CS 136 Fall 2008
Mark Giesbrecht

Assignment 4
Due Tuesday, November 11, 2008 at 4pm.

All coding questions are to be implemented in the C language. The prototypes of all functions which are required in the questions should be stored in an appropriately named `.h` file, which you should `#include` into your `.c` file of solutions, and into your `.c` file of drivers.

Questions with non-programming answers should be neatly written and deposited into the CS136 assignment box outside of MC4065. Please use a cover sheet. The answers do not need to be typeset.

1. Some older computer languages allow only for one-dimensional arrays, and indeed, our machine/memory model really only supports one-dimensional arrays directly. Nonetheless, we can use this to implement 2-dimensional arrays in a straightforward manner. In this question we will implement a set of functions for working with $n \times n$ matrices using one-dimensional arrays in C (you cannot use 2-dimensional C arrays!). We will use "mathematical" notation, rather than C notation for our matrices, so our indices will run from 1 to n (rather than 0 to $n-1$ as they would in C). We will also do our own memory management for matrices.

   All functions for this question should be stored in the file `matrix.c`, with headers for all functions in `matrix.h`. A file `matrix-driver.c` should contain a `main()` function with appropriate tests for all functions.

   Start by declaring two *global* variables `memory`, and `mem_next_unused` declared in the file `matrices.c` but outside the scope of any function:

   ```
   int memory[100000];
   int mem_next_unused = 1;
   ```

   A matrix will be stored as an index `a` into `memory`. At location `memory[a]` we will store the size of the matrix (say it is 50) and at locations `memory[a+1]`...`memory[a+50*50]` are stored the elements of the matrix.

   (a) Write a function `newmatrix` which consumes an `int n`, and an initial value `int init`, and returns the current value of `mem_next_unused`. The function should thus have prototype

   ```
       int newmatrix(int n, int init);
   ```

   `memory[mem_next_unused]` should be set to `n` (the dimension of the matrix), and all entries of `memory` from location `mem_next_unused+1` through `mem_next_unused+n*n` (inclusive) are set to `init`. After this, `1+n*n` is added to `mem_next_unused` (so it points to the beginning of "free" space for the next time we ask for a new matrix).

   If insufficient room is available in `memory` then 0 should be returned.

   The point is that these locations will be used to store the contents of an $n \times n$ matrix, and all entries in that matrix are set to `val`. A matrix in our representation will be represented by its starting location in `memory`.

(b) Write a function `readmat` which consumes an `int m` (representing a matrix). It then looks up the size n of the matrix (store in `memory[m]`) and reads $n \times n$ entries from standard input into locations `memory[m+1]`, ..., `memory[m+n*n]`. The standard input should simply consist of the integer data separated by spaces or newlines (exactly what `scanf` expects. The function should have prototype

```
void readmat(int m);
```

(c) Write a function `at` which consumes an `int a` (representing a matrix) and two `ints` `i` and `j`, and returns the value at the (i, j) location in the matrix. This is computed by first determining the size `n` of the matrix `a` (stored in memory). Then the first row of `a`, locations $(1, 1), \ldots, (1, n)$, are stored in `memory[a+1]`, `memory[a+2]`, ..., `memory[a+n]`. Next we store row 2 in `memory[a+n+1]`, ..., `memory[a+n+n]`, etc. Access to location i, j should take constant time, so you should figure out an easy formula for where location (i, j) is stored. If you attempt to access beyond the dimensions of the matrix, an error should be printed, and 0 returned. Your function should have prototype

```
int at(int a, int i, int j);
```

(d) Write a function `set`, which consumes an `int a` (representing a matrix), two `ints` `i` and `j` representing a position (i, j) in the matrix, and a value `int v`. The result of the function should be that location (i, j) in the matrix is set to have value v. Your function should have prototype

```
void set(int a, int i, int j, int v);
```

If (i, j) is beyond the dimensions of the matrix, an error should be printed.

(e) Write a function `print`, which consumes an `int a` (representing a matrix), and prints the matrix to standard output in a reasonable format. I.e., if the matrix has 3 rows and columns, it should be printed on 3 lines, with three `ints` per line, separated by spaces (don't worry about alignment). Your function should have prototype

```
void print(int a);
```

(f) Write a function `add`, which consumes two `ints`, `a` and `b` (representing matrices), and returns an `int c` representing a new matrix. The contents of the new matrix should be the contents of the matrix $a + b$. Your function should have prototype

```
int add(int a, int b);
```

The matrices represented by `a` and `b` should not be modified. If the dimensions of `a` and `b` are not the same, an error should be printed, and 0 returned (note that the first matrix started at location 1 in `memory`, so 0 indicates a error.

Include an analysis of the running time of your code as a function of `n` (the size of the matrices) in a comment in your code.

(g) Write a function `multiply`, which consumes two `ints`, `a` and `b` (representing matrices), and returns an `int c` representing a new matrix. The contents of the new matrix should be the contents of the matrix product $a \times b$. Recall from your linear algebra class that the (i, j) entry of $c = a \times b$ is

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Your function should have prototype

```
int multiply(int a, int b);
```

The matrices represented by `a` and `b` should not be modified. If the dimensions of `a` and `b` are not the same, an error should be printed, and 0 returned. Include an analysis of the running time of your code as a function of `n` (the size of the matrices) in a comment in your code.

(h) Write a function `transpose`, which consumes an `int a`, and *modifies* that matrix represented by `a` so that it contains its own *transpose*. Recall that the transpose of a matrix swaps rows and columns. The contents of row i column j in the original matrix is at location row j column i in the transpose. Do not create a new matrix for this question, just change the one passed as a parameter.

```
void transpose(int a);
```

Include an analysis of the running time of your code as a function of `n` (the size of the matrices) in a comment in your code.

(i) Write a function `power`, which consumes an `int a`, and an `int e` and returns a new matrix `b` such that `b` represents a^e (i.e., $b = a \times a \times \cdots \times a$, a multiplied with itself e times). Your function should have prototype

```
int power(int a, int e);
```

Your function should use as little extra space as possible. If `a` is an $n \times n$ matrix, you should be able to do this with $n^2 + 1$ space for `b` plus an extra $n^2 + 1$ space for work (use space in `memory` for this). Provide an analysis of both the time and the space required by your function in the comments of the function. If $e = 0$, you should return the $n \times n$ identity matrix.

For an bonus of up to 20% (on the assignment) implement a version of the repeated squaring algorithm described in the previous assignment for arithmetic modulo p. You should be able to get a much more efficient algorithm using this, but using slightly more space.

2. The sieve of Eratosthenes is informally described as follows. To find the prime numbers from 2 to n, write down all the numbers from 2 to n in a row. Circle 2 and cross out every second number (4, 6, etc.). Then find the next number not crossed out (it will be 3) and cross out every multiple of that number (6, 9, etc.) Repeat until all numbers are either circled or crossed out.

Write a C function with prototype

```
void sieve(int A[], int n);
```

which assumes that `A` is of size at least $n + 1$, and mutates `A` so that `A[i]` (for $2 \le i \le n$) has value 1 if `i` is prime and 0 otherwise.

Include an analysis of the running time of your code as a function of `n` in a comment. You may find the following inequalities useful. Let k be the largest integer such that $2^k \le n$. Then:

$$\sum_{i=1}^{n} 1/i \leq \sum_{i=1}^{2^{k+1}-1} 1/i \leq \sum_{j=0}^{k} \sum_{i=2^j}^{2^j+1-1} 1/i \leq \sum_{j=0}^{k} \sum_{i=2^j}^{2^j+1-1} 1/2^j \leq \sum_{j=0}^{k} 1 = k+1$$

All functions for this question should be stored in the file `sieve.c`, with the header for your function `sieve` in `sieve.h`. A file `sieve-driver.c` should contain a `main()` function with appropriate tests.

3. You have recently looked at the problem of computing the second, third and fourth largest elements from a set of data. In this question you are look for the kth largest element, where k is a parameter to the function. You should ignore multiple occurrences of the same number. Your function should have prototype

   ```
   int kthlargest(int k);
   ```

   It should read data from standard input until `EOF` is reached, at which point it should return the kth largest element. You may assume that $k < 1000$. Your program should not store all entries in the file into an array (and you do not know a maximum for the number of entries in the file). In a comment of your program, analyze the complexity of your code in terms of the number n of numbers in the file, and the number k.

   All functions for this question should be stored in the file `kthlargest.c`, with the header for your function `kthlargest` in `kthlargest.h`. A file `kthlargest-driver.c` should contain a `main()` function with appropriate tests for all functions.