CSE515 Advanced Algorithms
Lecture 26
Fast Fourier Transform II

Antoine Vigneron

Ulsan National Institute of Science and Technology

July 27, 2021

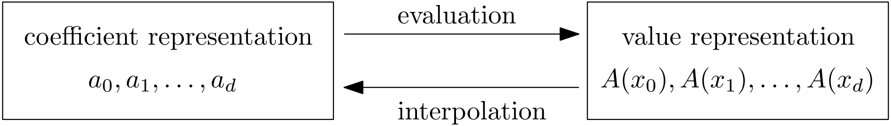
Introduction

- This is a second lecture on FFT. A few applications will be presented.

References:

- Section 2.6 of Algorithms by Dasgupta, Papadimitriou and Vazirani.
- Chapter 30 of Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein. (Available online from the UNIST library website.)

Interpolation



- In the previous lecture, we showed that when the points $(x_0, \ldots, x_n)$ are the nth roots of unity $(\omega^0, \omega^1, \ldots, \omega^{n-1})$ then *evaluation* can be done in $O(n \log n)$ time by FFT:

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega)$$

- We will now show that

$$\langle \text{coefficients} \rangle = \frac{1}{n} \text{FFT}(\langle \text{values} \rangle, \omega^{-1}).$$

- So *interpolation* can also be done in $O(n \log n)$ time.

Matrix Reformulation

- *Evaluation* can be reformulated as follows:

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

- Let M be the matrix above.
- M is a *Vandermonde* matrix, and it is known to be invertible whenever the x_is are distinct, which is the case here.

Matrix Reformulation

- So *interpolation* can be done as follows:

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = M^{-1} \cdot \begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix}$$

- Problem: we need to invert M.
- In general, a matrix can be inverted in $O(n^3)$ time.
- This matrix (a Vandermonde matrix) can be inverted in $O(n^2)$.
- This is still too slow for us, as we aim at $O(n \log n)$.
- We will be able to interpolate in $O(n \log n)$ time because the points x_i we use are nth roots of unity.

Matrix Reformulation

- For the FFT, we evaluate at $x_0 = \omega^0 = 1$, $x_1 = \omega^1$, $\dots x_{n-1} = \omega^{n-1}$, where $\omega = e^{2i\pi/n}$, so the matrix becomes

$$M_n(\omega) = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(n-1)j} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

- The coefficient at the jth row and kth column (starting the count at 0) is ω^{jk}.

Matrix Reformulation

- By definition, the FFT of the coefficients $(a_0, \dots, a_{n-1})$ of a polynomial $A(x) = a_0 + a_1 x + \cdots + a_n x$ is:

$$\begin{pmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{n-1}) \end{pmatrix} = M_n(\omega) \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Theorem (Inversion formula)

$$M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$$

- Proof done in class.

Matrix Reformulation

- It follows from previous slide that:

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} M_n(\omega^{-1}) \cdot \begin{pmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{n-1}) \end{pmatrix}$$

- In other words, we can perform interpolation by running our FFT algorithm on input vector $(A(1), A(\omega), \dots, A(\omega^n))$, and using ω^{-1} as the nth root of unity, and then dividing the result by n.

Matrix Reformulation

- In summary, FFT allows us to perform *evaluation* and *interpolation* at $x_0 = 1$, $x_1 = \omega$, ..., $x_{n-1} = \omega^{n-1}$.

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega) \qquad \text{evaluation}$$

$$\langle \text{coefficients} \rangle = \frac{1}{n}\text{FFT}(\langle \text{values} \rangle, \omega^{-1}) \qquad \text{interpolation}$$

- So both operations take time $O(n \log n)$.

Evaluation of a Polynomial at One point

- FFT allow us to evaluate a polynomial at n points $1, \omega, \dots \omega^{n-1}$ quickly. What if we only need evaluation at one point?
- So we just want to compute $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$.

Pseudocode

```
procedure Eval(A, x)
    result ← 0
    for i ← 0, n − 1 do
        result ← result + a_i · Pow(x, i)
    return result
procedure Pow(x, i)                         ▷ computing x^i
    pow ← 1
    for j ← 1, i do
        pow ← x · pow
    return pow
```

Evaluation of a Polynomial at One point

- Is this a good algorithm?
- No, because it runs in $O(n^2)$ time.
- We can easily make it $O(n)$ as follows:

Pseudocode

```
procedure FasterEval(A, x)
    result ← 0
    pow ← 1
    for i ← 0, n − 1 do
        result ← result + a_i · pow
        pow ← pow · x
    return result
```

- An even better approach is presented on next slide.

Horner's Method

Example

Let $A(x) = 2 + 3x - x^2 + 4x^3$. Then $A(x) = 2 + x(3 + x(-1 + 4x))$. So $A(3) = 2 + 3(3 + 3(-1 + 4 \times 3)) = 110$

In general:
$$\begin{aligned} & a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= a_0 + x\Big(a_1 + x\big(a_2 + \cdots + x(a_{n-2} + x a_{n-1}) \ldots \big)\Big). \end{aligned}$$

Pseudocode

procedure $\textsc{Horner}(A, x)$
    result $\leftarrow a_n$
    for $i \leftarrow n-1, 0$ **do**
        result $\leftarrow a_i + x \cdot$result
    return result

Horner's Method

- Horner's method runs in $O(n)$ time.
- FASTEREVAL also runs in $O(n)$ time.
- Why is Horner's method better?
- Horner's method makes n multiplications and n additions, while FASTEREVAL makes $2n$ multiplications and n additions.
- So we gain a *constant* factor.
- In addition, the code is shorter.

Polynomial Multiplication

- INPUT: two polynomials $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ and $B(x) = b_0 + b_1 x + \cdots + b_d x^d$ of degree at most $d \leqslant (n-1)/2$, given by their coefficients.
- OUTPUT: the coefficients of $C(x) = A(x) \cdot B(x)$.

Algorithm

- $\omega \leftarrow e^{2i\pi/n}$
- **Evaluation**: Compute $A(\omega^i)$ and $B(\omega^i)$ for $i = 0, \dots, n-1$ by FFT.
- Compute $C(\omega^i) = A(\omega^i) \cdot B(\omega^i)$ for $i = 0, \dots, n-1$.
- **Interpolation**: The coefficients of C are given by

$$\frac{1}{n}\mathrm{FFT}\left(\langle C(1), C(\omega), \dots, C(\omega^{n-1})\rangle, \omega^{-1}\right)$$

Polynomial Multiplication

- This algorithm was described in previous lecture.
- It runs in $O(n \log n)$ time, since FFT runs in $O(n \log n)$ time.
- Remark: The coefficients of C form a vector called the *convolution* $a \otimes b$ of the coefficient vectors of A and B.
- In other words, if $a = (a_0, \dots, a_{n-1})$ and $b = (b_0, \dots, b_{n-1})$, then their convolution is $a \otimes b = (c_0, \dots, c_{2n-2})$ where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

  for all k.
- It is the same as doing polynomial multiplication, so it can be computed in $O(n \log n)$ time.

Integer Multiplication

$$
\begin{array}{rrrr}
 & & 1 & 2 \\
\times & 1 & 3 & \\
\hline
 & & 3 & 6 \\
 & 1 & 2 & \\
\hline
 & 1 & 5 & 6
\end{array}
\qquad
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 0 \\
\times & & & & 1 & 1 & 0 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 0 \\
 & & & 0 & 0 & 0 & 0 & \\
 & & 1 & 1 & 0 & 0 & & \\
 & 1 & 1 & 0 & 0 & & & \\
\hline
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0
\end{array}
$$

- The *long multiplication* algorithm taught in primary school.
- Also works in binary.
- Running time: For two n-digits (or n-bits) numbers, takes $\Theta(n^2)$ time.

Integer Multiplication

- Suppose you want to multiply two integers $\alpha = (a_{n-1}a_{n-2}\dots a_1a_0)_{10}$ and $\beta = (b_{n-1}b_{n-2}\dots b_1b_0)_{10}$.
- For instance, if $\alpha = 2371$, then $n = 3$, $a_3 = 2$, $a_2 = 3$, $a_1 = 7$ and $a_0 = 1$.
- Let $A(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ and $B(x) = b_0 + b_1x + \dots + b_{n-1}x^{n-1}$.
- Then $\alpha = A(10)$ and $\beta = B(10)$.
- Let $C(x) = A(x) \cdot B(x)$.
- We compute the coefficients of $C(X)$ in $O(n \log n)$ time by FFT.
- Compute $C(10)$ in $O(n)$ time.
- The result is: $\alpha\beta = A(10) \cdot B(10) = C(10)$.
- Conclusion: We can multiply two n-digits integer in $O(n \log n)$ time.

Integer Multiplication

- Remark: The approach on previous slide is oversimplified.
- In reality, if we directly use FFT, it makes calculation using floating point numbers, and thus gives an inexact result.
- It can be remedied using modular arithmetic, and provide an exact result.
- It incurs a small increase in the running time.
- For instance, the Schönhage-Strassen algorithm multiplies n-digit integers in time $O(n \log n \log\log n)$; it performs FFT in modular arithmetic.
- In practice, it only improves on previous algorithms (such as Karatsuba's) for very large integers (more than 10,000 digits).

Pattern Matching

Problem

Let $p = p_0 p_1 \dots p_{m-1}$ and $t = t_0 t_1 \dots t_{n-1}$ be two strings over alphabet Σ, called pattern and text respectively. Find all occurrences of p as substrings of t.

Example

Suppose $\Sigma = \{A, T, C, G\}$, $p = GAT$ and $t = ATGACTGATCCGATTAC$.
Then there are two occurrences: $ATGACT\boldsymbol{GAT}CC\boldsymbol{GAT}TAC$.

- Variation: We may also allow "don't care" symbols $*$, so now the strings are in $\Sigma \cup \{*\}$.
- For instance if $p = C * T$ and $t = ATGACTTGATCGTGATTAC$. Then there are two matches $ATGA\boldsymbol{CTT}GAT\boldsymbol{CGT}GATTAC$.

Pattern Matching

Proposition

The pattern matching problem with "don't cares" can be solved in time $O(n \log(m) \log s)$, where $s = |\Sigma|$.

- See lecture notes.
- Remark: Brute force would be $O(nm)$.