CS487 - Symbolic Computation

University of Waterloo

Nicholas Pun

Winter 2020

Contents

List of Algorithms

Lecture 1: Introduction

1.1 Course Preview

Example 1.1: (Simplyfying Rational Expressions)

Suppose we have the two following expressions:

$$f := \frac{x+1}{x-1} - \frac{x^3 - 2x + x^2 + 2}{x^3 + 2x - x^2 - 2} + \frac{x^2+3}{x-1} \tag{1.1}$$

$$g := \frac{(x-1)^2 - x^2 - x + 2x}{(x+y+2)^{100}} \tag{1.2}$$

Question: How do we simplify these expressions to a single $\frac{poly}{poly}$ or return that it is 0?

One idea: Define a “normal” function:

1. If expression is 0, the normal function will be 0
2. If not, the normal function will be the simplest form

(More) Questions: What else do we need to consider?

- How do we represent polynomials (i.e. What data structure do we use?)
- How do we perform polynomial operations computationally?
- Do we need to consider the size of the integers in our computations?

Example 1.2: (Solving Recurrences)

Suppose we have the recurrence:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \frac{n}{2} & n > 1 \\ 1 & n = 1 \end{cases}$$

We can solve this by hand (using Master theorem or other techniques) to obtain the answer:

$$T(n) = n(1 + \log_2(n))$$

Question: How do we do this computationally?

Example 1.3

Consider the following identities:

$$\sum_{k=0}^{n} k = \frac{n(n-1)}{2} \tag{1.3}$$

$$\sum_{k=0}^{n} k^4 = \frac{n(n-1)(2n-1)(3n^3-3n-1)}{30} \tag{1.4}$$

Question: Can we return a closed form (without involving the index k) for any general expression or report that one doesn't exist?

1.2 Representation of Integers

Current computers are based on architecture with 64 bits (We will call this number of bits the word size)

Example 1.4

The unsigned long in C represents integers in exactly the range $[0, 2^{64}-1]$

Question: How do we represent larger numbers?

Idea. Use an array of word size numbers.

Any integer a can be expressed as the following summation:

$$a = (-1)^s \sum_{i=0}^{n} a_i 2^{64i}$$

where $s \in \{0,1\}$ represents the sign of a and $0 \leq a_i \leq 2^{64}-1$ are the individual elements in the array.

If we assume $0 \leq n+1 \leq 2^{63}$, then we can encode a as an array:

$$[s \cdot 2^{63} + n + 1, a_0, a_1, \ldots, a_n]$$

This is sufficient for all practical purposes.

Note. The length of a is given by: $\lfloor \log_{2^{64}} |a| \rfloor + 1 \in \mathcal{O}(\log |a|)$ words

1.3 Addition of Integers

Suppose our input is $a : a_0 + a_1\beta + a_2\beta^2 + \ldots a_n\beta^n$ and $b : b_0 + b_1\beta + b_2\beta^2 + \ldots b_m\beta^m$ (where $m \leq n$). Let $c = a + b = c_0 + c_1\beta + c_2\beta^2 + \ldots c_n\beta^n$, each $c_i = a_i + b_i$ if $i \leq m$ and $c_i = a_i$

otherwise.

$a_i + b_i$ may be greater than β. In this case, the addition creates a *carry* to the $(i+1)$-th term.

Question: How large can c get?

In particular, will our array drastically change in size?

We can begin with the case of $\beta = 2$. This gives us binary strings, a case we may be familiar with. We can simply every bit equal to 1 to obtain:

$$1 + 1 \cdot 2 + 1 \cdot 2^2 + \ldots + 1 \cdot 2^m = 2^{m+1} - 1$$

For general β this suggests the following:

$$\sum_{i=0}^{m} = (\beta - 1)\beta^i = \beta^{m+1} - 1 \tag{1.5}$$

So, given two equal length (array-wise) integers a, b:

$$\begin{aligned}(a_0 + a_1\beta + \ldots + a_m\beta^m) + (b_0 + b_1\beta + \ldots + b_m\beta^m) &\leq 2(\beta^{m+1} - 1) \\ &= (\beta^{m+1} - 2) + \beta^{m+1}\end{aligned}$$

This implies that the largest the carry bit can be is 1.

Lecture 2: Complexity of Arithmetic Operations

We want to talk about basic operations (i.e. $\{+, -, \times, \div\}$) over a ring. (Note: Division may not always be possible)

Example 2.1: (Rings)

The following rings will come up:

1. Integers ($\mathbb{Z}$)
2. Rationals ($\mathbb{Q}$)
3. Fields (E.g. $\mathbb{Z}_7$)
4. Polynomial Rings ($R[x]$), where R is any commutative ring. E.g. $\mathbb{Z}[x], \mathbb{Q}[x], \mathbb{Z}_p[x]$
5. Field of rational functions ($R(x)$). E.g. $\mathbb{Q}(x)$

2.1 Naïve upper bounds on costs

For polynomials, we are interested in $a, b \in R[x]$. We will let $n = deg(a),\ m = deg(b)$ and we will count ring operations from R.

For integers, we will count bit operations.

We'll also define the following operation: for $a \in \mathbb{Z}$, $\lg a = \begin{cases} 1 & \text{if } a = 0 \\ 1 + \lfloor \log_2 |a| \rfloor & \text{if } a \neq 0 \end{cases}$

The following table summarizes the upper bounds:

Operation	Polynomials	Integers
$a + b$	$n + m + 1$	$\lg a + \lg b$
$a - b$	$n + m + 1$	$\lg a + \lg b$
$a \times b$	$(n + 1)(m + 1)$	$(\lg a)(\lg b)$
$a = qb + r$	$(n - m + 1)(m + 1)$	$(\lg \frac{a}{b})(\lg b)$

2.1.1 Addition

$$\begin{array}{ll} a_0 + a_1x + \ldots + a_mx^m + & a_{m+1}x^{m+1} + \ldots + a_nx^n \\ b_0 + b_1x + \ldots + b_mx^m & \\ \hline c_0 + c_1x + \ldots + c_mx^m + & c_{m+1}x^{m+1} + \ldots + c_nx^n \end{array}$$

While we really only add the first $m + 1$ terms, the add operation returns a new polynomial c. As such, we really perform $\max\{m, n\} + 1 \in \Theta(n + m) + 1$ operations.

The same analysis can be used for the add operation on integers.

2.1.2 Multiplication

Consider $a = \sum^n a_i x^i, b = \sum^m b_i x^i$, and $c = a \times b = \sum^{n+m} c_k x^k$, where $c_k = \sum a_i b_j$.

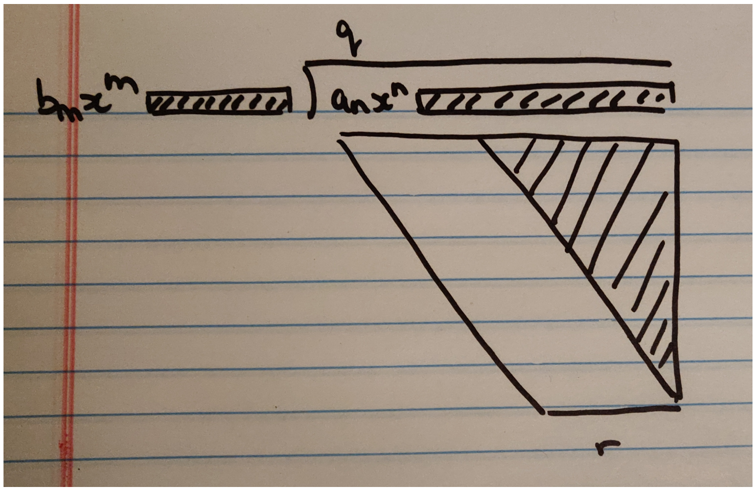
Classical "school" method: Cost is $(n+1)(m+1)$ multiplications and nm additions (exactly).

2.1.3 Division with Remainder

Given $a, b \in \mathbb{Z}$ (or $R[x]$), we want to find $q, r \in \mathbb{Z}$ with $size(r) < size(b)$ so that $a = bq + r$. Note that: $size(\cdot)$ for integers is just the magnitude, and for polynomials, $size(r) = \deg(r)$

We will require that for polynomials a, b, in $a \div b$, the constant term of b is a unit (and so an inverse exists)

Doing long division results in something that will look like the drawing below:



Within the shaded region of the trapezoid, no changes are made to the polynomial. In each step of the long division, we only perform changes to m terms within the unshaded band in the trapezoid. There are a total of $n - m$ steps (This is the resulting degree of q).

So, in total, long division of polynomials can be done in $\mathcal{O}((m+1)(n-m+1))$ operations.

Note. Why do we not just reuse our subtraction operation? We want this division operation to be primitive. The operation only performs ring operations as needed.

The same analysis can be performed on long division of integers.

2.2 Multimodular Reduction

Suppose $a \in \mathbb{Z}$, $p_1, \ldots, p_k \in \mathbb{Z}_{>1}$, with $a < p := p_1 \ldots p_k$. What is cost of computing $a \mod p_1$, $a \mod p_2$, …, $a \mod p_k$? (i.e. Obtaining the remainders)

Rough Bound: We can use the division with remainder operation. Both a and the p_i's are bounded by p. Since there are k p_i's, we will perform the operation at most k times. This gives the bound $\mathcal{O}(k(\lg p)^2)$

But, of course we can be more accurate with this bound. In total, the k division with remainders require $\sum_{i=1}^{k} C\left(\lg \frac{a}{p_i}\right)(\lg p)$ operations (The C comes from the big-$\mathcal{O}$ of the division with remainder operation). We get:

$$\begin{aligned}
&\sum_{i=1}^{k} C\left(\lg \frac{a}{p_i}\right)(\lg p) \\
&= C\sum_{i=1}^{k}\left(\lg \frac{a}{p_i}\right)(\lg p) \\
&\leq C\,(\lg p)\sum_{i=1}^{k}(\lg p) && (\lg \frac{a}{p_i} \leq \lg a \leq \lg p) \\
&\leq C(1+\log p)\sum_{i=1}^{k}(1+\log p) && (\text{Get rid of the } \lg) \\
&\leq C(2\log p)\sum_{i=1}^{k}(2\log p) && (\text{If } x > 1, 1+\log x \leq 2\log x) \\
&= 4C(\log p)^2
\end{aligned}$$

Lecture 3: Extended Euclidean Algorithm

3.1 Definitions

Went over definitions of: units, associates, zero divisors, integral domain, GCD, LCM, and Euclidean domain.

Note.

- On GCDs and LCMs: Often convenient to define them to be nonnegative to make them unique
- On $a = qb + r$, the quotient and remainder are not necessarily unique over $\mathbb{Z}$. (e.g. $7 = 5 \cdot 1 + 2 = 5 \cdot 2 - 3$). However, over $R = \mathbb{F}[x]$ (F field), the quotient and remainder *are* unique.

3.2 Extended Euclidean Algorithm

Input: $a, b \in R,\ b \neq 0$, R Euclidean Domain (e.g. $R = \mathbb{Z}$ or $R = \mathbb{F}[x]$)

Output: $s, t, g \in R$ such that $sa + tb = g$, where $g = \gcd(a, b)$

Example 3.1

One may recall the algorithm from MATH135 which begins with the following table:

s	t	r	q
1	0	a	0
0	1	b	0

At each step of the algorithm, we perform the operation: $Row_{i+1} \leftarrow Row_{i-1} - q_i Row_i$, where $q_i = \lfloor \frac{r_i}{r_{i-1}} \rfloor$. And we stop once the remainder is 0 and our answer can be read from the second last row.

For example, we can find $\gcd(91, 63)$:

s	t	r	q
1	0	91	0
0	1	63	0
1	-1	28	1
-2	3	7	2
9	-13	0	4

So, $(-2)(91) + 3(63) = 7$

Note. Behind the operation $Row_{i+1} \leftarrow Row_{i-1} - q_i Row_i$, we are really performing the 3 operations:

1. $r_{i+1} \leftarrow r_{i-1} - q_i r_i$
2. $s_{i+1} \leftarrow s_{i-1} - q_i s_i$
3. $t_{i+1} \leftarrow t_{i-1} - q_i t_i$

We build our way towards a matrix formulation of the algorithm. Consider the matrix:

$$Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix}$$

Observe that the matrix encodes the information of the *Row* operations above. To encode the operations on r_i, consider the matrix-vector multiplication:

$$Q_i \begin{bmatrix} r_{i-1} \\ r_i \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \begin{bmatrix} r_{i-1} \\ r_i \end{bmatrix} = \begin{bmatrix} r_i \\ r_{i-1} - q_i r_i \end{bmatrix} \begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix}$$

To encode the information on s_i and t_i, let $R_i = Q_i \dots Q_1$. We claim that:

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

Proof. We proceed by induction on i. This holds for $R_1 = Q_1$ since $s_1 = 0, t_1 = 1, s_2 = 1, t_2 = -q_1$.

Now suppose the statement holds for $R_1, \dots, R_{i-1}$. Then,

$$\begin{aligned} R_i &= Q_i Q_{i-1} \dots Q_1 \\ &= Q_i R_{i-1} \\ &= \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix} \begin{bmatrix} s_{i-1} & t_{i-1} \\ s_i & t_i \end{bmatrix} \\ &= \begin{bmatrix} s_i & t_i \\ s_{i-1} - q_i s_i & t_{i-1} - q_i t_i \end{bmatrix} \\ &= \begin{bmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{bmatrix} \end{aligned}$$

□

Let's formalize this:

Algorithm 1: Extended Euclidean Algorithm

```
Our input is: $a, b \in R, b \neq 0, d(a) \geq d(b))$ and $R$ a Euclidean Domain
Initialization:
    • Set $r_0 \leftarrow a$
    • Set $r_1 \leftarrow b$
for $i = 1 \ldots$ do
    • Compute $q_i = \lfloor \frac{r_i}{r_{i-1}} \rfloor$
    • Compute $r_{i+1}$ from $Q_i \begin{bmatrix} r_{i-1} \\ r_i \end{bmatrix}$
Stop loop at $i = \ell$ such that $r_{\ell+1} = 0$
```

Example 3.2

We can compute $\gcd(91, 63)$ using the matrix formulation:

$$Q_1 = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}, \; Q_2 = \begin{bmatrix} 0 & 1 \\ 1 & -2 \end{bmatrix}, \; Q_3 = \begin{bmatrix} 0 & 1 \\ 1 & -4 \end{bmatrix}$$

$$R_3 \begin{bmatrix} 91 \\ 63 \end{bmatrix} = Q_3 Q_2 Q_1 \begin{bmatrix} 91 \\ 63 \end{bmatrix} = \begin{bmatrix} -2 & 3 \\ 9 & -13 \end{bmatrix} \begin{bmatrix} 91 \\ 63 \end{bmatrix} = \begin{bmatrix} 7 \\ 0 \end{bmatrix}$$

so $(-2)(91) + 3(63) = 7$, which matches what we had before

3.3 Correctness

Proposition 3.1

$r_\ell = \gcd(r_0, r_1)$

Proof. We want to show:

1. $r_\ell | r_0$ and $r_\ell | r_1$
2. If $d | r_0$ and $d | r_1$, then $d | r_\ell$ for all $d \in R$

From the algorithm: $Q_\ell \ldots Q_1 \begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = \begin{bmatrix} r_\ell \\ 0 \end{bmatrix}$

Let $R_i = Q_i \ldots Q_1 = \begin{bmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{bmatrix}$

Then, $r_\ell \begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = \begin{bmatrix} s_\ell & t_\ell \\ s_{\ell+1} & t_{\ell+1} \end{bmatrix} = \begin{bmatrix} r_\ell \\ 0 \end{bmatrix}$

So $s_\ell r_0 + t_\ell r_1 = r_\ell$ (i.e. The second statment is true)

For the 1st statement: Each Q_i is invertible over R (Check!) So, each R_i is invertible over R and so in particular $\begin{bmatrix} r_0 \\ r_1 \end{bmatrix} = R_\ell^{-1} \begin{bmatrix} r_\ell \\ 0 \end{bmatrix}$ □

3.4 Cost Analysis

Consider $R = \mathbb{F}[x]$. Assume $\deg(r_0) \geq \deg(r_1)$.

We want to compute the cost of computing $(q_i, r_{i+1})_{1 \leq i \leq \ell}$

How many division steps ℓ?: $\ell \leq 1 + \deg(r_1)$ since $-\infty = \deg(r_{i+1}) < \deg(r_\ell) < \ldots < \deg(r_1)$

And dividing r_{i-1} by r_i with remainder costs $C(\deg(r_i + 1)(\deg(q_i + 1)))$ operations (C constant) from F.

Key observation: $\sum_{i=1}^{\ell} \deg(q_i) = \sum_{i=1}^{\ell} (\deg(r_{i-1}) - \deg(r_i)) = \deg(r_0)$

So the total cost is thus:

$$\begin{aligned}
&\leq \sum_{i=1}^{\ell} C(\deg(r_i + 1))(\deg(q_i + 1)) \\
&\leq C(\deg(r_1 + 1)) \sum_{i=1}^{\ell} (\deg(q_i + 1)) \qquad (r_i \text{ decreases by 1 in each iteration in the worst case}) \\
&\leq C(\deg(r_1 + 1))(\deg(r_0) + \ell) \\
&\in \mathcal{O}((1 + \deg r_0)(1 + \deg r_1))
\end{aligned}$$

Extension: what is the cost of computing $Q_\ell \ldots Q_1$ (Exercise: Can be done in approximately the same time)

3.5 Applications of the EEA

Computing over finite field (over a prime), given nonzero $a \in \mathbb{Z}_p$, use EEA to find $s, t \in \mathbb{Z}$ such that $sa + tp = 1$, then $sa \equiv 1 \mod p \Rightarrow s = a^{-1} \in \mathbb{Z}_p$

Rational Number Reconstruction: $\frac{-4}{5} \equiv 40 \mod 51$, call $\frac{-4}{5}$ the signed fraction, 30 the modular image and 51 the modulos m.

Input:

- A modulos $m \in \mathbb{Z}_{>0}$
- An image $u \in \mathbb{Z}_{\geq 0}$ such that $0 \leq u < m$
- Bounds $N, D \in \mathbb{Z}_{>0}$ such that $2ND < m$

Output: A signed and reduced rational number n/d such that $n/d \equiv u \mod m$, $|n| \leq N$, $d \leq D$

Fact: There is a unique n/d, if it exists, that satisfy the bounds.

Algorithm: Use EEA on m and u

Example 3.3

$u = 40, m = 51, N = D = 5$ There are 6 Q's. Look at $R_3 = Q_3 Q_2 Q_1$

Lecture 4: Polynomial Evaluation and Multiplication

4.1 Polynomial Evaluation

Suppose we were given the following polynomial:

$$f(x) = 5x^{1000} + 2x^{999} + \ldots + 3x + 2 \in \mathbb{Z}_7[x]$$

and an input $\alpha \in \mathbb{Z}_7^{300\times 300}$ (i.e. 300-by-300 matrix with elements from $\mathbb{Z}_7$)

Question: What is the cost of evaluating $f(\alpha)$?

Observations:

- The expensive operation is matrix multiplication
- It seems like we need at least 1000 multiplications to calculate each of: $\alpha^2, \alpha^3, \ldots, \alpha^{1000}$

However, by the end of the lecture, we will show a method that needs only 63 multiplications.

4.1.1 Naïve Algorithm

Algorithm 2: Naïve Algorithm

```
Input: $\alpha, a_0, a_1, \ldots a_n \in R$ ($R$ ring)
Output: $f(\alpha) \in R$, where $f(x) = a_0 + a_1x + \ldots + a_nx^n \in \mathbb{F}[x]$
Compute $\alpha^2, \alpha^3, \ldots, \alpha^n$ ($n-1$ multiplications)
Compute each $a_i\alpha^i$ $\forall i$ ($n$ multiplications)
Add ($n$ additions)
```

This method takes $2n-1$ multiplications and n additions.

4.1.2 Horner's Scheme

Horner's Scheme evaluates the polynomial in the following order:

$$f(\alpha) = (((\ldots(a_n\alpha + a_{n-1})\alpha\ldots)\alpha + a_2)\alpha + a_1)\alpha + a_0 \tag{4.1}$$

Note that each expression enclosed by parentheses cost 1 multiplication and 1 addition. Hence, overall, we have n multiplications and n additions. (We've decreased the number of multiplications by half!)

In 1954, Ostrowski asked if Horner's scheme is optimal. This lead to the development of the non-scalar complexity model.

4.1.3 Non-scalar Complexity Model

Let $R = \mathbb{F}[x, a_0, \ldots, a_n]$ be the ring of polynomials in indeterminates $x, a_0, \ldots, a_n$. We define scalar operations to be:

- Additions of 2 elements of R
- Multiplications of elements of R by fixed constants from $\mathbb{F}$

And, non-scalar operations to be: the multiplication of 2 inputs or non-scalar quantities.

Roughly speaking, the non-scalar operations will be the costly operations.

With this model in mind, let's rephrase our question: Is Horner's Scheme optimal with respect to non-scalar cost? No! (Victor Pan, 1959)

Let's calculate the non-scalar cost of Horner's method. Fix n and recall that evaluation is performed like so:

$$f(\alpha) = (((\ldots(a_n\alpha + a_{n-1})\alpha \ldots)\alpha + a_2)\alpha + a_1)\alpha + a_0$$

The innermost sum and multiplication $a_n\alpha + a_{n-1}$ is free. However, the multiplication $(a_n\alpha + a_{n-1})\alpha$ is a multiplication of two non-scalar quantities (the α). So, this counts towards our non-scalar cost.

Each subsequent multiplication will also be a non-scalar operation. In total, we perform $n-1$ non-scalar operations. However, we'll use even fewer non-scalar operations with the next method

4.1.4 Baby-Steps/Giant-Steps Method (By Patterson and Stockmeyer)

Theorem 4.1: (Patterson and Stockmeyer, 1973)

Let $f \in \mathbb{F}[x]$ of degree n. Then $f(\alpha)$ can be evaluated at any $\alpha \in \mathbb{F}$ with $2\lceil\sqrt{n}\rceil - 1$ non-scalar operations.

We'll prove this by exhibiting the algorithm. The idea is to partition f into $k \approx \sqrt{n}$ blocks of length $m \approx \sqrt{n}$. Then, we evaluate each block before evaluating the sum of the blocks. Let's see an example of this:

Example 4.1

Let $m = \lceil\sqrt{n}\rceil, k = 1 + \lceil\frac{n}{m}\rceil$, and $f(x) = 2x^8 + x^7 + 5x^6 + 2x^5 + 8x^4 + 2x^3 + x^2 + x + 4$.

So $m = 3$ is the length of each block and $k = 4$ is the upper bound on the number of blocks we'll have. Let F_0, F_1, F_2 be our blocks:

$$\begin{aligned} f(x) &= 2x^8 + x^7 + 5x^6 + 2x^5 + 8x^4 + 2x^3 + x^2 + x + 4 \\ &= \underbrace{(2x^2 + x + 5)}_{F_2} x^6 + \underbrace{(2x^2 + 8x + 2)}_{F_1} x^3 + \underbrace{(x^2 + x + 4)}_{F_0} \\ &= F_2(x) \cdot (x^3)^2 + F_1(x) \cdot (x^3) + F_0(x) \end{aligned}$$

Observe that this is just a polynomial with indeterminate x^3. Suppose we are given input α and we precompute α^2 and α^3. (This costs us $m-1 = 3-1 = 2$ non-scalar operations)

Then, we can first evaluate each $F_i(\alpha)$. This is free. (No non-scalar operations occur)

What remains is to evaluate our polynomial with input α^3, and we can use Horner's scheme. (This costs $k-1$ non-scalar operations)

Proof. Consider the algorithm:

Algorithm 3: Baby-Steps/Giant-Steps Method

```
Compute $\alpha^2, \alpha^3, \ldots, \alpha^m$ ($m \approx \sqrt{n}$ non-scalar operations)
Compute $\beta_i = F_i(\alpha)$ for $0 \leq i \leq k-1$ (0 non-scalar operations since powers of $\alpha$
  precomputed)
Compute $f(\alpha) = \beta_{k-1}(\alpha^m)^{k-1} + \beta_{k-2}(\alpha^m)^{k-2} + \ldots + \beta_0$
We can use Horner's Scheme here, which costs $k-1$ non-scalar operations.
```

In total, this requires $m-1+k-1 = 2\lceil\sqrt{n}\rceil - 1$ non-scalar operations □

4.2 Polynomial Multiplication

Input: $f, g \in R[x]$ of degree $n > 0$

Output: $f \times g$

The standard algorithm for this costs $\mathcal{O}(n^2)$ operations from R: $(n+1)^2 \times' s$ and $n^2 +' s$

4.2.1 Divide-and-Conquer Approach

Let us attempt to solve this using divide-and-conquer.

Let $n = 2^k,\ k \in \mathbb{N}$, $a, b \in R[x]$ with $\deg a, \deg b < n$ and $m = \frac{n}{2}$.

We will write $a = (A_1x^m + A_0),\ b = (B_1x^m + B_0)$, then $a \times b = A_1B_1x^n + (A_0B_1 - A_1B_0)x^m + A_0B_0$.

Example 4.2

For the following function a:

$$\begin{aligned} a &= x^5 + 3x^4 + 2x^3 + x^2 + 3x + 5 \\ &= \underbrace{(x+3)}_{A_1} x^4 + \underbrace{(2x^3 + x^2 + 3x + 5)}_{A_0} \end{aligned}$$

The cost of multiplying the two polynomials using this method is:

$$T(n) \leq \begin{cases} 4T\left(\frac{n}{2}\right) + 4n & n > 1 \\ 1 & n = 1 \end{cases} = n(5n-4) \in \Theta(n^2) \tag{4.2}$$

But ... that's not any better than what we had before ...

4.2.2 Karatsuba's Algorithm

It turns out we can reduce the number of multiplications earlier by 1. Consider writing $a \times b$ like so:

$$a \times b = A_1B_1(x^n - x^m) + (A_1 + A_0)(B_1 + B_0)x^m + A_0B_0(1 - x^m) \tag{4.3}$$

This only requires 3 multiplications:

$$T(n) \leq \begin{cases} 3T\left(\frac{n}{2}\right) + cn & n > 1 \\ 1 & n = 1 \end{cases} \in \Theta(n^{\log_2 3}) \quad (\log_2 3 \approx 1.59) \tag{4.4}$$

The calculation for $T(n)$ can be done using the Master theorem, or the following theorem:

Theorem 4.2

For $k \geq 1$:

$$T(2^k) \leq 3T(2^{k-1}) + c2^k \Rightarrow T(2^k) \leq 3^k - 2c2^k$$

Proof. We proceed by induction on k. The base case is easily verified. Assume the statement holds for some $k - 1 \geq 1$, then:

$$\begin{aligned} T(2^k) &\leq 3T(2^{k-1}) + c2^k \\ &\leq 3(3^{k-1} - 2c2^{k-1}) + c2^k \\ &= 3^k - 2c2^k \end{aligned}$$

□

and $3^k = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = n^{\log_2 3}$

4.3 Aside: Circuit Representations

We can use circuit drawings to model computations.

For example, in Figure 1, we can see that we perform no non-scalar operations.

But, in Figure 2, the multiplication at the 3rd level is a non-scalar operation.

Remark 4.1. The depth of a circuit is the parallel complexity

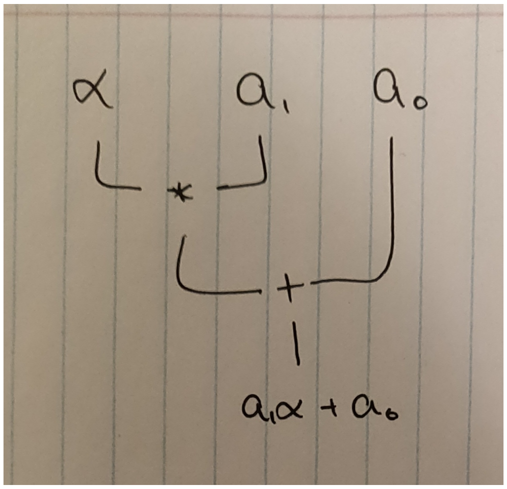


Figure 1: A circuit representation of $a_1\alpha + a_0$

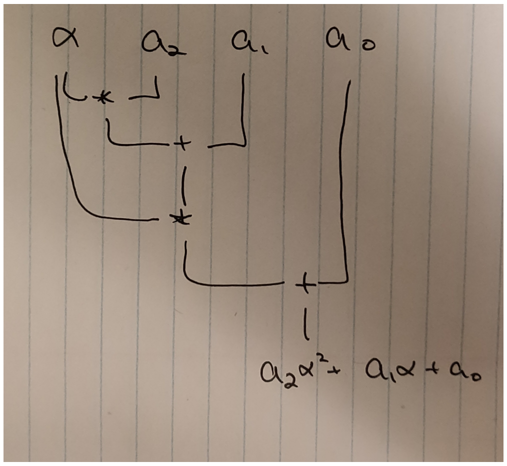


Figure 2: A circuit representation of $a_2\alpha^2 + a_1\alpha + a_0$

Lecture 5: Polynomial Multiplication using Lagrange Interpolation, Vandermonde Matrix

5.1 Polynomial Multiplication using Lagrange Interpolation

We continue our discussion on polynomial multiplication. Again, the motivation for the following algorithm is to reduce the non-scalar cost.

Theorem 5.1

Given $a, b \in \mathbb{F}[x]$, $\deg a, \deg b < n$, multiplying $a \times b$ has cost $2n - 1$ non-scalar multiplications if $\# \mathbb{F} \geq 2n - 1$

Note. The non-scalar multiplications refer to the coefficients of the polynomials we want to multiply

Idea. Use Polynomial Evaluation and Interpolation.

Let's see an example of this:

Example 5.1

We want to multiply the following polynomials $a(x) = 2 + 3x, b(x) = 1 + 2x$ using Lagrange Interpolation.

Let $c(x) = a(x) \times b(x)$ be the resulting polynomial. Note that $\deg c = 2$ so we'll need 3 evaluation points. Let $u_0 = 0, u_1 = 1, u_2 = 2$ be the 3 such points.

Evaluating our polynomials a and b at these 3 points give:

$$\begin{gathered} a(u_0) = 2,\ b(u_0) = 1 \\ a(u_1) = 5,\ b(u_1) = 3 \\ a(u_2) = 8,\ b(u_2) = 5 \end{gathered}$$

which gives us the value of c at the 3 points:

$$\begin{gathered} c(u_0) = a(u_0) \times b(u_0) = 2 \\ c(u_1) = a(u_1) \times b(u_1) = 15 \\ c(u_2) = a(u_2) \times b(u_2) = 40 \end{gathered}$$

We're nearly there! Now we have 3 data points:$(0, 2), (1, 15), (2, 40)$. Define the following Lagrange basis polynomials:

$$L_0 = \frac{(x-1)(x-2)}{(0-1)(0-2)},\ L_1 = \frac{(x-0)(x-2)}{(1-0)(1-2)},\ L_2 = \frac{(x-0)(x-1)}{(2-0)(2-1)}$$

Then,

$$c(x) = 2 \times L_0 + 15 \times L_1 + 40 \times L_2 = 2 + 7x + 6x^2$$

Proof. Consider the following algorithm:

Algorithm 4: Polynomial Multiplication using Lagrange Interpolation

```
(Our input is: $a, b \in \mathbb{F}[x]$ with $\deg a, \deg b < n$)
1 Choose $2n - 1$ evaluation points: $u_1, \dots u_{2n-1} \in \mathbb{F}$
2 Compute $\alpha_i = a(u_i)$ and $\beta_i = b(u_i)$ for $i = 1, \dots, 2n - 1$
3  Compute $\gamma_i = \alpha_i \beta_i$ for $i = 1, \dots, 2n - 1$
4 Interpolate to get $c = a \times b$ using Lagrange's formula:
      $c = \sum_{1 \leq i \leq 2n-1} \gamma_i L_i$        (5.1)
  where $L_i$ is defined as:
      $L_i = \prod_{j \neq i} \frac{x - u_j}{u_i - u_j} \in \mathbb{F}[x]$        (5.2)
```

Only Line 3 contributes to the non-scalar cost, and only $2n-1$ multiplications are made. □

5.2 A Slight Detour: The Vandermonde Matrix

Definition 5.1: (Vandermonde Matrix)

We define the Vandermonde Matrix to be the following $n \times n$ matrix:

$$VDM(u_1, u_2, \dots, u_n) = \begin{bmatrix} 1 & u_1^1 & \dots & u_1^{n-1} \\ 1 & u_2^1 & \dots & u_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & u_n^1 & \dots & u_n^{n-1} \end{bmatrix} \tag{5.3}$$

Now, given $a(x) = a_0 + a_1 x + \dots a_{n-1} x^{n-1}$, observe that we can express both polynomial evaluation and interpolation using the Vandermonde matrix.

5.2.1 Polynomial Evaluation

Given a as above and n evaluation points: $u_0, \ldots u_{n-1}$, the evaluation of a at these n points is:

$$VDM(u_0, \ldots, u_{n-1}) \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} a(u_0) \\ \vdots \\ a(u_{n-1}) \end{bmatrix} \tag{5.4}$$

5.2.2 Polynomial Interpolation

Proposition 5.1: (Determinant of a Vandermonde Matrix)

Let $V = VDM(u_1, \ldots, u_n)$. The determinant $\det(V)$ is:

$$\det(V) = \prod_{1 \leq i < j \leq n} (u_j - u_i) \tag{5.5}$$

Remark 5.1. Observe that when $u_1, \ldots, u_n$ are all distinct, then $\det(V)$ is non-zero and the matrix is invertible.

Given $(u_0, \alpha_0), (u_1, \alpha_1), \ldots, (u_{n-1}, \alpha_{n-1})$, where $u_0, \ldots u_{n-1}$ are all distinct, the interpolation of a at these n points is:

$$\begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} = VDM(u_0, \ldots, u_{n-1})^{-1} \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_{n-1} \end{bmatrix} \tag{5.6}$$

(The inverse exists by Proposition 5.1)

5.3 Another View on Polynomial Multiplication

We can rewrite the steps Algorithm 4 as expressions involving the Vandermonde matrix. Here, we'll just look at an example, and leave writing the algorithm out formally as an exercise to the reader.

Example 5.2

We will work over the field $\mathbb{Z}_7$. Consider the following polynomials:

$$f(x) = 2x^2 + 3x + 1$$
$$g(x) = x^2 + 5x + 2$$

Let $h(x) = f(x) \times g(x) = h_0 + h_1 x + \ldots + h_4 x^4$ and let's choose the evaluation points: 0, 1, 2, 3, 4 (We'll see very soon that we can choose better points)

1. Evaluation:

   To evaluate f, g at the 4 evaluation points, we'll use Equation (5.4):

$$VDM(0,1,2,3,4)\begin{array}{c} \begin{matrix} f & \; g \end{matrix} \\ \left[\begin{array}{c|c} 1 & 2 \\ 3 & 5 \\ 2 & 1 \\ 0 & 0 \\ 0 & 0 \end{array}\right] \end{array} = \begin{array}{c} \begin{matrix} f & \; g \end{matrix} \\ \left[\begin{array}{c|c} 1 & 2 \\ 6 & 1 \\ 1 & 2 \\ 4 & 2 \\ 4 & 4 \end{array}\right] \end{array}$$

   (Note that the 0's are used for padding since we want to evaluate 5 points but our polynomials are only of length 3)

2. Pointwise Multiplication:

   Now we take the resulting evaluated points and perform pointwise multiplication to obtain $h(x)$. That is:

$$\begin{aligned} h(0) &= f(0) \times g(0) = 1 \times 2 = 2 \\ h(1) &= f(1) \times g(1) = 6 \times 1 = 6 \\ h(2) &= f(2) \times g(2) = 1 \times 2 = 2 \\ h(3) &= f(3) \times g(3) = 4 \times 2 = 1 \\ h(4) &= f(4) \times g(4) = 4 \times 4 = 2 \end{aligned}$$

3. Interpolation:

   Finally, use Equation (5.6) to find $h_0, \ldots, h_4$:

$$VDM(0,1,2,3,4)^{-1} \begin{bmatrix} 2 \\ 6 \\ 2 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \\ 6 \\ 2 \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix}$$

So, $h(x) = 2x^4 + 6x^3 + 6x^2 + 4x + 2$

5.4 Choosing "Good" Evaluation Points

We'll see how we can choose better evaluation points to achieve polynomial multiplication in time $\mathcal{O}(n \log n)$ next lecture. For now, let's just make the following definition.

Definition 5.2: (Primitive n-th root of unity)

Let $n \in \mathbb{N}$ and $w \in \mathbb{F}$. w is a primitive n-th root of unity (n-PRU) if:

1. $w^n = 1$
2. n is a unit in $\mathbb{F}$
3. $w^k \neq 1$ for $1 \leq k < n$

Remark 5.2. For the 2nd property, we mean the n-fold sum of the additive identity $1_{\mathbb{F}}$ in $\mathbb{F}$. Further, we can define primitive n-th roots of unity arbitrary rings as well. In this case, the requirement that n is a unit is more significant. We will see later that the inverse of such n must exist for our particular usage of these elements.

Example 5.3: (PRUs)

1. Let $\mathbb{F} = \mathbb{C}$:
   - $w = e^{\frac{2\pi i}{8}}$ is an 8-PRU.
   - $w = i$ is an 4-PRU
   - $w = -1$ is an 2-PRU
2. Let $\mathbb{F} = \mathbb{Z}_{17}$:
   - $w = 3$ is an 16-PRU
   - $w = 7$ is an 4-PRU

Proposition 5.2

1. If w is an n-PRU, then w^{-1} is also an n-PRU
2. If w is an n-PRU and n is even, then w^2 is an $\frac{n}{2}$-PRU

Lecture 6: Discrete Fourier Transform and Fast Fourier Transform

6.1 Discrete Fourier Transform

Definition 6.1

Let w be an n-PRU in $\mathbb{F}$. Define $V(w)$ to be the following n-by-n matrix:

$$V(w) = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & w^1 & \dots & w^{n-1} \\ 1 & w^2 & \dots & w^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & w^{(n-1)} & \dots & w^{(n-1)^2} \end{bmatrix} = VDM(w^0, w^1, \dots, w^{n-1}) \tag{6.1}$$

and likewise for $V(w^{-1})$.

Theorem 6.1

Let w be an n-PRU, then $V(w) \cdot V(w^{-1}) = nI$

Proof.

$$\begin{aligned} \big(V(w)V(w^{-1})\big) &= i\text{-th row of } V(w) \times j\text{-th col of } V(w^{-1}) \\ &= \sum_{0 \le k < n} w^{ik} w^{-jk} \\ &= \sum_{0 \le k < n} w^{(i-j)k} \end{aligned}$$

If $i = j$, then the sum is $\sum_k 1 = n$

If $i \neq j$, then this is a geometric series:

$$\sum_{0 \le k < n} w^{(i-j)k} = \frac{w^{(i-j)n} - 1}{w^{(i-j)} - 1} = 0$$

since $w^{(i-j)n} = 1$ as w is an n-PRU □

Definition 6.2: (Discrete Fourier Transform)

Let $w \in \mathbb{F}$ be an n-PRU. DFT(w) is the linear map $F^n \to F^n$ defined by:

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \longmapsto \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} = V(w) \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \tag{6.2}$$

i.e. $b_j = \sum_{0 \leq k < n} a_k w^{jk}$

Note. We may write DFT$(w)(f)$, where f is a function with $\deg f = n - 1$, to denote performing the DFT on the coefficients of f

6.2 Fast Fourier Transform

Our goal is to develop a fast algorithm to evaluate DFT$(w)(f)$. The main idea is to divide-and-conquer. Let's look at a motivating example:

Let $f = a_0 + a_1 x + \ldots + a_k x^k$ (with k even). Consider the decomposition of f like so:

$$\begin{aligned} f(x) &= (a_0 + a_2x^2 + a_4x^4 + \ldots) + (a_1x + a_3x^3 + a_5x^5 + \ldots) \\ &= (a_0 + a_2x^2 + a_4x^4 + \ldots) + x(a_1 + a_3x^2 + a_5x^4 + \ldots) \\ &= \sum_{0 \leq i \leq k/2} a_{2i}x^{2i} + x \sum_{0 \leq j \leq k/2} a_{2j+1}x^{2j} \end{aligned}$$

That is, we divide the function into parts containing only even or only odd exponents. Then, we factor an x out of the odd exponents so that we only have even exponents in the sums. Define the following two functions:

$$f_{even}(x) = \sum_{0 \leq i \leq k/2} a_{2i}x^i \tag{6.3}$$

$$f_{odd}(x) = \sum_{0 \leq j \leq k/2} a_{2j+1}x^j \tag{6.4}$$

Then, f can be rewritten using these two functions like so:

$$f(x) = f_{even}(x^2) + x f_{odd}(x^2) \tag{6.5}$$

Now consider evaluating f at the 4 points: $1, i, -1, -i$. Plugging these 4 values into equation

Equation (6.5), we get the following expressions:

$$\begin{aligned} f(1) &= f_{even}(1) + (1)f_{odd}(1) \\ f(i) &= f_{even}(i^2) + (i)f_{odd}(i^2) \\ f(1) &= f_{even}(1) + (-1)f_{odd}(1) \\ f(1) &= f_{even}(i^2) + (-i)f_{odd}(i^2) \end{aligned}$$

Evaluating 1 and -1 amounts to computing $f_{even}(1)$ and $f_{odd}(1)$ and the combining the results appropriately. Likewise, we can do the same with i and $-i$

This suggests that we can reuse the computations of f_{even} and f_{odd}, which are polynomials of half the degree of f.

In general, if we can "pair up" our n points in the form:

$$(u_1, -u_1), (u_2, -u_2), \ldots, (u_{\frac{n}{2}}, -u_{\frac{n}{2}})$$

we can use Equation (6.5) to save evaluations

Lemma 6.1

Let w be an n-PRU. Then, there is always a pairing of points of the above form. More precisely:

$$w^{\frac{n}{2}+i} = -w^i$$

for $i = 1, \ldots, \frac{n}{2}$

Proof. $w^n = 1 \Rightarrow w^n - 1 = 0 \Rightarrow \left(w^{\frac{n}{2}} - 1\right)\left(w^{\frac{n}{2}} + 1\right) = 0$. So, $w^{\frac{n}{2}} = \pm 1$, but $w^{\frac{n}{2}} \neq 1$ since w is an n-PRU, so $w^{\frac{n}{2}} = -1$. Then, $w^{\frac{n}{2}+i} = w^{\frac{n}{2}} w^i = -w^i$ □

Theorem 6.2

Let n be a power of 2. Let $w \in \mathbb{F}$ be an n-PRU. Then, DFT(w) can be computed in $\mathcal{O}(n \log n)$ field operations.

Proof. Let $f = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$. We'll exhibit an algorithm to compute DFT(w)(f)

in $\mathcal{O}(n \log n)$ time:

Algorithm 5: Fast Fourier Transform (FFT)

```
    Input: $f = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$, $w \in \mathbb{F}$ an $n$-PRU.
    Output: $\mathrm{DFT}(w)(f)$
1   Compute $w^2, w^3, \ldots, w^{n-1}$
2   Recursively compute $\mathrm{DFT}(w)(f_{even})$ and $\mathrm{DFT}(w)(f_{odd})$:
        $\mathrm{DFT}(w)(f_{even}) = \begin{pmatrix} f_{even}(w^2) \\ f_{even}(w^4) \\ \vdots \\ f_{even}(w^n) \end{pmatrix} \quad \text{and} \quad \mathrm{DFT}(w)(f_{odd}) = \begin{pmatrix} f_{odd}(w^2) \\ f_{odd}(w^4) \\ \vdots \\ f_{odd}(w^n) \end{pmatrix}$
3   Compute $f(w^k) = f_{even}(w^{2k}) + w^k f_{odd}(w^{2k})$ for $k = 0, 1, \ldots, n-1$
```

Let $T(n)$ be the cost for $\deg f = n$. Line 1 costs less than n multiplications. Line 3 costs n multiplications and n additions. In total,the cost of Algorithm 5 is

$$T(n) = 2T\left(\frac{n}{2}\right) + 3n \in \mathcal{O}(n \log n)$$

□

6.3 Polynomial Multiplication using the FFT

We want to relate this method back to polynomial multiplication.

Theorem 6.3

Let $\mathbb{F}$ be a field, $n = 2^k$, $w \in \mathbb{F}$ an n-PRU. Then, polynomials in $\mathbb{F}[x]$ of degree $< \frac{n}{2}$ can be multiplied using $\mathcal{O}(n \log n)$ field ops.

Proof. Recall that polynomial multiplication can be performed by:

1. Evaluating the two functions at n points
2. Multiplying the images pointwise
3. Interpolating to obtain the multiplied function

Further, we can express evaluation and interpolation as matrix operations using the Vandermonde matrix (Example 5.2). Now, we'll use the DFT and Fast Fourier Transform algorithm to speed both operations up.

Let $a = a_0 + a_1 x + \ldots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$, $b = b_0 + b_1 x + \ldots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$ and

$$\overline{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{\frac{n}{2}-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad \overline{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{\frac{n}{2}-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where the 0's are padding elements so that the vectors are of length n.

Let $c = c_0 + c_1 + \ldots + c_n x^n = a \times b$ and $\overline{c}$ be the vector of its coefficients as with $\overline{a}$ and $\overline{b}$. Then,

$$\begin{aligned} \overline{c} &= (\mathrm{DFT}(w))^{-1} (\mathrm{DFT}(w)(a) \cdot \mathrm{DFT}(w)(b)) \\ &= \frac{1}{n} \left(\mathrm{DFT}(w^{-1})\right) (\mathrm{DFT}(w)(a) \cdot \mathrm{DFT}(w)(b)) \end{aligned}$$

(where $\cdot$ is pointwise multiplication)

And this uses $\mathcal{O}(n \log n)$ field operations □

Definition 6.3

We say $\mathbb{F}$ supports the FFT (Algorithm 5), if $\mathbb{F}$ has a 2^ℓ-PRU for any $\ell \in \mathbb{N}$

More generally, we'll use Definition 6.3 to extend Theorem 6.2.

Theorem 6.4

If $\mathbb{F}$ supports the FFT, the polynomials of degree at most n can be multiplied in $\mathcal{O}(n \log n)$ field ops.

Theorem 6.5: (Schönhage & Strassen, 1971)

Integer Multiplication can be done in time $\mathcal{O}(n \log n \log \log n)$

Theorem 6.6: (Cantor & Kaltofen, 1991)

Over any ring, polynomials of degree n can be multiplied in $\mathcal{O}(n \log n \log \log n)$

Lecture 7: Fast Division with Remainder and Newton Iteration

(See Appendix A for notes on Multiplication time)

7.1 Fast Division with Remainder

Goal: Given two polynomials:

$$\begin{aligned} a(x) &= a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \in \mathbb{F}[x] \\ b(x) &= b_m x^m + b_{m-1}x^{m-1} + \ldots + b_1 x + b_0 \in \mathbb{F}[x]. \end{aligned}$$

with $a_n, b_m \neq 0$, b monic, and $m \leq n$, find $q(x)$ and $r(x)$ such that $a(x) = q(x)b(x) + r(x)$, $\deg r < \deg b$.

7.1.1 Reversions

To solve the above problem, we'll need a new operation called reversion.

Given $a(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \in \mathbb{F}[x]$, we define the reversion of a, denoted $\text{rev}(a)$ or $\text{rev}_n(a)$ to be the following procedure:

1. Substitute x with $\frac{1}{y}$, then
2. Multiply by y^n.

This gives:

$$\begin{aligned} \text{rev}(a) = \text{rev}_n(a) &:= y^n a\left(\frac{1}{y}\right) \\ &= y^n \left(a_0 + a_1\left(\frac{1}{y}\right) + \ldots + a_n\left(\frac{1}{y^n}\right)\right) \\ &= y^n a_0 + y^{n-1}a_1 + \ldots + a_n \end{aligned} \tag{7.1}$$

So, we have reversed the ordering of the coefficients.

Remark 7.1. $\text{rev}\left(\text{rev}\left(a\right)\right) = a$

Note. Reversions don't cost any ring operations. For example, if the coefficients were stored as an array, then a reversion is just a reversal of the array.

7.1.2 Back to Fast Division

Using the idea of reversions, let's rewrite our goal. The reversion of $a(x) = q(x)b(x) + r(x)$ is:

$$\begin{aligned}
\operatorname{rev}_n(a) = y^n a\left(\frac{1}{y}\right) &= y^n\left(q\left(\frac{1}{y}\right) b\left(\frac{1}{y}\right) + r\left(\frac{1}{y}\right)\right) \\
&= y^n\left(q\left(\frac{1}{y}\right)\right) y^n\left(b\left(\frac{1}{y}\right)\right) + y^n\left(r\left(\frac{1}{y}\right)\right) \\
&= \operatorname{rev}_{n-m}(q)\operatorname{rev}_m(b) + y^{n-m+1}\operatorname{rev}_{m-1}(r)
\end{aligned}$$

For the last step: $\deg b = m$, so $\deg q = n - m$. Further, since $\deg r < \deg b$, $\deg r$ is at most $m - 1$.

And, if we take the equation modulo y^{n-m+1}, this gives:

$$\operatorname{rev}_n(a) = \operatorname{rev}_{n-m}(q)\operatorname{rev}_m(b) \pmod{y^{n-m+1}} \tag{7.2}$$

We revise our goal: Solve Equation (7.2) for the unknown $\operatorname{rev}_{n-m}(q)$

7.2 Newton Iteration

To solve Equation (7.2), we want to take the inverse of $\operatorname{rev}_m(b)$:

$$\operatorname{rev}_{n-m}(q) = \operatorname{rev}_n(a)\operatorname{rev}_m(b)^{-1} \pmod{y^{n-m+1}} \tag{7.3}$$

Generalizing this problem a bit: Given a function g and $k \in \mathbb{N}$, we want to find a function h so that $gh \equiv 1 \mod x^k$.

The main idea is to do this iteratively using Newton's method. Recall that Newton's method calculates a sequence $h_0, h_1, h_2, \ldots$ by using the formula:

$$h_{i+1} = h_i - \frac{\phi(h_i)}{\phi'(h_i)} \tag{7.4}$$

Toying around with the function for ϕ, we find that $\phi(h) = \frac{1}{h} - g$ will work for us since $1/g$ is a root of ϕ. Indeed, $\phi(1/g) = \frac{1}{1/g} - g = g - g = 0$.

Suppose we already have h_i, let's perform one iteration of newton's method with the above ϕ:

$$\begin{aligned}
h_{i+1} &= h_i - \frac{\phi(h_i)}{\phi'(h_i)} \\
&= h_i - \frac{\frac{1}{h_i} - g}{-\frac{1}{h_i^2}} \\
&= h_i + h_i - gh_i^2 \\
&= 2h_i - gh_i^2
\end{aligned}$$

How fast does this converge to our desired modulus x^k? Observe that h_i is squared in each iteration, so our precision is doubled in each step.

We summarize this in the following theorem:

Theorem 7.1

Let $h_0, h_1, h_2, \ldots \in \mathbb{F}[x]$ be such that $\deg h_i < 2^i$ and $gh_i \equiv 1 \pmod{x^{2^i}}$. Then, $h_0 = 1$ and $h_{i+1} \equiv 2h_i - gh_i^2 \pmod{x^{2^{i+1}}}$ for $i > 0$

And, this yields the following algorithm:

Algorithm 6: Quadratic Newton Iteration

```
    Input: $g \in \mathbb{F}[x]$ monic and $n = 2^r$
    Output: $h \in \mathbb{F}[x]$ such that $gh \equiv 1 \pmod{x^n}$
$h_0 := 1$
for $i = 0, 1, \ldots, r$ do
    $h_{i+1} = 2h_i - gh_i^2 \pmod{x^{2^{i+1}}}$
```

Theorem 7.2: (Time Complexity of Algorithm 6)

If $n = 2^r$, then $h_r \equiv g^{-1} \pmod{x^n}$ can be computed in $\mathcal{O}(M(n))$ field operations

Proof. Computing h_{i+1} requires at most $2M(2^{i+1}) + 2 \cdot 2^{i+1}$ field operations. (Polynomial multiplication, scalar multiplication and subtraction). Then, the total cost is:

$$\begin{aligned}
&2\sum_{i=0}^{r-1}\left(M\left(2^{i+1}\right)+2^{i+1}\right)\\
&=\left(2\sum_{i=0}^{r-1}M\left(2^{i+1}\right)\right)+\left(4\sum_{i=0}^{r-1}2^i\right)\\
&\leq\left(2\sum_{i=0}^{r-1}M\left(2^{i+1}\right)\right)+4n\\
&\leq 2M\left(2^r\right)\sum_{i=0}^{r-1}\frac{1}{2^{r-i}}+4n \qquad \left(\text{since } M(2^i)\leq\frac{M(2^r)}{2^{r-i}} \text{ by superlinearity}\right)\\
&\leq 2M\left(2^r\right)\sum_{i\geq 0}\frac{1}{2^i}+4n\\
&\in\mathcal{O}\left(M\left(2^r\right)\right)+\mathcal{O}(n)\\
&\in\mathcal{O}\left(M(n)\right)
\end{aligned}$$

□

7.3 Completing Fast Division with Remainder

We now have the necessary components to complete the algorithm for Fast Division with Remainder:

Algorithm 7: Fast Division with Remainder

```
Input: Two polynomials:
    • $a(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \in \mathbb{F}[x]$
    • $b(x) = b_m x^m + b_{m-1}x^{m-1} + \ldots + b_1 x + b_0 \in \mathbb{F}[x]$
with $a_n, b_m \neq 0$, $b$ monic, and $m \leq n$
Output: $q(x)$ and $r(x)$ such that $a(x) = q(x)b(x) + r(x)$, $\deg r < \deg b$.
Compute $\mathrm{rev}(a)$
Compute $\mathrm{rev}(b)^{-1}$ to precision $x^{n-m+1}$ using Algorithm 6
Compute $\mathrm{rev}(q) = \mathrm{rev}(a) \cdot \mathrm{rev}(b)^{-1} \pmod{x^{n-m+1}}$
$q \leftarrow \mathrm{rev}\left(\mathrm{rev}(q)\right)$
$r \leftarrow a - q \cdot b$
```

Corollary 7.1

Let $a, b \in \mathbb{F}[x]$ with $\deg a = n, \deg b = m, n \geq m$. Then, $q = a \operatorname{quo} b$ can be computed in $M(n-m)$ field ops

Proof. By Theorem 7.2 □

Corollary 7.2

For polynomials of degree at most n in $\mathbb{F}$, division with remainder requires at most $\mathcal{O}(M(n))$ operations.

Proof. Computing q uses at most $M(n-m)$ field ops and computing r uses at most $\mathcal{O}(M(n))$ field ops □

7.4 Concluding Remarks

Theorem 7.3: (Existence and Uniqueness of Inverse)

Let $g = g_0 + g_1 x + g_2 x^2 + \ldots \in F[[x]]$ have constant coefficient $g_0 = 1$. For any $k \in \mathbb{Z}_{>0}$, there exists a unique $b \in F[x]$ with $\deg b < k$ such that $bg \equiv 1 \pmod{x^k}$

Proof. We work in mod x^k. Write $g = 1 + g_1 x + \ldots + g_{k-1}x^{k-1} \pmod{x^{k-1}}$ and let $b =$

$b_0 + b_1 x + \ldots + b_{k-1}x^{k-1}$. Then,

$$bg = (b_0 + b_1 x + \ldots + b_{k-1}x^{k-1})(1 + g_1 x + \ldots + g_{k-1}x^{k-1}) \equiv 1$$

if and only if

$$\begin{array}{cccc} \mathrm{B} & \mathrm{G} & & \mathrm{I}_k \\ \begin{bmatrix} 1 & & & \\ b_1 & \ddots & & \\ \vdots & \ddots & \ddots & \\ b_{k-1} & \dots & b_1 & 1 \end{bmatrix} & \begin{bmatrix} 1 & & & \\ g_1 & \ddots & & \\ \vdots & \ddots & \ddots & \\ g_{k-1} & \dots & g_1 & 1 \end{bmatrix} & = & \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \end{array}$$

(Note: $BG = I_k$ and $GB = I_k$. This is easy to see since the multiplication of the polynomials is commutative)

if and only if

$$G \begin{bmatrix} 1 \\ b_1 \\ \vdots \\ b_{k-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

And this system has a unique solution for b. □

Note. B and G are known as Toeplitz matrices.

Lecture 8: Newton Iteration for Integers and $p$adic Inversion

I'll get back to this one ... this topic is less important

Lecture 9: Chinese Remainder Algorithm

Theorem 9.1: (Chinese Remainder Theorem)

Let R be an Euclidean Domain and $M = m_0 m_1 \dots m_{r-1}$, where $\gcd(m_i, m_j) = 1$ for $i \neq j$. (Note that this condition implies that $M = \operatorname{lcm}(m_0, \dots, m_1)$) Then,

$$R/_{(M)} \cong R/_{(m_0)} \times \dots \times R/_{(m_{r-1})} \tag{9.1}$$

Recall that in Section 2.2, that showed one direction of the bijection, i.e. the map taking $\operatorname{rem}(a, M) \mapsto (\operatorname{rem}(a, m_0), \operatorname{rem}(a, m_1), \dots, \operatorname{rem}(a, m_{r-1}))$. This takes $\mathcal{O}(\log^2 M)$ word ops.

In this lecture, we want to develop the reverse map using a similar amount of word ops. The idea of using small moduli and this Chinese Remainder algorithm (which we may refer to as "Chinese Remaindering") to obtain a pre-image will be a central idea in many of the remaining algorithms we will learn.

Example 9.1

Operations on modular residues are reflected when we perform Chinese Remainder to retrieve the preimage.

Let $m = 1001 = 7 \times 11 \times 13$, so in this example we'll consider $R = \mathbb{Z}_{1001} \cong \mathbb{Z}_7 \times \mathbb{Z}_{11} \times \mathbb{Z}_{13}$

Let $a \equiv 233 \pmod m$, $b \equiv 365 \pmod m$. By Theorem 9.1, $a \mapsto (2, 2, 12)$ and $b \mapsto (1, 2, 1)$

Consider the following operations:

1. $\operatorname{rem}(a + b, m) = (2, 2, 12) + (1, 2, 1) = (3, 4, 0) \mapsto 598 \pmod m$. Indeed: $233 + 365 \equiv 598 \pmod m$!
2. $\operatorname{rem}(a \cdot b, m) = (2, 2, 12) \cdot (1, 2, 1) = (2, 4, 12) \mapsto 961 \pmod m$. Indeed, we can check: $233 \cdot 365 = 85045 \equiv 961 \pmod m$

The main takeaway: The Chinese Remainder Theorem guarantees a unique preimage, so we can always work with small moduli and then work backwards. (Many times, it'll be much more computationally efficient to do so!)

Let's start developing the algorithm:

Goal: Given moduli $m_0, m_1, \dots m_{r-1}$ and images $v_0, v_1, \dots, v_{r-1}$. Find an $f \in R$ (R Euclidean Domain) such that $f \equiv v_i \pmod{m_i}$ for $0 \leq i \leq r - 1$

Let $M = m_0 m_1 \dots m_{r-1}$ and consider the following construction for f:

$$f = v_0 s_0 \frac{M}{m_0} + v_1 s_1 \frac{M}{m_1} + \dots + v_{r-1} s_{r-1} \frac{M}{m_{r-1}} \tag{9.2}$$

Consider $f \bmod m_0$, then each of $\frac{M}{m_1}, \dots, \frac{M}{m_{r-1}}$ will vanish and we are left with:

$$\begin{aligned} f = v_0 s_0 \frac{M}{m_0} &\equiv v_0 \pmod{m_0} \\ \Rightarrow s_0 \frac{M}{m_0} &\equiv 1 \pmod{m_0} \end{aligned}$$

To determine s_0 recall that we can apply EEA to get:

$$s_0 \frac{M}{m_0} + t_0 m_0 = 1$$

Since the choice of s_0 was arbitrary, this works for any s_i for $0 \leq i \leq r-1$. So our algorithm is:

Algorithm 8: Chinese Remainder Algorithm

```
1 Compute $M = m_0 m_1 \dots m_{r-1}$
2 Compute $\frac{M}{m_i}$ for $0 \leq i \leq r-1$
3 Compute $s_i$ such that $s_i \frac{M}{m_i} + t_i m_i = 1$ using EEA
4 Compute $f = v_0 s_0 \frac{M}{m_0} + v_1 s_1 \frac{M}{m_1} + \dots + v_{r-1} s_{r-1} \frac{M}{m_{r-1}}$
```

Theorem 9.2

If each v_i is in the range $[0, m_i - 1]$, then the cost of the Chinese Remainder algorithm is $\mathcal{O}(\log^2 M)$ word operations

Proof. Each step is bounded by $\mathcal{O}(\log^2 M)$ word operations □

Example 9.2

Let $m_0, m_1, m_2 = 7, 11, 13$ and $v_0, v_1, v_2 = 2, 2, 12$, so $M = 1001$

We want to find s_0, s_1, s_2 so that $f \equiv v_i \pmod{m_i}$ for each i, where:

$$\begin{aligned} f &= 2 \times \frac{7 \times 11 \times 13}{7} \times s_0 + 2 \times \frac{7 \times 11 \times 13}{11} \times s_1 + 12 \times \frac{7 \times 11 \times 13}{13} \times s_2 \\ &= 2 \times (11 \times 13) \times s_0 + 2 \times (7 \times 13) \times s_1 + 12 \times (7 \times 11) \times s_2 \end{aligned}$$

We use EEA to compute each s_i:

$$\gcd(11 \times 13, 7) = 1 \Rightarrow (-2)(11 \times 13) + (41)(7) = 1 \Rightarrow s_0 = -2$$
$$\gcd(7 \times 13, 11) = 1 \Rightarrow (4)(7 \times 13) + (-33)(11) = 1 \Rightarrow s_1 = 4$$
$$\gcd(7 \times 11, 13) = 1 \Rightarrow (-1)(7 \times 11) + (6)(13) = 1 \Rightarrow s_2 = -1$$

Note. Here, we perform the algorithm over the positive range, i.e. All our numbers are in the range $[0, m-1]$. We can perform the Chinese Remainder algorithm over the symmetric range as well. Then, our numbers must be in the range: $\left[-\left\lfloor \frac{m-1}{2} \right\rfloor, \left\lfloor \frac{m}{2} \right\rfloor\right]$

9.1 Variations to Chinese Remaindering

9.1.1 Incremental Chinese Remaindering

Suppose we are given some $a \in \mathbb{Z}$ and we start choosing small primes $m_0, m_1, m_2, \ldots$ to compute the sequence:

$$\text{rem}(a, m_0), \text{rem}(a, m_0 m_1), \text{rem}(a, m_0 m_1 m_2), \ldots$$

Eventually, $M = m_0 m_1 m_2 \ldots$ will become large enough such that $\text{rem}(a, M)$ will be the actual result. This occurs when the sequence fixed (There may be a chance that the sequence appears fixed but M isn't large enough yet, however, we can perform the analysis to determine that with low probability, we will get false positives).

9.1.2 Mixed Radix Representation

Let $0 \leq a < m_0 m_1 \ldots m_{r-1}$, each $m_i \in \mathbb{N}_{\geq 2}$ (not necessarily relatively prime)

Claim: We can write a uniquely as:

$$a = a_0 + a_1 m_0 + a_2 m_0 m_1 + \ldots + a_r m_0 m_1 \ldots m_{r-1}$$

with $0 \leq a_i < m_i$ for all i.

In fact, we can use this to help us compute the incremental chinese remaindering by letting $m_0 m_1 \ldots m_{r-2}$ be one radix and m_r be the second radix.

Appendix A: Multiplication Time

Definition A.1

A function $M : \mathbb{N}_{>0} \to \mathbb{R}_{>0}$ is a multiplication time for $R[x]$ (R ring) if polynomials in $R[x]$ of degree $< n$ can be multiplied using at most $M(n)$ ring operations in R.

We'll use M to add information to our cost estimates and improve our cost analysis for algorithms

Example A.1

We've seen a couple examples of multiplication time already:

- For **Naïve Multiplication** of polynomials, we know that $M(n) \in \mathcal{O}(n^2)$
- Using **Karatsuba's algorithm**, we can reduce that cost to $M(n) \in \mathcal{O}(n^{1.59})$
- Cantor & Kaltofen (Theorem 6.6) showed: $M(n) \in \mathcal{O}(n^2)$

And a couple interesting results for integers:

- Schöhage & Strassen (Theorem 6.5) showed that: $M(n) \in \mathcal{O}(n \log n \log \log n)$
- Fürer showed in 2007: $M(n) \in \mathcal{O}(n \log n K^{\log^* n})$, where K is some constant > 1 and $\log^*$ is the iterated logarithm. (Harvey and Van Der Hoeven showed that $K = 4$ in 2018)
- In March 2019, Harvey and Van Der Hoeven [1] showed that $M(n) \in \mathcal{O}(n \log n)$ (Note that the result has yet to be officially peer-reviewed as of the time these notes were taken)

Useful Assumptions about M:

1. Superlinearity:

   If $n \geq m$
   $$\frac{M(n)}{n} \geq \frac{M(m)}{m} \tag{A.1}$$

2. At most quadratic:
   $$M(mn) \leq m^2 M(n) \tag{A.2}$$

Proposition A.1

Equation (A.1) implies the following:

- $M(mn) \geq mM(n)$

- $M(n+m) \geq M(n) + M(m)$
- $M(n) \geq n$

Example A.2

Using the assumptions and Proposition A.1, we can say the following:

- $nM(n) + M(n^2) \leq 2M(n^2) \in \mathcal{O}(M(n^2))$
- $M(cn) \leq c^2 M(n) \in \mathcal{O}(M(n))$ for constant c
- $n^3 + nM(n) \geq M(n^3) + M(n^{\frac{3}{2}}) \in \Omega(M(n^3))$

References

[1] David Harvey and Joris Van Der Hoeven. Integer multiplication in time $\mathcal{O}(n \log n)$. 2019. hal-02070778.

[2] Joachim von zur Gathen and Gerhard Jürgen. *Modern Computer Algebra*. Cambridge University Press, 3 edition, 2013.