Design and Analysis of Algorithms: Lecture 3

Ben Chaplin

Contents

1 Polynomials

Definition. A **polynomial of degree** n is a function of the form $A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$, with the **coefficient vector** $\langle a_0, a_1, \ldots, a_{n-1} \rangle$.

1.1 Operations

- **Evaluation:** $f : A(x), x_0 \mapsto A(x_0)$
  A naive algorithm would take $O(n^2)$ time, but if one saves the values of x_0^k (**Horner's rule**), evaluation takes $O(n)$.
- **Addition:** $f : A(x), B(x) \mapsto A(x) + B(x)$
  Takes $O(n)$ time.
- **Multiplication:** $f : A(x), B(x) \mapsto A(x)B(x)$
  If we use coefficient form, it seems the best that can be done is $O(n^2)$.

Our problem for today: can we do polynomial multiplication faster than $O(n^2)$.

1.2 Representations

There are more ways to represent a polynomial $A(x) = a_0 + a_1 x + \ldots a_{n-1}x^{n-1}$.

- **Coefficient vector:** $\langle a_0, a_1, ldots, a_{n-1} \rangle$ (what we're used to).
- **Roots:** $r_0, r_1, \ldots, r_{n-1}$ where $A(x) = c(x - r_0)(x - r_1) \cdots (x - r_{n-1})$ for some constant c (by the Fundamental Theorem of Algebra, n roots uniquely determine a polynomial).
- **Samples:** (x_k, y_k) for $k = 0, 1, \ldots, n-1$, where $A(x_k) = y_k$, and the x_k's are unique (n unique samples also uniquely determine a polynomial, by the FTA).

Differing the representation of polynomials results in different complexities of the operations discussed above:

	Coefficient vector	**Roots**	**Samples**
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

2 Fast Fourier Transform

So how can we do polynomial multiplication in $\leq O(n^2)$ time?

The idea is to convert polynomials from coefficient form to sample form. Then, multiplying them takes $O(n)$ time, as seen above. Then, we will convert the product back to coefficient form. Thus, what we need is:

- A way to convert polynomials: **coefficients** $\rightarrow$ **samples** in $\leq O(n \log n)$ time.
- A way to convert polynomials: **samples** $\rightarrow$ **coefficients** in $\leq O(n \log n)$ time.

Fast Fourier Transform is the name of the algorithm we will use to do these conversions.

2.1 Naive approach

Coefficients $\rightarrow$ **Samples**:

Let $\langle a_0, a_1, \dots, a_{n-1} \rangle$ be the coefficient vectors of a polynomial A. Select some $X = \langle x_0, x_1, \dots, x_{n-1} \rangle$ to define the sample. Then, we want to calculate $Y = \langle y_0, y_1, \dots, y_{n-1} \rangle$, where $A(x_k) = y_k$. This can be represented as evaluating the following equation:

$$V = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix}, A = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}, Y = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

$$V \cdot A = Y$$

However, this takes $O(n^2)$ time.

Samples $\rightarrow$ **Coefficients**:

Now suppose we are given X and Y, and asked to determine A:

$$V^{-1} \cdot Y = A$$

This also takes $O(n^2)$ time.

2.2 Collapsing sets

We will use divide & conquer to evaluate the polynomial. However, if we continue using any values for X, we will face the following problem – our recursion will look like:

$$T(n, |X|) = 2T(\frac{n}{2}, |X|) + O(n + |X|)$$

Note, while n shrinks by half as we recurse, $|X|$ stays the same. This needs to change if we are to beat $O(n^2)$.

Definition. A set X is **collapsing** if $|X^2| = \dfrac{|X|}{2}$ and X^2 is collapsing, or $|X| = 1$.

Example. Let us build some collapsing sets:

- Let $X = \{1\}$, then X collapses by the base case.
- Let $X = \{-1, 1\}$, then $X^2 = \{1\}$, which collapses.
- Let $X = \{-i, i, -1, 1\}$, then $X^2 = \{-1, 1\}$, which collapses.
- $\cdots$

We see by this example that the $(2^n)^{th}$ roots of unity collapse for any $n \in \mathbb{N}$.

Definition. The **Discrete Fourier Transform** is the matrix-vector product $V \cdot A$ for $x_k = e^{ik\tau/n}$ (where $\tau = 2\pi$).

So what we want to compute is the **Discrete Fourier Transform**, as $e^{ik\tau/n}$ are the n^{th} roots of unity. This will give us our sample, and we can comupte it in $O(n \log n)$ time.

2.3 Algorithm

Algorithm 1 Divide & conquer algorithm for FFT

Input

A polynomial represented as coefficient vectors

Output

(X, Y) a valid sample of A

Pad A so that $|A| = 2^m$ for the smallest $m \in \mathbb{N}$ such that $2^m \geq |A|$.
$n \leftarrow 2^m$
$X \leftarrow \{e^{ik\tau/n} \mid k = 0, 1, \ldots, n-1\}$
$Y \leftarrow \text{DC}(A, X)$.values
return (X, Y)
procedure $\text{DC}(A, X)$
    $A_e = \sum_{k=0}^{n/2} a_{2k} x^k$
    $A_o = \sum_{k=0}^{n/2} a_{2k+1} x^k$
    if $|A| = 2$ **then**
        for $x \in X$ **do** ▷ $X = \{-1, 1\}$ here
            $M[x] = \text{MERGE}(A_e, A_o, x, \text{null})$
        end for
        return M
    end if
    $M_e = \text{DC}(A_e, X^2)$
    $M_o = \text{DC}(A_o, X^2)$
    return $\text{MERGE}(A_e, M_e, A_o, M_o, x)$
end procedure
procedure $\text{MERGE}(A_e, M_e, A_o, M_o, x)$
    return $A_e(x^2) + x(A_o(x^2))$ ▷ using M_e and M_o to calculate
end procedure

2.4 Runtime

Algorithm 1 runs in $O(n \log n)$ time, as each recursive call takes $O(\log n)$ time, and we make $O(n)$ recursive calls.

2.5 Complete Algorithm

Note that **Algorithm 1** simply transforms a polynomial from coefficient form to sample form. The entire FFT algorithm for polynomial multiplication is as follows:

1. Run **Algorithm 1** on both polynomials – $O(n \log n)$
2. Multiply the polynomials in sample form – $O(n)$
3. Using the inverse DFT, run the same **Algorithm 1** to convert back to coefficient form – $O(n \log n)$

Therefore, the complete FFT algorithm also runs in $O(n \log n)$ time.