Lecture 12: Polynomials

Fast multiplication with the Fast Fourier Transform

January 2021

- Polynomials are a data structure used frequently in sciences and engineering.
- We look at efficient methods to add and multiply two univariate polynomials $A(x)$ and $B(x)$ of degree n:
    - Straightforward methods: $\Theta(n)$ for addition; $\Theta(n^2)$ for multiplication
    - Advanced method, based on **F**ast **F**ourier **T**ransform (FFT): reduces time complexity of polynomial multiplication to $\Theta(n \log n)$

In this lecture we explain how FFT works for polynomial multiplication.

Preliminary notions

Polynomial in a variable x over an algebraic field F = a formal sum

$$A(x) = \sum_{j=0}^{n-1} a_j \, x^j$$

where $a_0, a_1, \ldots, a_{j-1} \in F$ are the coefficients of $A(x)$

- Usually, the field F is $\mathbb{C}$, $\mathbb{R}$ or $\mathbb{Q}$.
- The degree of $A(x)$ is $\deg(a) = \max\{j \mid a_j \neq 0\}$.
  - When $a_j = 0$ for all $0 \leq j < n$, we assume $\deg(A) = 0$.
  - Note that $0 \leq \deg(A) < n$.
- A degree-bound of $A(x)$ is an integer $m > 0$ such that $\deg(A) < m$.

Polynomial operations

For $A(x) = \sum_{j=0}^{n-1} a_j\, x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j\, x^j$ we define

Addition: If $C(x) = A(x) + B(x)$ then $C(x) = \sum_{j=0}^{n-1} c_j\, x^j$ where $c_j = a_j + b_j$ for $0 \le j < n$.

Multiplication: If $C(x) = A(x)\, B(x)$ then $C(x) = \sum_{j=0}^{2\,n-2} c_j\, x^j$ where $c_j = \sum_{k=0}^{j} a_k\, b_{j-k}$ for $0 \le k < 2n - 1$.

Example

If $A(x) = 2\,x^2 - 3\,x + 3$ and $B(x) = x^2 - 7\,x + 9$ then

$$A(x) + B(x) = 3\,x^2 - 10\,x + 12$$
$$A(x)\, B(x) = 2\,x^4 - 17\,x^3 + 42\,x^2 - 48\,x + 27$$

Remarks:

1. $\deg(A + B) \le \max(\deg(A), \deg(B))$
2. $\deg(A\, B) = \deg(A) \cdot \deg(B)$

The coefficient representation of the polynomial $A(x) = \sum_{j=0}^{n} a_j\, x^j$ is the vector of n coefficients $(a_0, a_1, \ldots, a_{n-1})$.

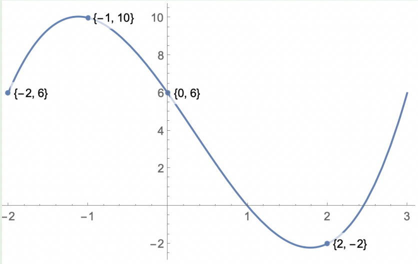
EXAMPLE: The coefficient representation of $x^3 - x + 7$ is the vector $(7, -1, 0, 1)$.

- The coefficient representation is convenient when we want to
  1. add two polynomials with degree bound n:
     $C(x) = A(x) + B(x)$ where $C(x) = \sum_{j=0}^{n-1} c_j\, x^j$ with
     $$c_j = a_j + b_j \quad \text{for } 0 \leq j < n$$
     Runtime complexity: $\Theta(n)$
  2. evaluate $A(x)$ at a given point x_0, with Horner's rule:
     $$A(x_0) = V_n = a_0 + x_0\,(a_1 + x_0\,(a_2 + \ldots + x_0(a_{n-1} + x_0\,(a_{n-1} + x_0 \cdot 0))\ldots))$$
     where $V_0 = 0$ and $V_j = a_{n-j} + x_0\, V_{j-1}$ for $1 \leq j \leq n$.
     Runtime complexity: $\Theta(n)$

A point-value representation of a polynomial $A(x)$ of degree-bound n is a set of n point-value pairs $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ such that

- $x_i \neq x_j$ whenever $0 \leq i < j < n$, and
- $y_j = A(x_j)$ for all $0 \leq j < n$.

Example (A point-value representation of $A(x) = x^3 - x^2 - 6x + 6$)



Properties of the point-value representation (1)

ASSUMPTION: $A(x)$ is a polynomial of degree-bound n

1. The point-value representation $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ of $A(x)$ is not unique: we can choose any n distinct points $x_0, x_1, \dots, x_{n-1}$
2. Computing $y_i = A(x_i)$ with Horner's rule takes $\Theta(n)$ time $\Rightarrow$ conversion from coefficient representation to point-value representation takes $\Theta(n^2)$ time.
3. For every point-value representation $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ there is a unique polynomial $A(x)$ of degree-bound n.
   - The coefficient representation of $A(x)$ can be computed by interpolation: we can use Lagrange's formula:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x - x_j)}{\prod_{j\neq k}(x_k - x_j)} = \sum_{k=0}^{n-1} a_j\, x^j \quad \text{where}$$

$$a_j = \text{(can be computed in time } \Theta(n^2)\text{; See [Cormen:2009])}$$

Properties of the point-value representation (2)

ASSUMPTIONS: $A(x)$ and $B(x)$ have degree-bound n, and

- an extended point-value representation of $A(x)$ is $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2\,n-1}, y_{2\,n-1})\}$
- an extended point-value representation of $B(x)$ is $\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2\,n-1}, y'_{2\,n-1})\}$

Then

1. A point-value representation of $A(x) + B(x)$ is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{2\,n-1} + y'_{2\,n-1})\}$$

2. A point-value representation of $A(x)\,B(x)$ is

$$\{(x_0, y_0 \cdot y'_0), (x_1, y_1 \cdot y'_1), \ldots, (x_{n-1}, y_{2\,n-1} \cdot y'_{2\,n-1})\}$$

The **runtime complexity** of polynomial addition and multiplication is $\Theta(n)$

Polynomial multiplication

What did we learn so far?

$$
\begin{array}{ccc}
\boxed{\begin{array}{l} a_0, a_1, \ldots, a_{n-1} \\ b_0, b_1, \ldots, b_{n-1} \end{array}} & \xrightarrow[\text{Time } \Theta(n^2)]{\text{Ordinary multiplication}} & \boxed{c_0, c_1, \ldots, c_{2\,n-1}} \\
\Big\downarrow {\scriptstyle \text{Evaluation, Time } \Theta(n^2)} & & \Big\uparrow {\scriptstyle \text{Interpolation, Time } \Theta(n^2)} \\
\boxed{\begin{array}{l} A(x_0), B(x_0), \\ A(x_1), B(x_1), \\ \vdots \\ A(x_{2\,n-1}), B(x_{2\,n-1}) \end{array}} & \xrightarrow[\text{Time } \Theta(n)]{\text{Pointwise multiplication}} & \boxed{\begin{array}{l} C(x_0), \\ C(x_1), \\ \vdots \\ C(x_{2\,n-1}) \end{array}}
\end{array}
$$

Good news

We can choose $x_0, x_1, \ldots, x_{2\,n-1}$ such that evaluation and interpolation can be performed in $\Theta(n \log n)$ time (see next slide)

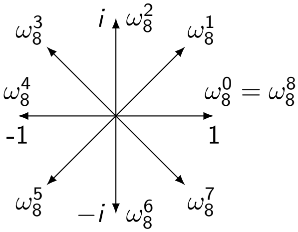
$\Rightarrow$ polynomial multiplication can be done in time $\Theta(n \log n)$

The complex roots of unity

Properties

The equation $z^m + 1 = 0$ as m distinct complex roots $\{\omega_m^k \mid 0 \leq k < m\}$ where $\omega_m = \cos \frac{2\pi}{m} + i \sin \frac{2\pi}{m} = e^{2\pi i/m}$.

- ω_m is called the principal m-th root of unity
- $\omega_m^k = e^{2\pi i k/m} = \cos \frac{2k\pi}{m} + i \sin \frac{2k\pi}{m}$
- These complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane. For example, when $m = 8$, the equation $z^m - 1 = z^8 - 1$ has 8 complex roots:



The discrete Fourier transform (DFT)

Given polynomial $A(x) = \sum_{j=1}^{n-1} a_j\, x^j$ (coefficient representation).

Compute $y_k = A(\omega_n^k) \in \mathbb{C}$ for $0 \leq k < n$

The discrete Fourier transform (DFT)

Given polynomial $A(x) = \sum_{j=1}^{n-1} a_j \, x^j$
(coefficient representation).

Compute $y_k = A(\omega_n^k) \in \mathbb{C}$ for $0 \leq k < n$

- This computation produces the point-value representation $\{(\omega_n^0, y_0), (\omega_n^1, y_1) \ldots, (\omega_n^{n-1}, y_{n-1})\}$ of $A(x)$.
- The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is called the discrete Fourier transform (DFT) of the vector $a = (a_0, a_1, \ldots, a_{n-1})$.
- We write $y = \mathrm{DFT}_n(a)$.

The fast Fourier transform (FFT)

FFT $=$ divide-and-conquer method to compute $\mathrm{DFT}_n(a)$ in time $\Theta(n \log n)$, as opposed to the $\Theta(n^2)$ time of the method of evaluation based on Horner's rule.

- Works well when n is a power of 2.

If $n = 2^N$ then $A(x) = A^{[0]}(x) + x\, A^{[1]}(x^2)$ where

$$A^{[0]}(x) = a_0 + a_2\, x + a_4\, x^2 + \ldots + a_{n-2} x^{n/2} - 1,$$
$$A^{[1]}(x) = a_1 + a_3\, x + a_5\, x^2 + \ldots + a_{n-1} x^{n/2} - 1.$$

- to evaluate $A(\omega_n^k)$, we must evaluate $A^{[0]}(\omega_n^{2k})$ and $A^{[1]}(\omega_n^{2\,k})$ for $0 \leq k < n$.
- n is even $\Rightarrow \{\omega_n^{2\,k} \mid 0 \leq k < n\} = \{\omega_{n/2}^k \mid 0 \leq k < n/2\}$

$\Rightarrow \mathrm{DFT}_n(\ldots)$ computation can be reduced recursively to two $\mathrm{DFT}_{n/2}(\ldots)$ computations.

The fast Fourier transform (FFT)

Pseudocode based on our previous remarks

RECURSIVE-FFT(a)

$n = a.length$ // n is a power of 2
if $n == 1$
    return a
$\omega_n = e^{2\pi i/n}$
$\omega = 1$
$a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$
$a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
$y^{[0]} =$ RECURSIVE-FFT($a^{[0]}$)
$y^{[1]} =$ RECURSIVE-FFT($a^{[1]}$)
for $k = 0$ **to** $n/2 - 1$
    $y_k = y_k^{[0]} + \omega\, y_k^{[1]}$
    $y_{k+(n/2)} = y_k^{[0]} - \omega\, y_k^{[1]}$
    $\omega = \omega\, \omega_n$
return y // y is assumed to be a column vector

Properties and applications

It can be proved with the Master Theorem that the runtime complexity of $\textsc{Recursive-FFT}(a)$ is $\Theta(n \log n)$ where $n = a.length$

Properties and applications

It can be proved with the Master Theorem that the runtime complexity of $\textsc{Recursive-FFT}(a)$ is $\Theta(n \log n)$ where $n = a.length$

$\Rightarrow$ $\mathrm{DFT}_n(a)$ can be computed in time $\Theta(n \log n)$

Properties and applications

It can be proved with the Master Theorem that the runtime complexity of $\textsc{Recursive-FFT}(a)$ is $\Theta(n \log n)$ where $n = a.length$

- $\Rightarrow$ $\text{DFT}_n(a)$ can be computed in time $\Theta(n \log n)$
- $\Rightarrow$ If we choose $x_0, x_1, \ldots, x_{n-1}$ to be the n complex roots of unity, we can compute the pointwise representation

$$\{(\omega_n^0, y_0), (\omega_n^1, y_1) \ldots, (\omega_n^{n-1}, y_{n-1})\}$$

of $A(x)$ in time $\Theta(n \log n)$

Properties and applications

It can be proved with the Master Theorem that the runtime complexity of $\textsc{Recursive-FFT}(a)$ is $\Theta(n \log n)$ where $n = a.length$

$\Rightarrow$ $\mathrm{DFT}_n(a)$ can be computed in time $\Theta(n \log n)$

$\Rightarrow$ If we choose $x_0, x_1, \ldots, x_{n-1}$ to be the n complex roots of unity, we can compute the pointwise representation

$$\{(\omega_n^0, y_0), (\omega_n^1, y_1) \ldots, (\omega_n^{n-1}, y_{n-1})\}$$

of $A(x)$ in time $\Theta(n \log n)$

- We remaining problem to solve is:
  How to perform interpolation at the complex roots of unity in time $\Theta(n \log n)$?

Interpolation at the complex roots of unity

Given $y_0, y_1, \ldots, y_{n-1} \in \mathbb{C}$

Find $a_0, a_1, \ldots, a_{n-1}$ such that

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix}}_{\text{Vandermonde matrix } V_n} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

$$\Rightarrow \; a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \, \omega_n^{-kj} \quad \text{for all } 0 \le j < n.$$

Interpolation at the complex roots of unity

Given $y_0, y_1, \ldots, y_{n-1} \in \mathbb{C}$

Find $a_0, a_1, \ldots, a_{n-1}$ such that

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2\,(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3\,(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2\,(n-1)} & \omega_n^{3\,(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix}}_{\text{Vandermonde matrix } V_n} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

$$\Rightarrow \; a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \, \omega_n^{-k\,j} \qquad \text{for all } 0 \leq j < n.$$

Question: How fast can be compute the vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$ with this formula?

Interpolation with the inverse Fourier transform

We can compute

$$y_j = A(\omega_n^j) = \sum_{k=0}^{n-1} a_k\, \omega_n^{k\,j}$$

for all $0 \leq j < n$ in time $\Theta(n \log n)$ with $\textsc{Recursive-FFT}(a)$.

$\Rightarrow$ we can compute

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k\, \omega_n^{-k\,j}$$

for all $0 \leq j < n$ in time $\Theta(n \log n)$ with $\textsc{Inverse-FFT}(y)$ obtained by changing $\textsc{Recursive-FFT}(a)$ as follows:

1. switch the roles of a and y
2. replace ω_n by ω_n^{-1}
3. divide each element by n

Interpolation with the inverse Fourier transform

We can compute

$$y_j = A(\omega_n^j) = \sum_{k=0}^{n-1} a_k\, \omega_n^{kj}$$

for all $0 \le j < n$ in time $\Theta(n \log n)$ with $\textsc{Recursive-FFT}(a)$.

$\Rightarrow$ we can compute

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k\, \omega_n^{-kj}$$

for all $0 \le j < n$ in time $\Theta(n \log n)$ with $\textsc{Inverse-FFT}(y)$ obtained by changing $\textsc{Recursive-FFT}(a)$ as follows:

1. switch the roles of a and y
2. replace ω_n by ω_n^{-1}
3. divide each element by n

$\Rightarrow$ **runtime complexity of** $\textsc{Inverse-FFT}(y)$**:** $O(n \log n)$

Exercises

1. Write down the pseudocode of INVERSE-FFT(y) by modifying the pseudocode of RECURSIVE-FFT(a) as suggested on the previous slide.
2. Suppose $a = (a_0, a_1, \ldots, a_{n-1})$ is the coefficient representation of a polynomial $A(x)$ with degree-bound n, and $B(x) = (x - b)\,A(x)$.
   (a) Write down the pseudocode of MULTIPLY1(a, b) which computes in time $\Theta(n)$ the coefficient representation $(b_0, b_1, \ldots, b_n)$ of polynomial $B(x)$.
   (b) Write down the pseudocode of QUOTIENT1(a, b) which computes in time $\Theta(n)$ the coefficient representation $(b_0, b_1, \ldots, b_{n-2})$ of the quotient of dividing $A(x)$ by $x - b$.
   (c) Write down the pseudocode of REMAINDER1(a, b) which computes in time $\Theta(n)$ the remainder of dividing $A(x)$ by $x - b$.
      - Remark: the remainder is the value of $A(b)$.

References

[Cormen:2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: INTRODUCTION TO ALGORITHMS. THIRD EDITION. The MIT Press. 2009.

Chapter 30: Polynomials and FFT.