CS 4424
Matrix multiplication

Reminder: matrix multiplication

Matrix-matrix product. Starting from

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & & \vdots \\ b_{n,1} & \cdots & b_{n,n} \end{bmatrix},$$

we get AB by multiplying A by all columns of B (or all rows of A by B).

Explicitly,

$$AB = \begin{bmatrix} \cdots & a_{1,1}b_{1,j} + \cdots + a_{1,n}b_{n,j} & \cdots \\ & \vdots & \\ \cdots & a_{n,1}b_{1,j} + \cdots + a_{n,n}b_{n,j} & \cdots \end{bmatrix}.$$

2×2 matrix multiplication

With

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix},$$

we get

$$AB = \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{bmatrix}.$$

Naive algorithm

for $i = 1, \ldots, n$

  for $j = 1, \ldots, n$

    $c_{i,j} = 0$

    for $k = 1, \ldots, n$

      $c_{i,j} = c_{i,j} + a_{i,k} b_{k,j}$

Total: n^3 mul, $n^3 - n^2$ add.

Main results

1. One can multiply matrices of size n using

   - $O(n^3)$ operations (naive algorithm)
   - $O(n^{\log_2 7})$ operations using Strassen's algorithm (1969)
   - (. . . many improvements)
   - $O(n^{2.39})$ operations using Coppersmith and Winograd's algorithm (1990).

We let ω be any number such that matrix multiplication can be done in $O(n^\omega)$ operations, so that

$$\omega \leq 2.39.$$

2. One can invert matrices (and do many other things) in $O(n^\omega)$.

Practical aspects

Many of these algorithms offer no interest for practical computations.

- One uses the naive algorithm (or a couple of variants of it) for sizes up to 100.
- For large sizes, algorithms by Strassen ($\omega = 2.81$) and Pan ($\omega = 2.77$) are sometimes used.
- None of the other ones is useful (threshold too high).
- For matrices with `double` entries, optimizing data access is more important.

Strassen's algorithm

Similar to Karatsuba's algorithm:

- find an improvement for a base case: 2×2 matrices
- use it recursively.

For the 2×2 case, given

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix},$$

- we compute **7** linear combinations of the $a_{i,j}$ and $b_{i,j}$
- multiply them pairwise
- recombine the results

Formulas

$$
\left|\begin{array}{lcl}
q_1 & = & (a_{1,1} - a_{1,2})b_{2,2} \\
q_2 & = & (a_{2,1} - a_{2,2})b_{1,1} \\
q_3 & = & a_{2,2}(b_{1,1} + b_{2,1}) \\
q_4 & = & a_{1,1}(b_{1,2} + b_{2,2}) \\
q_5 & = & (a_{1,1} + a_{2,2})(b_{2,2} - b_{1,1}) \\
q_6 & = & (a_{1,1} + a_{2,1})(b_{1,1} + b_{1,2}) \\
q_7 & = & (a_{1,2} + a_{2,2})(b_{2,1} + b_{2,2})
\end{array}\right.
\quad \text{and} \quad
\left|\begin{array}{lcl}
c_{1,1} & = & q_1 - q_3 - q_5 + q_7 \\
c_{1,2} & = & q_4 - q_1 \\
c_{2,1} & = & q_2 + q_3 \\
c_{2,2} & = & -q_2 - q_4 + q_5 + q_6
\end{array}\right.
$$

$n \times n$ matrices

Suppose that we have to multiply A and B in size n, with $n = 2^k$. We break them into blocks:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

where $A_{i,j}$ et $B_{i,j}$ have size $n/2 \times n/2$.

The formulas we used for the case 2×2 still work. They allow us to multiply A and B using

- 7 products in size $n/2$
- $O(n^2)$ extra operations.

Complexity analysis

Let $\mathsf{MM}(n)$ be the cost of multiplication in size n. Then we have

$$\mathsf{MM}(n) \leq 7\mathsf{MM}(n/2) + \lambda n^2$$

and so

$$\mathsf{MM}(n) \leq Cn^{\frac{\log(7)}{\log(2)}} \leq Cn^{2.81}.$$

Proof: master theorem.

Beyond Strassen

More generally:

- if you find an algorithm that does k multiplications in size n
- then you can take $\omega = \log(n)/\log(k)$
- $n = 2$: $k = 7$ is optimal
- $n = 3$: $k = 23$ is known; $k = 21$ would improve ω
- $n = 4$: $k = 49$
- $n = 5$: $k = 100$ is known; $k = 91$ would improve ω

Remark

- for a given n and k, many attemps done to find algorithms by computer search.

Rectangular matrices

Notation

Let's write

$$\langle n, m, p \rangle$$

for the number of multiplications it takes to multiply matrices

$$(n, m) \times (m, p).$$

Prop. (we've seen that before). If $\langle n, n, n \rangle = k$ then we can take

$$\omega = \frac{\log(k)}{\log(n)}.$$

Prop. If $\langle n, m, p \rangle = k$ then we can take

$$\omega = \frac{3\log(k)}{\log(mnp)}.$$

Steps of the proof

1. (block matrices)

$$\langle mm', nn', pp' \rangle \leq \langle m, n, p \rangle \langle m', n', p' \rangle.$$

2. (permutations)

$$\langle m, n, p \rangle = \langle n, p, m \rangle = \langle p, m, n \rangle.$$

3. (conclusion)

$$\begin{aligned} \langle mnp, mnp, mnp \rangle &\leq \langle m, n, p \rangle \langle n, p, m \rangle \langle p, m, n \rangle \\ &\leq \langle m, n, p \rangle \langle m, n, p \rangle \langle m, n, p \rangle \\ &\leq k^3 \end{aligned}$$

so we can take

$$\omega = \frac{\log(k^3)}{\log(mnp)} = \frac{3\log(k)}{\log(mnp)}.$$

Step 1: block matrices

Suppose we have to multiply A of size (mm', nn') by B of size (nn', pp').

We can decompose them into blocks:

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{bmatrix}, \quad B = \begin{bmatrix} B_{1,1} & \cdots & B_{1,p} \\ \vdots & & \vdots \\ B_{n,1} & \cdots & B_{n,p} \end{bmatrix},$$

where:

- each $A_{i,j}$ is a matrix of size (m', n')
- each $B_{i,j}$ is a matrix of size (n', p').

Step 1: block matrices

Their product is

$$C = \begin{bmatrix} C_{1,1} & \cdots & C_{1,p} \\ \vdots & & \vdots \\ C_{m,1} & \cdots & C_{m,p} \end{bmatrix},$$

where each $C_{i,j}$ is a block of size (m', p')

To compute AB,

- we apply the algorithm in size (m, n, p)
- each of the products is done on blocks of size (m', n', p')
- so the total number of multiplications is $\langle m, n, p \rangle \langle m', n', p' \rangle$.

Step 2: easy permutations

The transpose of a matrix is obtained by switching rows and columns.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \Longrightarrow \quad A^t = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Prop. $B^t A^t = (AB)^t$.

Consequence: (with A of size (m, n) and B of size (n, p))

$$\langle m, n, p \rangle = \langle p, n, m \rangle.$$

Polynomial notation

Multiplication algorithms

We have seen several algorithms for multiplying things

- polynomials (Karatsuba, FFT)
- power series
- matrices

which all have the same structure

- 1. compute combinations of the inputs
- 2. multiply them pairwise
- 3. recombine the products to get the result.

Polynomial notation

We want to describe the multiplication

$$(a_0 + a_1 X)(b_0 + b_1 X) = a_0 b_0 + (a_1 b_0 + a_0 b_1)X + a_1 b_1 X^2.$$

We can describe this operation using a polynomial in variables (A_0, A_1), (B_0, B_1), (C_0, C_1, C_2):

$$\mathbf{P}_{\text{poly2}} = A_0 B_0 C_0 + A_0 B_1 C_1 + A_1 B_0 C_1 + A_1 B_1 C_2.$$

translation

- compute $A_0 B_0$ and add it to C_0
- compute $A_0 B_1$ and add it to C_1
- compute $A_1 B_0$ and add it to C_1
- compute $A_1 B_1$ and add it to C_2

Polynomial notation for Karatsuba

We can rewrite the polynomial $\mathbf{P}_{\text{poly2}}$ as

$$\mathbf{P}_{\text{poly2}} = A_0B_0(C_0 - C_1) + (A_0 + A_1)(B_0 + B_1)C_1 + A_1B_1(C_2 - C_1)$$

This is the same polynomial, just written differently.

Now, the translation is:

- compute A_0B_0, add the result to C_0, subtract it from C_1
- compute $(A_0 + A_1)(B_0 + B_1)$, add the result to C_1
- compute A_1B_1, add the result to C_2, subtract it from C_1.

Polynomial notation for matrices

For the multiplication of 2×2 matrices, we have

$$\begin{aligned}\mathbf{P}_{\text{mat2}} = \; & A_{1,1}B_{1,1}C_{1,1} + A_{1,2}B_{2,1}C_{1,1} + A_{1,1}B_{1,2}C_{1,2} + A_{1,2}B_{2,2}C_{1,2} + \\ & A_{2,1}B_{1,1}C_{2,1} + A_{2,2}B_{2,1}C_{2,1} + A_{2,1}B_{1,2}C_{2,2} + A_{2,2}B_{2,2}C_{2,2}.\end{aligned}$$

translation

- compute $A_{1,1}B_{1,1}$ and add it to $C_{1,1}$,
- compute $A_{1,2}B_{2,1}$ and add it to $C_{1,1}$,
- ...

Polynomial notation for Strassen's algorithm

We can rewrite $\mathbf{P}_{\text{mat2}}$ as

$$\begin{aligned}\mathbf{P}_{\text{mat2}} \quad = \quad & (A_{1,1} - A_{1,2})B_{2,2}(C_{1,1} - C_{1,2}) + \\ & (A_{2,1} - A_{2,2})B_{1,1}C_{2,1} + \\ & A_{2,2}(B_{1,1} + B_{2,1})(-C_{1,1} + C_{2,1}) + \\ & A_{1,1}(B_{1,2} + B_{2,2})(C_{1,2} - C_{2,2}) + \\ & \ldots\end{aligned}$$

translation

- compute $(A_{1,1} - A_{1,2})B_{2,2}$, and add it to $C_{1,1}$, subtract it from $C_{1,2}$
- $\ldots$

Polynomial notation for $(1,2) \times (2,3)$

Let's compute

$$\begin{bmatrix} A_{1,1} & A_{1,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,1} & B_{2,2} & B_{2,3} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} \end{bmatrix}.$$

This gives

$$\begin{aligned} \mathbf{P}_{\text{mat123}} \quad = \quad & A_{1,1}B_{1,1}C_{1,1} + A_{1,2}B_{2,1}C_{1,1} + \\ & A_{1,1}B_{1,2}C_{1,2} + A_{1,2}B_{2,2}C_{1,2} + \\ & A_{1,1}B_{1,3}C_{1,3} + A_{1,2}B_{2,3}C_{1,3}. \end{aligned}$$

Polynomial notation for $(2,3) \times (3,1)$

Let's compute

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{bmatrix} \begin{bmatrix} C_{1,1} \\ C_{2,1} \\ C_{3,1} \end{bmatrix} = \begin{bmatrix} C_{1,1} \\ C_{2,1} \end{bmatrix}.$$

This gives

$$\begin{aligned} \mathbf{P}_{\text{mat231}} \quad = \quad & A_{1,1}B_{1,1}C_{1,1} \; + \; A_{1,2}B_{2,1}C_{1,1} \; + \; A_{1,3}B_{3,1}C_{1,1} \; + \\ & A_{2,1}B_{1,1}C_{2,1} \; + \; A_{2,2}B_{2,1}C_{2,1} \; + \; A_{2,3}B_{3,1}C_{2,1}. \end{aligned}$$

Comparison

$$\begin{aligned}\mathbf{P}_{\mathrm{mat123}} \;=\; & A_{1,1}B_{1,1}C_{1,1} + A_{1,2}B_{2,1}C_{1,1} + \\ & A_{1,1}B_{1,2}C_{1,2} + A_{1,2}B_{2,2}C_{1,2} + \\ & A_{1,1}B_{1,3}C_{1,3} + A_{1,2}B_{2,3}C_{1,3}.\end{aligned}$$

$$\begin{aligned}\mathbf{P}_{\mathrm{mat231}} \;=\; & A_{1,1}B_{1,1}C_{1,1} + A_{1,2}B_{2,1}C_{1,1} + A_{1,3}B_{3,1}C_{1,1} + \\ & A_{2,1}B_{1,1}C_{2,1} + A_{2,2}B_{2,1}C_{2,1} + A_{2,3}B_{3,1}C_{2,1}.\end{aligned}$$

Conclusion:

- up to replacing $A_{i,j}$ by $C_{j,i}$, $B_{i,j}$ by $A_{i,j}$ and $C_{i,j}$ by $B_{j,i}$, these are the same polynomials
- so an algorithm for $\mathbf{P}_{\mathrm{mat231}}$ gives an algorithm for $\mathbf{P}_{\mathrm{mat123}}$
- so $\langle 2,3,1\rangle = \langle 1,2,3\rangle$
- true in general

Approximate algorithms: an example with power series

Modular multiplication

Reminder: to multiply two polynomials A, B modulo a polynomial P (with $\deg(P) = d$)

- compute $C = AB$
- return $D = C \text{ rem } P$.

Alternative solution when all the roots $r_1, \ldots, r_d$ of P are known

- compute the values of A and B at $r_1, \ldots, r_d$
- return the polynomial E such that
  - $E(a_i) = A(r_i)B(r_i)$
  - $\deg(E) < d$

Prop: $D = E$.

Remark: FFT multiplication

Given A, B:

- evaluate A and B at roots of unity of high enough order n, by FFT
- multiply the values
- do an inverse FFT to get $C = AB$

In general, this will return

$$AB \text{ rem}(X^n - 1).$$

When n is large enough, there is no reduction, and we get AB.

Interpolation

Prop.

- Given two different elements r_0, r_1 in k and two values v_0, v_1, the unique polynomial P such that

$$P(r_0) = v_0, \quad P(r_1) = v_1, \quad \deg(P) < 2$$

is

$$P = v_0 \frac{X - r_1}{r_0 - r_1} + v_1 \frac{X - r_0}{r_1 - r_0}.$$

Remark: similar formula with more points, but we won't need it.

Two similar situations

1. computing modulo $P = X^2$

$$(a_0 + a_1 X)(b_0 + b_1 X) \bmod X^2 = a_0 b_0 + (a_0 b_1 + a_1 b_0) X$$

- naive: 3 multiplications
- no improvement possible over the naive algorithm

2. computing modulo $P = X^2 - 1 = (X - 1)(X + 1)$

$$(a_0 + a_1 X)(b_0 + b_1 X) \bmod (X^2 - 1) = a_0 b_0 + a_1 b_1 + (a_0 b_1 + a_1 b_0) X$$

- naive: 4 multiplications
- Karatsuba: 3 multiplications
- evaluation / interpolation: 2 multiplications

$$C = (a_0 + a_1)(b_0 + b_1)\frac{X+1}{2} - (a_0 - a_1)(b_0 - b_1)\frac{X-1}{2}.$$

More generally

Computing modulo $P = X^2 - r^2 = (X - r)(X + r)$:

$$(a_0 + a_1 X)(b_0 + b_1 X) \text{ rem } (X^2 - r^2) = a_0 b_0 + r^2 a_1 b_1 + (a_0 b_1 + a_1 b_0) X$$

- naive: 4 multiplications
- Karatsuba: 3 multiplications
- evaluation / interpolation: 2 multiplications

$$C = (a_0 + a_1 r)(b_0 + b_1 r)\frac{X + r}{2r} - (a_0 - a_1 r)(b_0 - b_1 r)\frac{X - r}{2r}.$$

An approximate solution

Let's suppose we can compute with real coefficients, and take

$$r = 10^{-10},$$

so we are computing modulo

$$X^2 - 10^{-20}.$$

We multiply $(10 + 12X)$ and $(-4 + 5X)$, which gives

$$\begin{aligned} & (10 + 12X)(-4 + 5X) \text{ rem } (X^2 - 10^{-20}) \\ = \quad & -39.999999999999999994 + 2X. \end{aligned}$$

This is (of course) very close to

$$(10 + 12X)(-4 + 5X) \text{ rem } X^2 = -40 + 2X.$$

We can get the former using two multiplications, but the latter requires three.

Making this formal

We introduce a new variable ε. Then, to compute

$$(a_0 + a_1 X)(b_0 + b_1 X) \text{ rem } X^2$$

we do the following:

- we compute

$$\begin{aligned} C_\varepsilon &= (a_0 + a_1 X)(b_0 + b_1 X) \text{ rem } (X^2 - \varepsilon^2) \\ &= (a_0 b_0 + a_1 b_1 \varepsilon^2) + (a_0 b_1 + a_1 b_0) X. \end{aligned}$$

  We do this by evaluation / interpolation:

$$C_\varepsilon = (a_0 + a_1 \varepsilon)(b_0 + b_1 \varepsilon)\frac{X + \varepsilon}{2\varepsilon} - (a_0 - a_1 \varepsilon)(b_0 - b_1 \varepsilon)\frac{X - \varepsilon}{2\varepsilon}.$$

- we replace ε by 0.

Using the polynomial notation

Multiplication modulo X^2 is represented by

$$\mathbf{P} = A_0B_0C_0 + A_0B_1C_1 + A_1B_0C_1.$$

Multiplication modulo $X^2 - \varepsilon^2$ is

$$\begin{aligned}\mathbf{P}_\varepsilon &= A_0B_0C_0 + \varepsilon^2 A_1B_1C_0 + A_0B_1C_1 + A_1B_0C_1 \\ &= \mathbf{P} + \varepsilon^2\mathbf{Q}.\end{aligned}$$

The evaluation / interpolation algorithm shows

$$\mathbf{P}_\varepsilon = \frac{(A_0 + A_1\varepsilon)(B_0 + B_1\varepsilon)(\varepsilon C_0 + C_1)}{2\varepsilon} - \frac{(A_0 - A_1\varepsilon)(B_0 - B_1\varepsilon)(\varepsilon C_0 + C_1)}{2\varepsilon}.$$

Summary

We get

$$\frac{(A_0 + A_1\varepsilon)(B_0 + B_1\varepsilon)(\varepsilon C_0 + C_1)}{2\varepsilon} - \frac{(A_0 - A_1\varepsilon)(B_0 - B_1\varepsilon)(\varepsilon C_0 + C_1)}{2\varepsilon} = \mathbf{P} + \varepsilon^2\mathbf{Q},$$

or

$$\frac{(A_0 + A_1\varepsilon)(B_0 + B_1\varepsilon)(\varepsilon C_0 + C_1)}{2} - \frac{(A_0 - A_1\varepsilon)(B_0 - B_1\varepsilon)(\varepsilon C_0 + C_1)}{2} = \varepsilon\mathbf{P} + \varepsilon^3\mathbf{Q}.$$

The left-hand side gives us

- an algorithm using 2 multiplications
- with coefficients involving ε
- that computes ε times what we want
- plus an higher-order term $\mathbf{Q}$ that we can dismiss afterwards.

Approximate algorithms for matrix multiplication

Multiplication of partial matrices

We consider the product $AB = C$, with

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & 0 \end{bmatrix}, \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix}.$$

The naive algorithm uses 6 multiplications:

$$\begin{aligned} \mathbf{P} \quad = \quad & A_{1,1}B_{1,1}C_{1,1} + A_{1,2}B_{2,1}C_{1,1} + \\ & A_{1,1}B_{1,2}C_{1,2} + A_{1,2}B_{2,2}C_{1,2} + \\ & A_{2,1}B_{1,1}C_{2,1} + \\ & A_{2,1}B_{1,2}C_{2,2}. \end{aligned}$$

It is optimal.

An approximate algorithm

The algorithm described by

$$\begin{aligned}\mathbf{P}_\varepsilon \quad = \quad & (A_{1,2}+\varepsilon A_{1,1})(B_{1,2}+\varepsilon B_{2,2})C_{1,2} \\ & (A_{2,1}+\varepsilon A_{1,1})B_{1,1}(C_{1,1}+\varepsilon C_{2,1}) \\ & -A_{1,2}B_{1,2}(C_{1,1}+C_{1,2}+\varepsilon C_{2,2}) \\ & -A_{2,1}(B_{1,1}+B_{1,2}+\varepsilon B_{2,1})C_{1,1} \\ & -(A_{1,2}+A_{2,1})(B_{1,2}+\varepsilon B_{2,1})(C_{1,1}+\varepsilon C_{2,2})\end{aligned}$$

gives us

$$\mathbf{P}_\varepsilon = \varepsilon\mathbf{P} + \varepsilon^2\mathbf{Q},$$

for some higher-order term $\mathbf{Q}$.

Rectangular matrix multiplication

We can use this trick to compute $AB = C$ with

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix}, \quad B = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix}.$$

This gives us an algorithm for $\langle 3, 2, 2 \rangle$

- using 10 multiplications
- with coefficients involving ε
- that computes ε times what we want
- plus an higher-order term $\mathbf{Q}$ that we can dismiss afterwards.

Towards an algorithm for square matrices

As we did for "usual" algorithms, we can apply permutations. This give algorithms for $\langle 3, 2, 2\rangle$, $\langle 2, 3, 2\rangle$ and $\langle 2, 2, 3\rangle$

- using 10 multiplications
- with coefficients involving ε
- that compute ε times what we want
- plus higher-order terms that we can dismiss afterwards.

Formalization

An algorithm

- using k multiplications
- with coefficients involving ε
- that compute ε^{r-1} times $\langle m, n, p \rangle$ plus higher-order terms.

is called an approximate algorithm of order r for $\langle m, n, p \rangle$.

Prop

- Given an approximate algorithm of order r for $\langle m, n, p \rangle$, involving k multiplications, we can deduce a (normal) algorithm for $\langle m, n, p \rangle$ involving $k\mathsf{M}(r)$ multiplications.

Proof: all operations are done using power series addition and multiplication, modulo ε^r.

Block products

Prop. Suppose we have

- an algorithm of order r for $\langle m, n, p\rangle$
- an algorithm of order r' for $\langle m', n', p'\rangle$

Then we can deduce an algorithm of order $r + r' - 1$ for $\langle mm', nn, pp'\rangle$.

Proof. We are doing a product $\langle m, n, p\rangle$ of matrices of size $\langle m', n', p'\rangle$.

1. We recursively compute products of the form

$$\varepsilon^{r'-1}\langle m', n'p'\rangle + \varepsilon^{r'} \cdots$$

2. We plug them into our approximate algorithm for $\langle m, n, p\rangle$. This gives

$$\varepsilon^{r'-1}\varepsilon^{r-1}\langle mm', nn', pp'\rangle + \varepsilon^{r'+r-1} \cdots$$

Using block multiplication

We have approximate algorithms of order 2 for $\langle 3,2,2\rangle$, $\langle 2,3,2\rangle$ and $\langle 2,2,3\rangle$, with 10 multiplications.

- This gives us an approximate algorithm of order 4 for $\langle 12,12,12\rangle$, with 1000 multiplications.
- and thus an approximate algorithm of order 7 for $\langle 12^2,12^2,12^2\rangle$, with 1000^2 multiplications.
- and thus an approximate algorithm of order 10 for $\langle 12^3,12^3,12^3\rangle$, with 1000^3 multiplications.
- and thus an approximate algorithm of order $3N+1$ for $\langle 12^N,12^N,12^N\rangle$, with 1000^N multiplications.
- and thus an algorithm for $\langle 12^N,12^N,12^N\rangle$, with $1000^N\mathsf{M}(3N+1)$ multiplications.

Conclusion

Each of these algorithms gives us a value for ω:

$$\begin{aligned}
\omega_N &= \frac{\log(1000^N \mathsf{M}(3N+1))}{\log(12^N)} \\
&= \frac{\log(1000^N)}{\log(12^N)} + \frac{\log(\mathsf{M}(3N+1))}{\log(12^N)} \\
&= 3\frac{\log(10)}{\log(12)} + \frac{\log(\mathsf{M}(3N+1))}{\log(12^N)}
\end{aligned}$$

so that $\omega_N \to 3\frac{\log(10)}{\log(12)} \simeq 2.78$.

In practice

i	12^i	ratio vs. naive algo
1	12	2.31
2	144	2.34
3	1728	1.93
4	20736	1.45
5	248832	1.03

i	12^i	ratio vs. Strassen
1	12	3.02
6	2985984	3.54
11	743008370688	1.56
16	184884258895036416	0.55

Swiss cheese matrices

Using sparsity

We consider matrices with many zero entries (we'll call them motives).

Theorem.

- Consider two motives A and B, of sizes (k, m) and (m, n).
- Suppose that you can compute $A \times B$ using ℓ products (possibly using approximate algorithms). Let f be the number of products in the naive product $A \times B$.

  Then we can take

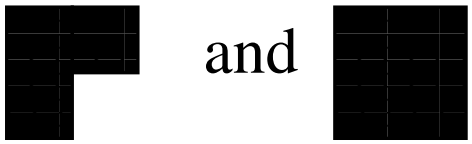
$$\omega \leq 3 \, \frac{\log \ell}{\log f}.$$

Simple examples.

- gives $\omega = 3$ for the naive algorithm
- gives $\omega = 2.81$ for Strassen.

Better examples

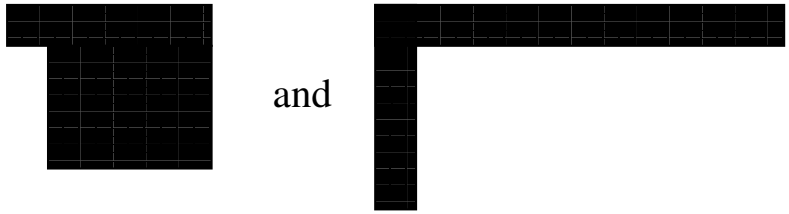
1. Back the previous example:



We had $\omega \leq 3\log(10)/\log(12) \simeq 2.78$

Now, $\ell = 5,\ f = 6 \implies \omega \leq 3\log(5)/\log(6) \simeq 2.70.$

2. The product of

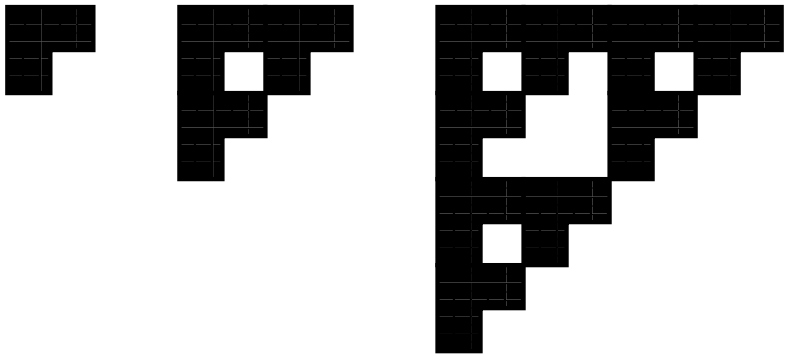


can be done with $\ell = 17$, whereas $f = 26 \implies \omega \leq 3\log(17)/\log(26) \simeq 2.61.$

Tensor powers

We write $A^{(s)}$ and $B^{(s)}$ the sth **tensor powers** of A and B.

On the previous example, $A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & 0 \end{bmatrix}$, the powers $A^{(1)}$, $A^{(2)}$ and $A^{(3)}$ are



(I am not writing the actual coefficients)

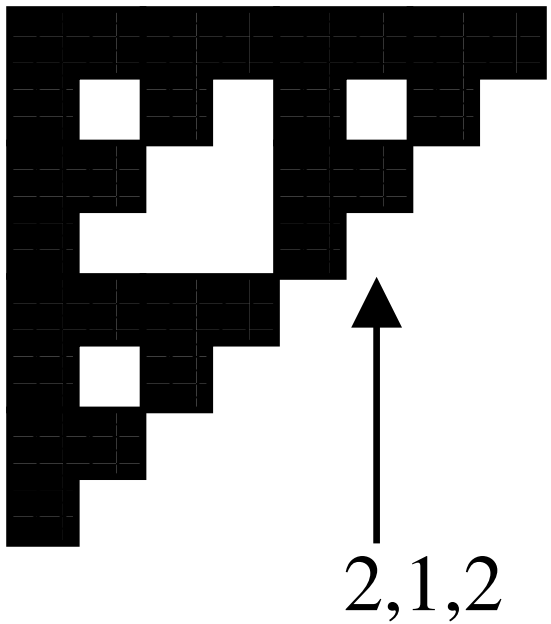
Rows and columns

Each column of $A^{(s)}$ is indexed by an s-uple μ of the form

$$\mu = (\mu_1, \ldots, \mu_s),$$

with $1 \leq \mu_i \leq m$.

Example. For $s = 3$, this gives:



Same thing for the rows of $B^{(s)}$.

Counting zeros

Prop.

- Let $k_1, \ldots, k_m$ be the numbers of non-zero entries in the columns $1, \ldots, m$ in A.
- Let C_μ be a column in $A^{(s)}$, with $\mu = (\mu_1, \ldots, \mu_s)$.
- Then the number of non-zero entries in C_μ is

$$K(\mu) = k_{\mu_1} \cdots k_{\mu_s}.$$

If we let σ_i be the number of μ_i's that are equal to i, we have

$$K(\mu) = k_1^{\sigma_1} \cdots k_m^{\sigma_m}.$$

Example. $\mu = (2, 1, 2) \implies \sigma = (1, 2) \implies K(\mu) = 2^1 1^2 = 2.$

Same thing for the columns of $B^{(s)}$; we write

$$N(\mu) = n_1^{\sigma_1} \cdots n_m^{\sigma_m}.$$

Cleaning

We fix $\sigma_1, \ldots, \sigma_m$. There are

$$M_\sigma = \frac{s!}{\sigma_1!\ \sigma_2!\ \cdots\ \sigma_m!}$$

choices $\mu = (\mu_1, \ldots, \mu_s)$ for the columns of $A^{(s)}$. For all these columns,

$$K(\mu) = K_\sigma = k_1^{\sigma_1} \cdots k_m^{\sigma_m}.$$

Same things for the rows of $B^{(s)}$, with

$$N(\mu) = N_\sigma = n_1^{\sigma_1} \cdots n_m^{\sigma_m}.$$

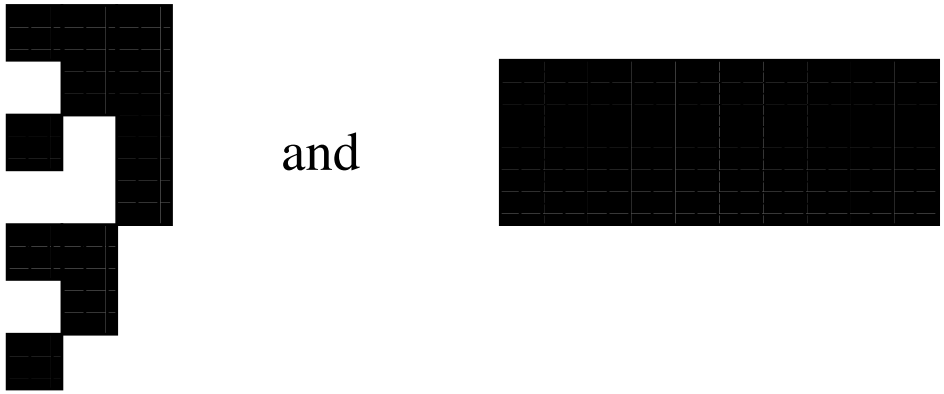
We remove all other columns of $A^{(s)}$ and rows of $B^{(s)}$ et we call $\mathfrak{A}_\sigma$ and $\mathfrak{B}_\sigma$ what is left.

Cleaning

Example: $\sigma = (2, 1)$.

We find the columns / rows $(1, 1, 2)$, $(1, 2, 1)$ and $(2, 1, 1)$, with $K_\sigma = 4, M_\sigma = 3, N_\sigma = 8$.

The remaining $\mathfrak{A}_\sigma$ and $\mathfrak{B}_\sigma$:



From sparse to full

Let U and V be any matrices of sizes (K_σ, M_σ) and (M_σ, N_σ).

Prop. On can find G and Q such that

$$U = G\mathfrak{A}_\sigma, \quad V = \mathfrak{B}_\sigma Q.$$

and thus $UV = G(\mathfrak{A}_\sigma \mathfrak{B}_\sigma)Q$.

So If we can compute $A \times B$ with ℓ product, then

$$\langle K_\sigma, M_\sigma, N_\sigma \rangle \leq \ell^s$$

and thus

$$\omega \leq 3 \frac{\log(\ell^s)}{\log(K_\sigma M_\sigma N_\sigma)}.$$

Asymptotically

Remember that

$$K_\sigma M_\sigma N_\sigma = \frac{s!}{\sigma_1!\ \sigma_2!\ \cdots\ \sigma_m!}(k_1 n_1)^{\sigma_1} \cdots (k_m n_m)^{\sigma_m}.$$

This is one of the terms in the expansion of

$$(k_1 n_1 + \cdots + k_m n_m)^s.$$

There are

$$\binom{s+m-1}{m-1}$$

terms in the expansion. So, there is at least one choice of σ such that

$$K_\sigma M_\sigma N_\sigma \geq \frac{(k_1 n_1 + \cdots + k_m n_m)^s}{\binom{s+m-1}{m-1}}.$$

Asymptotically

This gives

$$\log(K_\sigma M_\sigma N_\sigma) \geq s\log(k_1 n_1 + \cdots + k_m n_m) - \log(\binom{s+m-1}{m-1})$$

or

$$\frac{1}{\log(K_\sigma M_\sigma N_\sigma)} \leq \frac{1}{s\log(k_1 n_1 + \cdots + k_m n_m) - \log(\binom{s+m-1}{m-1})}$$

Since

$$\binom{s+m-1}{m-1} \leq (s+m-1)^{m-1}$$

we get

$$\frac{1}{\log(K_\sigma M_\sigma N_\sigma)} \leq \frac{1}{s\log(k_1 n_1 + \cdots + k_m n_m) - (m-1)\log(s+m-1)}.$$

Conclusion

Let f be the number of operations in the naive product $A \times B$. Then

$$f = k_1 n_1 + \cdots + k_m n_m.$$

So we get

$$\omega \leq 3 \frac{\log(\ell^s)}{s \log f - (m-1) \log(s+m-1)}.$$

With $s \to \infty$:

$$\omega \leq 3 \frac{\log \ell}{\log f}.$$