Programming Project 2

EE312 Fall 2013

Due Sept 18, 2013

No late submissions accepted!

FIVE POINTS

General: The purpose of this project is to get some experience with arrays, pointers and memory management. Mastery of these concepts is critical to C programming. Unless you really know what you're doing with pointers and memory allocation, you are a danger to society (well, figuratively speaking, we hope). When I wrote this assignment, I presumed that you understood the mathematical construct of a Matrix, and the calculation necessary to multiply two matrices together.

Your Mission: Edit the file "Project2.cpp" and implement the functions multiplyMatrices, allocateSpace and multiplyMatrixChain.

Stage 1, Matrix Multiplication: Go to main.cpp and comment out the calls inside main to testStage2 and testStage3. You only want to run the Stage 1 test at this time. All of the Stage2 code inside Project2.cpp should already be commented out (it's inside an **#ifdef** expression, more on this later). If you're getting error messages referring to program statements inside *allocateSpace* or *multiplyMatrixChain*, then either go ask for help or re-download the project. You are now ready to write the *multiplyMatrices* function.

Recall the mathematical definition of a matrix product. Given an M×N matrix A (M rows and N columns), and an N×K matrix B, calculate the M×K result matrix C as follows:

Each element $C_{ij} = A_{i0}B_{0j} + A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{i(n-1)}B_{(n-1)j}$

Every element of C must be computed this way. So, we'll need two nested **while** loops, one for i (which goes from 0 to M, the number of rows in A), and one loop for j, (which goes from 0 to K the number of columns in B). Nested at the innermost level will be yet another **while** loop (I call mine the "k-loop") which goes from 0 to N and calculates the sum for each C_{ij}.

Your function should have these three loops, one nested inside the other. You must, however, explicitly code the function to use row-major ordering for the matrix storage. That means that A_{ij} is stored in the location *a*[*i* * *a_cols* + *j*] where *a_cols* is the variable holding the number of columns in A (N in the discussion above). The matrices B and C are similarly stored in the arrays *b*[] and *c*[] respectively. For your convenience, the code you are given for *multiplyMatrices* defines the variables *a_rows*, *a_cols*, *b_rows*, *b_cols*, *c_rows* and *c_cols* (well, some are parameters, others are defined as local variables). **You may not need to use all these variables. If you decide not to use them, please delete the variable definitions.**

Stage 2, Matrix Chains: I hope Stage 1 went OK, ‘cause Stage 2 is essentially impossible without a correct solution to Stage 1. If you’ve finished Stage 1, you’ll need to activate the test program for Stage 2. Go to main.c and look for the *main*() routine (at the end of the file). Remove the comments before the *testStage2*() (and when you’re ready, *testStage3*() too).

For Stage 2, you are to implement a function that multiplies several matrices together. You should use your *multiplyMatrices* function from Stage 1 as a subroutine. However, whereas Stage 1 would multiply only two matrices, in Stage 2 you’ll compute the matrix product of N (an arbitrary number) of matrices! The input to your function will include an array called “*matrix_list*”. This array is literally implemented as an array of pointers. Each pointer in the array will point to the beginning of some matrix. All of the matrices are potentially different sizes. The i^{th} matrix (starting with i = 0) has rows[i] rows in it, and cols[i] columns in it. For example, if rows[0] were 5 and cols[0] were 3 then the first matrix is 5×3. Your function should multiply *matrix_list*[0] by *matrix_list*[1] (these are both matrices, so use the *mutiplyMatrices* function to perform the multiplication). The result of the first matrix multiplication should be multiplied by *matrix_list*[2], the result of that should be multiplied by *matrix_list*[3] and so on. The final result should be placed in the array *out*.

Accessing the matrices in the list: Each variable in the *matrix_list* array is a pointer. It points at the first variable in an array. The variables in that array are the elements from a matrix. My advice is that you declare three variables (each one should be a pointer to double, i.e., **double***), *a_mat, b_mat* and *c_mat*. To multiply the first two matrices, set *a_mat* equal to *matrix_list*[0] and set *b_mat* to *matrix_list*[1]. The variable *c_mat* will need to be set equal to the starting address of an array that you’ll allocate – more on this later. Once *c_mat* is assigned an address, you can invoke *multiplyMatrices* with *a_mat*, *b_mat* and *c_mat* as the arguments for the *a*, *b*, and *c* parameters respectively. Be sure that you get the right values for the number of rows and columns for each matrix.

Allocating Temporary Space: Unfortunately, you can’t store the result of multiplying *a_mat* and *b_mat* in the *out* matrix. The *out* matrix might be the wrong size! For example, if we were to multiply three matrices, say *matrix*[0] is 3×2, *matrix*[1] is 2×4 and *matrix*[2] is 4×3. The *out* matrix will be 3×3. However, the result of multiplying a 3×2 matrix by a 2×4 matrix is a 3×4 matrix, too big to fit inside a 3×3. There’s no way around this problem, except to allocate a brand new array to hold all these “intermediate” results. In this example, you’ll need to allocate an array large enough to hold 12 variables of type double. You should use the function *allocateSpace* for this purpose.

The *allocateSpace* function uses the memory inside the array *memory_pool* to satisfy requests for memory. The first time *allocateSpace* is called, it will return a pointer to the first element in *memory_pool*. It will also update the variable *top_of_pool*, increasing the top by the number of elements used by the first request. For example, if the first time you call *allocateSpace*, you ask for 10 elements, then *top_of_pool* will be set to 10. The next time you call *allocateSpace* asking for memory, you’ll receive a pointer to element 10 from *memory_pool*. This is the first element in the array not already allocated by a previous call to *allocateSpace*.

Memory will be doled out in this fashion for each array request. We keep slicing off more elements from the bottom of the pool, bumping the *top_of_pool* index up each time. Finally, when all the arrays we need have been allocated and we're done multiplying everything, we reset the *top_of_pool* to zero. This last step effectively deallocates all the memory. Next time we request memory, we'll start again at *memory_pool*[0]. There's already a statement to reset *top_of_pool* to zero at the end of *multiplyMatrixChain*. You should not need to change this line, or assign to *top_of_pool* anywhere else inside the *multiplyMatrixChain* function. Inside *allocateSpace,* you will need to increment *top_of_pool* by the correct number of elements (of course).

And that's all there is to it. Just be sure that on the last matrix multiplication you perform you put the final result into the *out* matrix (rather than putting it into a temporary array). I did this by assigning *c_mat* to the returned result from *allocateSpace* on the first n-2 iterations of the loop, and setting *c_mat* equal to *out* on the last iteration of the loop (to multiply n matrices together you must perform n-1 multiplications, hence n-1 iterations of the loop). If you don't want to be quite so tricky, go ahead and just copy all the elements out of *c_mat* into the *out* array using a big loop.

Stage 3, More Challenging Tests: If you implemented Stage 1 and Stage 2 correctly, then you're already done with Stage 3. Stage 3 just tests your program using rectangular (not square) matrices. If you have a bug that shows up in Stage 3, keep in mind that the mistake could be in your *multiplyMatrices* function (i.e., back in stage 1). Be sure you're using the rows and columns sizes correctly, and be sure you're allocating enough memory. Once you get all the bugs out of *multiplyMatrices* and *multiplyMatrixChain,* you'll pass Stage 3 with flying colors