Subject: CAAM 420/520 – Homework 1
Date Posted: January 23, 2023
Date Due: 7:00pm, January 30, 2023 via Canvas

Total number of points available: 85pts (CAAM 420), 66 to 85 pts (CAAM 520)

- **420:** All problems are required for students enrolled in CAAM 420.
- **520:** Problems marked with a * are not required for students in CAAM 520, however, they may attempt them if they wish. If these problems are attempted (i.e. included in your final report) they will be included as part of your grade; they do not count as bonus points. Note this means the total number of points each student's grade is taken out of can be different for each student enrolled in CAAM 520.

Problem 1 : Caching and Discontinuous Memory *7 pts (420)/ 5 pts (min 520)*

Consider the 3D array allocated using discontiguous memory (Method 2 in the slides) and the three different initialization routines given in Listing 1. Assume that all allocations succeeded and that `n1, n2, n3` are large enough to see caching effects.

Listing 1. Code snippet for Problem 1.

```
int p, i, j; // page, row, and column indices
int*** A = NULL;


// Allocate the pointers to each page:
A = new int**[n1];

// For each page:
for(p = 0; p < n1; p++) {
    A[p] = new int**[n2]; // Allocate the pointers to each row

    // For each row:
    for(i = 0; i < n2; i++) {
        A[p][i] = new int[n3]; // Allocate all the columns in row j
    }
}

// Initialization 1: p, i, j
for(p = 0; p < n1; p++) {
    for(i = 0; i < n2; i++) {
        for(j = 0; j < n3; j++) {
            A[p][i][j] = 1;
        }
    }
}

// Initialization 2: i, p, j
for(i = 0; i < n2; i++) {
    for(p = 0; p < n1; p++) {
        for(j = 0; j < n3; j++) {
            A[p][i][j] = 1;
        }
    }
}

// Initialization 3: p, j, i
for(p = 0; p < n1; p++) {
    for(j = 0; j < n3; j++) {
        for(i = 0; i < n2; i++) {
            A[p][i][j] = 1;
        }
    }
}
```

(a) *(2 pts)** Do you expect a difference in performance between Initializations 1 and 3? Why or why not?

(b) *(3 pts)* Do you expect a difference in performance between Initializations 1 and 2? Why or why not?

Hint: Every call to `new` allocates an array. How many arrays are being traversed without interruption?

Problem 2 Memory for Time: Memoization *16 pts (420)/ 10 pts (min 520)*

Time and memory complexity can sometimes be traded for one another. Here we consider increasing memory to improve speed.

In problems where the current solution depends on the solutions to previous iterations of the problem or subproblems *memoization* can be used to help offset the cost of computing solutions. In memoization, you save previous solutions to avoid recomputing them each time they are needed. Memoization is also called caching; it is especially advantageous when prior solutions are extremely expensive to compute.

Consider the two implementations given in Listing 2 for the Fibonacci numbers, a sequence of integers which can be defined recursively as

$$f_n = \begin{cases} 1, & n = 0, 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}.$$

Listing 2. Code snippet for Problem 2.

```
// No memoization
int fibonacci(int n) {
    // Base cases:
    if( n == 0 || n == 1)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}


// With memoization
int fibonacci_mem(int n) {
    // Static storage for previous solutions
    static int f[FIBONACCI_CACHE_SIZE] = {0};

    // Base cases:
    if( n == 0 || n == 1)
        return 1;

    // Check for memoized/cached solutions
    if( f[n-1] == 0 )
        f[n-1] = fibonacci_mem(n-1);

    if( f[n-2] == 0 )
        f[n-2] = fibonacci_mem(n-2);

    return f[n-1] + f[n-2];
}
```

(a) *(3 pts)** In terms of n, how many function calls are require to calculate f_n using the function `fibonacci`?

(b) *(3 pts)** In terms of n, what is the most and fewest number of function calls required to calculate f_n using the function `fibonacci_mem`?

(c) *(7 pts)* Suppose `fibonacci_mem` is called m times by the user using inputs $n_1, n_2, ..., n_m <$ `FIBONACCI_CACHE_SIZE`. Note the n_k's are not ordered (i.e. n_2 could be greater than, less than, or even equal to n_1). At most how many *recursive* calls to `fibonacci_mem` will be invoked?

(d) *(3 pts)* If you knew the requests for Fibonacci numbers were going to be in order (i.e., $n = 0, 1, 2, 3, ..., m$), would you still need to use an entire array for previous solutions?

Problem 3 The Modulus Operator and mD Indexing *10 pts (420 and 520)*

The modulus operator in C++, %, is used to compute the remainder of an integer number with respect to another integer. While it is not often seen outright outside of combinatorics, it is a very useful operator when dealing with mD arrays that have been dynamically allocated as 1D arrays (Method 1 in the slides).

Suppose you are going to allocate a mD array of size $n_1 \times n_2 \times ... \times n_m$ using a single 1D array. The formula for calculating the flat index for the 1D array can be written as the sum of the contributions of each of the Cartesian indices, I_k as:

$$i_{flat} = I_1 + I_2 + ... + I_m, \quad I_k = s_k i_k$$

where s_k is the stride of each dimension. Assume the indices are ordered in terms of the their stride; that is i_1 is the fastest index, then i_2, etc.

(a) *(2 pts)* What is formula for the k^{th} stride, s_k, in terms of $n_1, n_2, ..., n_m$?

(b) *(3 pts)* What are which strides are divisible by n_k? What is $I_j \% n_k$ for each of these?

(c) *(5 pts)* Write out the code snippet for calculating $i_1, i_2, ..., i_m$ given $i_{flat}, n_1, n_2, ..., n_m$ using the modulus operator.

Problem 4 Indexing Design *25 pts (420)/ 22 pts (min 520)*

Associated Files: `matrix_mult.h`

Name your files: `<netID>_matrix_mult.cpp`

WARNING: If you do not name your file using this pattern your code will not compile during grading and be subject to the appropriate penalties.

Because you have to handle indexing yourself when allocating mD arrays as 1D arrays, you can choose which index to make the fastest dimension. This choice is typically made based off of the routines you are going to do apply to your data. Additionally, you can sometimes alter your routines to best exploit the fastest dimension of your data (and thus caching). Here we will consider this for matrices (2D arrays) and matrix-vector multiplication.

In 2D arrays, row-major and column-major ordering refer to different ways of walking through the array; walking all the way through a row before going to the next row (row-major) or walking all the way through a column before going to the next column (column-major). Figure 1 shows an example of both orderings.

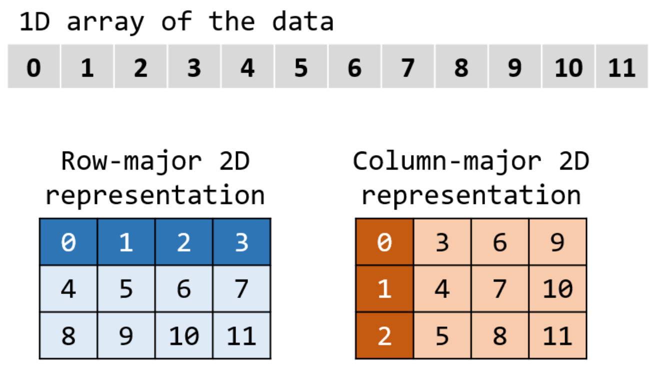

Figure 1. Example of mapping 12 values to a 3x4 matrix using row-major and column-major ordering.

(a) *(3 pts)* Implement the function `cartesian2flat_row_major` such that flat indices are computed in row-major ordering.

(b) *(3 pts)* Implement the function `cartesian2flat_col_major` such that flat indices are computed in column-major ordering.

(c) *(8 pts)* Implement the function `matrix_vec_mult_row_major` that performs matrix-vector multiplication using a row-major traversal of the matrix.

(d) *(8 pts)* Implement the function `matrix_vec_mult_col_major` that performs matrix-vector multiplication using a column-major traversal of the matrix.

(e) *(3 pts)** Which matrix-vector multiplication routine is better for a column-major indexed matrix?

Problem 5 Time for Memory: Sparse Data Structures *27 pts (420)/ 19 pts (min 520)*

Associated Files: `matrix_mult.h`

Name your file: `<netID>_matrix_mult.cpp` (use the same file as Problem 4)

As discussed in Problem 2, time and memory can sometimes be interchanged by changing our routines. In Problem 2, we used more memory (if we ignore stack allocations) to improve time costs by exploiting the problem's reliance on previous solutions. Here, we will trade longer time for some operations (but perhaps not others) for using less memory by exploiting sparsity.

Many real-world applications require matrix-vector algebra on incredibly large matrices. It is common for these matrices to be so large that they cannot actually fit into the computer memory regardless of allocation method. However, in such applications these matrices are often *sparse*, that is, the vast majority of their entries are zeros. To work with such matrices, scientific programmers define data structures that preserve the sparsity of their problems (and thus save memory), but these data structures may change the time complexity of operations that have to be performed on them. The example we will consider is the CSC (Compressed Sparse Column) matrix representation, which can be read about here.

For this problem, we will only consider square, $n \times n$ matrices. For answers on complexity, use the variables n, N, and N_{nz} where $N = n^2$ and N_{nz} = the number of non-zero entries in the matrix.

(a) *(2 pts)** What is the $\mathcal{O}$ and Ω complexity of accessing a single element of a dense matrix (i.e. a matrix represented as a regular 2D array)?

(b) *(3 pts)** What is the $\mathcal{O}$ and Ω complexity of accessing a single element of a CSC matrix?

(c) *(3 pts)** What is the complexity of matrix-vector multiplication with a dense matrix?

(d) *(4 pts)* What is the complexity of matrix-vector multiplication with a sparse matrix?

(e) *(5 pts)* What is the memory complexity of representing of a dense matrix and a CSC matrix (do not drop any coefficients in the final statement)? At what sparsity level (N_{nz}/N) do the memory requirements of the two become comparable? Assume n is large enough that $1/n$ is negligible.

(f) *(10 pts)* Implement the function `matrix_vec_mult_sparse` using the CSC data structure.

Hint: this function should be *similar to*, but not 100% the same as, the function you wrote for Problem 4.d. You must exploit sparsity to get full credit here.