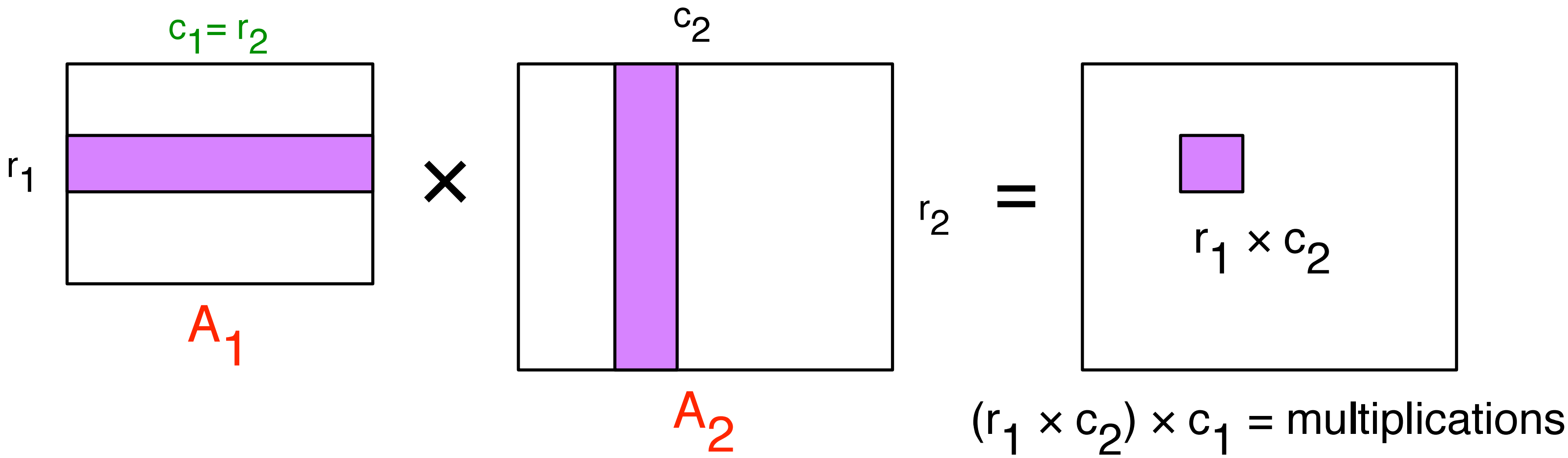
Matrix and Integer Multiplication

David Woodruff

CMU

(Thanks to Carl Kingsford for some of these slides)

Matrix Multiplication



If $r_1 = c_1 = r_2 = c_2 = N$, this standard approach takes $\Theta(N^3)$:

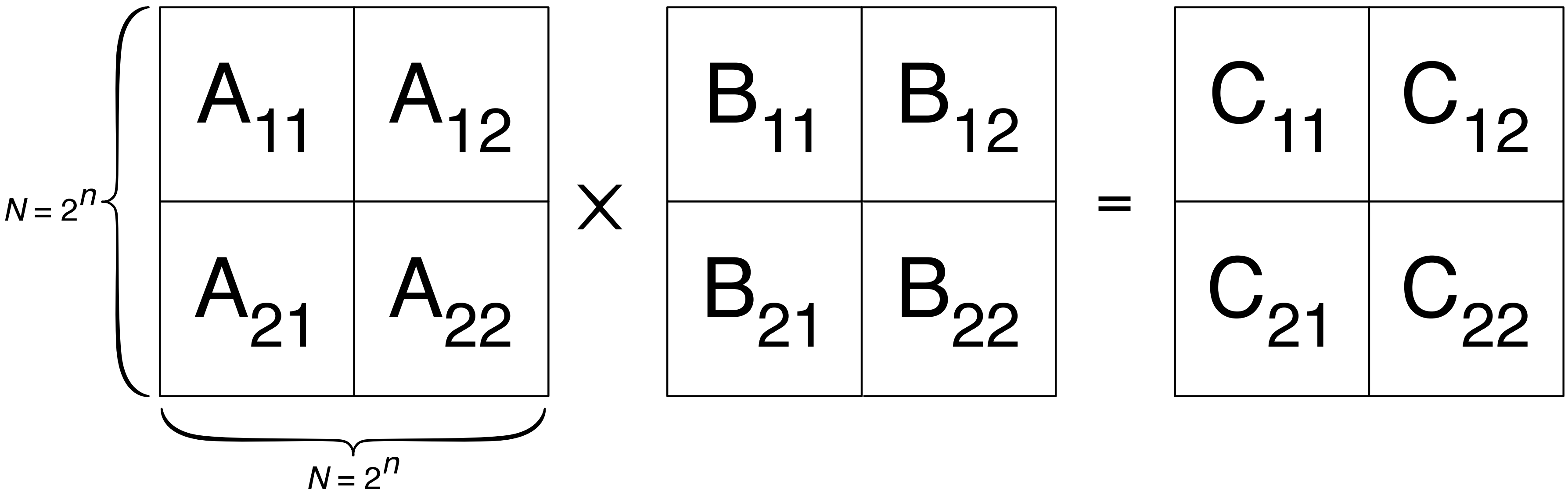
- For every row $\vec{r}$ (N of them)
- For every column $\vec{c}$ (N of them)
- Take their inner product: $r \cdot c$ using N multiplications

Matrix Multiplication Properties

- Suppose A is in $\mathrm{R}^{\mathrm{nx\,k}}$ and B is in $\mathrm{R}^{\mathrm{kx\,m}}$
- In general AB $\neq$BA
- If C is in $\mathrm{R}^{\mathrm{mxt}}$, then
  - (AB) C = A(BC)
  - A(B+C) = AB + AC

Can we multiply faster than $\Theta(N^3)$?

For simplicity, assume $N = 2^n$ for some n. The multiplication is:



- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Uses 8 multiplications

T(N) = 8T(N/2) + c N^2   Master Formula => T(N) = Theta(N^3)

Strassen's Algorithm

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22})B_{11}$$
$$P_3 = A_{11}(B_{12} - B_{22})$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12})B_{22}$$
$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 - P_2 + P_3 + P_6$$

Uses only 7 multiplications!

Since the submatrix multiplications are the expensive operations, we save a lot by eliminating one of them.

Apply the above idea recursively to perform the 7 matrix multiplications contained in $P_1, \ldots, P_7$.

Need to show how much savings this results in overall.

Recurrence

$$T(N) = T(2^n) = \underbrace{7\,T(2^n/2)}_{\text{recursive } \times} \quad + \quad \underbrace{c4^n}_{\textit{additions}}$$

Solving the recurrence:

$$\frac{T(2^n)}{7^n} = \frac{7\,T(2^{n-1})}{7^n} + \frac{c4^n}{7^n} = \frac{T(2^{n-1})}{7^{n-1}} + \frac{c4^n}{7^n}$$

The red term is same as the left-hand side but with $n-1$, so we can recursively expand:

$$\frac{T(2^n)}{7^n} = \gamma + \sum_{i=1}^{n} \frac{c4^i}{7^i} = \gamma + c \sum_{i=1}^{n} \left(\frac{4}{7}\right)^i \leq \alpha \quad \text{for some constants } \alpha,\ \gamma$$

So:

$$T(2^n) \leq 7^n \alpha = \alpha 2^{n \log_2(7)} = \alpha N^{2.807\ldots} = O(N^{2.807\ldots})$$

Space Complexity of Strassen's Algorithm

- Use the same memory for each recursive call

- Start with memory for the two input matrices and output matrix

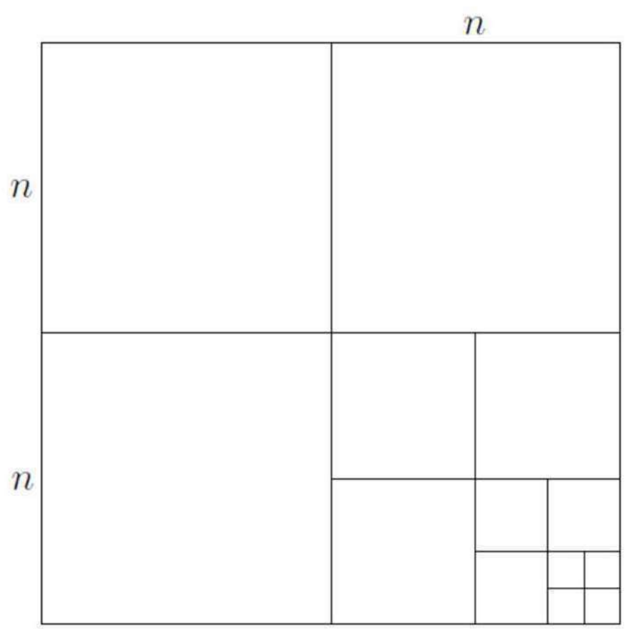
- Allocate $\mathrm{W}\left(\frac{\mathrm{n}}{2}\right)$ memory for recursive computation of P_1
  - When done, add the output to C_{11} and C_{22}
  - Then *reuse* your $\mathrm{W}\left(\frac{\mathrm{n}}{2}\right)$ memory to compute each of $\mathrm{P}_2, \ldots, \mathrm{P}_7$

- Let W(n) be the memory of Strassen's algorithm to multiply n x n matrices
- W(n) = $3\mathrm{n}^2 + \mathrm{W}\left(\frac{\mathrm{n}}{2}\right)$

Bounding the Space Complexity

$$W(n) = 3n^2 + W\left(\frac{n}{2}\right)$$

$$W(n) \leq 4n^2$$



Fast Matrix Multiplication: Practice

Implementation issues.

- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around $n = 128$.

Common misperception. *"Strassen is only a theoretical curiosity."*

- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2{,}500$.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" $Ax = b$, determinant, eigenvalues, SVD, ….

Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen 1969]

$$\Theta(n^{\log_2 7}) = O(n^{2.807})$$

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr 1971]

$$\Theta(n^{\log_2 6}) = O(n^{2.59})$$

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible.

$$\Theta(n^{\log_3 21}) = O(n^{2.77})$$

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, …]

- Two 20-by-20 matrices with 4,460 scalar multiplications. $O(n^{2.805})$
- Two 48-by-48 matrices with 47,217 scalar multiplications. $O(n^{2.7801})$
- A year later. $O(n^{2.7799})$
- December, 1979. $O(n^{2.521813})$
- January, 1980. $O(n^{2.521801})$

Fast Matrix Multiplication: Theory

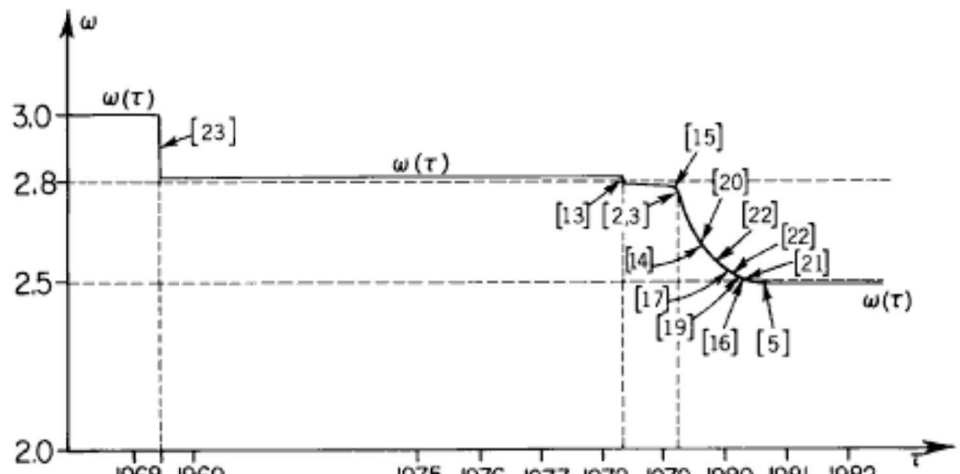


FIG. 1. *$\omega(t)$ is the best exponent announced by time τ.*

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Summary

- Strassen first to show matrix multiplication can be done faster than $O(N^3)$ time.
- Strassen's algorithm gives a performance improvement for large-ish N, depending on the architecture, e.g. $N > 100$ or $N > 1000$.
- Strassen's algorithm isn't optimal though! Over the years it's been improved:

Authors	Year	Runtime
Strassen	1969	$O(N^{2.807})$
⋮		
Coppersmith & Winograd	1990	$O(N^{2.3754})$
Stothers	2010	$O(N^{2.3736})$
Williams	2011	$O(N^{2.3727})$

- Conjecture: an $O(N^2)$ algorithm exists.

Karatsuba's Algorithm for Integer Multiplication

Complex Multiplication

Complex multiplication. $(a + bi)(c + di) = x + yi.$

Grade-school. $x = ac - bd, \; y = bc + ad.$

4 multiplications, 2 additions

Q. Is it possible to do with fewer multiplications?

A. Yes. [Gauss] $x = ac - bd, \; y = (a + b)(c + d) - ac - bd.$

3 multiplications, 5 additions

Remark. Improvement if no hardware multiply.

Integer Multiplication

```
       10101110
     × 01011101
     ----------
       10101110
     10101110
    10101110
   10101110
 10101110
---------------
 11111100110110
```

n numbers of n bits each
$O(n^2)$-time

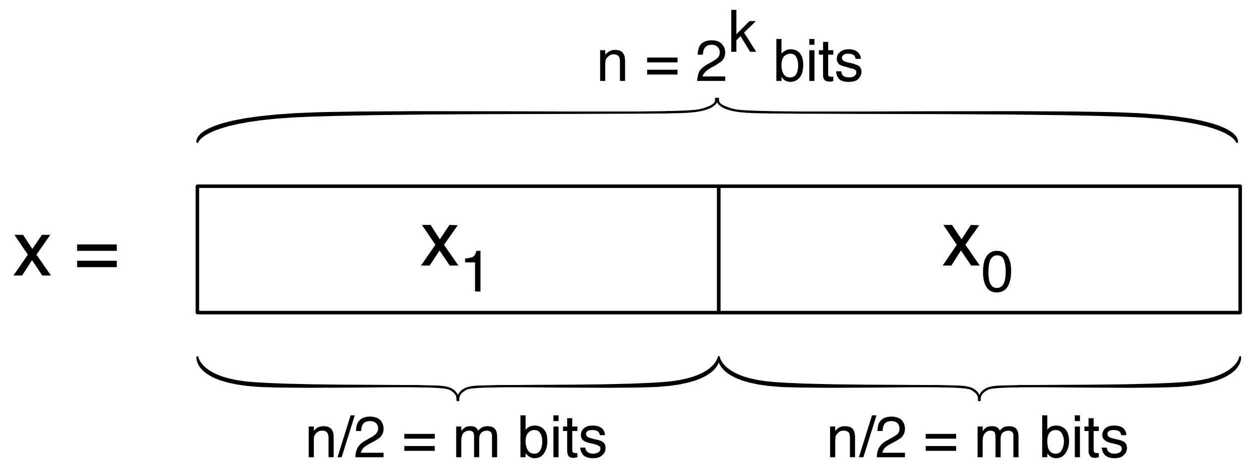
Start similar to Strassen's algorithm, breaking the items into blocks ($m = n/2$):

- $x = x_1 2^m + x_0$
- $y = y_1 2^m + y_0$

Then:

$$xy = (x_1 2^m + x_0)(y_1 2^m + y_0) = x_1 y_1 2^{2m} + (x_1 y_0 + x_0 y_1) 2^m + x_0 y_0$$

Breaking x and y into blocks



$x_1 2^m$ can be computed via "shift right by m"

So this multiplication only costs $O(n)$ operations.

T(n) = 4T(n/2) + O(n)   Master Formula => T(n) = Theta(n^2)

4 Multiplications $\rightarrow$ 3 Multiplications

$$xy = x_1y_12^{2m} + (x_1y_0 + x_0y_1)2^m + x_0y_0$$

We can write two multiplications as one, plus some subtractions:

$$x_1y_0 + x_0y_1 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0$$

But what we need to subtract is exactly what we need for the original multiplication!

- $p_0 = x_0y_0$
- $p_1 = x_1y_1$
- $p_2 = (x_1 + x_0)(y_1 + y_0) - p_1 - p_0$

$$xy = p_12^{2m} + p_22^m + p_0$$

Analysis

Assume $n = 2^k$ for some k (this is the common case when the integers are stored in computer words):

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$\frac{T(2^k)}{3^k} = \frac{T(2^{k-1})}{3^{k-1}} + \frac{c2^k}{3^k}$$

$$= \gamma + c\sum_{i=1}^{k} \frac{2^i}{3^i}$$

$$\leq \beta \quad \text{for some constants } \gamma, \beta$$

(γ handles the constant work for the base case.) So:

$$T(2^k) \leq \beta 3^k = \beta(2^k)^{\log_2(3)} = \beta n^{\log_2(3)} = O(n^{1.58\ldots})$$

Implementation Details

- Karatsuba is usually faster than naïve multiplication for 320-640 bit numbers
- $p_2 = (x_1 + x_0)(y_1 + y_0) - p_1 - p_0$
- $(x_1 + x_0)$ and $(y_1 + y_0)$ could be a number of size 2^{m+1}, which might need an extra bit
- But note $p_2 = (x_0 - x_1)(y_1 - y_0) + p_1 + p_0$
- We might need a bit to encode the sign of $(x_0 - x_1)$ and of $(y_1 - y_0)$
- You can instead record the sign, and multiply the absolute values of these numbers
- One advantage is the final computation of p_2 now involves no subtractions

Toom-Cook Multiplication

- Karatsuba's algorithm reduces 4 multiplications to 3
  - Runs in $\Theta\left(n^{(\log 3)/(\log 2)}\right) = \Theta(n^{1.58})$ time
- The Toom-3 algorithm splits numbers into 3 parts and reduces 9 multiplications to 5
  - Runs in $\Theta(n^{(\log 5)/(\log 3)}) = \Theta(n^{1.46})$ time
- The Toom-k algorithm splits numbers into k parts
  - Runs in $\Theta(c(k)\, n^{\frac{\log(2k-1)}{\log(k)}})$
  - Optimizing gives $\Theta(n2^{\sqrt{(2\log n)}} \log n)$ time

What's Really Going On?

- $x = x_1 \cdot 2^m + x_0$ and $y = y_1 \cdot 2^m + y_0$
- $P(z) = x_1 z + x_0$ and $Q(z) = y_1 z + y_0$
- $x \cdot y = P(2^m) \cdot Q(2^m)$, so integer multiplication can be solved with polynomial multiplication!
- Karatsuba's algorithm is a special case of a fast algorithm for polynomial multiplication. We will discuss polynomials more the next few lectures.
- Using the Fast Fourier Transform to multiply polynomials:
  - Schonage-Strassen algorithm for integer multiplication: O(n log n log log n) time
  - Harvey-van der Hoeven algorithm for integer multiplication: O(n log n) time

Facts About Polynomials

- $A(x) = \sum_{i=0,\ldots,n-1} a_i x^i$ is a degree n-1 polynomial
- A root of a polynomial is a number r for which A(r) = 0
- Fundamental theorem of algebra: a degree-d polynomial has at most d roots
  - Implies any distinct degree d polynomials A(x) and B(x) can evaluate to the same value on at most d different values x. Why?
  - A(x) – B(x) has degree at most d, so can have at most d roots
  - A degree d polynomial is determined by its evaluations on d distinct points $x_1, \ldots, x_d$

Polynomials and Fast Fourier Transform (FFT)

Polynomials

$A(x) = \sum_{i=0}^{n-1} a_i x^i$ a polynomial of degree n-1

Evaluate at a point x = b in time ?

Polynomials

$A(x) = \sum_{i=0}^{n-1} a_i x^i$ a polynomial of degree n-1

Evaluate at a point x = b in time O(n): Horner's rule:
Compute $a_{n-1} x$,

$a_{n-2} + a_{n-1}x^2$,

$a_{n-3} + a_{n-2} x + a_{n-1} x^3$

…

Each step O(1) operations, multiply by and add coefficient.

There are ≤ n steps. ➔ O(n) time

Summing Polynomials

$\sum_{i=0}^{n-1} a_i x^i$ a polynomial of degree n-1

$\sum_{i=0}^{n-1} b_i x^i$ a polynomial of degree n-1

$\sum_{i=0}^{n-1} c_i x^i$ the sum polynomial of degree n-1

$c_i = a_i + b_i$

Time O(n)

How to multiply polynomials?

$\sum_{i=0}^{n-1} a_i x^i$ a polynomial of degree n-1

$\sum_{i=0}^{n-1} b_i x^i$ a polynomial of degree n-1

$\sum_{i=0}^{2n-2} c_i x^i$ the product polynomial of degree n-1

$c_i = \sum_{j \leq i} a_j b_{i-j}$

Trivial algorithm: time $O(n^2)$
FFT gives time $O(n \log n)$

Polynomial representations

Coefficient: $(a_0, a_1, a_2, \ldots a_{n-1})$

Point-value: have points $x_0, x_1, \ldots x_{n-1}$ in mind

Represent polynomials A(X) by pairs

$\{ (x_0, y_0), (x_1, y_1), \ldots \}$ $A(x_i) = y_i$

To multiply in point-value, just need O(n) operations.

Approach to polynomial multiplication:

A, B given as coefficient representation

1) Convert A, B to point-value representation

2) Multiply C = AB in point-value representation

3) Convert C back to coefficient representation

2) done esily in time O(n)

FFT allows to do 1) and 3) in time O(n log n).

Note: For C we need 2n-1 points; we'll just think “n”