

Numbers

Examples of problems:

- Raise a number to a power.
- Find common factors for two numbers.
- Tell whether a number is prime.
- Generate a random integer.
- Multiply two integers.

These operations use all the digits, and cannot use floating point approximation.

For large numbers, cannot rely on hardware (constant time) operations.

- Measure input size by number of binary digits.
- Multiply, divide become expensive.

Analysis of Number Problems

Analysis problem: Cost may depend on properties of the number other than size.

- It is easy to check an even number for primeness.

If you consider the cost over all k -bit inputs, cost grows with k .

Features:

- Arithmetical operations are not cheap.
- There is only one instance of value n .
- There are 2^k instances of length k or less.
- The size (length) of value n is $\log n$.
- The cost may decrease when n increases in value, but generally increases when n increases in size (length).

Exponentiation

How do we compute m^n ?

We could multiply $n - 1$ times.

Can we do better?

Approaches to divide and conquer:

- Relate m^n to k^n for $k < m$.
- Relate m^n to m^k for $k < n$.

If n is even, then $m^n = m^{n/2}m^{n/2}$.

If n is odd, then $m^n = m^{\lfloor n/2 \rfloor}m^{\lfloor n/2 \rfloor}m$.

```
Power(base, exp) {  
    if exp = 0 return 1;  
    half = Power(base, exp/2);  
    half = half * half;  
    if (odd(exp)) then half = half * base;  
    return half;  
}
```

Analysis of Power

$$f(n) = \begin{cases} 0 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 + n \bmod 2 & n > 1 \end{cases}$$

Solution:

$$f(n) = \lfloor \log n \rfloor + \beta(n) - 1$$

where β is the number of 1's in the binary representation of n .

How does this cost compare with the problem size?

Is this the best possible? What if $n = 15$?

What if n stays the same but m changes over many runs?

In general, finding the best set of multiplications is expensive (probably exponential).

Largest Common Factor

The largest common factor of two numbers is the largest 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

$$n = km + l \text{ where } m > l \geq 0.$$

$$n = \lfloor n/m \rfloor m + n \bmod m.$$

$$\text{So, } f(n, m) = f(m, l) = f(m, n \bmod m).$$

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

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

Analysis of LCF

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:

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: $\Omega(n^2)$.

Another Approach

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.

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:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$

$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Requires 8 multiplications and 4 additions.

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.

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?

Divide and Conquer Recurrences

These have the form:

$$\begin{aligned}T(n) &= aT(n/b) + cn^k \\T(1) &= c\end{aligned}$$

... where a, b, c, k are constants.

A problem of size n is divided into a subproblems of size n/b , while cn^k is the amount of work needed to combine the solutions.

Divide and Conquer Recurrences

(cont)

Expand the sum; $n = b^m$.

$$\begin{aligned}T(n) &= a(aT(n/b^2) + c(n/b)^k) + cn^k \\&= a^m T(1) + a^{m-1}c(n/b^{m-1})^k + \dots + ac(n/b)^k + cn^k \\&= ca^m \sum_{i=0}^m (b^k/a)^i\end{aligned}$$

$$a^m = a^{\log_b n} = n^{\log_b a}$$

The summation is a geometric series whose sum depends on the ratio

$$r = b^k/a.$$

There are 3 cases.

D & C Recurrences (cont)

(1) $r < 1$

$$\sum_{i=0}^m r^i < 1/(1-r), \quad \text{a constant.}$$

$$T(n) = \Theta(a^m) = \Theta(n^{\log_b a}).$$

(2) $r = 1$

$$\sum_{i=0}^m r^i = m + 1 = \log_b n + 1$$

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n)$$

(3) $r > 1$

$$\sum_{i=0}^m r^i = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$$

So, from $T(n) = ca^m \sum r^i$,

$$\begin{aligned} T(n) &= \Theta(a^m r^m) \\ &= \Theta(a^m (b^k/a)^m) \\ &= \Theta(b^{km}) \\ &= \Theta(n^k) \end{aligned}$$

Summary

Theorem 3.4:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

Apply the theorem:

$$T(n) = 3T(n/5) + 8n^2.$$

$$a = 3, b = 5, c = 8, k = 2.$$

$$b^k/a = 25/3.$$

Case (3) holds: $T(n) = \Theta(n^2)$.

Prime Numbers

How do we tell if a number is prime?

One approach is the prime sieve: Test all prime up to $\lfloor \sqrt{n} \rfloor$.

This requires up to $\lfloor \sqrt{n} \rfloor - 1$ divisions.

- How does this compare to the input size?

Note that it is easy to check the number of times 2 divides n for the binary representation

- What about 3?
- What if n is represented in trinary?

Is there a polynomial time algorithm?

Facts about Primes

Some useful theorems from Number Theory:

Prime Number Theorem: The number of primes less than n is (approximately)

$$\frac{n}{\ln n}$$

- The average distance between primes is $\ln n$.

Prime Factors Distribution Theorem: For large n , on average, n has about $\ln \ln n$ different prime factors with a standard deviation of $\sqrt{\ln \ln n}$.

To prove that a number is composite, need only one factor.

What does it take to prove that a number is prime?

Do we need to check all \sqrt{n} candidates?

Probablistic Algorithms

Some probabilistic algorithms:

- $\text{Prime}(n) = \text{FALSE}$.
- With probability $1/\ln n$, $\text{Prime}(n) = \text{TRUE}$.
- Pick a number m between 2 and \sqrt{n} . Say n is prime iff m does not divide n .

Using number theory, we can create a cheap test that will determine that a number is composite (if it is) 50% of the time.

Algorithm:

```
Prime(n) {  
    for(i=0; i<COMFORT; i++)  
        if !CHEAPTEST(n)  
            return FALSE;  
    return TRUE;  
}
```

Of course, this does nothing to help you find the factors!

Random Numbers

Which sequences are random?

- 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
- 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- 2, 7, 1, 8, 2, 8, 1, 8, 2, ...

Meanings of “random”:

- Cannot predict the next item:
unpredictable.
- Series cannot be described more briefly
than to reproduce it: **equidistribution.**

There is no such thing as a random number sequence, only “random enough” sequences.

A sequence is **pseudorandom** if no future term can be predicted in polynomial time, given all past terms.

A Good Random Number Generator

Most computer systems use a deterministic algorithm to select pseudorandom numbers.

Linear congruential method:

- Pick a seed $r(1)$. Then,

$$r(i) = (r(i - 1) \times b) \bmod t.$$

Resulting numbers must be in range:

What happens if $r(i) = r(j)$?

Must pick good values for b and t .

- t should be prime.

Random Number examples

$$r(i) = 6r(i-1) \bmod 13 =$$

..., 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4,
11, 1, ...

$$r(i) = 7r(i-1) \bmod 13 =$$

..., 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4,
2, 1, ...

$$r(i) = 5r(i-1) \bmod 13 =$$

..., 1, 5, 12, 8, 1, ...
..., 2, 10, 11, 3, 2, ...
..., 4, 7, 9, 6, 4, ...
..., 0, 0, ...

Suggested generator:

$$r(i) = 16807r(i-1) \bmod 2^{31} - 1.$$

Introduction to the Slide Rule

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?

The slide rule 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 $n - 1$

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

Alternatively, a 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 (at the same points), we can simply multiply the corresponding pairs of values to get the corresponding values for polynomial AB .

Process:

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

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. (Why?)

An Example

Polynomial A: $x^2 + 1$.

Polynomial B: $2x^2 - x + 1$.

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

Note that evaluating a polynomial at 0 is easy.

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?

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

$$AB(0) = (1)(1) = 1$$

$$AB(1) = (2)(2) = 4$$

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

An Observation

In general, we can write $P_a(x) = E_a(x) + O_a(x)$ where E_a is the even powers and O_a is the odd powers. So,

$$P_a(x) = \sum_{i=0}^{n/2-1} a_{2i}x^{2i} + \sum_{i=0}^{n/2-1} a_{2i+1}x^{2i+1}$$

The significance is that when evaluating the pair of values x and $-x$, we get

$$\begin{aligned} E_a(x) + O_a(x) &= E_a(x) - O_a(-x) \\ O_a(x) &= -O_a(-x) \end{aligned}$$

Thus, we only need to compute the E's and O's once instead of twice to get both evaluations.

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 z is a **primitive nth root of unity** if

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

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

Example: For $n = 4$, $z = i$ or $z = -i$.

Identity: $e^{i\pi} = -1$.

In general, $z^j = e^{2\pi i j/n} = -1^{2j/n}$.

- Significance: We can find as many points on the circle as we need.

Evaluation

Define an $n \times n$ matrix A_z with row i and column j as

$$A_z = (z^{ij}).$$

Example: $n = 4$, $z = i$:

$$A_z = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

Let $a = [a_0, a_1, \dots, a_{n-1}]^T$ be a vector.

We can evaluate the polynomial at the n th roots of unity:

$$F_z = A_z a = b.$$

$$b_i = \sum_{k=0}^{n-1} a_k z^{ik}.$$

Another Example

For $n = 8$, $z = \sqrt{i}$. So,

$$A_z = \begin{matrix} & \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix} \\ \begin{matrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{matrix} & \begin{bmatrix} & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\ & i & -1 & -i & 1 & i & -1 & -i \\ & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\ & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\ & -i & -1 & i & 1 & -i & -1 & i \\ & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i} \end{bmatrix} \end{matrix}$$

We still have two problems:

1. We need to be able to do this fast. Its still n^2 multiplies to evaluate.
2. If we multiply the two sets of evaluations (cheap), we still need to be able to reverse the process (interpolate).

Interpolation

The interpolation step is nearly identical to the evaluation step.

$$F_z^{-1} = A_z^{-1}b' = a'.$$

What is A_z^{-1} ? This turns out to be simple to compute.

$$A_z^{-1} = \frac{1}{n}A_{1/z}$$

In other words, do the same computation as before but substitute $1/z$ for z (and multiply by $1/n$ at the end).

So, if we can do one fast, we can do the other fast.

Fast Polynomial Multiplication

An efficient divide and conquer algorithm exists to perform both the evaluation and the interpolation in $\Theta(n \log n)$ time.

- This is called the **Discrete Fourier Transform** (DFT).
- It is a recursive function that decomposes the matrix multiplications, taking advantage of the symmetries made available by doing evaluation at the n th roots of unity.

Polynomial multiplication of A and B :

- Represent an $n - 1$ -degree polynomial as $2n - 1$ coefficients:

$$[a_0, a_1, \dots, a_{n-1}, 0, \dots, 0]$$

- 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 .

Discrete Fourier Transform

```
Fourier_Transform(double *Polynomial, int n) {
    // Compute the Fourier transform of Polynomial
    // with degree n. Polynomial is a list of
    // coefficients indexed from 0 to n-1. n is
    // assumed to be a power of 2.
    double Even[n/2], Odd[n/2], List1[n/2], List2[n/2];

    if (n==1) return Polynomial[0];
    for (j=0; j<=n/2-1; j++) {
        Even[j] = Polynomial[2j];
        Odd[j] = Polynomial[2j+1];
    }
    List1 = Fourier_Transform(Even, n/2);
    List2 = Fourier_Transform(Odd, n/2);
    for (j=0; j<=n-1, j++) {
        Imaginary z = pow(E, 2*i*PI*j/n);
        k = j % (n/2);
        Polynomial[j] = List1[k] + z*List2[k];
    }
    return Polynomial;
}
```

This just does the transform on one of the two polynomials. The full process is:

1. Transform each polynomial.
2. Multiply resulting values ($O(n)$ multiplies).
3. Do the inverse transformation on the result.

Cost: $\Theta(n \log n)$