To: Christina Taylor
From: Eli Case
Date: January, 25, 2023
Subject: CAAM 420/520 – Homework 1

Problem 1 First, note that the array pointer, A, has a storage format of page, row, then column. For example, $A[p]$ points to an array of row pointers on the p^{th} page, $A[p][i]$ points to an array of the column entries on the p^{th} page in the i^{th} row. Lastly, $A[p][i][j]$ is simply the p, i, j entry of the array. With this storage format, the columns have the shortest stride, the rows have the second shortest stride, and the pages have the longest stride. Writing the for loop to have the inner-most loop align with the shortest stride, the second middle loop align with the second shortest stride, and the outer-most loop align with the longest stride will provide the fastest array traversal, as this will give the fewest amount of cache misses.

(a) A difference in performance would be expected between initializations 1 and 3 due to caching effects. Initialization 1 will provide a faster traversal over the array than initialization 3. Initialization 1 accesses elements in the array by page, row, then column, which is the optimal storage format. This storage format is optimal because the inner-most loop increments over contiguous memory, meaning there is the least amount of cache misses with initialization 1. Conversely, in initialization 3, the inner-most loop iterates over discontiguous memory, creating cache misses each inner-most loop iteration, as the rows are iterated over after the columns.

(b) A difference in performance would also be expected between initializations 1 and 2, as in initialization 2, the first loop iterates over discontiguous memory, creating cache misses as the rows are iterated over before the pages. As before, initialization 1 has the fewest amount of cache misses, meaning initialization 1 will provide a faster traversal over the array than initialization 2.
Corrected Answer: A difference in performance is not expected between initializaitons 1 and 2 because the cache misses only occur on the inner loop.

Problem 2 (a) Let the number of function calls required to calculate $f_n(n)$ with `fibonnaci` as a function of n be given by the function `num_calls(n)`. `num_calls(n)` is given by the following recursive formula.

$$\texttt{num\_calls(n)} = \begin{cases} 1 & n = 0, 1 \\ 1 + \texttt{num\_calls(n-1)} + \texttt{num\_calls(n-2)} & n \geq 2 \end{cases} \tag{1}$$

From inspection of equation 1, we can rewrite `num_calls(n)` as $\texttt{num\_calls(n)} = 2f_n(n) - 1$. Thus, we have that the number of calls to required valuate $f_n(n)$ using the function `fibonacci` is as shown below.

$$\texttt{num\_calls(n)} = 2f_n(n) - 1 \tag{2}$$

(b) Let the number of function calls required to calculate $f_n(n)$ with `fibonacci_mem` as a function of n be given by the function `num_calls(n)`. The maximum and

minimum `num_calls(n)` is given by the formula below.

$$\textbf{max}(\texttt{num\_calls(n)}) = n \tag{3}$$

$$\textbf{min}(\texttt{num\_calls(n)}) = 1 \tag{4}$$

We note that the maximum, or worst case, occurs when `fibonacci_mem` has not previously been called, and the minimum, or best case, occurs when `fibonacci_mem(n-1)` has previously been called.

(c) If we call `fibonacci_mem` m times with inputs $n_1, n_2, ..., n_m$, the largest possible number of recursive calls will be the maximum number in the list $n_1, n_2, ..., n_m$. Thus, if we let the maximum n in the given list be $n^* = \textbf{max}(\{n_i\}_{i=1}^m)$, we can write the largest possible number of recursive calls to `fibonnaci_mem` , given by `num_calls_recursive` , as shown below.

$$\texttt{num\_calls\_recursive} = n^* \tag{5}$$

(d) If it is known the requests for Fibonacci numbers are in sequence, an entire array for previous solutions would be unnecessary. A array of only size 2 would be required to store the previous two entries of the Fibonacci sequence for this case.

Problem 3 (a) We can write the k^{th} stride, s_k, in terms of $n_1, n_2, ..., n_m$ as follows.

$$s_k = \begin{cases} 1, & \text{k} = 1 \\ \prod_{i=1}^{k-1} n_i, & \text{k} \geq 2 \end{cases} \tag{6}$$

(b) From equation 6 above, it can be seen that the strides, s_j, with index $j \geq k+1$ are divisible n_k. Thus, it follows that $I_j \% n_k$ is as shown below.

$$I_j \% n_k = 0, \quad j > k+1 \tag{7}$$

(c)

```
int* get_cartesian_index(int i_flat, int* n, int n_size){
    int* i_cartesian = NULL;
    i_cartesian = new int[n_size];
    int prev_entry = 0;

    i_cartesian[0] = i_flat % n[0]; // set first index
    prev_entry = i_flat - i_cartesian[0]; // initialize mod value

    for(int i=1; i < n_size; i++){
        i_cartesian[i] = prev_entry % n[i]; // set remaining indices
        prev_entry = i_flat - prev_entry; // update mod value
    }

    return i_cartesian;
}
```

Problem 4 (e) For a column-major indexed matrix, the matrix-vector multiplication routine `matrix_vec_mult_col_major` is better. This matrix multiplication routine is better for column-major indexed matrices because each inner loop traversal over a given column will give more cache hits than if the inner-most loop traversed a row.

Problem 5 (a) In a dense matrix represented as a regular 2D array, we have the following time complexities to access a single element: $O(1)$ and $\Omega(1)$, or simply $\Theta(1)$.

(b) In a matrix represented in CSC format, we have the following time complexity to access a single element: $O(n)$ and $\Omega(1)$. The worst case time complexity is $O(n)$ for CSC format because one must iterate over a list of nonzero elements in a given column to access the element at the desired row number.

(c) For a dense matrix, the time complexity of matrix-vector multiplication is $\Theta(N)$.

(d) For a sparse matrix represented in CSC format, for example, the time complexity of matrix-vector multiplication is $O(N)$ and $\Omega(n)$. The best-case time complexity is $\Omega(n)$ for the case where there is only on non-zero entry per column.
Corrected Answer: $O(N_{nz} + n)$ to initialize the result vector and because we iterate over N_{nz} in the CSC matrix (Recall N_{nz} was a given variable).

(e) For a dense matrix, the memory complexity to represent the matrix is $\Theta(N)$. For a CSC Matrix, the memory complexity to represent the matrix is $\Theta(2N_{nz} + n)$. In order to determine when the memory requirements of the two storage formats become comparable, we can set the Big Theta memory complexities equal.

$$\begin{aligned} N &= 2N_{nz} + n \\ \implies 1 &= 2\frac{N_{nz}}{N} + \frac{n}{N} \\ \implies 1 &= 2\frac{N_{nz}}{N} + \frac{1}{n} \\ \implies 1 &\approx 2\frac{N_{nz}}{N} \\ \implies \frac{1}{2} &\approx \frac{N_{nz}}{N} \end{aligned}$$

where it is assumed that n is large enough so that $\frac{1}{n}$ is negligible. Thus, we have that the memory requirements for a dense matrix and CSC matrix become comparable when the sparsity level, $\frac{N_{nz}}{N}$ is as follows.

$$\frac{N_{nz}}{N} \geq \frac{1}{2} \tag{8}$$