Matrix Multiplication

input: $n \times n$ matrices A and B.
output: the matrix product $C = A \times B$. (So $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.)

We assume throughout that multiplication and addition of two numbers can be done in constant time. We also assume n is a power of 2 (we can reduce the general case to this one). The obvious algorithm takes time $O(n^3)$. We want time $o(n^3)$.

To warm up, consider the following recursive algorithm. Let A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, C_{11}, C_{12}, C_{21} and C_{22} be $\frac{n}{2} \times \frac{n}{2}$ submatrices such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \;\; \text{and } C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

We are given A and B, and want to compute C. The following matrix equation holds for each of the four pairs $i, j \in \{1, 2\}$:

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j}. \tag{1}$$

This gives the following recursive algorithm:

matrix-product-1(n, A, B) *(A and B are $n \times n$ matrices; n is a power of two)*
1. if $n = 1$: return $[a_{11} b_{11}]$ *(base case)*
2. for each of the four pairs $i, j \in \{1, 2\}$, compute A_{ij} and B_{ij} as defined above *(time $O(n^2)$)*
3. for each of the four pairs $i, j \in \{1, 2\}$:
4.1. let $C_{ij} =$ matrix-product-1$(n/2, A_{i1}, B_{1j})$ + matrix-product-1$(n/2, A_{i2}, B_{2j})$
5. return $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$

Correctness of the algorithm follows from Equation (1), by induction.

To compute the product of two $n \times n$ matrices, the algorithm makes eight recursive calls on $(n/2) \times (n/2)$ matrices. The time spent outside of the recursive calls is $O(n^2)$. So the running time satisfies $T(n) = 8T(n/2) + O(n^2)$. Using the recursion-tree method, the total time is proportional to

$$\sum_{i=0}^{\#\text{levels}} (\#\text{ subproblems in level } i) \times (\text{work per subproblem in level } i)$$
$$= \sum_{i=0}^{\log_2 n} (8^i) \times (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} 2^i = n^2\, \Theta(2^{\log_2 n}) = \Theta(n^3).$$

(We use above that the sum is geometric, so its value is proportional to its largest term. Then we use $2^{\log_2 n} = n$.) To improve the running time, we'll reduce the number of recursive calls from eight to seven. To do this, replace Steps 3 and 4.1 with the following steps. First, compute the following seven $\frac{n}{2} \times \frac{n}{2}$ matrices, using seven recursive matrix multiplications:

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) & M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} - B_{22}) & M_4 &= A_{22} \times (B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} & M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

Strassen figured out that those seven matrices satisfy the following equations:

Lemma 1 (Strassen)**.**

$$C_{11} = M_1 + M_4 - M_5 + M_7 \qquad C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4 \qquad C_{22} = M_1 - M_2 + M_3 + M_6$$

(We don't give the proof, but you can check each equation if you like.)

Then modify the algorithm to compute each C_{ij} using the four equations above in time $O(n^2)$.

Correctness of the algorithm follows by the lemma (and induction on n). The time now satisfies $T(n) = \mathbf{7}\,T(n/2) + O(n^2)$, so the total time is proportional to

$$\sum_{i=0}^{\log_2 n} (7^i) \times (n/2^i)^2 = n^2 \sum_{i=0}^{\log_2 n} (7/4)^i = n^2\,\Theta((7/4)^{\log_2 n}) = \Theta(n^{\log_2 7}) = \Theta(n^{2.80\cdots}).$$

(Above we use $x^{\log y} = y^{\log x}$ and $\log_2(7/4) = \log_2(7) - 2$.) We have the following result:

Theorem 1 (Strassen)**.** *There is an $O(n^{\log_2 7})$-time algorithm for multiplying $n \times n$ matrices.*

Large-Integer Multiplication

input: n-bit integers A and B, with binary representations $A = a_1 a_2 \ldots a_n$ and $B = b_1 b_2 \ldots b_n$.
output: the product $C = A\,B$.

This problem is about multiplying integers that are too large to fit in a machine word. (For example, RSA encryption currently requires arithmetic on 2048-bit integers, or larger.)

The standard grade-school algorithm takes time $\Theta(n^2)$. We want time $o(n^2)$.

To start, consider the following recursive algorithm:

int-product-1($n, A = a_1 a_2 \ldots a_n, B = b_1 b_2 \ldots b_n$) *($A$ and B are n-bit integers; n is a power of two)*
1. if $n = 1$: return $a_1 b_1$ *(base case)*
2. split A into $A_1 = a_1 \cdots a_{n/2}$ and $A_2 = a_{n/2+1} \cdots a_n$ so that $A = 2^{n/2} A_1 + A_2$.
3. split B into $B_1 = b_1 \cdots b_{n/2}$ and $B_2 = b_{n/2+1} \cdots b_n$ so that $B = 2^{n/2} B_1 + B_2$.
4. return $2^n(A_1 \times B_1) + 2^{n/2}(A_1 \times B_2 + A_2 \times B_1) + (A_2 \times B_2)$

The last step requires four multiplications of $n/2$-bit integers. These are done recursively.

Correctness of the algorithm follows by induction from the equation

$$\begin{aligned} A \times B &= (2^{n/2} A_1 + A_2) \times (2^{n/2} B_1 + B_2) \\ &= 2^n(A_1 \times B_1) + 2^{n/2}(A_1 \times B_2 + A_2 \times B_1) + (A_2 \times B_2). \end{aligned} \tag{2}$$

Each addition or multiplication by a power of two (shifting) can be done in $O(n)$ time. So the running time satisfies $T(n) = 4T(n/2) + n$, which implies

$$T(n) = \sum_{i=0}^{\log_2 n} (4^i)(n/2^i) = n \sum_{i=0}^{\log_2 n} 2^i = n\,\Theta(2^{\log_2 n}) = \Theta(n^2).$$

To reduce the time, note that the middle term in Step 4, $A_1 \times B_2 + A_2 \times B_1$, satisfies

$$A_1 \times B_2 + A_2 \times B_1 = (A_1 + A_2) \times (B_1 + B_2) - A_1 \times B_1 - A_2 \times B_2. \tag{3}$$

The products $A_1 \times B_1$ and $A_2 \times B_2$ have to be computed anyway for Step 4, so computing the right-hand side above requires only *one* additional recursive call instead of two.

This gives us Karatsuba's algorithm:

int-product$(n, A = a_1 a_2 \ldots a_n, B = b_1 b_2 \ldots b_n)$ *(A and B are n-bit integers; n is a power of two)*
1. if $n = 1$: return $a_1 b_1$ *(base case)*
2. split A into $A_1 = a_1 \cdots a_{n/2}$ and $A_2 = a_{n/2+1} \cdots a_n$ so that $A = 2^{n/2} A_1 + A_2$.
3. split B into $B_1 = b_1 \cdots b_{n/2}$ and $B_2 = b_{n/2+1} \cdots b_n$ so that $B = 2^{n/2} B_1 + B_2$.
4. let $M_1 =$ int-product$(n/2, A_1, B_1)$ *($M_1 = A_1 \times B_1$)*
5. let $M_2 =$ int-product$(n/2, A_2, B_2)$ *($M_2 = A_2 \times B_2$)*
6. let $M_3 =$ int-product$(n/2, A_1 + A_2, B_1 + B_2)$ *($M_3 = (A_1 + A_2) \times (B_1 + B_2)$)*
7. return $2^n M_1 + 2^{n/2}(M_3 - M_1 - M_2) + M_2$

Correctness follows by induction from Equations (2) and (3).

Recalling that each addition or multiplication by a power of two can be done in $O(n)$ time, the running time satisfies $T(n) = \mathbf{3}\,T(n/2) + n$, which implies

$$T(n) = \sum_{i=0}^{\log_2 n} (3^i)(n/2^i) = n \sum_{i=0}^{\log_2 n} (3/2)^i = n\,\Theta((3/2)^{\log_2 n}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.58\ldots}).$$

(Above we use $x^{\log y} = y^{\log x}$ and $\log_2(3/2) = \log_2(3) - 1$.) We have the following result:

Theorem 2 (Karatsuba)**.** *There is an $O(n^{\log_2 3})$-time algorithm for multiplying n-bit integers.*

External resources

- CLRS Chapter 4.2; Dasgupta et al. Chapter 2.1; Kleinberg & Tardos Chapter 5.5
- Jeff Edmond's Chapter 1.8 (mentions even faster integer-multiplication algorithms)
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/01-recursion.pdf
- MIT lecture videos
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-11-integer-arithmetic-karatsuba-multiplication/
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-3-divide-and-conquer-strassen-fibonacci-polynomial-multiplicatio