Algorithms (V)

Yu Yu

Shanghai Jiaotong University

Review of the Previous Lecture

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

The fast Fourier transform

Polynomial multiplication

If $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ and $B(x) = b_0 + b_1 x + \cdots + b_d x^d$, their product

$$C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$$

has coefficients

$$c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}$$

where for $i > d$, take a_i and b_i to be zero.

Computing c_k from this formula takes $O(k)$ steps, and finding all $2d + 1$ coefficients would therefore seem to require $\Theta(d^2)$ time.

An alternative representation of polynomials

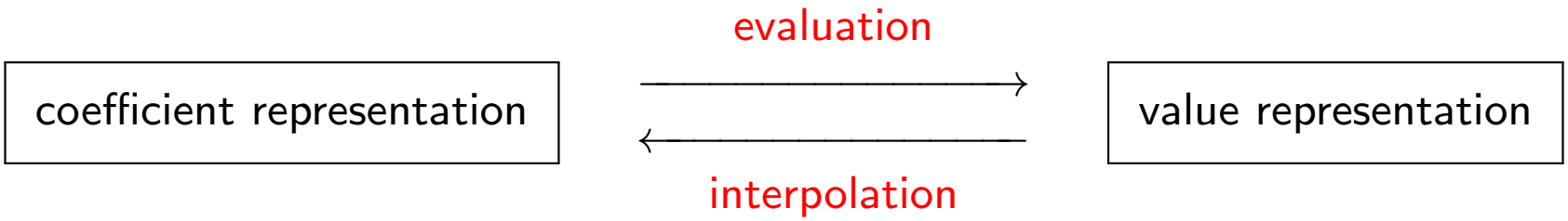
Fact

A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points. E.g., any two points determine a line.

We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ by either one of the following:

1. Its coefficients $a_0, a_1, \ldots, a_d$. (coefficient representation)
2. The values $A(x_0), A(x_1), \ldots, A(x_d)$. (value representation)

An alternative representation of polynomials (cont'd)



- The product $C(x)$ has degree $2d$, it is completely determined by its value at any $2d + 1$ points.
- The value of $C(z)$ at any given point z is just $A(z)$ times $B(z)$.

Thus polynomial multiplication takes linear time in the value representation.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection

Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$

Evaluation

Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$

Multiplication

Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$

Interpolation

Recover $C(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$

The **selection** step and the n multiplications are just linear time: In a *typical setting* for polynomial multiplication, the coefficients of the polynomials are real numbers and, moreover, are small enough that basic arithmetic operations (adding and multiplying) take unit time.

Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$, and so the baseline for n points is $\Theta(n^2)$.

The **fast Fourier transform (FFT)** does it in just $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$ in which the computations required by the individual points overlap with one another and can be shared.

Evaluation by divide-and-conquer

We pick the n points:

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because *the even powers of x_i coincide with those of $-x_i$*.

To investigate this, we need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4).$$

More generally

$$A(x) = A_e(x^2) + xA_o(x^2),$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that n is even).

Evaluation by divide-and-conquer (cont'd)

Given paired points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$$
$$A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2).$$

In other words, evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ (which each have half the degree of $A(x)$) at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

If we could recurse, we would get a divide-and-conquer procedure with running time

$$T(n) = 2T(n/2) + O(n) = O(n \log n).$$

How to choose n points?

We have a problem:

> *To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.*

How can a square be negative?

We use **complex numbers**.

At the very bottom of the recursion, we have a *single point*. This point might as well be 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.
The next level up then has $\pm\sqrt{+1} = \pm 1$ as well as the complex numbers $\pm\sqrt{-1} = \pm i$, where i is the imaginary unit.

By continuing in this manner, we eventually reach the initial set of n points: *the complex nth roots of unity*, that is, the n complex solutions to the equation

$$z^n = 1.$$

The complex roots of unity

The nth roots of unity: the complex numbers

$$1, \omega, \omega^2, \ldots, \omega^{n-1},$$

where

$$\omega = e^{2\pi i/n}.$$

If n is even, then:

1. The nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$.
2. Squaring them produces the $(n/2)$nd roots of unity.

Therefore, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $(n/2^k)$th roots of unity, for $k = 0, 1, 2, 3, \ldots$.
All these sets of numbers are plus-minus paired, and so our divide-and-conquer, works perfectly.

The resulting algorithm is the **fast Fourier transform**.

The fast Fourier transform (polynomial formulation)

$\mathrm{FFT}(A, \omega)$

Input: coefficient representation of a polynomial $A(x)$ of degree $\leq n-1$, where n is a power of 2; ω, an nth root of unity

Output: value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

1. **if** $\omega = 1$ **then** return $A(1)$
2. express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$
3. call $\mathrm{FFT}(A_e, \omega^2)$ to evaluate A_e at even powers of ω
4. call $\mathrm{FFT}(A_o, \omega^2)$ to evaluate A_o at even powers of ω
5. **for** $j = 0$ **to** $n-1$ **do**
6. compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$
7. return $A(\omega^0), \ldots, A(\omega^{n-1})$

Interpolation

We designed the FFT, a way to move from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex nth roots of unity

$$\left(1, \omega, \omega^2, \ldots, \omega^{n-1}\right).$$

That is

$$\langle \text{value} \rangle = \mathrm{FFT}\left(\langle \text{coefficients} \rangle, \omega\right).$$

We will see that the **interpolation** can be computed by

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

A matrix reformulation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

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

Let M be the matrix in the middle, which is a **Vandermonde** matrix:

If $x_0, \dots, x_{n-1}$ are distinct numbers, then M is invertible.

Hence

Evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.

Interpolation resolved

In linear algebra terms, the FFT multiplies an arbitrary n-dimensional vector – which we have been calling the coefficient representation – by the $n \times n$ matrix

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

Its (j, k)th entry (starting row- and column-count at zero) is ω^{jk}.

We will show:

Inversion formula

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

Interpolation resolved (cont'd)

Take ω to be $e^{2\pi i/n}$ and think of the columns of M as vectors in $\mathbb{C}^n$. Recall that the **angle** between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v = (v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their *inner product*

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \cdots + u_{n-1} v_{n-1}^*,$$

where z^* denotes the **complex conjugate** of z.

> Recall the complex conjugate of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$. The complex conjugate of a vector (or matrix) is obtained by taking the complex conjugates of all its entries.

The above quantity is maximized when the vectors lie in the *same direction* and is zero when the vectors are *orthogonal* to each other.

Interpolation resolved (cont'd)

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

Take the inner product of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}.$$

This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio ω^{j-k}. Therefore, if $j \neq k$, then it evaluates to

$$\frac{1 - \omega^{n(j-k)}}{1 - \omega^{(j-k)}} = 0$$

If $j = k$, then it evaluates to n □

Corollary

$MM^* = nI$, *i.e.*,

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

The fast Fourier transform (general formulation)

$\mathrm{FFT}(a, \omega)$

Input: An array $a = (a_0, a_1, \ldots, a_{n-1})$ for n a power of 2
A primitive nth root of unity, ω

Output: $M_n(\omega)$

1. **if** $\omega = 1$ **then** return a
2. $(s_0, s_1, \ldots, s_{n/2-1}) = \mathrm{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
3. $(s'_0, s'_1, \ldots, s'_{n-2}) = \mathrm{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
4. **for** $j = 0$ **to** $n/2 - 1$ **do**
5. $\quad r_j = s_j + \omega^j s'_j$
6. $\quad r_{j+n/2} = s_j - \omega^j s'_j$
7. return $(r_0, r_1, \ldots, r_{n-1})$

Chapter 3. Decompositions of graphs

Why graphs?

- A wide range of problems can be expressed with clarity and precision in the concise pictorial language of graphs.
  - Graph coloring.
  - Graph connectivity and reachability.
  - Flow.
- Formally, a graph is specified by a set of **vertices** (also called **nodes**) V and by **edges** E between select pairs of vertices.
  - **Undirected graphs**, i.e., E is a symmetric relation.
  - **Directed graphs**.

How is a graph represented?

We can represent a graph by an *adjacency matrix*: if there are $n = |V|$ vertices $v_1, \ldots, v_n$, this is an $n \times n$ array whose (i, j)th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

For undirected graph, the matrix is *symmetric* since an edge $\{u, v\}$ can be taken in either direction.

Pros: the presence of a particular edge can be checked in constant time, with just one memory access.

Cons: Takes up $O(n^2)$ space, which is wasteful if the graph does not have very many edges.

How is a graph represented? (cont'd)

We can represent a graph by an *adjacency list*: It consists of $|V|$ linked lists, one per vertex. The linked list for vertex u holds the names of vertices to which u has *an outgoing edge* – that is – vertices v for which $(u, v) \in E$.
Therefore, each edge appears in exactly one of the linked lists if the graph is directed or two of the lists if the graph is undirected.

Pros: The total size of the data structure is $O(|E|)$.

Cons: Checking for a particular edge (u, v) is no longer constant time.