} \in F$ independently and uniformly at random
3. Let $f = \varphi_0 + \varphi_1 x + \varphi_2 x^2 + \ldots + \varphi_{k-1} x^{k-1} \in F[x]$
4. For $j = 1, 2, \ldots, s$, share j is the pair $(\xi_j, f(\xi_j)) \in F^2$

- Using fast batch evaluation and interpolation, preparing the shares takes $O(M(s) \log s)$ operations in F, and recovering the secret takes $O(M(k) \log k)$ operations in F

Randomization and primal–dual

- The secret $\varphi_0 \in F$ resides in the primal (coefficient representation)
- Selecting $\varphi_1, \varphi_2, \ldots, \varphi_{k-1} \in F$ independently and uniformly at random masks the secret in the dual (evaluation representation) unless we know k shares
- This is our first example of the use of **randomization** during this course
- The evaluation–interpolation duality enables us to spread the information in the coefficient representation uniformly to evaluations in the evaluation representation
- The following lectures will explore both randomization as a tool in algorithm design and the aforementioned "uniformity" further, the latter in particular as regards error-correcting codes and error-tolerant computation

Recap of key content for Lecture 3

- **Division** (**quotient** and **remainder**) for polynomials
- Fast division by **reduction** to fast multiplication
- Polynomial division via **reversal**
- **Newton iteration**
- Newton iteration for the inverse of the reverse of the divisor
- **Convergence analysis** for Newton iteration
- Fast **batch evaluation** and **interpolation** of polynomials
- Reduction to fast quotient and remainder
  —divide-and-conquer recursive remaindering along a **subproduct tree**

Problem Set 3 – I

1. Polynomial division by reversal. For a polynomial $f = \varphi_0 + \varphi_1 x + \ldots + \varphi_n x^n \in R[x]$ of degree at most $n \in \mathbb{Z}_{\geq 0}$ over a ring R, let the n-*reversal* of f be the polynomial $\operatorname{rev}_n f = \varphi_n + \varphi_{n-1} x + \ldots + \varphi_0 x^n \in R[x]$. Let $a, b \in R[x]$ be polynomials with b monic and $n = \deg a \geq \deg b = m$. Show that the quotient $q \in R[x]$ and the remainder $r \in R[x]$ with $a = qb + r$ and $\deg r \leq m - 1$ satisfy the reversal identity

$$\operatorname{rev}_n a = (\operatorname{rev}_{n-m} q)(\operatorname{rev}_m b) + x^{n-m+1} \operatorname{rev}_{m-1} r\,.$$

Hint: Observe that for all $i = 0, 1, \ldots, n-m$ and $j = 0, 1, \ldots, m$ we have $x^{n-i-j} = x^{n-m-i} x^{m-j}$.

Problem Set 3 – II

2. Evaluation via recursive remaindering. Let R be a ring.
   (a) Show that for all $\xi \in R$ and $f \in R[x]$ we have $f(\xi) = f \operatorname{rem} (x - \xi)$.
   (b) Let $a, b, c \in R[x]$, with b and c monic, and suppose that c divides b. Show that $a \operatorname{rem} c = (a \operatorname{rem} b) \operatorname{rem} c$.

   Hints: Recall that the quotient and remainder are unique for $a, b \in R[x]$ with b monic. Use the defining equality $a = qb + r$ with $\deg r < \deg b$ for both parts. For part (a), investigate what happens when you evaluate the defining equality at ξ.

Problem Set 3 – III

3. Preliminaries for fast interpolation. Let R be a ring, let $\xi_0, \xi_1, \ldots, \xi_{e-1} \in R$, and $\lambda_0, \lambda_1, \ldots, \lambda_{e-1} \in R$ be given as input. The form of the Lagrange interpolation polynomial (27) suggests that one should first seek to construct the coefficients of the polynomial

$$\ell = \sum_{i=0}^{e-1} \lambda_i \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (x - \xi_j) \in R[x] \,. \tag{24}$$

Show that we can compute the coefficients of ℓ in $O(M(e) \log e)$ operations in R. You may assume that $e = 2^k$ for a nonnegative integer k. Here $M(e) = e \log e \log \log e$.

Hints: Work with binary strings and the representation of the perfect binary tree using binary strings in $\{0, 1\}^{\underline{k}}$. To construct the coefficients of the polynomial (24), first construct a subproduct tree with polynomials s_u for all $u \in \{0, 1\}^{\underline{k}}$ from $\xi_0, \xi_1, \ldots, \xi_{e-1}$

Problem Set 3 – IV

as during fast evaluation. Next, annotate the tree with another family of polynomials such that the polynomial at the root will be equal to (24). You may want to try associating with each leaf $v \in \{0, 1\}^k$ the polynomial

$$\ell_v = \lambda_v \tag{25}$$

and with each internal node $u \in \{0, 1\}^{\underline{k-1}}$ the polynomial

$$\ell_u = \ell_{u0} s_{u1} + s_{u0} \ell_{u1} \,. \tag{26}$$

Why is this a good choice? Prepare a small example, say with $k = 2$ or $k = 3$ as necessary. Show that $\ell_\epsilon = \ell$, where ϵ is the empty binary string.

Problem Set 3 – V

4. Fast interpolation via subproducts and fast evaluation. Let R be a ring and let $\xi_0, \xi_1, \ldots, \xi_{e-1} \in R$ and $\eta_0, \eta_1, \ldots, \eta_{e-1} \in R$ such that $\xi_i - \xi_j$ is a unit in R for all $0 \leq i < j \leq e-1$. Show that we can compute the coefficients of the Lagrange interpolation polynomial

$$\ell = \sum_{i=0}^{e-1} \left(\eta_i \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (\xi_i - \xi_j)^{-1} \right) \prod_{\substack{j=0 \\ j \neq i}}^{e-1} (x - \xi_j) \in R[x] \tag{27}$$

that satisfies $\ell(\xi_i) = \eta_i$ for all $i = 0, 1, \ldots, e-1$ in $O(M(e) \log e)$ operations in R. You may assume that $e = 2^k$ for a nonnegative integer k.

Hints: Apply your solution to Problem 3 in two passes. In the first pass, set $\lambda_v = 1$ for all $v \in \{0, 1\}^k$ and compute the coefficients of the polynomial $f = \ell_\epsilon$ using (25) and (26). Evaluate f at $\xi_0, \xi_1, \ldots, \xi_{e-1}$ using fast evaluation. Then do a second pass (with a different choice for the values λ_v) so that at the root you recover the Lagrange interpolation polynomial (27).

4. Extended Euclidean algorithm and interpolation from erroneous data

Computer Science Club, St Petersburg
17–18 November 2018

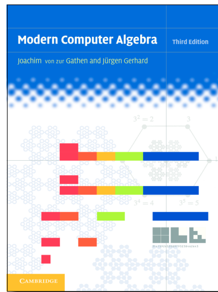
Petteri Kaski
Department of Computer Science
Aalto University

Recap of last lecture

- **Division** (**quotient** and **remainder**) for polynomials
- Fast division by **reduction** to fast multiplication
- Polynomial division via **reversal**
- **Newton iteration**
- Newton iteration for the inverse of the reverse of the divisor
- **Convergence analysis** for Newton iteration
- Fast **batch evaluation** and **interpolation** of polynomials
- Reduction to fast quotient and remainder
  —divide-and-conquer recursive remaindering along a **subproduct tree**

Goal: Near-linear-time toolbox for univariate polynomials

- Multiplication
- Division (quotient and remainder)
- Batch evaluation
- Interpolation
- Extended Euclidean algorithm (gcd) (this lecture)
- Interpolation from partly erroneous data (this lecture)



Chapter 5

A NEW ALGORITHM FOR DECODING REED-SOLOMON CODES

Shuhong Gao
Department of Mathematical Sciences
Clemson University,
Clemson, SC 29634-0975, USA.

Abstract A new algorithm is developed for decoding Reed-Solomon codes. It uses fast Fourier transforms and computes the message symbols directly without explicitly finding error locations or error magnitudes. In the decoding radius (up to half of the minimum distance), the new method is easily adapted for error and erasure decoding. It can also detect all errors outside the decoding radius. Compared with the Berlekamp-Massey algorithm, discovered in the late 1960's, the new method seems simpler and more natural yet it has a similar time complexity.

1. Introduction

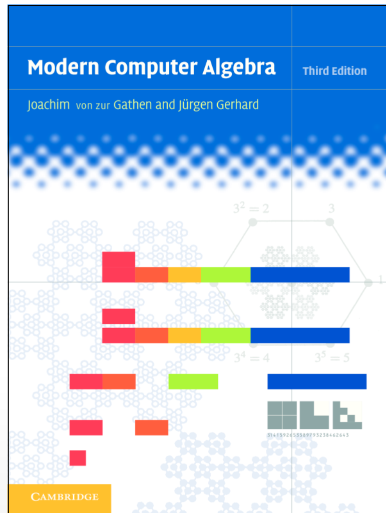
Reed-Solomon codes are the most popular codes in practical use today with applications ranging from CD players in our living rooms to spacecrafts in deep space exploration. Their main advantage lies in two facts: high capability of correcting both random and burst errors; and existence of efficient decoding algorithm for them, namely the Berlekamp-Massey algorithm, discovered in the late 1960's [1, 9]. The Berlekamp-Massey

Further motivation for this lecture

- After this lecture we have completed our work on the near-linear time toolbox for univariate polynomials
- This lecture is also our first encounter with **uncertainty** in computation
- In this lecture we learn how to cope with uncertainty in the form of **errors in data** by using **error-correcting codes**
- Next lecture looks at **errors in computation** ...

Fast extended Euclidean algorithm (for polynomials)

(von zur Gathen and Gerhard [6], Section 11.1)



Fast interpolation from partly erroneous data

(Gao [5])

Chapter 5

A NEW ALGORITHM FOR DECODING REED-SOLOMON CODES

Shuhong Gao
Department of Mathematical Sciences
Clemson University,
Clemson, SC 29634-0975, USA.

Abstract A new algorithm is developed for decoding Reed-Solomon codes. It uses fast Fourier transforms and computes the message symbols directly without explicitly finding error locations or error magnitudes. In the decoding radius (up to half of the minimum distance), the new method is easily adapted for error and erasure decoding. It can also detect all errors outside the decoding radius. Compared with the Berlekamp-Massey algorithm, discovered in the late 1960's, the new method seems simpler and more natural yet it has a similar time complexity.

1. Introduction

Reed-Solomon codes are the most popular codes in practical use today with applications ranging from CD players in our living rooms to spacecrafts in deep space exploration. Their main advantage lies in two facts: high capability of correcting both random and burst errors; and existence of efficient decoding algorithm for them, namely the Berlekamp-Massey algorithm, discovered in the late 1960's [1, 9]. The Berlekamp-Massey

Key content for Lecture 4

- **Extended Euclidean algorithm** for polynomials recalled and expanded
  - The **quotient sequence**, the **Bézout coefficients**, and the **halting threshold**
- Fast extended Euclidean algorithm for polynomials by **divide and conquer**
  - The two polynomial operands **truncated** to a prefix of the highest-degree monomials determine the prefix of the quotient sequence (exercise)
- Coping with **errors in data** using **error-correcting codes**
- A family of error-correcting codes (**Reed–Solomon codes**) based on evaluation–interpolation duality for univariate polynomials
  - Key observation: low-degree polynomials have few roots (exercise)
  - Fast **encoding** and **decoding** of Reed–Solomon codes via the fast univariate polynomial toolkit and **Gao's** (2003) **decoder**

Extended Euclidean algorithm (for polynomials)

- Let F be a field and let $f, g \in F[x]$ with $\deg f \geq \deg g \geq 0$
- Traditional extended Euclidean algorithm:
  1. $r_0 \leftarrow f$, $s_0 \leftarrow 1$, $t_0 \leftarrow 0$,
     $r_1 \leftarrow g$, $s_1 \leftarrow 0$, $t_1 \leftarrow 1$
  2. $i \leftarrow 1$,
     while $r_i \neq 0$ **do**
     $q_i \leftarrow r_{i-1} \text{ quo } r_i$
     $r_{i+1} \leftarrow r_{i-1} - q_i r_i$
     $s_{i+1} \leftarrow s_{i-1} - q_i s_i$
     $t_{i+1} \leftarrow t_{i-1} - q_i t_i$
     $i \leftarrow i + 1$
  3. $\ell \leftarrow i - 1$
     return ℓ, r_i, s_i, t_i for $i = 0, 1, \ldots, \ell + 1$, and q_i for $i = 1, 2, \ldots, \ell$
- We want a faster algorithm

Example (over $\mathbb{Z}_2[x]$)

- Let $f = x^5 + x^4 + x^3 + x^2 + x + 1 \in \mathbb{Z}_2[x]$ and $g = x^5 + x^4 + 1 \in \mathbb{Z}_2[x]$
- We obtain

i	r_i	s_i	t_i	q_i
0	$x^5 + x^4 + x^3 + x^2 + x + 1$	1	0	
1	$x^5 + x^4 + 1$	0	1	1
2	$x^3 + x^2 + x$	1	1	$x^2 + 1$
3	$x^2 + x + 1$	$x^2 + 1$	x^2	x
4	0	$x^3 + x + 1$	$x^3 + 1$	

- In particular $\ell = 3$ and $r_\ell = x^2 + x + 1$ is a greatest common divisor of $x^5 + x^4 + x^3 + x^2 + x + 1$ and $x^5 + x^4 + 1$

Terminology

- The sequence $q_1, q_2, \ldots, q_\ell$ is the **quotient sequence** produced by the algorithm
- The polynomial r_i is the **remainder** at iteration i
- The polynomials s_i and t_i are the **Bézout coefficients** at iteration i
- The Bézout coefficients satisfy $r_i = s_i r_0 + t_i r_1$

Desiderata for a fast algorithm

- Let F be a field and let $f, g \in F[x]$ with $d \geq \deg f \geq \deg g \geq 0$
- Desired output:
  The quotients $q_1, q_2, \ldots, q_h$ and two consecutive rows r_h, s_h, t_h and $r_{h+1}, s_{h+1}, t_{h+1}$ for a choice of $h = 1, 2, \ldots, \ell$
- Using $O(M(d) \log d)$ operations in F

The degree sequences m_i and n_i

- It will be convenient to work with the following two sequences
- For $i = 1, 2, \ldots, \ell + 1$ let

$$m_i = \deg q_i$$

  where, for convenience, we let $m_{\ell+1} = \infty$
- For $i = 0, 1, \ldots, \ell + 1$, let

$$n_i = \deg r_i$$

  recalling that $n_{\ell+1} = \deg 0 = -\infty$
- By assumption, we have $\deg r_0 \geq \deg r_1 \geq 0$
- Since we have $r_{i+1} = r_{i-1} - q_i r_i$ and $\deg r_i > \deg r_{i+1}$ for all $i = 1, 2, \ldots, \ell$, it follows that

$$n_{i-1} = n_i + m_i$$

Example (over $\mathbb{Z}_2[x]$)

- Let $f = x^5 + x^4 + x^3 + x^2 + x + 1 \in \mathbb{Z}_2[x]$ and $g = x^5 + x^4 + 1 \in \mathbb{Z}_2[x]$
- We obtain

i	r_i	s_i	t_i	q_i	m_i	n_i
0	$x^5 + x^4 + x^3 + x^2 + x + 1$	1	0			5
1	$x^5 + x^4 + 1$	0	1	1	0	5
2	$x^3 + x^2 + x$	1	1	$x^2 + 1$	2	3
3	$x^2 + x + 1$	$x^2 + 1$	x^2	x	1	2
4	0	$x^3 + x + 1$	$x^3 + 1$		∞	$-\infty$

- In particular $\ell = 3$ and $r_\ell = x^2 + x + 1$ is a greatest common divisor of $x^5 + x^4 + x^3 + x^2 + x + 1$ and $x^5 + x^4 + 1$

The halting threshold $h = h(k)$

- Given a threshold parameter $k = 0, 1, \ldots, n_0$ as input, we want the algorithm to halt at iteration $h = h(k)$ determined by

$$m_1 + m_2 + \ldots + m_h \leq k$$

  and

$$m_1 + m_2 + \ldots + m_h + m_{h+1} > k$$

- In particular, we observe that $0 \leq h \leq \ell$

The halting threshold $h = h(k)$

- Equivalently, since $n_i = n_{i-1} - m_i$ for $i = 1, 2, \ldots, \ell + 1$, we have

  $$n_h \geq n_0 - k$$

  and

  $$n_{h+1} < n_0 - k$$

- That is, the algorithm halts at the unique iteration $h = 0, 1, \ldots, \ell$ when the degree of r_{h+1} for the first time decreases below $n_0 - k$

Truncating a polynomial

- Let

$$f = \varphi_n x^n + \varphi_{n-1} x^{n-1} + \ldots + \varphi_1 x + \varphi_0 \in F[x]$$

  with **leading coefficient** $\operatorname{lc} f = \varphi_n \neq 0$

- For $k \in \mathbb{Z}$, define the **truncated polynomial**

$$f \restriction k = \varphi_n x^k + \varphi_{n-1} x^{k-1} + \ldots + \varphi_{n-k+1} x + \varphi_{n-k} \in F[x]$$

  where we set $\varphi_i = 0$ for $i < 0$ as necessary

- For $k \geq 0$ we have that $f \restriction k$ is a polynomial of degree k whose coefficients are the $k+1$ highest coefficients of f
- For $k < 0$ we have $f \restriction k = 0$
- For all $i = 0, 1, \ldots$ we have $(fx^i) \restriction k = f \restriction k$

Example: Truncating a polynomial

- Let us work with the polynomial

$$f = 2 + 9x + 10x^2 + 4x^3 \in \mathbb{Z}_{11}[x]$$

- We obtain the truncations

$$\begin{aligned} &\vdots \\ f \upharpoonright -2 &= 0 \\ f \upharpoonright -1 &= 0 \\ f \upharpoonright 0 &= 4 \\ f \upharpoonright 1 &= 10 + 4x \\ f \upharpoonright 2 &= 9 + 10x + 4x^2 \\ f \upharpoonright 3 &= 2 + 9x + 10x^2 + 4x^3 \\ f \upharpoonright 4 &= 2x + 9x^2 + 10x^3 + 4x^4 \\ f \upharpoonright 5 &= 2x^2 + 9x^3 + 10x^4 + 4x^5 \\ &\vdots \end{aligned}$$

Coinciding pairs of polynomials

- Let $f, g, \tilde{f}, \tilde{g} \in F[x] \setminus \{0\}$ with $\deg f \geq \deg g$ and $\deg \tilde{f} \geq \deg \tilde{g}$
- For $k \in \mathbb{Z}$, we say that (f, g) and $(\tilde{f}, \tilde{g})$ **coincide up to** k and write $(f, g) \equiv_k (\tilde{f}, \tilde{g})$ if

$$f \upharpoonright k = \tilde{f} \upharpoonright k$$
$$g \upharpoonright (k - (\deg f - \deg g)) = \tilde{g} \upharpoonright (k - (\deg \tilde{f} - \deg \tilde{g}))$$

- Remark:
  If $(f, g) \equiv_k (\tilde{f}, \tilde{g})$ and $k \geq \deg f - \deg g$, then $\deg f - \deg g = \deg \tilde{f} - \deg \tilde{g}$

Example: Coinciding pairs of polynomials

- The pairs

$$f = 7 + 2x + x^2 + x^3 + 10x^4 + 7x^5 + x^6 + 5x^7 + 9x^8 + 5x^9 + 7x^{10} \in \mathbb{Z}_{11}[x]$$
$$g = 3 + 7x + 4x^2 + 2x^3 + 2x^4 + 6x^5 + 3x^6 + 2x^7 + 4x^8 \in \mathbb{Z}_{11}[x]$$

  and

$$\tilde{f} = 1 + 5x + 9x^2 + 5x^3 + 7x^4 \in \mathbb{Z}_{11}[x]$$
$$\tilde{g} = 3 + 2x + 4x^2 \in \mathbb{Z}_{11}[x]$$

  coincide up to 4

- Indeed, we have $\deg f = 10$, $\deg g = 8$, $\deg \tilde{f} = 4$, and $\deg \tilde{g} = 2$, with

$$f \upharpoonright 4 = \tilde{f} \upharpoonright 4 = 1 + 5x + 9x^2 + 5x^3 + 7x^4$$
$$g \upharpoonright 2 = \tilde{g} \upharpoonright 2 = 3 + 2x + 4x^2$$

Quotients of coinciding pairs of polynomials

- The following lemma enables us to design a divide-and-conquer extended Euclidean algorithm by truncating the operands to division

Lemma 8 (Sufficiently coinciding pairs of polynomials have identical quotients)

Suppose that $(f, g) \equiv_{2k} (\tilde{f}, \tilde{g})$ *for* $k \in \mathbb{Z}$ *with* $k \geq \deg f - \deg g \geq 0$. *Define* $q, r, \tilde{q}, \tilde{r} \in F[x]$ *by division with quotients and remainders as follows*

$$\begin{aligned} f &= qg + r, & \deg r &< \deg g\,, \\ \tilde{f} &= \tilde{q}\tilde{g} + \tilde{r}, & \deg \tilde{r} &< \deg \tilde{g}\,. \end{aligned}$$

Then, $q = \tilde{q}$ *and at least one of the following holds* $(g, r) \equiv_{2(k-\deg q)} (\tilde{g}, \tilde{r})$ *or* $r = 0$ *or* $k - \deg q < \deg g - \deg r$.

Proof.

Exercise

Example: Quotient of coinciding pairs of polynomials

- The pairs

$$f = 7 + 2x + x^2 + x^3 + 10x^4 + 7x^5 + x^6 + 5x^7 + 9x^8 + 5x^9 + 7x^{10} \in \mathbb{Z}_{11}[x]$$
$$g = 3 + 7x + 4x^2 + 2x^3 + 2x^4 + 6x^5 + 3x^6 + 2x^7 + 4x^8 \in \mathbb{Z}_{11}[x]$$

and

$$\tilde{f} = 1 + 5x + 9x^2 + 5x^3 + 7x^4 \in \mathbb{Z}_{11}[x]$$
$$\tilde{g} = 3 + 2x + 4x^2 \in \mathbb{Z}_{11}[x]$$

coincide up to 4, with $4 \geq \deg f - \deg g = 2$

- Accordingly (by Lemma 8), the quotients agree:

$$f \operatorname{quo} g = 9 + 10x + 10x^2$$
$$\tilde{f} \operatorname{quo} \tilde{g} = 9 + 10x + 10x^2$$

Quotient sequences of coinciding pairs of polynomials

- Now let us study what happens in the extended Euclidean algorithm if we execute it for two inputs, (r_0, r_1) and $(\tilde{r}_0, \tilde{r}_1)$, with $\deg r_0 \geq \deg r_1 \geq 0$ and $\deg \tilde{r}_0 \geq \deg \tilde{r}_1 \geq 0$:

$$\begin{aligned} r_0 &= q_1 r_1 + r_2, & \tilde{r}_0 &= \tilde{q}_1 \tilde{r}_1 + \tilde{r}_2 \\ r_1 &= q_2 r_2 + r_3, & \tilde{r}_1 &= \tilde{q}_2 \tilde{r}_2 + \tilde{r}_3 \\ &\vdots & &\vdots \\ r_{i-1} &= q_i r_i + r_{i+1}, & \tilde{r}_{i-1} &= \tilde{q}_i \tilde{r}_i + \tilde{r}_{i+1} \\ &\vdots & &\vdots \\ r_{\ell-1} &= q_\ell r_\ell, & \tilde{r}_{\tilde{\ell}-1} &= \tilde{q}_{\tilde{\ell}} \tilde{r}_{\tilde{\ell}} \end{aligned}$$

- In particular, our interest is on the case $(r_0, r_1) \equiv_{2k} (\tilde{r}_0, \tilde{r}_1)$ ...

Quotient sequences of coinciding pairs of polynomials

- We can now study the execution on two *coinciding* inputs (r_0, r_1) and $(\tilde{r}_0, \tilde{r}_1)$ with $\deg r_0 \geq \deg r_1 \geq 0$ and $\deg \tilde{r}_0 \geq \deg \tilde{r}_1 \geq 0$ as follows

Lemma 9 (Identical quotient sequences up to the halting threshold)

Let $k \in \mathbb{Z}$ with $(r_0, r_1) \equiv_{2k} (\tilde{r}_0, \tilde{r}_1)$. Then, $h(k) = \tilde{h}(k)$ with $q_i = \tilde{q}_i$ for all $i = 1, 2, \ldots, h(k)$.

Proof sketch.

By induction on i and using Lemma 8 for the induction step, the following holds for all $0 \leq i \leq h(k)$: we have $i \leq \tilde{h}(k)$, $q_i = \tilde{q}_i$, and at least one of the following holds: $i = h(k)$ or $(r_i, r_{i+1}) \equiv_{2(k-\sum_{j=1}^{i} m_j)} (\tilde{r}_i, \tilde{r}_{i+1})$. □

Example: Quotient sequences of coinciding pairs

- Let us run the extended Euclidean algorithm for a pair of polynomials in $\mathbb{Z}_{11}[x]$:

i	q_i	r_i	s_i	t_i
0		$7+x+3x^2+5x^3+9x^4+10x^5+7x^6$	1	0
1	4	$4+10x+7x^2+4x^3+7x^4+4x^5+10x^6$	0	1
2	$4+2x$	$2+5x+8x^2+3x^4+5x^5$	1	7
3	$4+10x$	$7+8x+9x^2+10x^3+6x^4$	$7+9x$	$6+8x$
4	$2+3x$	$7+2x+2x^2+2x^3$	$6+4x+9x^2$	$5+7x+8x^2$
5	$10+9x$	$4+5x+10x^2$	$6+5x+3x^2+6x^3$	$7+x+7x^2+9x^3$
6	$4+8x$	$4x$	$1+10x+x^3+x^4$	$1+6x^2+x^3+7x^4$
7	x	4	$2+x+2x^3+10x^4+3x^5$	$3+4x+5x^2+x^3+8x^4+10x^5$
8		0	$1+8x+10x^2+x^3+10x^4+x^5+8x^6$	$1+8x+2x^2+7x^3+6x^4+3x^5+x^6$

- Here is a run on a pair that coincides with the first pair up to length $2k = 4$:

i	q_i	r_i	s_i	t_i
0		$3+5x+9x^2+10x^3+7x^4$	1	0
1	4	$7+4x+7x^2+4x^3+10x^4$	0	1
2	$4+2x$	$8+3x^2+5x^3$	1	7
3	$4+10x$	$8+10x+6x^2$	$7+9x$	$6+8x$
4	$6x$	$9+x$	$6+4x+9x^2$	$5+7x+8x^2$
5	$8+7x$	8	$7+6x+9x^2+x^3$	$6+2x^2+7x^3$
6		0	$5+6x+5x^2+6x^3+4x^4$	$1+9x+3x^2+7x^3+6x^4$

- Observe that the quotient sequences agree up to total degree $\deg q_1 + \deg q_2 + \ldots + \deg q_{h(k)} \leq k$ with $h(k) = 3$

A divide-and-conquer extended Euclidean algorihtm

- We now use Lemma 9 to design a fast divide-and-conquer version of the extended Euclidean algorihtm
- For a given input $(r_0, r_1) \in F[x]^2$ with $\deg r_0 \geq \deg r_1 \geq 0$ and halting parameter $k \geq 0$, the key idea is to truncate the input using the "$\upharpoonright$"-operator and build the quotient sequence $q_1, q_2, \ldots, q_{h(k)}$ using two recursive calls with halting parameter at most $\lfloor k/2 \rfloor$ each
- That is, the idea essentially to use the first recursive call to recover $q_1, q_2, \ldots, q_{h(\lfloor k/2 \rfloor)}$, then compute (as needed) the next quotient $q_{h(\lfloor k/2 \rfloor)+1}$ explicitly, and then make a second recursive call (as needed) to recover the rest of the quotient sequence $q_1, q_2, \ldots, q_{h(k)}$
- With careful implementation, this leads to an algorithm that runs in $O(M(k) \log k)$ operations in F
- Before describing the algorithm in detail, let us recall some further terminology ...

Invariants of the extended Euclidean algorithm

- Recall the matrices

$$R_0 = \begin{bmatrix} s_0 & t_0 \\ s_1 & t_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \quad \text{for } i = 1, 2, \ldots, \ell$$

  and $R_i = Q_i Q_{i-1} \cdots Q_1 R_0 \in F[x]^{2\times 2}$ for $i = 0, 1, \ldots, \ell$ from the analysis of the traditional extended Euclidean algorithm in Problem Set 1

- We recall that for all $i = 0, 1, \ldots, \ell$ we have $R_i = \begin{bmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{bmatrix}$ and $R_i \begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = \begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix}$

- Our algorithm design will be such that on input (r_0, r_1) and k it produces as output (i) the value $h(k)$, (ii) the quotient sequence $q_1, q_2, \ldots, q_{h(k)}$, and (iii) the matrix $R_{h(k)}$ ...

Truncating inputs to the extended Euclidean algorithm

- Let us write $h(k), q_1, q_2, \ldots, q_{h(k)}, R_{h(k)} \leftarrow \operatorname{extgcd}(k, r_0, r_1)$ to indicate that the algorithm produces the output $h(k), q_1, q_2, \ldots, q_{h(k)}, R_{h(k)}$ on input k, r_0, r_1 with $\deg r_0 \geq \deg r_1 \geq 0$
- Lemma 9 now implies that we have

$$\operatorname{extgcd}(k, r_0, r_1) = \operatorname{extgcd}\Big(k, r_0 \upharpoonright 2k, r_1 \upharpoonright (2k - (\deg r_0 - \deg r_1))\Big) \tag{28}$$

- In particular, we can assemble the output recursively so that the input polynomials to each recursive call are truncated in degree to the minimum enabled by (28)
- We are now ready for the detailed pseudocode of the algorithm ...

A divide-and-conquer extended Euclidean algorithm I

- Let F be a field and let $k \in \mathbb{Z}$ and $r_0, r_1 \in F[x]$ with $\deg r_0 \geq \deg r_1$ and $r_0 \neq 0$ be given as input

1. If $k < \deg r_0 - \deg r_1$ holds, then return with output $h(k) \leftarrow 0$ and $R_{h(k)} \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
2. If $k = 0$ and $\deg r_0 = \deg r_1$ hold, then return with output $h(k) \leftarrow 1$, $q_1 = \frac{\operatorname{lc} r_0}{\operatorname{lc} r_1}$, and $R_{h(k)} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -\frac{\operatorname{lc} r_0}{\operatorname{lc} r_1} \end{bmatrix}$
3. Set $k_1 \leftarrow \lfloor k/2 \rfloor$
4. Make the first recursive call $h_1, q_1^{(1)}, q_2^{(1)}, \ldots, q_{h_1}^{(1)}, R^{(1)} \leftarrow \operatorname{extgcd}\big(k_1, r_0 \upharpoonright 2k_1, r_1 \upharpoonright (2k_1 - (\deg r_0 - \deg r_1))\big)$
5. Compute the matrix-vector product $\begin{bmatrix} \tilde{r}_{h_1} \\ \tilde{r}_{h_1+1} \end{bmatrix} \leftarrow R^{(1)} \begin{bmatrix} r_0 \upharpoonright 2k \\ r_1 \upharpoonright (2k - (\deg r_0 - \deg r_1)) \end{bmatrix}$

A divide-and-conquer extended Euclidean algorithm II

6. If $\deg q_1^{(1)} + \deg q_2^{(1)} + \ldots + \deg q_{h_1}^{(1)} + \deg \tilde{r}_{h_1} - \deg \tilde{r}_{h_1+1} > k$ holds, then return with output $h(k) \leftarrow h_1$, $q_1, q_2, \ldots, q_{h(k)} \leftarrow q_1^{(1)}, q_2^{(1)}, \ldots, q_{h_1}^{(1)}$, and $R_{h(k)} \leftarrow R^{(1)}$
7. Compute the quotient $q_{h_1+1} \leftarrow \tilde{r}_{h_1}$ quo $\tilde{r}_{h_1+1}$ and the matrix $Q_{h_1+1} \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & -q_{h_1+1} \end{bmatrix}$
8. Compute the remainder $\tilde{r}_{h_1+2} \leftarrow \tilde{r}_{h_1} - q_{h_1+1}\tilde{r}_{h_1+1}$
9. Set $k_2 \leftarrow k - (\deg q_1^{(1)} + \deg q_2^{(1)} + \ldots + \deg q_{h_1}^{(1)} + \deg q_{h_1+1})$
10. Make the second recursive call
    $h_2, q_1^{(2)}, q_2^{(2)}, \ldots, q_{h_2}^{(2)}, R^{(2)} \leftarrow \operatorname{extgcd}\big(k_2, \tilde{r}_{h_1+1} \upharpoonright 2k_1, \tilde{r}_{h_1+2} \upharpoonright (2k_1 - (\deg \tilde{r}_{h_1+1} - \deg \tilde{r}_{h_1+2}))\big)$
11. Return with output $h(k) \leftarrow h_1 + 1 + h_2$,
    $q_1, q_2, \ldots, q_{h(k)} \leftarrow q_1^{(1)}, q_2^{(1)}, \ldots, q_{h_1}^{(1)}, q_{h_1+1}, q_1^{(2)}, q_2^{(2)}, \ldots, q_{h_2}^{(2)}$, and
    $R_{h(k)} \leftarrow R^{(2)} Q_{h_1+1} R^{(1)}$

Remarks and analysis

- Caveat: In Step 1 we may have $\deg r_1 = -\infty$ (that is, $r_1 = 0$) and in Step 6 we may have $\deg \tilde{r}_{h_1+1} = -\infty$ (that is, $\tilde{r}_{h_1+1} = 0$)
- After Step 1 it holds that $k \geq \deg r_0 - \deg r_1 \geq 0$, after Step 2 it holds that $k \geq 1$ and $\deg r_0 > \deg r_1 \geq 0$; thus, $0 \leq k_1 \leq k - 1$
- After Step 5 we have
$$\deg q_1^{(1)} + \deg q_2^{(1)} + \ldots + \deg q_{h_1}^{(1)} \leq k_1$$
and, also recalling that $k_1 = \lfloor k/2 \rfloor$,
$$\deg q_1^{(1)} + \deg q_2^{(1)} + \ldots + \deg q_{h_1}^{(1)} + \deg \tilde{r}_{h_1} - \deg \tilde{r}_{h_1+1} \geq k_1 + 1 \geq \lceil k/2 \rceil$$
- Assuming that $\tilde{r}_{h_1+1} \neq 0$, we have $\deg q_{h_1+1} = \deg \tilde{r}_{h_1} - \deg \tilde{r}_{h_1+1}$
- Thus, $k_2 \leq \lfloor k/2 \rfloor \leq k - 1$
- The algorithm runs in $T(k) \leq T(k_1) + T(k_2) + O(M(k)) \leq 2T(\lfloor k/2 \rfloor) + O(M(k))$ operations in F; that is, $T(k) = O(M(k) \log k)$ operations in F

Key content for Lecture 4 (recalled)

- **Extended Euclidean algorithm** for polynomials recalled and expanded
  - The **quotient sequence**, the **Bézout coefficients**, and the **halting threshold**
- Fast extended Euclidean algorithm for polynomials by **divide and conquer**
  - The two polynomial operands **truncated** to a prefix of the highest-degree monomials determine the prefix of the quotient sequence (exercise)
- Coping with **errors in data** using **error-correcting codes**
- A family of error-correcting codes (**Reed–Solomon codes**) based on evaluation–interpolation duality for univariate polynomials
  - Key observation: low-degree polynomials have few roots (exercise)
  - Fast **encoding** and **decoding** of Reed–Solomon codes via the fast univariate polynomial toolkit and **Gao's** (2003) **decoder**

Number of roots

- Let F be a field
- A **root** of a polynomial $f \in F[x]$ is an element $\xi \in F$ with $f(\xi) = 0$

Theorem 10 (Number of roots)

A nonzero polynomial $f \in F[x]$ of degree at most d has at most d distinct roots.

Proof.

Exercise

Two distinct polynomials mostly disagree

- Let F be a field
- Let $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) \in F^e$ be a vector of e distinct elements of F
- Associate with $f \in F[x]$ the vector of evaluations

$$f(\Xi) = (f(\xi_1), f(\xi_2), \ldots, f(\xi_e)) \in F^e$$

Lemma 11 (Bounded agreement of low-degree polynomials)

Let $f_0, f_1 \in F[x]$ be distinct polynomials of degree at most d.
Then, $f_0(\Xi)$ and $f_1(\Xi)$ agree in at most d coordinates.

Proof.

The difference $f_0 - f_1 \neq 0$ is a polynomial of degree at most d and thus has at most d distinct roots

Reconstructibility from partly erroneous data

- Let $f \in F[x]$ be a polynomial of degree at most d
- Let $e \geq d + 1$ and let $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) \in F^e$ consist of distinct elements

Lemma 12 (Unique reconstructibility)

Suppose that the vectors $\Gamma \in F^e$ and $f(\Xi)$ disagree in at most $(e - d - 1)/2$ coordinates. Then, Γ uniquely identifies f

Proof.

Let $f_0, f_1 \in F[x]$ be two polynomials of degree at most d such that $f_0(\Xi)$ and $f_1(\Xi)$ each disagree with Γ in at most $(e - d - 1)/2$ coordinates. In total there are e coordinates, so $f_0(\Xi)$ and $f_1(\Xi)$ and Γ must thus all agree in at least $e - 2(e - d - 1)/2 = d + 1$ coordinates. By Lemma 11 thus $f_0 = f_1$. □

(Furthermore, we can, very inefficiently, recover f from Γ by considering in turn each vector $\tilde{\Gamma} \in F^e$ that disagrees with Γ in at most $(e - d - 1)/2$ coordinates: for each such $\tilde{\Gamma}$, interpolate f from $f(\Xi) = \tilde{\Gamma}$, and stop when f has degree at most d.)

Reed–Solomon codes

- Suppose we want to protect a sequence $\Phi = (\varphi_0, \varphi_1, \ldots, \varphi_d) \in F^{d+1}$ of elements of a field F against errors
- We may represent Φ as a polynomial $f = \varphi_0 + \varphi_1 x + \ldots + \varphi_d x^d \in F[x]$ of degree at most d
- Let $e \geq d + 1$ and let $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) \in F^e$ consist of distinct elements
- Let us use $\Psi = f(\Xi) \in F^e$ as the encoded representation of Φ
- Suppose that $\hat{\Psi}$ disagrees with Ψ in at most $(e - d - 1)/2$ coordinates. Then, Lemma 12 implies that we can recover Φ from $\hat{\Psi}$
- That is, $\hat{\Psi}$ may have up to $\lfloor (e - d - 1)/2 \rfloor$ errors and we can still recover Φ
- Encoding can be done in near-linear-time by fast batch evaluation ...
- ... but how efficiently can we decode in the presence of errors?

Example: Encoding

- Let us work with $e = 8$, $d = 3$, $F = \mathbb{Z}_{11}$, and the evaluation points $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) = (0, 1, 2, 3, 4, 5, 6, 7) \in \mathbb{Z}_{11}^e$
- Suppose we want to protect the data vector $\Phi = (5, 3, 1, 9) \in \mathbb{Z}_{11}^{d+1}$
- We view Φ as the degree-at-most-d polynomial $f = 5 + 3x + x^2 + 9x^3 \in \mathbb{Z}_{11}[x]$
- The encoded representation of Φ is

$$\Psi = f(\Xi) = (f(\xi_1), f(\xi_2), \ldots, f(\xi_e)) = (5, 7, 10, 2, 4, 4, 1, 5) \in \mathbb{Z}_{11}^e$$

Gao's (2003) decoder for Reed–Solomon codes

- Let $f \in F[x]$ be a polynomial of degree at most d
- Let $e \geq d + 1$ and let $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) \in F^e$ consist of distinct elements
- Suppose that the vectors $\Gamma \in F^e$ and $f(\Xi)$ disagree in at most $(e - d - 1)/2$ coordinates. Then, Γ uniquely identifies f (Lemma 12)
- Moreover, given Ξ, Γ, d as input, f can be computed in $O(M(e) \log e)$ operations in F (Gao [5])

Gao's decoding algorithm

- Let $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) \in F^e$ consisting of distinct elements, $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_e) \in F^e$, and $d \in \mathbb{Z}_{\geq 0}$ with $d + 1 \leq e$ be given as input
- Gao's algorithm [5] proceeds as follows:
  1. Using a subproduct tree, construct the polynomial $g_0 = \prod_{i=1}^{e}(x - \xi_i)$
  2. Interpolate the unique polynomial $g_1 \in F[x]$ of degree at most $e - 1$ that satisfies $g_1(\xi_i) = \gamma_i$ for all $i = 1, 2, \ldots, e$
  3. Apply the extended Euclidean algorithm to g_0 and g_1 to produce the consecutive remainders g_h, g_{h+1} with $\deg g_h \geq D$, and $\deg g_{h+1} < D$ for $D = (e + d + 1)/2$. Let $s_{h+1}, t_{h+1} \in F[x]$ be the associated Bézout coefficients with $g_{h+1} = s_{h+1}g_0 + t_{h+1}g_1$
  4. Divide g_{h+1} by t_{h+1} to obtain the quotient $f_1 \in F[x]$ and the remainder $r \in F[x]$ with $g_{h+1} = t_{h+1}f_1 + r$ and $\deg r < \deg t_{h+1}$
  5. Output f_1 as the result of interpolation if both $\deg f_1 \leq d$ and $r = 0$; otherwise assert decoding failure
- It is immediate that the algorithm runs in $O(M(e) \log e)$ operations in F

Example: Decoding I

- Let us work with $e = 8$, $d = 3$, $F = \mathbb{Z}_{11}$, and the evaluation points $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) = (0, 1, 2, 3, 4, 5, 6, 7) \in \mathbb{Z}_{11}^e$
- Suppose we have the vector $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_e) = (5, 7, 1, 2, 9, 4, 1, 5) \in \mathbb{Z}_{11}^e$
- First, we construct the polynomial

$$g_0 = \prod_{i=1}^{e} (x - \xi_i) = 9x + 2x^3 + 4x^4 + 9x^5 + 3x^6 + 5x^7 + x^8$$

- Then, we interpolate the polynomial

$$g_1 = 5 + 7x + 5x^2 + 2x^3 + 10x^4 + 9x^5 + 6x^6 + 7x^7$$

  that satisfies $g_1(\xi_i) = \gamma_i$ for all $i = 1, 2, \ldots, e$

Example: Decoding II

- Next we apply the extended Euclidean algorithm to g_0 and g_1 to produce the consecutive remainders g_h, g_{h+1} with $\deg g_h \geq D$, and $\deg g_{h+1} < D$ for $D = (e + d + 1)/2 = 6$ ...
- For convenience, we display the entire output of the extended Euclidean algorithm (but omitting the first Bézout coefficient sequence):

i	q_i	g_i	t_i
0		$9x + 2x^3 + 4x^4 + 9x^5 + 3x^6 + 5x^7 + x^8$	0
1	$8 + 8x$	$5 + 7x + 5x^2 + 2x^3 + 10x^4 + 9x^5 + 6x^6 + 7x^7$	1
2	$7 + 10x$	$4 + x + 3x^2 + x^3 + 7x^4 + 4x^6$	$3 + 3x$
3	$3 + 3x$	$10 + 4x + 7x^2 + 9x^3 + 6x^4 + 5x^5$	$2 + 4x + 3x^2$
4	$6 + 10x$	$7 + 3x + 3x^2 + 8x^3 + 6x^4$	$8 + 7x + x^2 + 2x^3$
5	$10 + 9x$	$1 + 4x + 3x^2 + 8x^3$	$9 + 3x + 4x^2 + 2x^4$
6	$4 + 10x$	$8 + 9x + 3x^2$	$6 + 6x + 10x^3 + 2x^4 + 4x^5$
7	$5 + 4x$	$2 + 9x$	$7 + 7x + 10x^2 + 4x^3 + 4x^4 + 8x^5 + 4x^6$
8	$10 + x$	9	$4 + 9x + 10x^2 + 5x^3 + 10x^4 + 3x^5 + 3x^6 + 6x^7$
9		0	$x + 10x^3 + 9x^4 + x^5 + 4x^6 + 3x^7 + 5x^8$

- (In a fast implementation we would of course use the divide-and-conquer extended Euclidean algoritm and would not produce the entire sequence of remainders g_i)

Example: Decoding III

- From the extended Euclidean algorithm we obtain that $h = 2$ with

$$g_{h+1} = 10 + 4x + 7x^2 + 9x^3 + 6x^4 + 5x^5$$
$$t_{h+1} = 2 + 4x + 3x^2$$

- Dividing g_{h+1} by t_{h+1} we obtain the quotient

$$f_1 = 5 + 3x + x^2 + 9x^3$$

  and the remainder $r = 0$
- In particular, the decoding is successful, and the reconstructed data vector is $(5, 3, 1, 9) \in \mathbb{Z}_{11}^{d+1}$
- Re-encoding the reconstructed vector as appropriate, we can also observe that the vector Γ has two errors, namely $f(\xi_3) = 10 \neq \gamma_3 = 2$ and $f(\xi_5) = 4 \neq \gamma_5 = 9$

Correctness I

- First, suppose that the algorithm does not assert failure
- Then, $f_1 = g_{h+1}/t_{h+1}$ has degree at most d
- Since $t_{h+1}f_1 = g_{h+1} = s_{h+1}g_0 + t_{h+1}g_1$, we have $s_{h+1}g_0 = t_{h+1}(f_1 - g_1)$ and hence for all $i = 1, 2, \ldots, e$ we have $t_{h+1}(\xi_i) = 0$ or $f_1(\xi_i) = g_1(\xi_i) = \gamma_i$
- Since g_{h+1} is the first remainder with $\deg g_{h+1} < D$ and $\deg g_0 = e$, by the structure of the Bézout coefficients we have $\deg t_{h+1} \leq e - D = (e - d - 1)/2$
- Indeed, from the definition of Bézout coefficients we have $\deg s_{h+1}, \deg t_{h+1} \leq \sum_{i=1}^{h} \deg q_i = \deg g_0 - \deg g_h \leq e - D$ since $\deg g_i + \deg q_i = \deg g_{i-1}$ and $\deg g_h \geq D$
- Since t_{h+1} has at most $\deg t_{h+1}$ roots, we have $f_1(\xi_i) \neq \gamma_i$ for at most $(e - d - 1)/2$ coordinates $i = 1, 2, \ldots, e$
- Thus, f_1 is a valid output for input Ξ, Γ, d

Correctness II

- Next, let $f \in F[x]$ be a polynomial of degree at most d, let $\Xi = (\xi_1, \xi_2, \ldots, \xi_e) \in F^e$ consist of distinct elements, and let $\Gamma = (\gamma_1, \gamma_2, \ldots, \gamma_e) \in F^e$ be a vector that disagrees with $f(\Xi)$ in at most $(e - d - 1)/2$ coordinates for $d + 1 \leq e$
- By Lemma 12, we know that Γ uniquely determines f
- We show that Gao's algorithm outputs $f_1 = f$ on input Ξ, Γ, d
- Let $B = \{i \in \{1, 2, \ldots, e\} : f(\xi_i) \neq \gamma_i\}$ be the set of "bad" coordinates
- That is, B is the set of coordinates where Γ and $f(\Xi)$ disagree
- By assumption we have $|B| \leq (e - d - 1)/2$
- To understand the operation of the algorithm, let us split the polynomials g_0 and g_1 into parts based on B and $G = \{1, 2, \ldots, e\} \setminus B$ (the "bad" and "good" coordinates)

Correctness III

- Toward this end, let
$$q = \prod_{i \in G} (x - \xi_i) \in F[x], \qquad r_0 = \prod_{i \in B} (x - \xi_i) \in F[x]$$
- It is immediate that $g_0 = qr_0$
- Let $r_1 \in F[x]$ be the unique polynomial of degree at most $(e - d - 1)/2 - 1$ with $r_1(\xi_i) = q(\xi_i)^{-1}(\gamma_i - f(\xi_i)) \neq 0$ for all $i \in B$
- Thus, we have $g_1 = qr_1 + f$
- We have that $\gcd(r_0, r_1) = 1$ since no root of r_0 is a root of r_1 and r_0 factors into a product of degree 1 polynomials
- The following lemma will imply that the algorithm outputs $f_1 = f$; we postpone the proof and give it as Lemma 13

Correctness IV

- **Gao's Lemma.** (Lemma 13 below) Let $c, d, D \in \mathbb{Z}_{\geq 0}$ and let $q, r_0, r_1, f_0, f_1 \in F[x]$ with $\gcd(r_0, r_1) = 1$, $\deg q \geq D \geq c + d + 1$, and $\deg r_i \leq c$, $\deg f_i \leq d$ for $i = 0, 1$. Run the extended Euclidean algorithm on input $g_0 = qr_0 + f_0$ and $g_1 = qr_1 + f_1$ to obtain the remainders g_h and $g_{h+1} = s_{h+1}g_0 + t_{h+1}g_1$ for $s_{h+1}, t_{h+1} \in F[x]$ with $\deg g_h \geq D$ and $\deg g_{h+1} < D$. Then, $s_{h+1} = -\alpha r_1$ and $t_{h+1} = \alpha r_0$ for some $\alpha \in F \setminus \{0\}$
- Take $f_0 = 0, f_1 = f$, $c = |B|$ in the lemma and recall that we have $D = (e + d + 1)/2$
- Thus, $c \leq (e - d - 1)/2$, $\deg q = |G| = e - |B| \geq D \geq c + d + 1$, and the lemma applies to the polynomials $g_0 = qr_0$ and $g_1 = qr_1 + f$ constructed in the algorithm
- Let $g_{h+1}, s_{h+1}, t_{h+1}$ be the output of the lemma (also constructed by the algorithm)
- Because $f_0 = 0$ and $f_1 = f$, we have $g_{h+1} = -\alpha r_1 q r_0 + \alpha r_0(qr_1 + f) = t_{h+1}f$
- In particular, the algorithm outputs $f_1 = f = g_{h+1}/t_{h+1}$ $\square$

Preparation for Gao's Lemma

- Recall the matrices
$$R_0 = \begin{bmatrix} s_0 & t_0 \\ s_1 & t_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \quad \text{for } i = 1, 2, \ldots, \ell$$
and $R_i = Q_i Q_{i-1} \cdots Q_1 R_0 \in F[x]^{2\times 2}$ for $i = 0, 1, \ldots, \ell$ from the analysis of the traditional extended Euclidean algorithm in Problem Set 1
- We recall that for all $i = 0, 1, \ldots, \ell$ we have $R_i = \begin{bmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{bmatrix}$ and $R_i \begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = \begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix}$
- Since $\det Q_i = -1$ we have $\det R_i = (-1)^i$ and thus $R_i^{-1} = (-1)^i \begin{bmatrix} t_{i+1} & -t_i \\ -s_{i+1} & s_i \end{bmatrix}$
- Since $r_{\ell+1} = 0$, we have $\begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = R_\ell^{-1} \begin{bmatrix} r_\ell \\ 0 \end{bmatrix} = \begin{bmatrix} (-1)^\ell t_{\ell+1} r_\ell \\ (-1)^{\ell+1} s_{\ell+1} r_\ell \end{bmatrix}$
- We conclude that $s_{\ell+1} = (-1)^{\ell+1} r_1 / r_\ell$ and $t_{\ell+1} = (-1)^\ell r_0 / r_\ell$

Gao's Lemma

Lemma 13 (Gao [5])

Let $c, d, D \in \mathbb{Z}_{\geq 0}$ *and let* $q, r_0, r_1, f_0, f_1 \in F[x]$ *with* $\gcd(r_0, r_1) = 1$, $\deg q \geq D \geq c + d + 1$, *and* $\deg r_i \leq c$, $\deg f_i \leq d$ *for* $i = 0, 1$. *Run the extended Euclidean algorithm on input* $g_0 = qr_0 + f_0$ *and* $g_1 = qr_1 + f_1$ *to obtain the remainders* g_h *and* $g_{h+1} = s_{h+1}g_0 + t_{h+1}g_1$ *for* $s_{h+1}, t_{h+1} \in F[x]$ *with* $\deg g_h \geq D$ *and* $\deg g_{h+1} < D$. *Then,* $s_{h+1} = -\alpha r_1$ *and* $t_{h+1} = \alpha r_0$ *for some* $\alpha \in F \setminus \{0\}$

Proof of Gao's Lemma I

- Let $r_0, r_1, \ldots, r_\ell, r_{\ell+1}$ and $q_1, q_2, \ldots, q_\ell$ be the sequences of remainders and quotients in the extended Euclidean algorithm on input r_0, r_1
- Since $\gcd(r_0, r_1) = 1$, we have $r_\ell \in F \setminus \{0\}$ and $r_{\ell+1} = 0$
- Let $s_i, t_i \in F[x]$ for $i = 0, 1, \ldots, \ell + 1$ be the associated sequence of Bézout coefficients
- For all $i = 1, 2, \ldots, \ell$, we have

$$r_{i+1} = r_{i-1} - q_i r_i \,, \quad s_{i+1} = s_{i-1} - q_i s_i \,, \quad t_{i+1} = t_{i-1} - q_i t_i \tag{29}$$

- For all $i = 2, 3, \ldots, \ell + 1$ define $g_i = s_i g_0 + t_i g_1$
- From (29) it follows that $g_{i+1} = g_{i-1} - q_i g_i$ for all $i = 1, 2, \ldots, \ell$
- Let us show that $\deg g_i$ is a monotone decreasing sequence for $i = 1, 2, \ldots, \ell$

Proof of Gao's Lemma II

- We have $r_i = s_i r_0 + t_i r_1$ for all $i = 1, 2, \dots, \ell + 1$. Furthermore, $\deg s_i \leq c$ and $\deg t_i \leq c$ for all $i = 1, 2, \dots, \ell + 1$
- Since $g_0 = qr_0 + f_0$, $g_1 = qr_1 + f_1$, and $g_i = s_i g_0 + t_i g_1$, for all $i = 0, 1, \dots, \ell$ we have $g_i = qr_i + s_i f_0 + t_i f_1$
- Since $\deg(s_i f_0 + t_i f_1) \leq c + d$ and $\deg q \geq D \geq c + d + 1$, we have $\deg g_i = \deg q + \deg r_i \geq D$ for all $i = 0, 1, \dots, \ell$
- Since $\deg r_i$ is monotone decreasing for $i = 1, 2, \dots, \ell$, we have that the same holds for $\deg g_i$
- Thus, we have that $g_0, g_1, \dots, g_\ell$ and $q_1, q_2, \dots, q_\ell$ form a prefix of the sequence of remainders and quotients in the extended Euclidean algorithm on input g_0, g_1
- Since $\deg r_\ell = 0$, we have $\deg g_\ell = \deg q \geq D$

Proof of Gao's Lemma III

- Since $s_{\ell+1} = (-1)^{\ell+1} r_1 / r_\ell$ and $t_{\ell+1} = (-1)^{\ell} r_0 / r_\ell$, we have

$$g_{\ell+1} = s_{\ell+1} g_0 + t_{\ell+1} g_1 = (-1)^{\ell} (-f_0 r_1 + f_1 r_0) / r_\ell$$

- Thus, $\deg g_{\ell+1} \leq c + d < D$ and it follows that $g_{\ell+1} = g = s g_0 + t g_1$ with $\alpha = (-1)^{\ell} / r_\ell$, $s = -\alpha r_1$, and $t = \alpha r_0$ $\square$

Recap of Lecture 4

- **Extended Euclidean algorithm** for polynomials recalled and expanded
  - The **quotient sequence**, the **Bézout coefficients**, and the **halting threshold**
- Fast extended Euclidean algorithm by **divide and conquer**
  - The two operands **truncated** to a prefix of the highest-degree monomials determine the prefix of the quotient sequence (exercise)
- Coping with **errors in data** using **error-correcting codes**
- A family of error-correcting codes (**Reed–Solomon codes**) based on evaluation–interpolation duality for univariate polynomials
  - Key observation: low-degree polynomials have few roots (exercise)
  - Fast **encoding** and **decoding** of Reed–Solomon codes via the fast univariate polynomial toolkit and **Gao's** (2003) **decoder**

Problem Set 4 – I

1. Let F be a field. Show that a nonzero polynomial $f \in F[x]$ of degree at most d has at most d distinct roots.

   Hints: To reach a contradiction, assume that you have at least $d + 1$ distinct roots. Recall what we know about Vandermonde matrices from our earlier problem sets.

Problem Set 4 – II

2. Reed–Solomon codes.
   (a) Encoding. Suppose we want to encode the data vector $\Phi = (7, 6, 5, 4, 3) \in \mathbb{F}_{11}^5$ using the evaluation points $\Xi = (0, 1, 2, 3, 4, 5, 6) \in \mathbb{F}_{11}^7$. Find the encoding $\Psi = f(\Xi) \in \mathbb{F}_{11}^7$.
   (b) Decoding in the presence of errors. Suppose that $\Xi = (1, 2, 3, 4, 5, 6) \in \mathbb{F}_{13}^6$ and that $\Gamma = (3, 8, 6, 0, 7, 1) \in \mathbb{F}_{13}^6$. Find the unique polynomial $f \in \mathbb{F}_{13}[x]$ of degree at most 1 such that $f(\Xi)$ agrees with Γ in all but at most 2 coordinates, or conclude that no such f exists.

Hints: For part (a), we have $f = 7 + 6x + 5x^2 + 4x^3 + 3x^4 \in \mathbb{F}_{11}[x]$, $d = 4$, and $e = 7$. For part (b), we have $d = 1$ and $e = 6$. One possibility to decode is to try out all polynomials f of degree at most 1 over $\mathbb{F}_{13}$. How many such polynomials are there? Another is to use Gao's algorithm.

Problem Set 4 – III

3. Coinciding pairs of polynomials and polynomial quotient. Let us study the two pairs of polynomials

$$f = 4 + 5x + 3x^2 + 2x^3 + 9x^4 + 8x^5 + x^6 + 3x^7 + 9x^8 + 5x^9 + 7x^{10} \in \mathbb{Z}_{11}[x],$$
$$g = 5 + 7x + 5x^2 + 5x^3 + x^4 + 7x^5 + 4x^6 + 5x^7 + 8x^8 \in \mathbb{Z}_{11}[x]$$

and

$$\tilde{f} = x^6 + 3x^7 + 9x^8 + 5x^9 + 7x^{10} \in \mathbb{Z}_{11}[x],$$
$$\tilde{g} = 4x^6 + 5x^7 + 8x^8 \in \mathbb{Z}_{11}[x].$$

Observe that $(f, g) \equiv_4 (\tilde{f}, \tilde{g})$. Using the classical algorithm for polynomial division (recall Lecture 1), divide f by g and divide $\tilde{f}$ by $\tilde{g}$. Observe that both divisions produce the same quotient. Using the structure of the classical algorithm, justify why the two divisions must produce the same quotient.

Problem Set 4 – IV

Hints: Study carefully how the classical division algorithm obtains the coefficients of the quotient, one coefficient at a time. Which coefficients of the dividend and the divisor can have an effect on a particular coefficient of the quotient?

Problem Set 4 – V

4. Coinciding pairs of polynomials, polynomial quotient, and further coincidence. Let F be a field and let $f, g, \tilde{f}, \tilde{g} \in F[x]$ with $\deg f \geq \deg g \geq 0$ and $\deg \tilde{f} \geq \deg \tilde{g} \geq 0$. Suppose that $(f, g) \equiv_{2k} (\tilde{f}, \tilde{g})$ for $k \in \mathbb{Z}$ with $k \geq \deg f - \deg g \geq 0$. Define $q, r, \tilde{q}, \tilde{r} \in F[x]$ by division with quotients and remainders as follows

$$
\begin{aligned}
f &= qg + r, && \deg r < \deg g\,, \\
\tilde{f} &= \tilde{q}\tilde{g} + \tilde{r}, && \deg \tilde{r} < \deg \tilde{g}\,.
\end{aligned}
$$

Prove that we have $q = \tilde{q}$ and at least one of the following holds: $(g, r) \equiv_{2(k-\deg q)} (\tilde{g}, \tilde{r})$ or $r = 0$ or $k - \deg q < \deg g - \deg r$.

Hints: Recall that $(f, g) \equiv_{2k} (\tilde{f}, \tilde{g})$ holds if and only if $f \upharpoonright 2k = \tilde{f} \upharpoonright 2k$ and $g \upharpoonright (2k - (\deg f - \deg g)) = \tilde{g} \upharpoonright (2k - (\deg \tilde{f} - \deg \tilde{g}))$. Show first that without loss generality (by multiplying each pair (f, g) and $(\tilde{f}, \tilde{g})$ with x^m for a nonnegative integer m as necessary), we can assume that $\deg f = \deg \tilde{f}$. Then conclude that

Problem Set 4 – VI

$k \geq \deg f - \deg g \geq 0$ implies $\deg g = \deg \tilde{g}$. To show that $q = \tilde{q}$ holds, study the identity

$$f - \tilde{f} = q(g - \tilde{g}) + (q - \tilde{q})\tilde{g} + r - \tilde{r}$$

and seek to control the degrees of the differences $f - \tilde{f}$, $g - \tilde{g}$, and $r - \tilde{r}$ from above. For example, $f \upharpoonright 2k = \tilde{f} \upharpoonright 2k$ and $\deg f = \deg \tilde{f}$ imply that we have $\deg(f - \tilde{f}) < \deg f - 2k$. Finally, show that $r \neq 0$ and $k - \deg q \geq \deg g - \deg r$ together imply $(g, r) \equiv_{2(k - \deg q)} (\tilde{g}, \tilde{r})$.

5. Identity testing and probabilistically checkable proofs

Computer Science Club, St Petersburg
17–18 November 2018

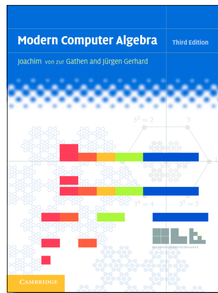
Petteri Kaski
Department of Computer Science
Aalto University

Recap of last lecture

- **Extended Euclidean algorithm** for polynomials recalled and expanded
  - The **quotient sequence**, the **Bézout coefficients**, and the **halting threshold**
- Fast extended Euclidean algorithm for polynomials by **divide and conquer**
  - The two polynomial operands **truncated** to a prefix of the highest-degree monomials determine the prefix of the quotient sequence (exercise)
- Coping with **errors in data** using **error-correcting codes**
- A family of error-correcting codes (**Reed–Solomon codes**) based on evaluation–interpolation duality for univariate polynomials
  - Key observation: low-degree polynomials have few roots (exercise)
  - Fast **encoding** and **decoding** of Reed–Solomon codes via the fast univariate polynomial toolkit and **Gao's** (2003) **decoder**

Have: Near-linear-time toolbox for univariate polynomials

- Multiplication
- Division (quotient and remainder)
- Batch evaluation
- Interpolation
- Extended Euclidean algorithm (gcd)
- Interpolation from partly erroneous data



Chapter 5

A NEW ALGORITHM FOR DECODING REED-SOLOMON CODES

Shuhong Gao
Department of Mathematical Sciences
Clemson University,
Clemson, SC 29634-0975, USA.

Abstract A new algorithm is developed for decoding Reed-Solomon codes. It uses fast Fourier transforms and computes the message symbols directly without explicitly finding error locations or error magnitudes. In the decoding radius (up to half of the minimum distance), the new method is easily adapted for error and erasure decoding. It can also detect all errors outside the decoding radius. Compared with the Berlekamp-Massey algorithm, discovered in the late 1960's, the new method seems simpler and more natural yet it has a similar time complexity.

1. Introduction

Reed-Solomon codes are the most popular codes in practical use today with applications ranging from CD players in our living rooms to spacecrafts in deep space exploration. Their main advantage lies in two facts: high capability of correcting both random and burst errors; and existence of efficient decoding algorithm for them, namely the Berlekamp-Massey algorithm, discovered in the late 1960's [1, 9]. The Berlekamp-Massey

Motivation for this lecture

- In the last lecture we encountered **uncertainty** in computation
- We saw how to cope with uncertainty in the form of **errors in data** by using **error-correcting codes**
- In this lecture we look at (fine-grained) **proof systems** and **errors in computation** ...
- Our motivation is to be able to **delegate computation** ...

Delegating computation

Client

modest resources
reliable

Problem instance

Solution

Service-provider

massively SIMD-parallel resources
error-prone

- How to verify that the solution is correct ?

- How to design an algorithm to tolerate (a small number of) errors *during computation* ?

- How to convince the client or a third party that the solution is correct ?

Key content for Lecture 5

- We look at yet further applications of the evaluation–interpolation duality and randomization in algorithm design
- Randomized **identity testing** for polynomials and matrices (exercise)
- **Delegating computation** and **proof systems**
- **Completeness** and **soundness** of a proof system,
  cost of **preparing** a proof, cost of **verifying** a proof
- Williams's (2016) [14] probabilistic proof system for #CNFSAT
- Coping with **errors in computation** using error-correcting codes with multiplicative structure (Reed–Solomon codes revisited)
- Proof systems that tolerate errors during proof preparation (Björklund & K. 2016) [3]

Proof systems

- Let I be a **claim**
  (an instance of a computational problem with a yes/no (true/false) solution)
- Let us assume that I is decidable, that is, there exists an algorithm D that given I as input outputs whether I is true
- Deciding whether I is true can often be assisted by supplying a **proof** Π for I
- A **proof system** consists of a verification algorithm (the **verifier**) V that takes as input I together with a putative proof $\tilde{\Pi}$ and either accepts or rejects $\tilde{\Pi}$ as a proof for I

Completeness and soundness

- A proof system with verifier V is
  - **complete** if for every true I there exists a proof Π such that V accepts on input I and Π
  - **sound** if for every false I and every putative proof $\tilde{\Pi}$ it holds that V rejects on input I and $\tilde{\Pi}$

Probabilistic soundness

- Let us relax the notion of soundness somewhat by allowing the verifier V to make random choices during its execution
- A proof system with a randomized verifier V is **probabilistically sound** if for every false I and every putative proof $\tilde{\Pi}$ it holds that V rejects with high probability on input I and $\tilde{\Pi}$
- By "high probability" we mean with probability $1 - o(1)$ as a function of the size of I, where probability is over the random choices made by V

Efficiency (verifier)

- In addition to completeness and soundness, in general we want a proof system also to be *efficient*
- That is, V on input I and $\tilde{\Pi}$ should consume less computational resources than it takes to decide I (using the best known algorithm for deciding I)

Efficiency (prover)

- Besides verifier efficiency, a yet further aspect to a proof system are the computational resources to **prepare** a proof
- Let P be an algorithm (the **prover**) that given a claim I as input outputs whether I is true, and if I is true, also outputs a proof Π such that V accepts on input I and Π
- We would like P to be efficient in the sense that P should not consume substantially more computational resources than it takes to decide I (using the best known algorithm for deciding I)

(Some of) recent work on fine-grained proof systems

- Goldwasser, Kalai, Rothblum [7]
- Walfish and Blumberg [13]
- Carmosino, Gao, Impagliazzo, Mihajlin, Paturi, Schneider [4]
- Williams [14]
- Björklund, K. [3, 8]

- In what follows we look at Williams's [14] proof system for #CNFSAT ...

Boolean satisfiability

- Let $x_1, x_2, \dots, x_n$ be n variables that take values in $\{0, 1\}$
- A **truth assignment** A is a mapping that assigns a value in $\{0, 1\}$ to each of the variables $x_1, x_2, \dots, x_n$
- A **literal** is a variable (x_i) or its negation ($\bar{x}_i$)
- A literal x_i (respectively, $\bar{x}_i$) is **satisfied** by A if $A(x_i) = 1$ (respectively, $A(x_i) = 0$)
- A **clause** C is a set of literals
- A clause C is **satisfied** by A if at least one literal in C is satisfied by A
- A collection of clauses $C_1, C_2, \dots, C_m$ is **satisfied** by A if A satisfies every clause $C_1, C_2, \dots, C_m$

Conjunctive-normal-form satisfiability (CNFSAT)

- The **CNFSAT** problem asks, given a collection $C_1, C_2, \ldots, C_m$ of clauses over variables $x_1, x_2, \ldots, x_n$ as input, whether there exists a truth assignment that satisfies all the clauses $C_1, C_2, \ldots, C_m$
- CNFSAT is NP-complete
- The **#CNFSAT** problem asks, given a collection $C_1, C_2, \ldots, C_m$ of clauses over variables $x_1, x_2, \ldots, x_n$ as input, for the number of truth assignments that satisfy all the clauses $C_1, C_2, \ldots, C_m$
- #CNFSAT is #P-complete
- It is not known how to solve CNFSAT in worst-case time $O^*((2-\epsilon)^n)$ for any constant $\epsilon > 0$; the best known algorithms run in $O^*(2^n)$ time
- Here the $O^*(\,)$ notation suppresses a multiplicative factor polynomial in the size of the input

CNFSAT and #CNFSAT

- It is easy to convince a verifier that an instance $C_1, C_2, \ldots, C_m$ of CNFSAT is satisfiable – just give the verifier a truth assignment A that satisfies $C_1, C_2, \ldots, C_m$
- The verifier can check that A actually satisfies $C_1, C_2, \ldots, C_m$ in time $O(mn)$
- But how to convince a verifier that $C_1, C_2, \ldots, C_m$ has exactly N satisfying truth assignments?
- For example, how to convince a verifier that $C_1, C_2, \ldots, C_m$ has *no* (zero) satisfying truth assignments?

A probabilistic proof system for #CNFSAT

- Williams (2016) [14]:
  There exists a randomized algorithm V (the verifier) such that for all collections $\mathscr{C}$ of m clauses over n variables and all integers N it holds that
  1. if $\mathscr{C}$ has exactly N satisfying truth assignments, then there exists a bit string Π of length $O^*(2^{n/2})$ such that V accepts the triple $\mathscr{C}, N, \Pi$ with probability 1;
  2. if $\mathscr{C}$ does not have exactly N satisfying truth assignments, then for every bit string $\tilde{\Pi}$ it holds that V rejects the triple $\mathscr{C}, N, \tilde{\Pi}$ with probability $1 - o(1)$.

  Moreover, V runs in time $O^*(2^{n/2})$

Multivariate polynomial representation

- Let us work over $\mathbb{F}_q$, a finite field with $q \geq 2$ elements, q prime
- Let $x_1, x_2, \ldots, x_n$ be indeterminates that take values in $\mathbb{F}_q$
- Let us work with multivariate polynomials in $\mathbb{F}_q[x_1, x_2, \ldots, x_n]$
- We will transform a collection $\mathscr{C}$ of m clauses over $x_1, x_2, \ldots, x_n$ into a multivariate polynomial $p_{\mathscr{C}}(x_1, x_2, \ldots, x_n)$ such that for all $\alpha_1, \alpha_2, \ldots, \alpha_n \in \{0, 1\} \subseteq \mathbb{F}_q$ we have $p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$ if and only if the truth assignment A with $A(x_1) = \alpha_1, A(x_2) = \alpha_2, \ldots, A(x_n) = \alpha_n$ satisfies $\mathscr{C}$, and $p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0$ otherwise

A literal as a multivariate polynomial

- For a literal ℓ over the variables $x_1, x_2, \ldots, x_n$, define the multivariate polynomial

$$p_\ell(x_1, x_2, \ldots, x_n) = \begin{cases} 1 - x_i & \text{if } \ell = x_i; \\ x_i & \text{if } \ell = \bar{x}_i \end{cases}$$

- p_ℓ has degree 1
- For all $\alpha_1, \alpha_2, \ldots, \alpha_n \in \{0, 1\}$ we have $p_\ell(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0$ if and only if the truth assignment A with $A(x_1) = \alpha_1, A(x_2) = \alpha_2, \ldots, A(x_n) = \alpha_n$ satisfies ℓ, and $p_\ell(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$ otherwise

A clause as a multivariate polynomial

- Let C be a clause over the variables $x_1, x_2, \ldots, x_n$
- For a clause C, define the multivariate polynomial

$$p_C(x_1, x_2, \ldots, x_n) = 1 - \prod_{\ell \in C} p_\ell(x_1, x_2, \ldots, x_n)$$

- Since C has at most $2n$ literals, p_C has degree at most $2n$
- For all $\alpha_1, \alpha_2, \ldots, \alpha_n \in \{0, 1\}$ we have $p_C(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$ if and only if the truth assignment A with $A(x_1) = \alpha_1, A(x_2) = \alpha_2, \ldots, A(x_n) = \alpha_n$ satisfies C, and $p_C(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0$ otherwise

A collection of clauses as a multivariate polynomial

- Let $\mathscr{C}$ be a collection $C_1, C_2, \ldots, C_m$ of clauses over the variables $x_1, x_2, \ldots, x_n$
- Define the multivariate polynomial

$$p_{\mathscr{C}}(x_1, x_2, \ldots, x_n) = \prod_{j=1}^{m} p_{C_j}(x_1, x_2, \ldots, x_n)$$

- $p_{\mathscr{C}}$ has degree at most $2mn$
- For all $\alpha_1, \alpha_2, \ldots, \alpha_n \in \{0, 1\}$ we have $p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$ if and only if the truth assignment A with $A(x_1) = \alpha_1, A(x_2) = \alpha_2, \ldots, A(x_n) = \alpha_n$ satisfies $\mathscr{C}$, and $p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0$ otherwise

#CNFSAT as a multivariate polynomial

- Let us work over $\mathbb{F}_q$, a finite field with $q \geq 2$ elements, q a prime
- Let $x_1, x_2, \ldots, x_n$ be indeterminates that take values in $\mathbb{F}_q$
- Let $\mathscr{C}$ be a collection of m clauses over $x_1, x_2, \ldots, x_n$
- We now have a multivariate polynomial $p_{\mathscr{C}}(x_1, x_2, \ldots, x_n)$ of degree at most $2mn$ such that for all $\alpha_1, \alpha_2, \ldots, \alpha_n \in \{0, 1\}$ we have $p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) = 1$ if and only if the truth assignment A with $A(x_1) = \alpha_1, A(x_2) = \alpha_2, \ldots, A(x_n) = \alpha_n$ satisfies $\mathscr{C}$, and $p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) = 0$ otherwise
- That is, the number N of satisfying truth assignments to $\mathscr{C}$ satisfies

$$N \equiv \sum_{\alpha_1, \alpha_2, \ldots, \alpha_n \in \{0,1\}} p_{\mathscr{C}}(\alpha_1, \alpha_2, \ldots, \alpha_n) \pmod{q}$$

#CNFSAT as a univariate polynomial (1/2)

- Without loss of generality we may assume that n is even
- With some foresight, let us now assume that $2^{n/2+2}mn \le q \le 2^{n/2+3}mn$ (for large enough n we can find the two smallest such primes q_1, q_2 in time $O^*(2^{n/2})$, cf. [2] and [1])
- Let $a_1, a_2, \ldots, a_{n/2} \in \mathbb{F}_q[x]$ be univariate polynomials of degree at most $2^{n/2} - 1$ such that

$$\{0,1\}^{n/2} = \{(a_1(\alpha), a_2(\alpha), \ldots, a_{n/2}(\alpha)) : \alpha \in \{0, 1, \ldots, 2^{n/2} - 1\}\}$$

- In particular we can construct such polynomials $a_1, a_2, \ldots, a_{n/2}$ in time $O^*(2^{n/2})$ using fast interpolation (exercise)
- Now define the univariate polynomial $P_{\mathscr{C}} \in \mathbb{F}_q[x]$ in the indeterminate x by

$$P_{\mathscr{C}}(x) = \sum_{\alpha_{n/2+1}, \alpha_{n/2+2}, \ldots, \alpha_n \in \{0,1\}} p_{\mathscr{C}}\Big(a_1(x), a_2(x), \ldots, a_{n/2}(x), \alpha_{n/2+1}, \alpha_{n/2+2}, \ldots, \alpha_n\Big)$$

#CNFSAT as a univariate polynomial (2/2)

- Recalling from the previous slide, we have

$$P_{\mathscr{C}}(x) = \sum_{\alpha_{n/2+1},\alpha_{n/2+2},\ldots,\alpha_n \in \{0,1\}} p_{\mathscr{C}}\big(a_1(x), a_2(x), \ldots, a_{n/2}(x), \alpha_{n/2+1}, \alpha_{n/2+2}, \ldots, \alpha_n\big)$$

- We observe that $P_{\mathscr{C}}$ has degree at most $2^{n/2+1}mn \le q/2$
- Using near-linear-time algorithms for univariate polynomials, given a collection $\mathscr{C}$ of clauses and a point $\xi \in \mathbb{F}_q$ as input, we can compute the value $P_{\mathscr{C}}(\xi)$ in time $O^*(2^{n/2})$ (exercise)
- From the definition of the polynomials $a_1, a_2, \ldots, a_{n/2}$ we observe that the number N of satisfying truth assignments to $\mathscr{C}$ satisfies

$$N \equiv \sum_{\alpha=0}^{2^{n/2}-1} P_{\mathscr{C}}(\alpha) \pmod{q} \tag{30}$$

The proof string

- Recall that for large enough n we can assume that we work modulo a prime q with $2^{n/2+2}mn \leq q \leq 2^{n/2+3}mn$
- Given $\mathscr{C}$ as input, in time $O^*(2^{n/2}e)$ we can produce e evaluations of $P_{\mathscr{C}}$ at distinct points
- If $e \geq 2^{n/2+1}mn + 1$, these evaluations enable us to interpolate $P_{\mathscr{C}}$ in time $O^*(2^{n/2})$ using fast interpolation
- We can represent the prime q and the coefficients of $P_{\mathscr{C}} \in \mathbb{F}_q[x]$ (of degree at most $2^{n/2+1}mn$) as a (prefix-coded) binary string Π_q of length $O^*(2^{n/2})$
- Let q_1, q_2 be the two least primes in the interval $[2^{n/2+2}mn, 2^{n/2+3}mn]$
- Take as the proof string Π the concatenation of Π_{q_1} and Π_{q_2}

Completeness

- Suppose $\Pi = \Pi_{q_1}\Pi_{q_2}$ is a correct proof string (of length $O^*(2^{n/2})$)
- Using Π_{q_1} and Π_{q_2} together with fast batch evaluation and (30) we can recover $N \bmod q_1$ and $N \bmod q_2$ in time $O^*(2^{n/2})$, where N is the number of satisfying truth assignments to $\mathscr{C}$
- Since $0 \le N \le 2^n$ and $q_1 q_2 \ge 2^n + 1$, from $N \bmod q_1$ and $N \bmod q_2$ we can reconstruct the correct N using the Chinese Remainder Theorem
- Thus the verifier will always accept a correct triple $\mathscr{C}, \tilde{N}, \tilde{\Pi}$ with $\tilde{\Pi} = \Pi$ and $\tilde{N} = N$ in time $O^*(2^{n/2})$

Soundness (probabilistic) I

- Suppose the verifier is given as input a collection $\mathscr{C}$ of m clauses over the variables $x_1, x_2, \ldots, x_n$, an integer $\tilde{N}$, and a binary string $\tilde{\Pi}$
- The verifier first checks that $\tilde{\Pi} = \tilde{\Pi}_{q_1}\tilde{\Pi}_{q_2}$ such that $\tilde{\Pi}_{q_1}$ and $\tilde{\Pi}_{q_2}$ encode the coefficients of a polynomial $\tilde{P}$ of degree at most $2^{n/2+1}mn$ modulo the two least primes q_1 and q_2 in the interval $[2^{n/2+2}mn, 2^{n/2+3}mn]$; if this is not the case, the verifier rejects
- Next, consider each $q \in \{q_1, q_2\}$ in turn
- To verify that $\tilde{P} = P_{\mathscr{C}} \in \mathbb{F}_q[x]$ the verifier repeats the following test $\lceil \log_2 n \rceil + 1$ times: select $\xi \in \mathbb{F}_q$ independently and uniformly at random, and test that $\tilde{P}(\xi) = P_{\mathscr{C}}(\xi)$ holds; if this is not the case, the verifier rejects
- The left-hand side $\tilde{P}(\xi)$ can be evaluated in time $O^*(2^{n/2})$ using Horner's rule; the right-hand side $P_{\mathscr{C}}(\xi)$ can be evaluated in time $O^*(2^{n/2})$ using the dedicated evaluation algorithm for $P_{\mathscr{C}}$ (in the exercises)

Soundness (probabilistic) II

- Since $\tilde{P} - P_{\mathscr{C}}$ has degree at most $2^{n+1}mn \leq q/2$, if $\tilde{P} \neq P_{\mathscr{C}} \in \mathbb{F}_q[x]$ then the verifier rejects with probability at least $1 - 1/n$ (exercise)
- Thus the verifier rejects with probability $1 - o(1)$ unless the string $\tilde{\Pi}$ is in fact the correct proof string Π; from Π the verifier can recover the correct solution N and reject unless $\tilde{N} = N$; the verifier runs in time $O^*(2^{n/2})$

Complexity of preparing and verifying the proof

- Given $\mathscr{C}$ as input, in time $O^*(2^{n/2}e)$ we can produce e evaluations of $P_{\mathscr{C}}$ at distinct points modulo q
- If $e \geq 2^{n/2+1}mn + 1$, these evaluations enable us to interpolate $P_{\mathscr{C}}$ in time $O^*(2^{n/2})$ using fast interpolation
- Thus, the total effort to prepare the proof is $O^*(2^n)$, which essentially matches the best known algorithms for counting the number of satisfying assignments to $\mathscr{C}$ (that is, no algorithm that runs in worst-case time $O^*((2-\epsilon)^n)$ is known for any constant $\epsilon > 0$)
- The total effort to (probabilistically) verify the proof is $O^*(2^{n/2})$

Proof preparation with tolerance for errors [3, 8]

- Beyond #CNFSAT, a number of other computational problems admit proof systems in the following framework ...
- The proof is a polynomial $p(x)$ of degree at most d over $\mathbb{F}_q$ (one or more polynomials with Chinese Remaindering)
- **Prepare** the proof in **evaluation** representation with distinct e points

$$(\xi_1, p(\xi_1)),\ (\xi_2, p(\xi_2)),\ \ldots,\ (\xi_e, p(\xi_e))$$

- Preparation is vector-parallel, tolerates at most $(e-d-1)/2$ errors for $e \geq d+1$
- **Decode** the proof from evaluation representation to **coefficient** representation

$$p(x) = \pi_0 + \pi_1 x + \pi_2 x^2 + \ldots + \pi_d x^d$$

- **Verify** the proof by selecting a uniform random $\xi \in \mathbb{F}_q$ and testing whether

$$p(\xi) = \pi_0 + \pi_1 \xi + \pi_2 \xi^2 + \ldots + \pi_d \xi^d$$

Delegating computation

Client

modest resources
reliable

Problem instance

Solution

Service-provider

massively SIMD-parallel resources
error-prone

- How to verify that the solution is correct ?

- How to design an algorithm to tolerate (a small number of) errors *during computation* ?

- How to convince the client or a third party that the solution is correct ?

Recap of Lecture 5

- We look at yet further applications of the evaluation–interpolation duality and randomization in algorithm design
- Randomized **identity testing** for polynomials and matrices (exercise)
- **Delegating computation** and **proof systems**
- **Completeness** and **soundness** of a proof system,
  cost of **preparing** a proof, cost of **verifying** a proof
- Williams's (2016) [14] probabilistic proof system for #CNFSAT
- Coping with **errors in computation** using error-correcting codes with multiplicative structure (Reed–Solomon codes revisited)
- Proof systems that tolerate errors during proof preparation (Björklund & K. 2016) [3]

Problem Set 5 – I

1. Randomized polynomial identity testing. Let F be a field with at least q elements.
   - 1.1 Let $f, \hat{f} \in F[x]$ be polynomials of degree at most d. Show that if $f \neq \hat{f}$ then a uniform random $\xi \in F$ satisfies $f(\xi) \neq \hat{f}(\xi)$ with probability at least $1 - d/q$.
   - 1.2 Let $a, b, c \in F[x]$ be three polynomials, each of degree at most d and each given as a sequence of coefficients. Present a randomized test that verifies $c = ab$ and uses $O(d)$ operations in F. If $c = ab$ the test must accept with probability 1; if $c \neq ab$ the test must reject with probability at least $1 - d/q$.

Hints: For part (a), recall what we know about low-degree polynomials. For part (b), reduce to part (a) and carefully justify that your algorithm uses $O(d)$ operations in F.

Problem Set 5 – II

2. Testing a matrix product. Let A, B, C be three $n \times n$ matrices with entries in a field F. Present a randomized algorithm that tests whether $C = AB$ using $O(n^2)$ operations in F. When $C = AB$, your algorithm must always assert that $C = AB$. When $C \neq AB$, your algorithm must assert that $C \neq AB$ with probability at least $1/2$.

   Hints: Select a uniform random $x \in \{0, 1\}^n \subseteq F^n$. Study the probability that $Cx \neq A(Bx)$ when $C \neq AB$.

Problem Set 5 – III

3. Evaluation algorithm for the #CNFSAT proof polynomial $P_{\mathscr{C}}$. Let $\mathscr{C}$ be a collection of clauses $C_1, C_2, \ldots, C_m$ over n variables $x_1, x_2, \ldots, x_n$ taking values in $\{0, 1\}$. Present detailed pseudocode for an algorithm that, given as input $\mathscr{C}$, a prime q with $2^{n/2+2}mn \leq q \leq 2^{n/2+3}mn$, and $\xi \in \mathbb{F}_q$, computes the value $P_{\mathscr{C}}(\xi) \in \mathbb{F}_q$ in time $O(2^{n/2}(mn)^c)$ for some constant $c > 0$. Carefully justify the running time of your algorithm. You may use the near-linear-time toolbox for univariate polynomials and algorithms for modular arithmetic in $\mathbb{F}_q$ as subroutines without detailed pseudocode, but make sure that you specify with care the input to each subroutine.

   Hints: The polynomial $P_{\mathscr{C}} \in \mathbb{F}_q[x]$ is defined in the lecture slides. Observe that your algorithm needs to work for an arbitrary $\xi \in \mathbb{F}_q$, not only for $\xi \in \{0, 1\}$. Also observe that the given input is $\mathscr{C}$, q, and ξ. In particular, the polynomials $a_1, a_2, \ldots, a_{n/2}$ need to be constructed inside your algorithm.

Problem Set 5 – IV

4. Delegating matrix multiplication. Suppose you have two $n \times n$ matrices, X and Y, with entries in a finite field F with at least four elements. You want to delegate the task of computing the product matrix XY to your three friends Alice, Bob, and Charlie so that none of your three friends individually gains any information about the matrices X and Y other than the size parameter n. Describe a protocol that employs Alice, Bob, and Charlie to help you so that you obtain the product matrix XY without you yourself putting in more work than $O(n^2)$ operations in F. You can assume you have a subroutine that returns independent uniform random elements of F.

   Hints: Recall Shamir's secret sharing. Extend each matrix X, Y to a matrix whose entries are polynomials of degree at most one with coefficients in F, where the constant of each polynomial is the original matrix entry. Have Alice, Bob, and Charlie each multiply a pair of $n \times n$ matrices $X^{(\mathrm{A})}, Y^{(\mathrm{A})}$, $X^{(\mathrm{B})}, Y^{(\mathrm{B})}$, and $X^{(\mathrm{C})}, Y^{(\mathrm{C})}$ with entries in F. Recover the product matrix XY by interpolation from the products $X^{(\mathrm{A})}Y^{(\mathrm{A})}$, $X^{(\mathrm{B})}Y^{(\mathrm{B})}$, and $X^{(\mathrm{C})}Y^{(\mathrm{C})}$ that Alice, Bob, and Charlie supply to you. Carefully justify

Problem Set 5 – V

that each of your friends on his or her own does not gain any information about X and Y other than the size parameter n.

References I

[1] M. Agrawal, N. Kayal, and N. Saxena, PRIMES is in P, *Ann. of Math. (2)* 160 (2004), 781–793.
[doi:10.4007/annals.2004.160.781].

[2] R. C. Baker, G. Harman, and J. Pintz, The difference between consecutive primes. II, *Proc. London Math. Soc. (3)* 83 (2001), 532–562.
[doi:10.1112/plms/83.3.532].

[3] A. Björklund and P. Kaski, How proofs are prepared at Camelot: extended abstract, in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016* (G. Giakkoupis, Ed.). ACM, 2016, pp. 391–400.
[doi:10.1145/2933057.2933101].

References II

[4] M. L. Carmosino, J. Gao, R. Impagliazzo, I. Mihajlin, R. Paturi, and S. Schneider, Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility, in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016* (M. Sudan, Ed.). ACM, 2016, pp. 261–270. [doi:10.1145/2840728.2840746].

[5] S. Gao, A new algorithm for decoding Reed–Solomon codes, in *Communications, Information, and Network Security* (V. K. Bhargava, H. V. Poor, V. Tarokh, and S. Yoon, Eds.), Springer, 2003, pp. 55–68.

[6] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, third ed., Cambridge University Press, Cambridge, 2013. [doi:10.1017/CBO9781139856065].

References III

[7] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, Delegating computation: Interactive proofs for muggles, *J. ACM* 62 (2015), 27:1–27:64. [doi:10.1145/2699436].

[8] P. Kaski, Engineering a delegatable and error-tolerant algorithm for counting small subgraphs, in *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018.* (R. Pagh and S. Venkatasubramanian, Eds.). SIAM, 2018, pp. 184–198. [doi:10.1137/1.9781611975055.16].

[9] A. Schönhage, Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2, *Acta Informat.* 7 (1976/77), 395–398. [doi:10.1007/BF00289470].

[10] A. Schönhage and V. Strassen, Schnelle Multiplikation grosser Zahlen, *Computing (Arch. Elektron. Rechnen)* 7 (1971), 281–292.

References IV

[11] A. Shamir, How to share a secret, *Comm. ACM* 22 (1979), 612–613. [doi:10.1145/359168.359176].

[12] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992. [doi:10.1137/1.9781611970999].

[13] M. Walfish and A. J. Blumberg, Verifying computations without reexecuting them, *Commun. ACM* 58 (2015), 74–84. [doi:10.1145/2641562].

[14] R. R. Williams, Strong ETH breaks with Merlin and Arthur: Short non-interactive proofs of batch evaluation, in *31st Conference on Computational Complexity, CCC 2016, May 29 to June 1, 2016, Tokyo, Japan* (R. Raz, Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:17. [doi:10.4230/LIPIcs.CCC.2016.2].