COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 5

Dr. Joe Anderson

Due: 3 October 2019

1 Objectives

In this lab you will focus on the following objectives:

1. Design and implement a new, mathematically motivated, data structure.
2. Develop familiarity with matrices and matrix operations.
3. Study the algorithmic problems of matrix algebra both in theory and in practice.

2 Preliminaries

Recall that in `c++`, one may declare multi-dimensional arrays. For example: `int arr[3][5]` declares an array of three five-element integer arrays, containing fifteen total integers. One way to visualize this is as follows:

```
[
  [1, 2, 3, 4, 5],  // arr[0]
  [6, 7, 8, 9, 10], // arr[1]
  [11,12,13,14,15]  // arr[2]
]
```

To access the second element in the third array, one would use the syntax `arr[2][1]`.

To rectify this with the traditional mathematical definition of a matrix, we say that A is a real $n \times m$ matrix if

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,m} \\ \vdots & & \ddots & & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,m} \end{pmatrix}$$

where each $a_{i,j} \in \mathbb{R}$ for $i = 1, 2, 3, \dots, n$ and $j = 1, 2, 3, \dots, m$. In this notation, the matrix has n rows and m columns. The link in notation is that, if the matrix is stored in a `c++` program as described above (i.e. an array of arrays), accessing element $a_{i,j}$ is accomplished by the syntax $A[i][j]$; that is, each "row" can be seen as a single array, and the full matrix is an array of rows of same length.

If A and B are both of dimension $n \times m$, then the sum is the $n \times m$ matrix C with entries given by

$$C_{i,j} = a_{i,j} + b_{i,j}$$

for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, m$.

If A is an $n \times m$ matrix and B is an $m \times k$ matrix, then the product AB is matrix C, where

$$C_{i,j} = \sum_{l=1}^{m} a_{i,l} b_{l,j}$$

for $i = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, k$; so C is a new $n \times k$ matrix. Thus the (i, j) entry of C is, in another manner of speaking, the ith row of A, dot product with the jth column of B.

Your task below is to design and implement a data structure in `c++` to model a matrix of real numbers, then to define the operations of 1) addition, 2) subtraction, and 3) multiplication. You will then analyze the computational complexity of each operation for your data structure, and provide both theoretical and practical analysis.

3 Tasks

1. Put your code in a folder to be zipped and turned in at the end.
2. Design a `class` in `c++` to represent a matrix.
   (a) Put your class declarations and prototypes in file: `matrix.h` (use include guards. See appendix if you haven't used these before.)
   (b) Put your class definitions in file: `matrix.cpp`
   (c) Your class should model a single matrix of size $n \times m$, where n and m are positive integers, and the size should not be limited except by the integers/memory available on the platform.
   (d) The storage of the array elements should use dynamic memory (i.e. the heap), and hence your class should have appropriate copy constructor, destructor, and overloaded assignment operator, all of which prevent memory leaks when moving/deleting matrix data.
   (e) Because the matrices you test should be quite large, consider using `long unsigned` or `size_t` to index the arrays. This will avoid overflow errors during loops and iterative processes.
   (f) The class should have methods (and, helpfully, overloaded operators) for addition, subtraction, and multiplication of matrices.
      i. Note that matrix multiplication is not valid for matrices A and B when the number of columns of A does not match the number of rows of B. In this case, your class should gracefully abort the multiplication and throw an exception.
      ii. Similarly, the size of the matrices in each dimension must match for addition and subtraction. These should also be validated, to avoid segfaults or unexpected memory corruption.
      iii. Each of these operations should return a *new* matrix object, and not modify either operand, as in standard arithmetic with program variables (`x+y` does not inherently affect either of the values, but simply returns a third one).
3. Write a test harness (main function, in a third file) to test your matrix code with some examples. Be sure to try different edge cases, and "shapes" of matrices; for instance,
   (a) Diagonal matrices: all entries off the main diagonal ($A_{0,0}, A_{1,1}, \ldots, A_{n,n}$) are 0
   (b) Upper (or Lower) triangular matrices: entries below (or above) the main diagonal are 0.
   (c) The identity matrix: 1 on the main diagonal, 0 otherwise.
   (d) A vector (an $n \times 1$ matrix)!
4. Furthermore, include some benchmark data, timing how long it takes to multiply matrices of varying sizes and with varying magnitudes of entries.

5. Include a `Makefile` to build your code.

6. Include a `README` file to document your code, any interesting design choices you made, and answer the following questions:

   (a) Summarize your approach to the problem, and how your code addresses the abstractions needed for matrix arithmetic.

   (b) Analyze your benchmark observations: what does the timing suggest for the time complexity of addition, subtraction, and multiplication for two $n \times n$ matrices?

   (c) What is the arithmetic time complexity of your implementation of each operation, when done on two $n \times n$ matrices? Give tight asymptotic bounds.

   (d) How could the code be improved in terms of usability, efficiency, and robustness?

4 Submission

All submitted labs must compile with your provided `Makefile` and run on the COSC Linux environment.

Upload your project files to MyClasses in a single `.zip` file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.

5 Bonus

For 5 bonus points, you may implement an overloaded multiplication operator to multiply a matrix by a scalar, which has the result of multiplying each entry of the matrix by that scalar.

For 10 bonus points, implement (in a separate program, but which uses the matrix class you created) the fast Fibonacci algorithm to compute the nth Fibonacci number, and reports the time and number of arithmetic operations needed.

For 10 bonus points, implement Strassen's matrix multiplication algorithm. Do a performance comparison versus the naïve algorithm, can you find an instance where Strassen's is more practical?

A Appendix: Include Guards

To avoid re-declaration errors when compiling `c++` code for which multiple files `#include` some common library, one uses *include guards*. For the matrix class, in file `matrix.h`, your include guards should follow the template:

```
#ifndef MATRIX_H
#define MATRIX_H

class Matrix{
  // Your class delcarations and prototypes
};

#endif
```

The statements above and below the `class` are pre-compiler macros: they get "executed" not as part of your program, but by the compiler itself. In this case, if the compiler has not read this file yet, the value `MATRIX_H` has not been defined internally to the compiler yet; as a result, the first time through, the `#ifndef` will be true, and the next line will define the value `MATRIX_H`, and the rest of the file will be processed. The upshot is that if *another* file has `#include "matrix.h"`, the constant `MATRIX_H` will have been defined, and the compiler will skip over reading this file again, thus avoiding the error of defining the class `Matrix` and its members multiple times!

For further reading and links, see the course webpage.