To: Christina Taylor
From: Anwar Khaddaj
Date: January 31, 2023
Subject: CAAM 420/520 – Homework 1
I collaborated with Arshia Singhal and Jonathan Cangelosi on this homework.

Problem 1 (a) Yes, there will be a difference in performance between initializations 1 and 3. Initialization 1 doesn't have any interruption while traversing the arrays in a row-major fashion while initialization 3 will have some interruptions as it's traversing the inner arrays in a column-major fashion which is different that how it's actually stored.

(b) No, there will be no difference in performance between initialization 1 and 2 since the final loop responsible for accessing the data is the same. Even if we switch the two pointers, the performance will be the same especially that $n3$ are large enough to see caching effects so the cache will reset while traversing the final loop leading to the same cache misses in both initializations.

Problem 2 (a) Let $F(n)$ denote the number of function calls to calculate $f_n, F(n-1)$ the number of function calls to calculate f_{n-1}, and $F(n-2)$ the number of function calls to calculate f_{n-2}. Thus, $F(n) = 1 + F(n-1) + F(n-2)$. and $f_n = f_{n-1} + f_{n-2}$.
We claim that $F(n) = 2f_n - 1$, and we proceed to prove it by induction.
Base case: $n = 0$: $F(0) = 1 = 2f_0 - 1$.
Inductive case: Suppose it is true for all n. Show it is true for all $n+1$. Thus, $F(n) = 2f_n - 1$. Show $F(n+1) = 2f_{n+1} - 1$.
$F(n+1) = 1+F(n)+F(n-1) = 1+2f_n-1+2f_{n-1}-1 = 2(f_n+f_{n-1})-1 = 2f_{n+1}-1$.
Therefore, it is true for all n.
Now, to write it in terms of n, we have that $f_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$.
Plugging it into the formula that we just proved by induction, we get the desired result:

$$F(n) = \frac{2}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{2}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n - 1.$$

(b) The fewest number of function calls is 1 call (for example for $f_n = 1$ or 0). The most number of function calls is $n+1$ calls. Since memoization is applied, the values computed are stored in a static array. Thus, to get f_n, 1 call is enough to store all values all the way from 0 to n, i.e. $n+1$ calls.

(c) Since the $n_k's$ are not ordered, we should take into account the number of occurrences where $n_{i+1} > n_i + 1 \ \forall i \in 1, ..., m$, call this variable p. Now, since we are dealing with recursive calls (i.e. calls after the initial call) and we are using a memoization approach, we get at most $\max(n_1, n_2, ..., n_m) - p$ recursive calls.

(d) No, an array of size 2 is enough as the requests are in order, so we only need to store the previous 2 Fibonacci numbers. Every time we need to compute a certain Fibonacci number, we only need to use the 2 previously computed values stored in the array and then we replace the oldest number by the newest and possibly shift/reverse the elements of the array.

Problem 3 (a) Knowing that i_1 is the fastest index, the corresponding s_1 should be the smallest. Thus,

$$s_k = \begin{cases} 1 & \text{for } k = 1, \\ \prod_{i=1}^{k-1} n_i & \text{for } 1 < k \leq m \end{cases}$$

(b) It is obvious from the formula given in part a that $s_{k+1}, ..., s_m$ are divisible by n_k. Knowing that $I_j = s_j * i_j$ and that the above strides are divisible by n_k, $I_j \% n_k = 0$ for all strides $s_{k+1}, ..., s_m$.

(c)
```cpp
void indexer(int* n, int iflat, int* result,int size_n) {
    //n contains n1,..,nm and
    //result array returns the indices i1,i2,..im
    //Calculating the strides
    int* stride=NULL;
    stride = new int[size_n];
    stride[0]=1;
    for (int i=1;i<size_n;i++){
        stride[i]=stride[i-1]*n[size_n-i];
    }
    //Calculating the indices
    int remainder=0;
    for (int i=size_n-1;i>=0;i--){
        result[size_n-i-1]=(int)iflat/stride[i];
        remainder=iflat%stride[i];
        iflat=remainder;
    }
}
```

Problem 4 (a) Submitted as ask15_matrix_mult.cpp

(b) Submitted as ask15_matrix_mult.cpp

(c) Submitted as ask15_matrix_mult.cpp

(d) Submitted as ask15_matrix_mult.cpp

(e) For a column-major indexed matrix, it is better to implement the matrix_vec_mult_col_major routine as it uses a column=major traversal of the matrix suitable to how this matrix is indexed.

Problem 5 (a) $\mathcal{O}(1)$ and $\Omega(1)$ as it is a simple arithmetic computation to get the element from the index.

(b) $\mathcal{O}(n)$ (worst case scenario: you have to traverse the whole row_ind array) and $\Omega(1)$. However, if the row index array is sorted per column, we can apply binary search and get $\mathcal{O}(\log n)$.

(c) $\mathcal{O}(N)$ as you need to traverse all the values in the dense matrix which sum up to n^2.

(d) $\mathcal{O}(N_{nz})$ as you need to traverse all the values in the sparse matrix which are the non-zero values N_{nz}.

(e) Memory complexity of representing a dense matrix: $\mathcal{O}(N)$.
Memory complexity of representing a CSC matrix:
$(n + 1) + N_{nz} + N_{nz} = 2N_{nz} + n + 1$.

For both memory requirements to be comparable,

$$\frac{2N_{nz} + n + 1}{N} = \frac{N}{N}$$

so

$$\frac{2N_{nz}}{N} + \frac{1}{n} + \frac{1}{n^2} = 1.$$

Knowing that n is large enough, $\frac{1}{n}$ and $\frac{1}{n^2}$ is negligible. Therefore, the sparsity level is $\frac{N_{nz}}{N} \approx \frac{1}{2}$ (it approaches 1/2 as $n \to \infty$) .

(f) Submitted as ask15_matrix_mult.cpp