Algebraic and Numeric Algorithms

- Measuring cost of arithmetic and numerical operations:
  - Measure size of input in terms of \( \text{bits} \).
- Algebraic operations:
  - Measure size of input in terms of \( \text{numbers} \).
- In both cases, measure complexity in terms of basic arithmetic operations: \(+, -, \times, /\).
- 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:

\[
p = 1; \\
\text{for} \ (i=1 \text{ to } k) \ \\
p = p \times n;
\]

Analysis:

- Input size: \( \Theta(\log n + \log k) \).
- Time complexity: \( \Theta(k) \) multiplications.
- This is \textbf{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:

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

New algorithm:

\[
p = n; \\
\text{for} \ (i = t-1 \text{ downto } 0) \ \\
p = p \times p \times \exp(n, b[i])
\]

Analysis:

- Time complexity: \( \Theta(t) = \Theta(\log k) \) multiplications.
- This is \textbf{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
  \[
  n = km + l \text{ where } m > l \geq 0.
  \]
- So,
  \[
  f(n, m) = f(m, l) = f(m, n \mod m).
  \]

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

GCD Algorithm

\[
 f(n, m) = \begin{cases} 
  n & m = 0 \\
  f(m, n \mod m) & m > 0 
\end{cases}
\]

Mulitplying Polynomials (1)

\[
 P = \sum_{i=0}^{n-1} p_i x^i \quad Q = \sum_{i=0}^{n-1} q_i x^i.
\]

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

- Divide and Conquer:
  \[ P_1 = \sum_{i=0}^{n/2-1} p_i x^{i} \quad P_2 = \sum_{i=n/2}^{n-1} p_i x^{i-n/2} \]
  \[ Q_1 = \sum_{i=0}^{n/2-1} q_i x^{i} \quad Q_2 = \sum_{i=n/2}^{n-1} q_i x^{i-n/2} \]

  \[ PQ = \left( P_1 + x^{n/2} P_2 \right) \left( Q_1 + x^{n/2} Q_2 \right) = P_1 Q_1 + x^{n/2} (P_1 Q_2 + P_2 Q_1) + x^n P_2 Q_2. \]

- Recurrence:
  \[ T(n) = 4T(n/2) + O(n). \]
  \[ T(n) = \Theta(n^2). \]

**Observation:**

- Divide and Conquer.
- Trade more additions/subtractions for fewer multiplications in the \( 2 \times 2 \) case.

**Multiplying Polynomials (3)**

Observation:

\[ (P_1 + P_2)(Q_1 + Q_2) = P_1 Q_1 + (P_1 P_2 + P_1 Q_2) + P_2 Q_2 \]
\[ (Q_1 P_2 + P_2 Q_1) = (P_1 + P_2)(Q_1 + Q_2) - P_1 Q_1 - P_2 Q_2 \]

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

Recurrence:

\[ T(n) = 3T(n/2) + O(n) \]
\[ T(n) = aT(n/b) + cn^d. \]
\[ \log_b a = \log_2 3 \approx 1.59. \]
\[ T(n) = \Theta(n^{1.59}). \]

**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 the \( 2 \times 2 \) case.

2. Divide and conquer.

In the straightforward implementation, \( 2 \times 2 \) case is:

\[ c_{11} = a_{11} b_{11} + a_{12} b_{21} \]
\[ c_{12} = a_{11} b_{12} + a_{12} b_{22} \]
\[ c_{21} = a_{21} b_{11} + a_{22} b_{21} \]
\[ c_{22} = a_{21} b_{12} + a_{22} b_{22} \]

Requires 8 multiplications and 4 additions.
Another Approach (1)

Compute:

\[ m_1 = (a_{12} - a_{22})(b_{21} + b_{22}) \]
\[ m_2 = (a_{11} + a_{22})(b_{11} + b_{22}) \]
\[ m_3 = (a_{11} - a_{21})(b_{11} + b_{12}) \]
\[ m_4 = (a_{11} + a_{12})b_{22} \]
\[ m_5 = a_{11}(b_{12} - b_{22}) \]
\[ m_6 = a_{22}(b_{21} - b_{11}) \]
\[ m_7 = (a_{21} + a_{22})b_{11} \]

Then:

\[ c_{11} = m_1 + m_2 - m_4 + m_6 \]
\[ c_{12} = m_4 + m_5 \]
\[ c_{21} = m_6 + m_7 \]
\[ c_{22} = m_2 - m_3 + m_5 - m_7 \]

7 multiplications and 18 additions/subtractions.

Another Approach (2)

By Strassen’s algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of \( n \times n \) matrices.

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)

Recurrence:

\[ T(n) = 7T(n/2) + 18(n/2)^2 \]
\[ T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81}). \]

Current “fastest” algorithm is \( \Theta(n^{2.376}) \).

Open question: Can matrix multiplication be done in \( O(n^2) \) time?

But, this has a high constant due to the additions. This makes it rather impractical in real applications.

But this “fastest” algorithm is even more impractical due to overhead.
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?

This is an example of a transform. We do transforms to convert a hard problem into a (relatively) easy problem.

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 \( \mathbf{a} \) of \( n \) values can uniquely represent a polynomial of degree \( n - 1 \)

\[ P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} a_i x^i. \]

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 \textbf{evaluation}.
- Finding the coefficients for the polynomial given the values at \( n \) points is called \textbf{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:

$AB(-1) = (2)(4) = 8$
$AB(0) = (1)(1) = 1$
$AB(1) = (2)(2) = 4$

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