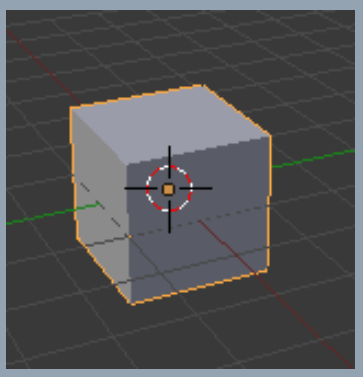
CoE 164

Computing Platforms

Software Exercise 03

Academic Period: 2nd Semester AY 2021-2022
Workload: 6 hours
Synopsis: A Faster Matrix Multiplication?
Submission Platform: UVLe Submission Bin

Introduction



Working with matrices has been a boon for a lot of fields, including computer graphics. 3D models nowadays can have thousands to millions of vertices, and transforming them (i.e. scaling, rotating, etc.) can take a long time. This is because transformation requires multiplication between two matrices.

The usual way of multiplying matrices is by iterating through each element of the resulting matrix and dot multiplying the corresponding row and column in the first and second operand matrices, respectively, using the indices of this element. With a time complexity of $O(n^3)$ and relatively easy implementation, this general matrix-matrix multiplication (GEMM) algorithm seems in hindsight to be the fastest one available. However, a mathematician named Volker Strassen has pushed study into overcoming that barrier. He claimed that, for a multiplication between two 2×2 matrices with originally eight (8) multiplications, we can forego one so that we only need to do seven (7).

Given two 2×2 square (possibly block) matrices A and B, we can multiply these matrices using the ordinary method as shown.

$$AB = C$$

$$\mathbf{AB} = \begin{bmatrix} \overrightarrow{a_{1x}} \cdot \overrightarrow{b_{x1}} & \overrightarrow{a_{1x}} \cdot \overrightarrow{b_{x2}} \\ \overrightarrow{a_{2x}} \cdot \overrightarrow{b_{x1}} & \overrightarrow{a_{2x}} \cdot \overrightarrow{b_{x2}} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Using the Strassen algorithm, we need to first compute for seven new matrices M_x as shown below.

$$M_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$

$$M_2 = (a_{21} + a_{22})b_{11}$$
$$M_3 = a_{11}(b_{12} - b_{22})$$
$$M_4 = a_{22}(b_{21} - b_{11})$$
$$M_5 = (a_{11} + a_{12})b_{22}$$
$$M_6 = (a_{21} - a_{11})(b_{11} + b_{12})$$
$$M_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

These matrices will form the solution matrix as shown below:

$$\mathbf{AB} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

As observed, the computation for the M matrices need only seven (7) multiplications while the rest of the computations are element-wise additions. With a time complexity of $\approx O(n^{2.87})$, and using up a significant amount of memory than GEMM, such algorithms are usually used only for large matrices. Also, compared to the common approach, the Strassen algorithm is recursive, meaning that we can split a matrix into four roughly equal parts to generate a 2×2 block matrix, generate the seven intermediate matrices by using the algorithm again when multiplication is needed, and combine the matrices to form the answer block matrix. Division of such matrices are easy if they are square and their dimensions are a multiple of a power of two. However, in practice, the matrices are padded with zeroed-out columns and rows such that their dimensions are even before they are split, and the paddings are discarded on merging.

You were tasked in your undergraduate research to write algorithms for 3D graphics transformation. After reading a lot about the Strassen algorithm, you came across a paper trying to dispel some of the “myths” associated with the algorithm. They have demonstrated that the Strassen algorithm can be re-implemented to use around the same memory space as that of the usual GEMM, and that it can be used optimally on smaller matrices. They have discussed that applying the Strassen algorithm by dividing the matrix at most twice, and performing GEMM on the resulting split matrices, performs slightly faster than using GEMM alone, especially on multi-core processors.

In addition, this paper has outlined a specific chain of steps to perform the Strassen algorithm to reduce the memory footprint. The steps for one recursion *in order* are as follows. Note that this arrangement still contains the classic seven required multiplications with the addition of two temporary buffers X and Y and usage of the answer matrix C as an in-place buffer.

1. $X = A_{11} - A_{21}$ 2. $Y = B_{22} - B_{12}$ 3. $C_{21} = XY$ 4. $X = A_{21} + A_{22}$ 5. $Y = B_{12} - B_{11}$ 6. $C_{22} = XY$ 7. $X = X - A_{11}$ 8. $Y = B_{22} - Y$ 9. $C_{12} = XY$ 10. $X = A_{12} - X$ 11. $C_{11} = XB_{22}$	12. $X = A_{11}B_{11}$ 13. $C_{12} = X + C_{12}$ 14. $C_{21} = C_{12} + C_{21}$ 15. $C_{12} = C_{12} + C_{22}$ 16. $C_{22} = C_{21} + C_{22}$ 17. $C_{12} = C_{12} + C_{11}$ 18. $Y = Y - B_{21}$ 19. $C_{11} = A_{22}Y$ 20. $C_{21} = C_{21} - C_{11}$ 21. $C_{11} = A_{12}B_{21}$ 22. $C_{11} = X + C_{11}$

With this observation, you are now aiming to implement a variation of the Strassen algorithm - one that uses the Strassen algorithm to perform multiplication in at most two recursions, and performing GEMM on the remaining recursions. Note than when the two matrices have dimensions 1×1, the result of their multiplication is trivial while two matrices that have dimensions 2×2 can be processed with only one recursion of the Strassen algorithm.

Input

The input to the program starts with a number T indicating the number of test cases. T test cases follow, with each test case starting with a line containing four integers - A_{row}, A_{col}, B_{row}, and B_{col}, indicating the dimensions of matrices A (of dimension $A_{row} \times A_{col}$) and B (of dimension $B_{row} \times B_{col}$), respectively. The next A_{row} lines will contain matrix A, with each line having A_{col} integers each. Then, the next B_{row} lines will contain matrix B, with each line having B_{col} integers each.

Output

The output should consist of T blocks corresponding to the T test cases. Each test case t_i should start with a line containing the string `Case #`$t_i + 1$`:`, with t_i starting at 0. Then, the next line contains two integers C_{row}, and C_{col} , denoting the dimensions of the answer matrix C (of dimension $C_{row} \times C_{col}$). Then, the next C_{row} lines contain matrix C, each having C_{col} integers each.

Example

Input

2
2 2 2 2
1 -1
-1 1
1 2
3 4
4 4 4 4
3 4 5 4
5 6 3 -1
2 3 4 -6
-4 3 4 -6
2 -5 9 0
0 0 1 2
3 4 -5 2
8 9 -1 3

Output

Case #1:
2 2
-2 -2
2 2
Case #2:
4 4
53 41 2 30
11 -22 37 15
-32 -48 7 -4
-44 -18 -47 -4

Additional Description/Requirements

Your program can only support the following limits. Note that this is a special problem because you can create your program to pass either the normal inputs only or both the normal and hard inputs.

$1 \leq T \leq 10$
$A_{col} = B_{row}$
$A_{ij}, B_{ij} \in Z$
$-100 \leq A_{ij}, B_{ij} \leq 100$

Normal Input

$A_{row} = A_{col} = B_{row} = B_{col}$
$A_{row}, A_{col}, B_{row}, B_{col} \in \{2^k \mid 2 \leq k \leq 7\}$

Hard Input

$2 \leq A_{row}, A_{col}, B_{row}, B_{col} \leq 500$

You can assume that all of the inputs are well-formed and within the above constraints, and you are not required to handle any errors arising from them. After all, your thesis adviser is waiting for your progress report!

Since you are trying to replicate the results of the paper to the best of your abilities, your program should **not** use any library that implements some sort of matrix-matrix multiplication. Additionally, your program should also **not** use any libraries that need to be downloaded off the internet (e.g. libraries that have to be downloaded from pip (for Python) or npm (for Javascript) are prohibited).

Grading Rubric

30% Strassen algorithm (recursive limit and split)
40% Strassen algorithm (order of operations and result)
20% Normal GEMM (after two-level Strassen algorithm)
10% Program passes on normal input
10% Program passes on hard input