

CMPSCI 611: Advanced Algorithms

Lecture 3: Fast Polynomial Multiplication, Greedy Algorithms and Matroids

Andrew McGregor

Last Compiled: September 17, 2009

Outline

Finishing off Fast Polynomial Multiplication

Greedy Algorithms and Matroids

Minimum Spanning Tree and Kruskal's Algorithm

Polynomial Multiplication

Problem: Suppose $A(x)$ and $B(x)$ are polynomials of degree $n - 1$:

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$$

Want to compute $C(x) = A(x)B(x)$ in $O(n \log n)$ time.

Polynomial Multiplication

Problem: Suppose $A(x)$ and $B(x)$ are polynomials of degree $n - 1$:

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$$

Want to compute $C(x) = A(x)B(x)$ in $O(n \log n)$ time.

Definition (CR)

The *coefficient representation*: $A(x) \equiv (a_0, \dots, a_{n-1})$.

Polynomial Multiplication

Problem: Suppose $A(x)$ and $B(x)$ are polynomials of degree $n - 1$:

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$$

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$$

Want to compute $C(x) = A(x)B(x)$ in $O(n \log n)$ time.

Definition (CR)

The *coefficient representation*: $A(x) \equiv (a_0, \dots, a_{n-1})$.

Definition (PVR)

The *point-value representation*: for n distinct points x_0, \dots, x_{n-1}

$$A \equiv \{(x_0, A(x_0)), (x_1, A(x_1)), \dots, (x_{n-1}, A(x_{n-1}))\}$$

Recall: Given PVR of A and B with respect to distinct x_0, \dots, x_{2n-2} , can get PVR of C with respect to x_0, \dots, x_{2n-2} in $O(n)$ time.

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

Naive implementation of step 1 takes. . .

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

Naive implementation of step 1 takes... $O(n^2)$ time.

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

Naive implementation of step 1 takes... $O(n^2)$ time. We'll do steps 1 and 3 in $O(n \log n)$ time.

Framework for Fast Polynomial Multiplication

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

Naive implementation of step 1 takes... $O(n^2)$ time. We'll do steps 1 and 3 in $O(n \log n)$ time.

Important: We can choose any distinct points for the PVR. Let's use the complex roots of unity...

Complex Roots of Unity

Definition

The n -th roots of unity are the n complex solutions to the equation $x^n = 1$, i.e.,

$$e^{2\pi ik/n} = \cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n} \quad k = 0, \dots, n-1.$$

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

Complex Roots of Unity

Definition

The n -th roots of unity are the n complex solutions to the equation $x^n = 1$, i.e.,

$$e^{2\pi ik/n} = \cos \frac{2\pi k}{n} + i \sin \frac{2\pi k}{n} \quad k = 0, \dots, n-1.$$

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

Lemma (Halving Lemma)

If n is even, then the squares of the n -th roots of unity are two copies of the $n/2$ -th roots of unity:

$$\{(\omega_n^0)^2, (\omega_n^1)^2, \dots, (\omega_n^{n-1})^2\}$$

equals two copies of $\{\omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n/2-1}\}$.

Divide and Conquer for Polynomial Evaluation

- Write degree $n - 1$ polynomial to be evaluated in terms of two degree $n/2 - 1$ polynomials:

$$\begin{aligned}A(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \\ &= (a_0 + a_2x^2 + \dots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + \dots + a_{n-1}x^{n-2}) \\ &= A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)\end{aligned}$$

Divide and Conquer for Polynomial Evaluation

- ▶ Write degree $n - 1$ polynomial to be evaluated in terms of two degree $n/2 - 1$ polynomials:

$$\begin{aligned}A(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \\ &= (a_0 + a_2x^2 + \dots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + \dots + a_{n-1}x^{n-2}) \\ &= A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)\end{aligned}$$

- ▶ To evaluate A at $x = \omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ it suffices to evaluate A_{even} and A_{odd} at $x = \omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n/2-1}$

Divide and Conquer for Polynomial Evaluation

- ▶ Write degree $n - 1$ polynomial to be evaluated in terms of two degree $n/2 - 1$ polynomials:

$$\begin{aligned}A(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \\ &= (a_0 + a_2x^2 + \dots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + \dots + a_{n-1}x^{n-2}) \\ &= A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)\end{aligned}$$

- ▶ To evaluate A at $x = \omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ it suffices to evaluate A_{even} and A_{odd} at $x = \omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n/2-1}$
- ▶ Let $T(n)$ be time to evaluate degree $n - 1$ polynomial at n -th roots of unity. Then $T(1) = \Theta(1)$ and

$$T(n) = 2T(n/2) + \Theta(n)$$

Divide and Conquer for Polynomial Evaluation

- ▶ Write degree $n - 1$ polynomial to be evaluated in terms of two degree $n/2 - 1$ polynomials:

$$\begin{aligned}A(x) &= a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} \\ &= (a_0 + a_2x^2 + \dots + a_{n-2}x^{n-2}) \\ &\quad + x(a_1 + a_3x^2 + \dots + a_{n-1}x^{n-2}) \\ &= A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2)\end{aligned}$$

- ▶ To evaluate A at $x = \omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ it suffices to evaluate A_{even} and A_{odd} at $x = \omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n/2-1}$
- ▶ Let $T(n)$ be time to evaluate degree $n - 1$ polynomial at n -th roots of unity. Then $T(1) = \Theta(1)$ and

$$T(n) = 2T(n/2) + \Theta(n)$$

- ▶ Use Master Theorem to conclude that $T(n) = \Theta(n \log n)$.

Back to Framework. . .

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$

Back to Framework. . .

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points

Back to Framework. . .

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR

Back to Framework. . .

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

Back to Framework. . .

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

We now know:

1. Step 1 can be done in $O(n \log n)$ time.
2. Step 2 can be done in $O(n)$ time.

Back to Framework. . .

- ▶ Input: Coefficient representation of $A(x)$ and $B(x)$
- ▶ Step 1: Transform into PVR by evaluating at $2n - 1$ points
- ▶ Step 2: Multiply polynomials to get $C(x)$ in PVR
- ▶ Step 3: Transform PVR of $C(x)$ back into CR.

We now know:

1. Step 1 can be done in $O(n \log n)$ time.
2. Step 2 can be done in $O(n)$ time.

It turns out that Step 3 is almost identical to Step 1!

Polynomial Evaluation and Interpolation

Step 1 Revisited: Transform $(a_0, a_1, \dots, a_{n-1})$ to

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

where $y_i = A(\omega_n^i)$.

Polynomial Evaluation and Interpolation

Step 1 Revisited: Transform $(a_0, a_1, \dots, a_{n-1})$ to

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

where $y_i = A(\omega_n^i)$. In other words, we need to evaluate:

$$V_n \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

where

$$V_n = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}$$

Polynomial Evaluation and Interpolation

Step 3 as inverse of Step 1: Need to transform $\{(\omega_n^0, y_0), (\omega_n^1, y_1), \dots, (\omega_n^{n-1}, y_{n-1})\}$ into $(a_0, a_1, \dots, a_{n-1})$ where $y_i = A(\omega_n^i)$.

Polynomial Evaluation and Interpolation

Step 3 as inverse of Step 1: Need to transform $\{(\omega_n^0, y_0), (\omega_n^1, y_1), \dots, (\omega_n^{n-1}, y_{n-1})\}$ into $(a_0, a_1, \dots, a_{n-1})$ where $y_i = A(\omega_n^i)$. In other words, we need to evaluate:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = V_n^{-1} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Polynomial Evaluation and Interpolation

Step 3 as inverse of Step 1: Need to transform $\{(\omega_n^0, y_0), (\omega_n^1, y_1), \dots, (\omega_n^{n-1}, y_{n-1})\}$ into $(a_0, a_1, \dots, a_{n-1})$ where $y_i = A(\omega_n^i)$. In other words, we need to evaluate:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = V_n^{-1} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

The inverse of V_n is just V_n with ω_n replaced by ω_n^{-1} :

$$V_n^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \omega_n^{-3} & \dots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \omega_n^{-6} & \dots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \omega_n^{-3(n-1)} & \dots & \omega_n^{-(n-1)(n-1)} \end{pmatrix}$$

Solving Step 3 Outline

- ▶ Let $\hat{A}(x) = y_0 + y_1x + \dots + y_{n-1}x^{n-1}$

Solving Step 3 Outline

- ▶ Let $\hat{A}(x) = y_0 + y_1x + \dots + y_{n-1}x^{n-1}$
- ▶ Need to compute:

$$a_i = \frac{\hat{A}(\omega_n^{-k})}{n} \quad \text{for } k = 0, \dots, n-1$$

Solving Step 3 Outline

- ▶ Let $\hat{A}(x) = y_0 + y_1x + \dots + y_{n-1}x^{n-1}$
- ▶ Need to compute:

$$a_i = \frac{\hat{A}(\omega_n^{-k})}{n} \quad \text{for } k = 0, \dots, n-1$$

- ▶ Rewrite $\hat{A}(x) = \hat{A}_{\text{even}}(x^2) + x\hat{A}_{\text{odd}}(x^2)$

Solving Step 3 Outline

- ▶ Let $\hat{A}(x) = y_0 + y_1x + \dots + y_{n-1}x^{n-1}$
- ▶ Need to compute:

$$a_i = \frac{\hat{A}(\omega_n^{-k})}{n} \quad \text{for } k = 0, \dots, n-1$$

- ▶ Rewrite $\hat{A}(x) = \hat{A}_{\text{even}}(x^2) + x\hat{A}_{\text{odd}}(x^2)$
- ▶ To evaluate \hat{A} on

$$\{\omega_n^0, \omega_n^{-1}, \dots, \omega_n^{-(n-1)}\}$$

it suffices to evaluate \hat{A}_{even} and \hat{A}_{odd} on

$$\{\omega_{n/2}^0, \omega_{n/2}^{-1}, \dots, \omega_{n/2}^{-(n/2-1)}\}$$

because Halving Lemma also applies to ω_n^{-1} .

- ▶ Step 3 can also be done in $O(n \log n)$ steps.

Outline

Finishing off Fast Polynomial Multiplication

Greedy Algorithms and Matroids

Minimum Spanning Tree and Kruskal's Algorithm

Greedy Algorithms Overview

“An algorithm that finds a solution by adding elements one by one, where each element that is added is the best current choice without regard to the future consequences of this choice.”

Greedy Algorithms Overview

“An algorithm that finds a solution by adding elements one by one, where each element that is added is the best current choice without regard to the future consequences of this choice.”

- ▶ Minimum Spanning Tree and Kruskal's algorithm
- ▶ Matroids and Subset Systems
- ▶ Bipartite Matching and Intersections of Matroids
- ▶ Union-Find Data Structure

Outline

Finishing off Fast Polynomial Multiplication

Greedy Algorithms and Matroids

Minimum Spanning Tree and Kruskal's Algorithm

Minimum Spanning Tree and Kruskal's Algorithm

Problem: Given an undirected, connected graph $G = (V, E)$ with edge weights find the minimum-weight subset $E' \subset E$ such that the graph $G = (V, E')$ is acyclic and connected, i.e., a minimum spanning tree (MST).

Minimum Spanning Tree and Kruskal's Algorithm

Problem: Given an undirected, connected graph $G = (V, E)$ with edge weights find the minimum-weight subset $E' \subset E$ such that the graph $G = (V, E')$ is acyclic and connected, i.e., a minimum spanning tree (MST).

Algorithm (Kruskal)

1. *Sort edges by non-decreasing weight*
2. $F = \emptyset$
3. *Until F is a spanning tree of G*
 - 3.1 *Get the next edge e*
 - 3.2 *If $F + e$ is acyclic then $F = F + e$*

Running Time of Kruskal's Algorithm

Implementation: Maintain an array A with an entry for each $v \in V$ that indicates which connected component it belongs to.

- ▶ Sorting: $O(|E| \log |E|)$

Running Time of Kruskal's Algorithm

Implementation: Maintain an array A with an entry for each $v \in V$ that indicates which connected component it belongs to.

- ▶ Sorting: $O(|E| \log |E|)$
- ▶ Checking if acyclic: $|E|$ checks and each is $O(1)$ time.

Running Time of Kruskal's Algorithm

Implementation: Maintain an array A with an entry for each $v \in V$ that indicates which connected component it belongs to.

- ▶ Sorting: $O(|E| \log |E|)$
- ▶ Checking if acyclic: $|E|$ checks and each is $O(1)$ time.
- ▶ Adding e to F : Updating array takes $O(|V|)$ time and array is updated exactly $|V| - 1$ times.

Total Running Time: $O(|E| \log |E| + |V|^2)$

Running Time of Kruskal's Algorithm

Implementation: Maintain an array A with an entry for each $v \in V$ that indicates which connected component it belongs to.

- ▶ Sorting: $O(|E| \log |E|)$
- ▶ Checking if acyclic: $|E|$ checks and each is $O(1)$ time.
- ▶ Adding e to F : Updating array takes $O(|V|)$ time and array is updated exactly $|V| - 1$ times.

Total Running Time: $O(|E| \log |E| + |V|^2)$

Will make this $O(|E| \log |E|)$ later via the union-find data structure

For Next Time...

- ▶ Read sections 3.1 and 3.2.