CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2010

Algebraic and Numeric Algorithms

- Measuring cost of arithmetic and numerical operations:
  - Measure size of input in terms of **bits**.
- Algebraic operations:
  - Measure size of input in terms of **numbers**.
- In both cases, measure complexity in terms of basic arithmetic operations: $+, -, *, /$.
  - Sometimes, measure complexity in terms of bit operations to account for large numbers.
- Size of numbers may be related to problem size:
  - Pointers, counters to objects.
  - Resolution in geometry/graphics (to distinguish between object positions).

Exponentiation

Given positive integers n and k, compute n^k.

Algorithm:

```
p = 1;
for (i=1 to k)
  p = p * n;
```

Analysis:

- Input size: $\Theta(\log n + \log k)$.
- Time complexity: $\Theta(k)$ multiplications.
- This is **exponential** in input size.

Faster Exponentiation

Write k as:

$$k = b_t 2^t + b_{t-1} 2^{t-1} + \cdots + b_1 2 + b_0, b \in \{0, 1\}.$$

Rewrite as:

$$k = ((\cdots (b_t 2 + b_{t-1})2 + \cdots + b_2)2 + b_1)2 + b_0.$$

New algorithm:

```
p = n;
for (i = t-1 downto 0)
  p = p * p * exp(n, b[i])
```

Analysis:

- Time complexity: $\Theta(t) = \Theta(\log k)$ multiplications.
- This is **exponentially** better than before.

Greatest Common Divisor

- The Greatest Common Divisor (GCD) of two integers is the greatest integer that divides both evenly.
- Observation: If k divides n and m, then k divides $n - m$.
- So,

$$f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n).$$

- Observation: There exists k and l such that

$$\begin{aligned} n &= km + l \text{ where } m > l \geq 0. \\ n &= \lfloor n/m \rfloor m + n \mod m. \end{aligned}$$

- So,

$$f(n, m) = f(m, l) = f(m, n \mod m).$$

GCD Algorithm

$$f(n,m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```

Analysis of GCD

- How big is $n \bmod m$ relative to n?

$$\begin{aligned}
n \geq m &\Rightarrow n/m \geq 1 \\
&\Rightarrow 2\lfloor n/m \rfloor > n/m \\
&\Rightarrow m\lfloor n/m \rfloor > n/2 \\
&\Rightarrow n - n/2 > n - m\lfloor n/m \rfloor = n \bmod m \\
&\Rightarrow n/2 > n \bmod m
\end{aligned}$$

- The first argument must be halved in no more than 2 iterations.
- Total cost:

Multiplying Polynomials (1)

$$P = \sum_{i=0}^{n-1} p_i x^i \qquad Q = \sum_{i=0}^{n-1} q_i x^i.$$

- Our normal algorithm for computing PQ requires $\Theta(n^2)$ multiplications and additions.

Multiplying Polynomials (2)

- Divide and Conquer:

$$P_1 = \sum_{i=0}^{n/2-1} p_i x^i \qquad P_2 = \sum_{i=n/2}^{n-1} p_i x^{i-n/2}$$

$$Q_1 = \sum_{i=0}^{n/2-1} q_i x^i \qquad Q_2 = \sum_{i=n/2}^{n-1} q_i x^{i-n/2}$$

$$\begin{aligned} PQ &= (P_1 + x^{n/2}P_2)(Q_1 + x^{n/2}Q_2) \\ &= P_1Q_1 + x^{n/2}(Q_1P_2 + P_1Q_2) + x^n P_2Q_2. \end{aligned}$$

- Recurrence:

$$\begin{aligned} T(n) &= 4T(n/2) + O(n). \\ T(n) &= \Theta(n^2). \end{aligned}$$

Multiplying Polynomials (3)

Observation:

$$(P_1 + P_2)(Q_1 + Q_2) = P_1Q_1 + (Q_1P_2 + P_1Q_2) + P_2Q_2$$

$$(Q_1P_2 + P_1Q_2) = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2$$

Therefore, PQ can be calculated with only 3 recursive calls to a polynomial multiplication procedure.

Recurrence:

$$\begin{aligned} T(n) &= 3T(n/2) + O(n) \\ &= aT(n/b) + cn^1. \end{aligned}$$

$\log_b a = log_2 3 \approx 1.59.$
$T(n) = \Theta(n^{1.59}).$

Matrix Multiplication

Given: $n \times n$ matrices A and B.

Compute: $C = A \times B$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Straightforward algorithm:

- $\Theta(n^3)$ multiplications and additions.

Lower bound for any matrix multiplication algorithm: $\Omega(n^2)$.

Strassen's Algorithm

(1) Trade more additions/subtractions for fewer multiplications in 2×2 case.

(2) Divide and conquer.

In the straightforward implementation, 2×2 case is:

$$
\begin{aligned}
c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\
c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\
c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\
c_{22} &= a_{21}b_{12} + a_{22}b_{22}
\end{aligned}
$$

Requires 8 multiplications and 4 additions.

Another Approach (1)

Compute:

$$
\begin{aligned}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &= (a_{11} + a_{12})b_{22} \\
m_5 &= a_{11}(b_{12} - b_{22}) \\
m_6 &= a_{22}(b_{21} - b_{11}) \\
m_7 &= (a_{21} + a_{22})b_{11}
\end{aligned}
$$

Another Approach (2)

Then:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

7 multiplications and 18 additions/subtractions.

Strassen's Algorithm (cont)

Divide and conquer step:

Assume n is a power of 2.

Express $C = A \times B$ in terms of $\frac{n}{2} \times \frac{n}{2}$ matrices.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Strassen's Algorithm (cont)

By Strassen's algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$$\begin{aligned} T(n) &= 7T(n/2) + 18(n/2)^2 \\ T(n) &= \Theta(n^{\log_2 7}) = \Theta(n^{2.81}). \end{aligned}$$

Current "fastest" algorithm is $\Theta(n^{2.376})$
Open question: Can matrix multiplication be done in $O(n^2)$ time?

Introduction to the Sliderule

Compared to addition, multiplication is hard.

In the physical world, addition is merely concatenating two lengths.

Observation:

$$\log nm = \log n + \log m.$$

Therefore,

$$nm = \text{ antilog}(\log n + \log m).$$

What if taking logs and antilogs were easy?

Introduction to the Sliderule (2)

The sliderule does exactly this!

- It is essentially two rulers in log scale.
- Slide the scales to add the lengths of the two numbers (in log form).
- The third scale shows the value for the total length.

Representing Polynomials

A vector **a** of n values can uniquely represent a polynomial of degree $n-1$

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a degree $n-1$ polynomial can be uniquely represented by a list of its values at n distinct points.

- Finding the value for a polynomial at a given point is called **evaluation**.
- Finding the coefficients for the polynomial given the values at n points is called **interpolation**.

Multiplication of Polynomials

To multiply two $n-1$-degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications.

However, if we evaluate both polynomials, we can simply multiply the corresponding pairs of values to get the values of polynomial AB.

Process:

- Evaluate polynomials A and B at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

Multiplication of Polynomials (2)

This can be faster than $\Theta(n^2)$ IF a fast way can be found to do evaluation/interpolation of $2n - 1$ points (normally this takes $\Theta(n^2)$ time).

Note that evaluating a polynomial at 0 is easy, and that if we evaluate at 1 and -1, we can share a lot of the work between the two evaluations.

Can we find enough such points to make the process cheap?

An Example

Polynomial A: $x^2 + 1$.
Polynomial B: $2x^2 - x + 1$.
Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

Notice:

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

Nth Root of Unity

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient.

Complex number ω is a **primitive nth root of unity** if

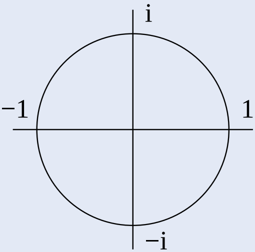
1. $\omega^n = 1$ and
2. $\omega^k \neq 1$ for $0 < k < n$.

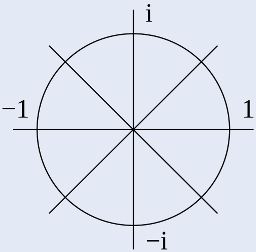
$\omega^0, \omega^1, ..., \omega^{n-1}$ are the **nth roots of unity**.

Example:

- For $n = 4$, $\omega = i$ or $\omega = -i$.

Nth Root of Unity (cont)





$n = 4$, $\omega = i$.
$n = 8$, $\omega = \sqrt{i}$.

Discrete Fourier Transform

Define an $n \times n$ matrix $V(\omega)$ with row i and column j as

$$V(\omega) = (\omega^{ij}).$$

Example: $n = 4$, $\omega = i$:

$$V(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let $\overline{a} = [a_0, a_1, ..., a_{n-1}]^T$ be a vector.
The **Discrete Fourier Transform** (DFT) of $\overline{a}$ is:

$$F_\omega = V(\omega)\overline{a} = \overline{v}.$$

This is equivalent to evaluating the polynomial at the nth roots of unity.

Array example

For $n = 8$, $\omega = \sqrt{i}$, $V(\omega) =$

$$\begin{array}{rrrrrrrr}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\
1 & i & -1 & -i & 1 & i & -1 & -i \\
1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\
1 & -i & -1 & i & 1 & -i & -1 & i \\
1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i}
\end{array}$$

Inverse Fourier Transform

The inverse Fourier Transform to recover $\overline{a}$ from $\overline{v}$ is:

$$F_{\omega}^{-1} = \overline{a} = [V(\omega)]^{-1} \cdot \overline{v}.$$

$$[V(\omega)]^{-1} = \frac{1}{n} V(\frac{1}{\omega}).$$

This is equivalent to interpolating the polynomial at the nth roots of unity.

An efficient divide and conquer algorithm can perform both the DFT and its inverse in $\Theta(n \lg n)$ time.

Fast Polynomial Multiplication

Polynomial multiplication of A and B:

- Represent an $n-1$-degree polynomial as $2n-1$ coefficients:
$$[a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0]$$
- Perform DFT on representations for A and B.
- Pairwise multiply results to get $2n-1$ values.
- Perform inverse DFT on result to get $2n-1$ degree polynomial AB.

FFT Algorithm

```
FFT(n, a0, a1, ..., an-1, omega, var V);
Output: V[0..n-1] of output elements.
begin
  if n=1 then V[0] = a0;
  else
    FFT(n/2, a0, a2, ... an-2, omega^2, U);
    FFT(n/2, a1, a3, ... an-1, omega^2, W);
    for j=0 to n/2-1 do
      V[j] = U[j] + omega^j W[j];
      V[j+n/2] = U[j] - omega^j W[j];
end
```