CSE 417: Algorithms and Computational Complexity

Divide & Conquer

Autumn 2002
Paul Beame

1

Master Divide and Conquer Recurrence

- If $T(n)= a\cdot T(n/b)+c\cdot n^k$ for $n>b$ then
  - if $a>b^k$ then $T(n)$ is $Q(n^{\log_b a})$
  - if $a<b^k$ then $T(n)$ is $Q(n^k)$
  - if $a=b^k$ then $T(n)$ is $Q(n^k \log n)$
- Works even if it is $\lceil n/b \rceil$ instead of n/b.

2

Another Divide &Conquer Example: Multiplying Faster

- On the first HW you analyzed our usual algorithm for multiplying numbers
  - $Q(n^2)$ time
  - On real machines each "digit" is, e.g., **32** bits long but still get $Q(n^2)$ running time with this algorithm when run on **n**-bit multiplication
- We can do better!
  - We'll describe the basic ideas by multiplying polynomials rather than integers
  - Advantage is we don't get confused by worrying about carries at first

3

Notes on Polynomials

- These are just formal sequences of coefficients
  - when we show something multiplied by x^k it just means shifted **k** places to the left – basically no work

Usual polynomial multiplication

$$
\begin{array}{r}
4x^2 + 2x + 2 \\
x^2 - \ \ 3x + 1 \\
\hline
4x^2 + 2x + 2 \\
-12x^3 - 6x^2 - 6x \\
4x^4 + 2x^3 + 2x^2 \\
\hline
4x^4 - 10x^3 + 0x^2 - 4x + 2
\end{array}
$$

4

Polynomial Multiplication

- **Given:**
  - Degree **n-1** polynomials **P** and **Q**
    - $P = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}$
    - $Q = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-2}x^{n-2} + b_{n-1}x^{n-1}$
- **Compute:**
  - Degree **2n-2** Polynomial **P Q**
  - $P\,Q = a_0b_0 + (a_0b_1+a_1b_0)\,x + (a_0b_2+a_1b_1+a_2b_0)\,x^2 + \ldots + (a_{n-2}b_{n-1}+a_{n-1}b_{n-2})\,x^{2n-3} + a_{n-1}b_{n-1}\,x^{2n-2}$
- **Obvious Algorithm:**
  - Compute all a_ib_j and collect terms
  - $Q\,(n^2)$ time

5

Naive Divide and Conquer

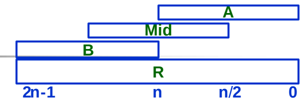
- Assume $n=2k$
  - $P = (a_0 + a_1\ x + a_2 x^2 + \ldots + a_{k-2} x^{k-2} + a_{k-1} x^{k-1}) + (a_k + a_{k+1} x + \ \ldots + a_{n-2}x^{k-2} + a_{n-1}x^{k-1})\ x^k$
    $= P_0 + P_1 x^k$ where P_0 and P_1 are degree **k-1** polynomials
  - Similarly $Q = Q_0 + Q_1 x^k$
- $P\,Q = (P_0+P_1x^k)(Q_0+Q_1x^k)$
  $= P_0Q_0 + (P_1Q_0+P_0Q_1)x^k + P_1Q_1x^{2k}$
- **4** sub-problems of size $k=n/2$ plus linear combining
  - $T(n)=4\cdot T(n/2)+cn$ Solution $T(n) = Q(n^2)$

6

Karatsuba's Algorithm

- A better way to compute the terms
  - Compute
    - $A \leftarrow P_0Q_0$
    - $B \leftarrow P_1Q_1$
    - $C \leftarrow (P_0+P_1)(Q_0+Q_1) = P_0Q_0+P_1Q_0+P_0Q_1+P_1Q_1$
  - Then
    - $P_0Q_1+P_1Q_0 = C-A-B$
    - So $PQ=A+(C-A-B)x^k+Bx^{2k}$
  - 3 sub-problems of size $n/2$ plus $O(n)$ work
    - $T(n) = 3\,T(n/2) + cn$
    - $T(n) = O(n^{a})$ where $a = \log_2 3 = 1.59\ldots$

Karatsuba: Details

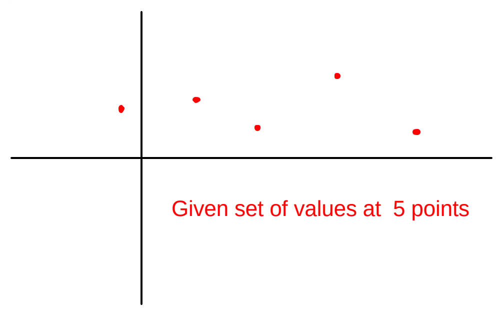


PolyMul(**P**, **Q**):
// **P**, **Q** are length **n** =**2k** vectors, with **P**[**i**], **Q**[**i**] being
// the coefficient of x^i in polynomials **P**, **Q** respectively.
// Let **Pzero** be elements **0..k-1** of **P**; **Pone** be elements **k..n-1**
// **Qzero**, **Qone** : similar
A ← PolyMul(**Pzero**, **Qzero**); // result is a (**2k-1**)-vector
B ← PolyMul(**Pone**, **Qone**); // ditto
Psum ← **Pzero** + **Pone**; // add corresponding elements
Qsum ← **Qzero** + **Qone**; // ditto
C ← polyMul(**Psum**, **Qsum**); // another (**2k-1**)-vector
Mid ← **C** – **A** – **B**; // subtract corresponding elements
R ← **A** + Shift(**Mid**, **n/2**) +Shift(**B**,**n**) // a (**2n-1**)-vector
Return(**R**);

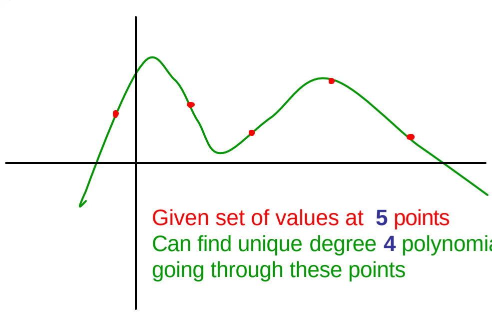
Multiplication

- Polynomials
  - Naïve: $Q(n^2)$
  - Karatsuba: $Q(n^{1.59\ldots})$
  - Best known: $Q(n \log n)$
    - "Fast Fourier Transform"
    - FFT widely used for signal processing
- Integers
  - Similar, but some ugly details re: carries, etc. gives $Q(n \log n \log\log n)$,
    - mostly unused in practice except for symbolic manipulation systems like Maple

Hints towards FFT: Interpolation



Hints towards FFT: Interpolation



Hints towards FFT: Evaluation & Interpolation

P: $a_0, a_1, \ldots, a_{n-1}$
Q: $b_0, b_1, \ldots, b_{n-1}$

ordinary polynomial multiplication $Q(n^2)$

$c_k \leftarrow \sum_{i+j=k} a_i b_j$

R: $c_0, c_1, \ldots, c_{2n-1}$

evaluation at $y_0, \ldots, y_{2n-1}$ $O(?)$

interpolation from $y_0, \ldots, y_{2n-1}$ $O(?)$

$P(y_0), Q(y_0)$
$P(y_1), Q(y_1)$
…
$P(y_{2n-1}), Q(y_{2n-1})$

point-wise multiplication of numbers $O(n)$

$R(y_0) \leftarrow P(y_0)\cdot Q(y_0)$
$R(y_1) \leftarrow P(y_1)\cdot Q(y_1)$
…
$R(y_{2n-1}) \leftarrow P(y_{2n-1})\cdot Q(y_{2n-1})$

Karatsuba's algorithm and evaluation and interpolation

- Strassen gave a way of doing 2x2 matrix multiplies with fewer multiplications
- Karatsuba's algorithm can be thought of as a way of multiplying degree 1 polynomials (which have 2 coefficients) using fewer multiplications
  - $PQ=(P_0+P_1z)(Q_0+Q_1z)$
    $= P_0Q_0 + (P_1Q_0+P_0Q_1)z + P_1Q_1z^2$
  - Evaluate at **0,1,-1** (Could also use other points)
    - $A = P(0)Q(0)= P_0Q_0$
    - $C = P(1)Q(1)=(P_0+P_1)(Q_0+Q_1)$
    - $D = P(-1)Q(-1)=(P_0-P_1)(Q_0-Q_1)$
  - Interpolating, Karatsuba's **Mid=(C-D)/2** and **B=(C+D)/2-A**

13

Hints towards FFT: Evaluation at Special Points

- Evaluation of polynomial at **1** point takes **O(n)**
  - So **2n** points (naively) takes $O(n^2)$—no savings
- Key trick:
  - use carefully chosen points where there's some sharing of work for several points, namely various powers of $w = e^{2p i/n}$, $i = \sqrt{-1}$
- Plus more Divide & Conquer.
- Result:
  - both evaluation and interpolation in **O(n log n)** time

14

Fun facts about $w=e^{2p i/n}$ for even n

- $w^n = 1$
- $w^{n/2} = -1$
- $w^{n/2+k} = -w^k$ for all values of **k**
- $w^2 = e^{2p i/m}$ where **m=n/2**
- $w^k = \cos(2kp/n)+i\sin(2kp/n)$ so can compute with powers of w

15

The key idea for n even

- $P(w) = a_0+a_1w+a_2w^2+a_3w^3+a_4w^4+\ldots+a_{n-1}w^{n-1}$
  $= a_0 +a_2w^2 +a_4w^4 +\ldots+ a_{n-2}w^{n-2}$
  $+ a_1w+a_3w^3 +a_5w^5 +\ldots+a_{n-1}w^{n-1}$
  $= P_{even}(w^2) + w\,P_{odd}(w^2)$
- $P(-w)=a_0 -a_1w+a_2w^2 -a_3w^3+a_4w^4-\ldots\ -a_{n-1}w^{n-1}$
  $= a_0 +a_2w^2 +a_4w^4 +\ldots+ a_{n-2}w^{n-2}$
  $- (a_1w+a_3w^3 +a_5w^5 +\ldots+a_{n-1}w^{n-1})$
  $= P_{even}(w^2) - w\,P_{odd}(w^2)$

where $P_{even}(x) = a_0 +a_2x +a_4x^2 +\ldots+ a_{n-2}x^{n/2-1}$

and $P_{odd}(x) = a_1+a_3x +a_5x^2 +\ldots+a_{n-1}x^{n/2-1}$

16

The recursive idea for n a power of 2

- Also
  - P_{even} and P_{odd} have degree **n/2** where
  - $P(w^k)=P_{even}(w^{2k})+w^kP_{odd}(w^{2k})$
  - $P(-w^k)=P_{even}(w^{2k})-w^kP_{odd}(w^{2k})$
- Recursive Algorithm
  - Evaluate P_{even} at $1,w^2,w^4,\ldots,w^{n-2}$
  - Evaluate P_{odd} at $1,w^2,w^4,\ldots,w^{n-2}$
  - Combine to compute **P** at $1,w,w^2,\ldots,w^{n/2-1}$
  - Combine to compute P at $-1,-w,-w^2,\ldots,-w^{n/2-1}$ (i.e. at $w^{n/2}, w^{n/2+1}, w^{n/2+2},\ldots, w^{n-1}$)

w^2 is $e^{2p i/m}$ where **m=n/2** so problems are of same type but smaller size

17

Analysis and more

- Run-time
  - $T(n)=2T(n/2)+cn$ so $T(n)=O(n \log n)$
- So much for evaluation ... what about interpolation?
  - Given
    - $r_0=R(1)$, $r_1=R(w)$, $r_2=R(w^2)$,..., $r_{n-1}=R(w^{n-1})$
  - Compute
    - $c_0, c_1,\ldots,c_{n-1}$ s.t. $R(x)=c_0+c_1x+\ldots+c_{n-1}x^{n-1}$

18

Interpolation » Evaluation: strange but true

- Weird fact:
  - If we define a new polynomial $S(x) = r_0 + r_1 x + r_2 x^2 + \ldots + r_{n-1} x^{n-1}$ where $r_0, r_1, \ldots, r_{n-1}$ are the evaluations of R at $1, w, \ldots, w^{n-1}$
  - Then $c_k = S(w^{-k})/n$ for $k=0,\ldots,n-1$
- So...
  - evaluate S at $1, w^{-1}, w^{-2}, \ldots, w^{-(n-1)}$ then divide each answer by n to get the $c_0, \ldots, c_{n-1}$
  - w^{-1} behaves just like w did so the same $O(n \log n)$ evaluation algorithm applies !

Divide and Conquer Summary

- Powerful technique, when applicable
- Divide large problem into a few smaller problems of the same type
- Choosing sub-problems of roughly equal size is usually critical
- Examples:
  - Merge sort, quicksort (sort of), polynomial multiplication, FFT, Strassen's matrix multiplication algorithm, powering, binary search, root finding by bisection, …

Why this is called the discrete Fourier transform

- Real Fourier series
  - Given a real valued function f defined on $[0,2\pi]$ the Fourier series for f is given by $f(x)=a_0+a_1 \cos(x) + a_2 \cos(2x) + \ldots + a_m \cos(mx) + \ldots$ where

$$\boxed{a_m = \frac{1}{2\pi}\int_0^{2\pi} f(x)\cos(mx)\,dx}$$

  - is the component of f of frequency m
  - In signal processing and data compression one ignores all but the components with large a_m and there aren't many since

Why this is called the discrete Fourier transform

- Complex Fourier series
  - Given a function f defined on $[0,2\pi]$ the complex Fourier series for f is given by $f(z)=b_0+b_1 e^{iz} + b_2 e^{2iz} + \ldots + b_m e^{miz} + \ldots$ where

$$\boxed{b_m = \frac{1}{2\pi}\int_0^{2\pi} f(z)\, e^{-miz}\,dz}$$

    is the component of f of frequency m
  - If we **discretize** this integral using values at n **2p/n apart** equally spaced points between 0 and 2π we get

$$\boxed{\bar{b}_m = \frac{1}{n}\sum_{k=0}^{n-1} f_k\, e^{-2km\pi i/n} = \frac{1}{n}\sum_{k=0}^{n-1} f_k\, \omega^{-km} \quad \text{where } f_k = f(2k\pi/n)}$$

just like interpolation!