CSE 202 - Algorithms

Polynomial Representations,

Fourier Transfer,

and other goodies.

(Chapters 28-30)

Representation matters

Example: simple graph algorithms

Adjacency matrix	Adjacency lists
M[i,j] = 1 iff (i,j) is an edge.	M[i] is list of nodes attached to node i by edge
Requires $\Omega(V^2)$ just to find some edges (if you're unlucky and graph is sparse)	O(E) algorithm for connected components, O(E lg E) for MST ...

Representation matters

Example: long integer operations

Roman numerals	Arabic notation
multiplication isn't hard, but if "size" of instance is number characters, can be $O(N^2)$ (why??)	$O(N^{\lg 3})$, or even better.

Another long integer representation

Let $q_1, q_2, \ldots, q_k$ be relatively prime integers.

Assume each is short, (e.g. $< 2^{31}$)

For $0 < i < q_1\, q_2\, q_k$, represent i as vector:

$< i \pmod{q_1}, i \pmod{q_2}, \ldots, i \pmod{q_k} >$

Converting to this representation takes perhaps $O(k^2)$

k divides of k-long number by a short one.

Converting back requires k Chinese Remainder operations.

Why bother?

+ or x on two such number takes O(k) time.

but comparisons are very time-consuming.

If you have many x's per convert, you can save time!

Representing $(N-1)^{th}$ degree polynomials

To convert:
Evaluate at N points
$O(N^2)$

Use N coefficients	Use N points
$a_0 + a_1 x + a_2 x^2 + \ldots + a_{N-1} x^{N-1}$	$b_0=f(0), \ldots, b_{N-1}=f(N-1)$.
Addition: point wise	Addition: point wise
Multiplication: convolution	Multiplication: point wise
$c_i = \sum a_k b_{i-k}$	

Add N basis functions
$O(N^2)$

So what?

Slow convolutions can be changed to faster point wise ops.

E.g., in signal processing, you often want to form dot product of a set of weights $w_0, w_1, \ldots, w_{N-1}$ with all N cyclic shifts of data $a_0, a_1, \ldots, a_{N-1}$. This is a convolution.

Takes N^2 time done naively.

- Pretend data and weights are coefficients of polynomials.
- Convert each set to point value representations
  Might take $O(N^2)$ time.
- Now do one point wise multiplications
  $O(N)$ time.
- And finally convert each back to original form
  Might take $O(N^2)$ time.

Ooops ... we haven't saved any time.

Fast Fourier Transform (FFT)

It turns out, we can switch back and forth between two representations (often called "time domain" and "frequency domain") in O(N lg N) time.

Basic idea: rather than evaluating polynomial at 0, ..., N-1, do so at the N "roots of unity"

These are the complex numbers that solve $X^N = 1$.

Now use fancy algebraic identities to reduce work.

Bottom line – to form convolution, it only takes

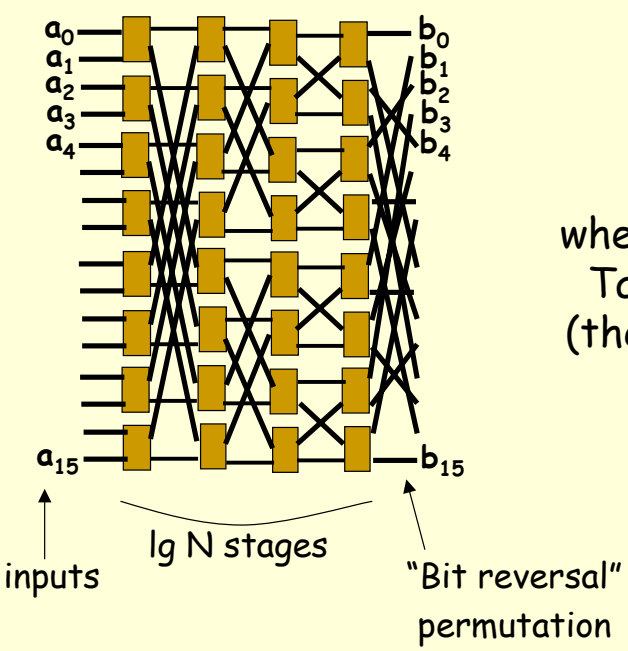
O(N lg N) to convert to frequency domain

O(N) to form convolution (point wise multiplication)

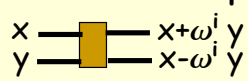
O(N lg N) to convert back.

O(N lg N) instead of $O(N^2)$.

The FFT computation "Butterfly network"



Each box computes ...



where ω^i is some root of unity. Takes 10 floating-point ops (these are complex numbers).

Factoid: In 1990, 40% of all Cray Supercomputer cycles were devoted to FFT's

Fast Matrix Multiplication
(chapter 28)

Naïve matrix multiplication is $O(N^3)$

for multiplying two NxN matrices.

Strassen's Algorithm uses 7 multiplies of N/2 x N/2 matrixes (rather than 8).

$T(N) = 7T(N/2) + c\,N^2$. Thus, T(N) is O(??) = $O(N^{2.81\ldots})$.

It's faster than naïve method for N > 45-ish.

Gives slightly different answer, due to different rounding.

It can be argued that Strassen is more accurate.

Current champion: $O(N^{2.376})$ [Coppersmith & Winograd]

Linear Programming
(chapter 29)

Given n variables $x_1, x_2, \ldots x_n$,

- maximize "objective function" $c_1x_1 + c_2x_2 + \ldots + c_nx_n$,
- subject to the m linear constraints:

$$a_{11}x_1 + a_{12}x_2 + \ldots + c_{1n}x_n \le b_1$$
$$\ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + c_{mn}x_n \le b_m$$

Simplex method: Exponential worst case, but very fast in practice.

Ellipsoid method is polynomial time. There exists some good implementations (e.g. in IBM's OSL library).

NOTE: Integer linear programming (where answer must be integers) is NP-hard (i.e., probably not polynomial time).