CS 225 – Data Structures

ZJUI – Spring 2022

Lecture 4: Precomputation and Domain Transformation

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

4 Precomputation and Domain Transformation

4.1 Precomputation

Precomputation in general refers to a technique to apply first a different algorithm on (parts of) the input and to use the result in (multiple instances) of the main algorithm in order to improvce efficiency

Returning to sequence structures let us consider the following **string searching problem**:

Let A be a finite alphabet, consider strings $S, P \in A^*$ of lengths $|S| = n$ and $|P| = m$, decide whether P is a substring of S (i.e. $S = S_1 + P + S_2$), and find the first position r, at which the substring P appears in S (i.e. $|S_1| = r - 1$)

Without loss of generality we can assume $n \geq m$

We usually call S a **target string** and P a **pattern**

The Knuth-Morris Pratt Algorithm

A naive approach would for every $0 \leq i \leq n - m$ compare $P(j)$ and $S(i + j)$ for all $1 \leq j \leq m$—if all these comparisons yield true, then $i + 1$ is a position in S, where a substring P starts

Then the total number of comparisons (and hence the time complexity) is in $\Omega(m \cdot (n - m))$, which is $\Omega(m \cdot n)$ for $n \gg m$

The **Knuth-Morris Pratt algorithm** (for short: **KMP algorithm**) works with time complexity in $O(n)$, thus significally improves the time complexity

The clue is the precomputation of a sequence $NEXT$ of length m, where in case a mismatch occurs with $P(j)$

- $j - NEXT(j)$ is the number of positions P needs to be shifted, if $NEXT(j) > 0$
- otherwise, the first element in P needs to be lined up with the next position of S that has not yet been examined

In either case either P or the position of S under consideration is shifted, which leads to the linear time complexity—$NEXT$ only depends on P, and can be computed in time in $O(m)$

Example / 1

Let P be the string abcabcacab with the following values for $NEXT$:

j	1	2	3	4	5	6	7	8	9	10
$P(j)$	a	b	c	a	b	c	a	c	a	b
$NEXT(j)$	0	1	1	0	1	1	0	5	0	1

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P	a	b	c	a	b	c	a	c	a	b																
	↑																									

The comparison at position 1 gives a mismatch, so we simply shift the cursor to position 2 and align the beginning of P with this position

Example / 2

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P		a	b	c	a	b	c	a	c	a	b															
		↑	↑	↑	↑																					

The next three comparisons match, but for $j = 4$ we obtain another mismatch

As $NEXT(4) = 0$, we proceed with the next position 6 of S and align the beginning of P with this position

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P						a	b	c	a	b	c	a	c	a	b											
						↑	↑	↑	↑	↑	↑	↑	↑													

Now the mismatch occurs with the eigh'th position of P

Example / 3

As $NEXT(8) = 5$, we need to shift P by 3 positions; there is no need to check the first four positions again

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P									a	b	c	a	b	c	a	c	a	b								
													↑													

We continue this way, finding the next mismatch at position 5, which requires a shift of P by 4 positions

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P													a	b	c	a	b	c	a	c	a	b				
													↑	↑	↑	↑	↑	↑	↑	↑						

Example / 4

We have again a mismatch in the eight'th position leading to a shift of P by three—the first four positions need not to be rechecked

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P																a	b	c	a	b	c	a	c	a	b	
																				↑	↑	↑	↑	↑	↑	

This time the comparisons reach the end of P, so we found a match at position 16 of the target string S

The Boyer-Moore Algorithm

The KMP algorithm has one disadvantage: if P is not a substring of S, then every element in S will be compared at least once, which gives complexity in $\Omega(n)$

This is still the case, if the first occurrence of P as a substring of S occurs in a position close to $n - m$

The **Boyer-Moore algorithm** (for short: **BM algorithm**) avoids this problem

The BM algorithm has also worst case complexity in $O(n)$, but tends to be sub-linear, in particular for increasing values of m

In the best case, the BM algorithms finds all occurrences of P as a substring of S in time in $O(m + n/m)$

Same as the KMP algorithm the BM algorithm slides P along S, but it compares characters of P from right to left

Boyer-Moore Shifting Rules

The BM algorithm applies two shifting rules that are applied in case of a mismatch:

- If the mismatch occurs with the last character in P and the corresponding character in S is $c \in A$, then shift P by $d_1(c)$ positions

  The function d_1 only depends on P and can be pre-computed as

$$d_1(c) \; = \; \begin{cases} m & \text{if } c \neq P(i) \text{ for all } 1 \leq i \leq m \\ m - \max\{i \mid P(i) = c\} & \text{else} \end{cases}$$

- If the mismatch occurs for $P(j)$ with $j < m$, then P is shifted by $d_2(j)$ positions, where $r = j - d_2(j) + 1$ is maximal such that $P(j+1) \dots P(m)$ occurs as substring of P at position r

  Also d_2 only depends on P and can be pre-computed—we omit the details

Example / 1

We use the same example for S and P as for the KMP algorithm

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P	a	b	c	a	b	c	a	c	a	b																
							↑	↑	↑	↑																

The first mismatch occurs with the fourth comparison from the right, i.e. $j = 7$, so we shift P by $d_2(7) = 5$ positions—note that cab is a substring of P at position $3 = 7 - 5 + 1$

S	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
P						a	b	c	a	b	c	a	c	a	b											
															↑											

The immediate mismatch requires a shift of P by $d_1(c) = 2$ positions—the largest i with $P(i) = c$ is $i = 8$

Example / 2

```
S   b a b c b a b c a b c a a b c a b c a b c a c a b c
P                 a b c a b c a c a b
                              ↑ ↑ ↑ ↑
```

After four comparisons we detect a mismatch at position $j = 7$, so we shift P again by $d_2(7) = 5$ positions

```
S   b a b c b a b c a b c a a b c a b c a b c a c a b c
P                           a b c a b c a c a b
                                              ↑
```

In three more steps we always have an immediate mismatch, so we shift by $d_1(a) = 1$ and $d_1(c) = 2$ positions

```
S   b a b c b a b c a b c a a b c a b c a b c a c a b c
P                                 a b c a b c a c a b
                                  ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
```

Finally, there is no more mismatch

4.2 Domain Transformation

Suppose we need to compute a function $f : D^n \to D$

An **algebraic domain transformation** comprises a bijective ***transformation function*** $\sigma : D \to D'$ and a ***transformed function*** $g : D'^n \to D'$ such that

$$f(x_1, \dots, x_n) \; = \; \sigma^{-1}(g(\sigma(x_1), \dots, \sigma(x_n)))$$

holds for all $x_1, \dots, x_n \in D$

A standard example is the symbolic multiplication of two polynomials in $\mathbb{Z}[x]$ of degree $\leq d$ with, say $P(x) = a_d x^d + \cdots + a_1 x + a_0$ and $Q(x) = b_d x^d + \cdots + b_1 x + b_0$

Clearly, we can represent polynomials by $(d+1)$-tuples in $\mathbb{Z}^{d+1}$

Then a naive algorithm on such tuples using multiplications and additions over $\mathbb{Z}$ can be easily defined with time complexity in $\Theta(d^2)$

Example / cont.

A different representation of the domain would be to use $2d+1$ values $P(x_0), \dots, P(x_d)$ ($d+1$ values uniquely determine the polynomial)

Then we can use efficient component-wise multiplication (and $2d+1$ points suffice to determine the product polynomial)

However, for this idea to make sense the following problems need to be solved:

- The evaluation of a polynomial $P(x)$ must be carried out efficiently (better than naive computation in $\Omega(d^2)$)—this concerns the efficiency of the transformation function σ

- The computation of the coefficients of the product polynomial must be carried out efficiently (better than naive interpolation with complexity in $\Omega(d^3)$)—this concerns the efficiency of the inverse transformation function σ^{-1}

Discrete Fourier Transformation

We will now show that if parameters are set appropriately, both problems—efficiency of the transformation function σ and efficiency of the inverse transformation function σ^{-1}—can be solved

In the following all our computations are done in $\mathbb{Z}_m$ or in $\mathbb{C}$, because we need to exploit **principal roots of unity**

Assumptions. Let $n > 1$ be a power of 2, and then choose a constant ω with $\omega^{n/2} = -1$

For instance, for $m = 257$ and $n = 8$ we could take $\omega = 4$ in $\mathbb{Z}_{257}$, as $\omega^4 \equiv -1 \mod 257$

For $n = 8$ we could take $\omega = e^{i\pi/4} = (1+i)/\sqrt{2} \in \mathbb{C}$, as $\omega^4 = e^{i\pi} = -1$

Fast Fourier Transformation / Preliminaries

Let $P_{\bar{a}}(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$ be a polynomial of degree $< n$ represented by $\bar{a} = (a_0, \dots, a_{n-1})$

The **discrete Fourier transform** of $\bar{a}$ is the n-tuple

$$F_\omega(\bar{a}) \;=\; (P_{\bar{a}}(\omega^0), \dots, P_{\bar{a}}(\omega^{n-1}))$$

For $t = n/2$ consider the t-tuples $\bar{b} = (a_0, a_2, \dots, a_{n-2})$ and $\bar{c} = (a_1, a_3, \dots, a_{n-1})$

Then we have $P_{\bar{a}}(x) = P_{\bar{b}}(x^2) + x \cdot P_{\bar{c}}(x^2)$

In particular, with $\alpha = \omega^2$ we obtain $P_{\bar{a}}(\omega^i) = P_{\bar{b}}(\alpha^i) + \omega^i \cdot P_{\bar{c}}(\alpha^i)$

Fast Fourier Transformation / cont.

Furthermore, we have

$$\alpha^{t/2} = (\omega^2)^{t/2} = \omega^t = \omega^{n/2} = -1$$

We can there replace n by t and ω by α, which allows us to use the discrete Forier transforms $F_\alpha(\bar{b})$ and $F_\alpha(\bar{c})$

As $\alpha^t = 1$ and $\omega^t = -1$ hold, we further get

$$\alpha^{t+i} = \alpha^i \qquad \text{and} \qquad \omega^{t+i} = -\omega^i \ ,$$

which implies

$$P_{\bar{a}}(\omega^{t+i}) \ = \ P_{\bar{b}}(\alpha^i) - \omega^i P_{\bar{c}}(\alpha^i)$$

This altogether suggest to exploit divide-and-conquer for the computation of the discrete Fourier transform $F_\omega(\bar{a})$

Recursive FFT Algorithm

With the convention that n is a power of 2, ω is a constant with $\omega^{n/2} = -1$, the inputs $\text{CoeffA}(i)$ are defined for $0 \leq i \leq n-1$, and the outputs $\text{FourA}(i)$ as well we define the following recursive rule:

```
$\text{FourA} \leftarrow \text{FFT}(\text{CoeffA}, \omega, n) =$
    **IF** $n = 1$ **THEN** $\text{FourA}(0) := \text{CoeffA}(0)$
    **IF** $n > 1$
    **THEN LET** $t = n/2$ **IN**
        **FORALL** $i$ **WITH** $0 \leq i \leq t-1$ **DO**
            $\text{CoeffB}(i) := \text{CoeffA}(2i)$
            $\text{CoeffC}(i) := \text{CoeffA}(2i+1)$
        $\text{FourB} \leftarrow \text{FFT}(\text{CoeffB}, \omega^2, t)$
        $\text{FourC} \leftarrow \text{FFT}(\text{CoeffC}, \omega^2, t)$
        **FORALL** $i$ **WITH** $0 \leq i \leq t-1$ **DO**
            **LET** $\alpha = \omega^i$ **IN**
                $\text{FourA}(i) := \text{FourB}(i) + \alpha \cdot \text{FourC}(i)$
                $\text{FourA}(i+t) := \text{FourB}(i) - \alpha \cdot \text{FourC}(i)$
```

FFT Algorithm / cont.

The **fast Fourier transform** algorithm is then invoked by
Fourier $\leftarrow$ **FFT**(Coefficients, ω, n), where

- Coefficients is a unary function symbol of arity 1, such that Coefficients(i) contain the input coefficients $\bar{a} = (a_0, \dots, a_{n-1})$
- Fourier is a unary function symbol of arity 1, such that Fourier(i) contain the input coefficients $F_\omega(\bar{a})$

Counting the number of steps of **FFT** for an input sequence of length $n = 2^k$ we obtain a linear recurrence equation $f_k = 2f_{k-1} + g(n)$ with $g(n) \in O(n)$

Solving this linear recurrence gives rise to time complexity in $\Theta(n \log n)$

While the iterative FFT algorithm is conceptually simple, an iterative implementation is preferred—we omit the details (see Exercise 9.2.2 in the Brassard/Bratley textbook, marked with **)

Example

Let $n = 8$, $\omega = 4$ and $\bar{a} = (255, 8, 0, 226, 37, 240, 3, 0)$—we calculate in $\mathbb{Z}_{257}$

Split $\bar{a}$ into $\bar{b} = (255, 0, 37, 3,)$ and $\bar{c} = (8, 226, 240, 0)$ and call **FFT** recursively with $\omega^2 = 16$ and $t = 4$

We obtain FourB $= F_{16}(\bar{b}) = (38, 170, 32, 9)$ and FourC $= F_{16}(\bar{c}) = (217, 43, 22, 7)$, which is combined into

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\text{FourA}(0) & = & 38 + 217 = 255 & \text{FourA}(4) & = & 38 - 217 = 78 \\
\text{FourA}(1) & = & 170 + 43\omega = 85 & \text{FourA}(5) & = & 170 - 43\omega = 255 \\
\text{FourA}(2) & = & 32 + 22\omega^2 = 127 & \text{FourA}(6) & = & 32 - 22\omega^2 = 194 \\
\text{FourA}(2) & = & 9 + 7\omega^3 = 200 & \text{FourA}(7) & = & 9 - 7\omega^3 = 75
\end{array}$$

So the result is $F_4(\bar{a}) = (255, 85, 127, 200, 78, 255, 194, 75)$

Principal Roots of Unity

Let us now look into the inverse transformation, i.e. given $d+1$ value points for a polynomial $P(x)$ of degree d, find the coefficients of $P(x)$

For the inverse Fourier transform we will exploit principal roots of unity

We say that ω is a **principal n'th root of unity** iff $\omega \neq 1$, $\omega^n = 1$ and

$$\sum_{j=0}^{n-1} \omega^{jp} = 0 \quad \text{holds for all } 1 \leq p < n$$

We will show that if n is a power of 2 and we have $\omega^{n/2} = -1$, then ω is a principal n'th root of unity

Foundations

Theorem. Consider any commutative ring R, in which $1 + 1 \neq 0$ holds. Let $n > 1$ be a power and let $\omega \in R$ with $\omega^{n/2} = -1$. Then the following hold:

(i) ω is a principal n'th root of unity

(ii) ω has a multiploicative inverse $\omega^{-1} = \omega^{n-1}$—then

(iii) ω^{-1} is a principal n'th root of unity

(iv) Let n denote $\overbrace{1 + \cdots + 1}^{n \text{ times}}$. If $n \neq 0$, then all ω^i with $0 \leq i \leq n-1$ are pairwise different

(v) If $n \in R$ has a multiplicative inverse n^{-1} then for any principal n'th root of unity ω we get $\omega^{n/2} = -1$

Let ω^{-i} denote $(\omega^{-1})^i$ for any $i \in \mathbb{Z}$

In (iv) the powers ω^i are called **n'th roots of unity**

Proof

Proof. (i) $\omega \neq 1$ and $\omega^n = 1$ are obvious, so we concentrate on the third condition in the definition of principal roots of unity

Let $n = 2^k$ and $p = 2^u \cdot v$ with an odd v and define $s = 2^{k-u-1}; \omega^{sp} = \omega^{v \cdot n/2} = (-1)^v = -1$

This implies $\omega^{(j+s)p} = \omega^{jp} \cdot \omega^{sp} = -\omega^{jp}$

With this we get

$$\sum_{j=0}^{n-1} \omega^{jp} = \sum_{r=0}^{s-1} \sum_{\substack{j=0 \\ j \equiv r(s)}}^{n-1} \omega^{jp} = \sum_{r=0}^{s-1} \sum_{\substack{j=0 \\ j \equiv r(s)}}^{n-1} \omega^{rp}(-1)^j = \sum_{r=0}^{s-1} \omega^{rp} \sum_{\substack{j=0 \\ j \equiv r(s)}}^{n-1} (-1)^j = 0$$

(ii) is obvious, as $\omega^{n-1} \cdot \omega = \omega^n = 1$

(iii) follows from (i), as $(\omega^{-1})^{n/2} = (-1)^{-1} = -1$ holds

Proof / cont.

(iv) Assume $\omega^i = \omega^j$; then $\omega^p = 1$ with $p = j - i > 0$ and consequently also $\omega^{jp} = 1$ for all $0 \le j \le n-1$

Due to the third property of principal roots of unity we have $0 = \sum_{j=0}^{n-1} \omega^{jp} = \sum_{j=0}^{n-1} 1 = n$ contradicting our assumption $n \neq 0$

(v) With $p = n/2$ we have $\sum_{j=0}^{n-1} \omega^{jp} = 0$

If j is even, then $\omega^{jp} = 1$, and if j is odd, then $\omega^{jp} = \omega^p$, hence

$$0 = 2 \cdot \sum_{j=0}^{n-1} \omega^{jp} = n + n \cdot \omega^p = n(1 + \omega^p)$$

As n^{-1} exists, we get $1 + \omega^p = 0$, i.e. $\omega^p = -1$

Examples

- First consider the field $\mathbb{C}$ of complex numbers

  For $\omega = e^{2\pi i/n} \in \mathbb{C}$ we have $\omega^{n/2} = e^{i\pi} = -1$, so part (i) of the Theorem above implies that ω is a principal n'th root of unity

- Consider the commutative ring $\mathbb{Z}_m$

  Let n and ω be positive powers of 2, and let $m = \omega^{n/2} + 1 > 2$

  Then clearly $1 + 1 \neq 0$ in $\mathbb{Z}_m$ and $\omega^{n/2} = -1$ holds in $\mathbb{Z}_m$, so by part (i) of the Theorem above ω is a principal n'th root of unity

  Furthermore, in $\mathbb{Z}_m$ we have $(m-(m-1)/n)\cdot n = m\cdot n-(m-1) = -\omega^{n/2} = 1$, which implies $n^{-1} = m - (m-1)/n$

Inversion Theorem

Now let R be a commutative ring, let $\omega \in R$ be a principlal n'th root of unity, and assume that the multiplicative inverse n^{-1} exists in R

Define two $n \times n$ matrices A and B with $A_{ij} = \omega^{ij}$, $B_{ij} = n^{-1}\omega^{-ij}$ for $0 \le i, j < n$

Inversion Theorem. *B is the inverse of A, i.e. $AB = I_n$.*

With the definition of A we get

$$\begin{aligned} \bar{a} \cdot A = (a_0, \dots, a_{n-1}) \cdot \left(\omega^{ij}\right) &= \left(\sum_{i=0}^{n-1} a_i, \sum_{i=0}^{n-1} a_i \omega^i, \dots, \sum_{i=0}^{n-1} a_i \omega^{i(n-1)} \right) \\ &= (P_{\bar{a}}(1), P_{\bar{a}}(\omega), \dots, P_{\bar{a}}(\omega^{n-1}) = F_\omega(\bar{a}) \end{aligned}$$

It therefore makes sense to define the **inverse Fourier transform** of $\bar{a}$ as

$$F_\omega^{-1}(\bar{a}) = \bar{a} \cdot B = (n^{-1}P_{\bar{a}}(1), n^{-1}P_{\bar{a}}(\omega^{-1}), \dots, n^{-1}P_{\bar{a}}(\omega^{-(n-1)})$$

Proofs

Corollary. We have $F_\omega^{-1}(F_\omega(\bar{a})) = F_\omega(F_\omega^{-1}(\bar{a})) = \bar{a}$ for any n-tuple $\bar{a} \in R^n$.

Proof. We have $F_\omega^{-1}(F_\omega(\bar{a})) = F_\omega(\bar{a}) \cdot B = \bar{a} \cdot A \cdot B = \bar{a}$ and

$$F_\omega(F_\omega^{-1}(\bar{a})) = F_\omega^{-1}(\bar{a}) \cdot A = \bar{a} \cdot B \cdot A = \bar{a}$$

Proof of the Theorem. Let $C = A \cdot B$, then

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj} = n^{-1} \sum_{k=0}^{n-1} \omega^{k(i-j)}$$

For $i = j$ we get $\omega^{k(i-j)} = 1$ and hence $C_{ii} = n^{-1} \cdot n = 1$

For $i > j$ let $p = i - j$, which gives $C_{ij} = n^{-1} \sum_{k=0}^{n-1} \omega^{kp} = 0$, as ω is a principal n'th root of unity

For $i < j$ let $p = j - i$, which gives $C_{ij} = n^{-1} \sum_{k=0}^{n-1} (\omega^{-1})^{kp} = 0$, as ω^{-1} is a principal n'th root of unity

Implementation Issues

To compute the inverse Fourier transform $F_\omega^{-1}(\bar{a})$ we only have to invoke **FFT** with input $\bar{a}$, $\omega^{-1} = \omega^{n-1}$ and n, then multiply all components with n^{-1}

For computations in $\mathbb{C}$ rounding errors may occur, so computations in $\mathbb{Z}_m$ are preferred

First, multiplications in $\mathbb{Z}_m$ with $m = 2^u + 1$ (as in our example of roots of unity) can be realised without using division

For this let a, b be integers with $0 \leq a, b < m$ and let $c = ab$ be their product

Decompose c into two blocks of u bits, i.e. write $c = 2^u j + i$ with $0 \leq i < 2^u$ and $0 \leq j \leq 2^u$, and use $d = \begin{cases} i - j & \text{for } i \geq j \\ i - j + m & \text{for} i < j \end{cases}$

We always have $0 \leq d < m$, so in $\mathbb{Z}_m$ we have $c = 2^u j + i = mj + (i - j) = d$

For computarions in $\mathbb{Z}_m$ it may be necessary to deal with large integers, so we have to drop the assumption that arithmetic operations can be performed at unit cost (see Brassard/Bratley, pp.283f.)

Example

Take again $n = 8$, $\omega = 4$ (a principal n'th root of unity in $\mathbb{Z}_{257}$) and let $\bar{a} = (255, 85, 127, 200, 78, 255, 194, 75)$

In order to compute $F_\omega^{-1}(\bar{a})$ in $\mathbb{Z}_{257}$ first compute $\mathbf{FFT}(\bar{a}, \omega^{-1}, n)$ using $\omega^{-1} = \omega^7 = 193$

Then $\bar{a}$ is decomposed into $\bar{b} = (255, 127, 78, 194)$ and $\bar{c} = (85, 200, 255, 75)$

The recursive calls $\mathbf{FFT}(\bar{b}, \omega^{-2}, 4)$ and $\mathbf{FFT}(\bar{c}, \omega^{-2}, 4)$ with $\omega^{-2} = 241$ yield the results $B = (140, 221, 12, 133)$ and $C = (101, 143, 65, 31)$

The combination of these results lead to $\mathbf{FFT}(\bar{a}, \omega^{-1}, n) = (241, 64, 0, 9, 39, 121, 24, 0)$

Finally, we multiply with $n^{-1} = m - (m-1)/n = 225$, which gives the final result $F_\omega^{-1}(\bar{a}) = (255, 8, 0, 226, 37, 240, 3, 0)$

Symbolic Operations on Polynomials

In principle, we have shown that the discrete Fourier transform and its inverse fulfill the requirements of an efficuent algebraic domain transformation

However, it is still necessary to determine appropriate values for n and ω (both should be powers of 2), such that computations can be executed in $\mathbb{Z}_m$ with $m = \omega^{n/2} + 1$

Let $P(x)$ and $Q(x)$ be polynomials in $\mathbb{Z}[x]$, let $u = \max\{degP, degQ\}$ and let a and b be the maxima of the absolute values of the coefficients in $P(x)$ and $Q(x)$, respectively.
Then the absolute value of the coefficients of $P(x)Q(x)$ is $\leq ab(u+1)$.

Using this result we can choose n as the smallest power of 2 that is larger than the degree of the product

If v is the smallest power of 2 that is $\geq 2ab(u+1)$, then we can take $m = v + 1$ and ω as a principal n'th root of unity in $\mathbb{Z}_m$

Example

Suppose we want to multiply $P(x) = 3x^3 - 5x^2 - x + 1$ and $Q(x) = x^3 - 4x^2 + 6x - 2$

The degree of the product will be 6, so we take $n = 8$

With $u = 3$, $a = 5$ and $b = 6$ we know that the coefficients of the product will be in the interval $[-120, 120]$

We therefore choose $m = 257$ and $\omega = 4$ with $\omega^{n/2} = -1$

ω is a principal 8'th root of unity in $\mathbb{Z}_m$, and we get $n^{-1} = 225$ in $\mathbb{Z}_m$

Applying the discrete Fourier transform to the coefficients of P and Q, then multiplying the resulting tuples component-wise, and applying the inverse Fourier transform yield the tuple $(255, 8, 0, 226, 37, 240, 3, 0)$, which represents the polynomial

$$R(x) = 3x^6 - 17x^5 + 37x^4 - 31x^3 + 8x - 2$$

with coefficients in $[-120, 120]$

Multiplication of Large Integers

Now consider the problem to multiply two arbitrary large integers $a, b \in \mathbb{Z}$ (without loss of generality consider only non-negative integers)

The key idea is to exploit a double domain transformation

First consider a p-adic representation $a = \sum_{i=0}^{q-1} a_i p^i$ and represent a by the polynomial $P_a(x) = \sum_{i=0}^{q-1} a_i x^i$

Then we can multiply $P_a(x)$ and $P_b(x)$ using the discrete Fourier transform to obtain the polynomial $P_{ab}(x)$, so finally we have $ab = P_{ab}(p)$

Example. Take the base $p = 10$, and let $a = 2,301$ and $b = 1,095$, which defines the polynomials $P_a(x) = 2x^3 + 3x^2 + 1$ and $P_b(x)x^3 + 9x + 5$

We compute the product

$$P_{ab}(x) = P_a(x)P_b(x) = 2x^6 + 3x^5 + 18x^4 + 38x^3 + 15x^2 + 9x + 5 \ ,$$

which gives the result $ab = P_{ab}(10) = 2,519,595$

The Degree Problem

In general, assume that we have to multiply two n-bit integers (assume that n is a power of 2)

This is reduced to the symbolic multiplication of two polynomials of degree $< n$

Then the central step in this symbolic multiplication comprises d multiplications of integers $< \omega^{d/2} + 1$ with $d = 2n$

Unfortunately, even for $\omega = 2$ these integers require $n + 1$ bits

That is, the problem of multiplying two n-bit integers has been "reduced" to $2n$ multiplications of slightly larger integers—this is no improvement at all!

The only way to correct this miserable approach is to reduce the degree of the polynomials representing the integers to be multiplied, but the discrete Fourier transform in order to be efficient requires sufficiently high degrees

Furthermore, the price for this is an increase of the size of the coefficients, so the degree reduction requires care

The Degree Problem / cont.

For instance, take the base 4 instead of 2—then coefficients are between 0 and 3 and the polynomials have degree $< n/2$

Thus, coefficients in the product are bounded by $9n/2$, and it sufficis to choose ω with $9n/2 \leq \omega^{n/2}$

For $n \geq 16$ we can simply choose $\omega = 2$, so the time complexity becomes $M(n) \in nM(n/2) + O(n^2 \log n)$

This defines a non-linear recurrence equation $M(n) = nM(n/2)$ for n being a power of 2 with initial condition $M(1) = 1$

Unfortunately, we can show that the solution of this recurrence leads to $M(n) = n^{(1+\log_2 n)/2} \notin O(n^k)$ for any k

Thus, the degree of the polynomial has to be further decreased

The Degree Problem / Solution

Define $\ell = 2^{\lceil \log_2 n/2 \rceil}$, i.e. $\ell = \sqrt{n}$ or $\ell = \sqrt{2n}$, and $k = n/\ell$

With these choices let $P_a(x)$ be a polynomial of degree $< k$ with coefficients corresponding to the k blocks of ℓ successive bits in the binary representation of a

We also have $P_a(2^\ell) = a$

Using this representation of integers by polynomials gives rise to efficient symbolic multiplication, so we can show $M(n) \in O(n(\log n)^{2.59})$ $(2.59 \approx \log_2 6)$

For further details we refer to Section 9.5 in the Brassard/Bratley textbook

The algorithm is not yet optimal, but further improvements are very complicated

The methods sketched above have been used in the computation of 134 million decimal positions of the number π

Example

Let $a = 9,885$ and $b = 21,260$, so $n = 16$, $\ell = 4$ and $k = 4$

Decomposing the binary representation of a into four blocks of four bits gives 0010 0110 1001 1101, and for b we get 0101 0011 0000 1100

We therefore build the polynomials $P_a(x) = 2x^3 + 6x^2 + 9x + 13$ and $P_b(x) = 5x^3 + 3x^2 + 12$

The symbolic product is

$$P_{ab}(x) = P_a(x) \cdot P_b(x) = 10x^6 + 36x^5 + 63x^4 + 116x^3 + 111x^2 + 108x + 156 \ ,$$

which finally yields

$$ab = P_{ab}(16) = 210,155,100$$