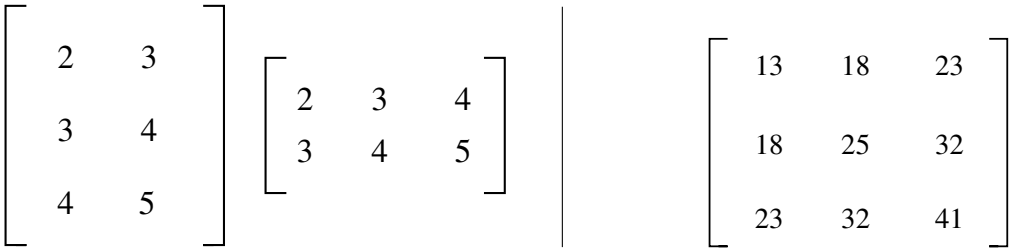

13.3 Matrix Multiplication

Input description: An $x \times y$ matrix A and a $y \times z$ matrix B.

Problem description: Compute the $x \times z$ matrix $A \times B$.

Discussion: Matrix multiplication is a fundamental problem in linear algebra. Its main significance for combinatorial algorithms is its equivalence to many other problems, including transitive closure/reduction, parsing, solving linear systems, and matrix inversion. Thus, a faster algorithm for matrix multiplication implies faster algorithms for all of these problems. Matrix multiplication arises in its own right in computing the results of such coordinate transformations as scaling, rotation, and translation for robotics and computer graphics.

The following tight algorithm computes the product of $x \times y$ matrix A and $y \times z$ matrix B runs in $O(xyz)$. Remember to first initialize $M[i,j]$ to 0 for all $1 \leq i \leq x$ and $i \leq j \leq z$:

```
for i = 1 to x do
    for j = 1 to z
        for k = 1 to y
            M[i,j] = M[i,j] + A[i,k] · A[k,j]
```

An implementation in C appears in Section 2.5.4 (page 45). This straightforward algorithm would *seem* to be tough to beat in practice. That said, observe that the three loops can be arbitrarily permuted without changing the resulting answer. Such a permutation will change the memory access patterns and thus how effectively the cache memory is used. One can expect a 10-20% variation in run time among the six possible implementations, but could not confidently predict the winner (typically ikj) without running it on your machine with your given matrices.

When multiplying bandwidth-b matrices, where all non-zero elements of A and B lie within b elements of the main diagonals, a speedup to $O(xbz)$ is possible, since zero elements cannot contribute to the product.

Asymptotically faster algorithms for matrix multiplication exist, using clever divide-and-conquer recurrences. However, these prove difficult to program, require very large matrices to beat the trivial algorithm, and are less numerically stable to boot. The most famous of these is Strassen's $O(n^{2.81})$ algorithm. Empirical results (discussed next) disagree on the exact crossover point where Strassen's algorithm beats the simple cubic algorithm, but it is in the ballpark of $n \approx 100$.

There is a better way to save computation when you are multiplying a chain of more than two matrices together. Recall that multiplying an $x \times y$ matrix by a $y \times z$ matrix creates an $x \times z$ matrix. Thus, multiplying a chain of matrices from left to right might create large intermediate matrices, each taking a lot of time to compute. Matrix multiplication is not commutative, but it is associative, so we can parenthesize the chain in whatever manner we deem best without changing the final product. A standard dynamic programming algorithm can be used to construct the optimal parenthesization. Whether it pays to do this optimization will depend upon whether your matrix dimensions are sufficiently irregular and your chain multiplied often enough to justify it. Note that we are optimizing over the sizes of the dimensions in the chain, not the actual matrices themselves. No such optimization is possible if all your matrices are the same dimensions.

Matrix multiplication has a particularly interesting interpretation in counting the number of paths between two vertices in a graph. Let A be the adjacency matrix of a graph G, meaning $A[i,j] = 1$ if there is an edge between i and j. Otherwise, $A[i,j] = 0$. Now consider the square of this matrix, $A^2 = A \times A$. If $A^2[i,j] \geq 1$. This means that there must be a vertex k such that $A[i,k] = A[k,j] = 1$, so i to k to j is a path of length 2 in G. More generally, $A^k[i,j]$ counts the number of paths of length exactly k between i and j. This count includes nonsimple paths, where vertices are repeated, such as i to k to i to j.

Implementations: D'Alberto and Nicolau [DN07] have engineered a very efficient matrix multiplication code, which switches from Strassen's to the cubic algorithm at the optimal point. It is available at *http://www.ics.uci.edu/~fastmm/*. Earlier experiments put the crossover point where Strassen's algorithm beats the cubic algorithm at about $n = 128$ [BLS91, CR76].

Thus, an $O(n^3)$ algorithm will likely be your best bet unless your matrices are very large. The linear algebra library of choice is LAPACK, a descendant of LINPACK [DMBS79], which includes several routines for matrix multiplication. These Fortran codes are available from Netlib as discussed in Section 19.1.5 (page 659).

Algorithm 601 [McN83] of the Collected Algorithms of the ACM is a sparse matrix package written in Fortran that includes routines to multiply any combination of sparse and dense matrices. See Section 19.1.5 (page 659) for details.

Notes: Winograd's algorithm for fast matrix multiplication reduces the number of multiplications by a factor of two over the straightforward algorithm. It is implementable, although the additional bookkeeping required makes it doubtful whether it is a win. Expositions on Winograd's algorithm [Win68] include [CLRS01, Man89, Win80].

In my opinion, the history of theoretical algorithm design began when Strassen [Str69] published his $O(n^{2.81})$-time matrix multiplication algorithm. For the first time, improving an algorithm in the asymptotic sense became a respected goal in its own right. Progressive improvements to Strassen's algorithm have gotten progressively less practical. The current best result for matrix multiplication is Coppersmith and Winograd's [CW90] $O(n^{2.376})$ algorithm, while the conjecture is that $\Theta(n^2)$ suffices. See [CKSU05] for an alternate approach that recently yielded an $O(n^{2.41})$ algorithm.

Engineering efficient matrix multiplication algorithms requires careful management of cache memory. See [BDN01, HUW02] for studies on these issues.

The interest in the squares of graphs goes beyond counting paths. Fleischner [Fle74] proved that the square of any biconnected graph has a Hamiltonian cycle. See [LS95] for results on finding the square roots of graphs—i.e. , finding A given A^2.

The problem of Boolean matrix multiplication can be reduced to that of general matrix multiplication [CLRS01]. The four-Russians algorithm for Boolean matrix multiplication [ADKF70] uses preprocessing to construct all subsets of $\lg n$ rows for fast retrieval in performing the actual multiplication, yielding a complexity of $O(n^3/\lg n)$. Additional preprocessing can improve this to $O(n^3/\lg^2 n)$ [Ryt85]. An exposition on the four-Russians algorithm, including this speedup, appears in [Man89].

Good expositions of the matrix-chain algorithm include [BvG99, CLRS01], where it is given as a standard textbook example of dynamic programming.

Related Problems: Solving linear equations (see page 395), shortest path (see page 489).