CS 211: Computer Architecture, Fall 2020
Programming Assignment 1: Introduction to C (50 points)

Instructor: Prof. Santosh Nagarakatte

Due: September 21, 2020 at 5pm Eastern Time.

Introduction

This assignment's goal is to get you started with programming in C and get familiar with compiling, linking, running, and debugging C programs. Your task is to write 5 small C programs. Your program must follow the input-output guidelines listed in each section **exactly**, with no additional or missing output.

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are powerful. Hence, you should not look at your friend's code. See Rutgers academic integrity policy at:
`http://academicintegrity.rutgers.edu/`. We will report any violation to office of student conduct.

First: Twin Prime Number (5 Points)

You have to write a program that will read a list of integers from a file and print if each number in the file is a twin prime number or not.

A prime number x is a twin prime number if $x - 2$ is a prime number or $x + 2$ is a prime number. For example, 3 is a twin prime number since 5 is prime and 5 is a prime number since 3 is prime. There are infinitely many twin prime numbers: $3, 5, 7, 11, 13, 17, 19, \ldots$.

Input-Output format: Your program will take the file name as input. The file contains a number of positive integers. There is one integer per line. For example, a sample input file "file1.txt" can be:

```
3
14
19
23
31
```

Your output will contain the same number of lines as the number of lines in the input file. Each line will either say `yes` if the corresponding integer is a twin prime or `no` if the corresponding integer is not a twin prime.

```
$./first file1.txt
yes
no
yes
no
yes
```

We will not give you improperly formatted files. You can assume that the files exist and all the input files are in proper format as above.

Second: Ordered Linked List (10 points)

In this part, you have to implement operations on the linked list such that it maintains a list of integers in sorted order. For example, if a list already contains 2, 5 and 8, then 1 will be inserted at the start of the list, 3 will be inserted between 2 and 5 and 10 will be inserted at the end.

Input format: This program takes a file name as an argument from the command line. The file contains successive lines of input. Each line contains a string, either `INSERT` or `DELETE`, followed by a space and then an integer. For each of the lines that starts with `INSERT`, your program should insert that number in the linked list in sorted order if it is not already there. Your program should not insert any duplicate values. If the line starts with a `DELETE`, your program should delete the value if it is present in the linked list. Your program should silently ignore the line if the requested value is not present in the linked list. After every `INSERT` and `DELETE`, your program should print the contents of the linked list. The values should be printed in a single line separated by a single space. There should be no leading or trailing white spaces in each line of the output. You should print `EMPTY` if the linked list is empty.

Output format: At the end of the execution, your program should have printed the contents of the linked list after each `INSERT` or `DELETE` operation. Each time the contents are printed, the values should be on a single line separated by a single space. There should be no leading or trailing white spaces in each line of the output.You should print `EMPTY` if the linked list is empty. Your program should print "error" (and nothing else) if the file does not exist. You can assume that there will be at least one `INSERT` or `DELETE` in each file.

Example Execution:

Let's consider that we have 2 text files with the following contents:

```
file1.txt:
INSERT 1
INSERT 2
DELETE 1
INSERT 3
INSERT 4
DELETE 4
INSERT 5
DELETE 5
```

```
file2.txt:
INSERT 1
DELETE 1
INSERT 2
DELETE 2
INSERT 3
DELETE 3
INSERT 4
DELETE 4
INSERT 5
DELETE 5
```

Then the result of executing your program will be as follows:

```
$./second file1.txt
1
1 2
2
2 3
2 3 4
2 3
2 3 5
2 3
```

```
$./first file2.txt
1
EMPTY
2
EMPTY
3
EMPTY
4
EMPTY
5
EMPTY
```

```
$./second file3.txt
error
```

Third: Stack and Queue (10 points)

In this part, you will implement a linked list that supports both stack and queue operation. The idea is to have a single linked list that supports three operations:

- `ENQUEUE`: Queues a value at the end of the linked list
- `PUSH`: Pushes a value at the beginning of the linked list
- `POP`: Pops and removes the value at the beginning of the linked list.

Input format: This program takes a file name as an argument from the command line. The file contains successive lines of input. Each line contains a string, either `ENQUEUE` or `PUSH` followed by a space and then an integer OR just the word `POP` without anything following it. For each line that starts with `ENQUEUE`, your program should insert that number at the end of the linked list (like a queue). If the line starts with a `PUSH`, your program should insert that number at the beginning of the linked list (like a stack). If the line says `POP`, your program should pop and delete the first value at the beginning of the linked list.

After every `ENQUEUE`, `PUSH`, and `POP`, your program should print the contents of the linked list. The values should be printed in a single line separated by a single space. There should be no leading or trailing white spaces in each line of the output. You should print `EMPTY` if the linked list is empty.

Output format: At the end of the execution, your program should have printed the content of the linked list after each `ENQUEUE`, `PUSH`, or `POP` operation. Each time the content is printed, the values should be on a single line separated by a single space. There should be no leading or trailing white spaces in each line of the output.You should print `EMPTY` if the linked list is empty. Your program should print "error" (and nothing else) if the file does not exist. You can assume that there will be at least one `ENQUEUE`, `PUSH`, or `POP` operation in each file.

Example Execution:

Assume we have a text file with the following contents:

```
file.txt:
PUSH 1
ENQUEUE 2
PUSH 3
PUSH 4
POP
ENQUEUE 5
POP
POP
POP
POP
```

The the results will be:

```
$./third file.txt
1
1 2
3 1 2
4 3 1 2
3 1 2
3 1 2 5
1 2 5
2 5
5
EMPTY
```

Fourth: Magic Square (10 Points)

A magic square is an arrangement of the numbers from 1 to n^2 in an (n x n) matrix, with each number occurring exactly once, and such that the sum of the entries of any row, any column, or any main diagonal is the same.

An example of a Magic Square is as such:

```
8 1 6
3 5 7
4 9 2
```

In this case, the sum of all entries in a given row, column or main diagonal is equal to 15.

More examples and information about magic squares can be found here: `http://mathforum.org/alejandre/magic.square/adler/adler.whatsquare.html`

In this part, you will create a program that automatically creates a magic square for an odd-ordered matrix, *i.e.* $n \times n$ matrix where n is an odd number. There is a famous method for creating magic squares for matrix of odd order: `https://en.wikipedia.org/wiki/Magic_square#A_method_for_constructing_a_magic_square_of_odd_order`

Method: This method is applicable for all odd-ordered matrix. We illustrate the method by creating a 3×3 magic square.

(1) The first step starts with a number 1 at the center column of the first row:

	1	

(2) After that, we fill incrementally larger number to the cell diagonally up and right, one at a time. If the new cell goes outside of the matrix, we wrap around to the other side. For example, since there is no cell above and to the right of our `1`, we fill the bottom right cell with a `2`. It can also be considered that the sides of the matrices are connected to the opposite side when traversing the cells.

	1	
		2

	1	
3		
		2

(3), If the cell diagonally up and right of the current cell is already filled with a number, then we move one cell vertically down and fill that cell with the next number:

	1	
3		
4		2

It is important to remember that step (2) takes precedence over step (3):

	1	
3	5	
4		2

	1	6
3	5	
4		2

	1	6
3	5	7
4		2

8	1	6
3	5	7
4		2

8	1	6
3	5	7
4	9	2

The same principle can be used for any odd-ordered matrix to create a magic square.

Input/Output format Your program will accept a positive number n as a command line argument. You can assume that we will give a positive number n as the one and only command line argument, but n may be an even number or an odd number.

If n is an odd number, then your program should output the magic square created using the above method. The matrix should be produced with each rows separated by a line and each cell in a row separated by a tab. If n is an even number, then your program should print “error” (and nothing else).

Example Execution:

Here is an example of the input and the expected result:

```
$ ./fourth 3
8 1 6
3 5 7
4 9 2
```

```
$ ./fourth 4
error
```

Fifth: Matrix Determinant(15 points)

In linear algebra, the determinant is a value that can be computed with a square matrix. The determinant describes some properties about the square matrix. Determinants are used for solving linear equations, computing inverses, etc, and is an important concept in linear algebra. In the fifth part of the assignment, you will write a program that computes the determinant of any $n \times n$ matrix. You will have to carefully manage `malloc` and `free` instructions to successfully compute the determinants.

Determinant

Given a square $n \times n$ matrix M, we will symbolize the determinant of M as $Det(M)$. You can compute $Det(M)$ as follows:

1×1 **matrix** The determinant of the 1×1 matrix is the value of the element itself. For example,

$$Det(\begin{bmatrix} 3 \end{bmatrix}) = 3$$

2×2 **matrix** The determinant of a 2×2 matrix can be computed using the following formula:

$$Det(\begin{bmatrix} a & b \\ c & d \end{bmatrix}) = ad - bc$$

For example,

$$Det(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}) = 1 \times 4 - 2 \times 3 = 4 - 6 = -2$$

3×3 **matrix** The determinant of a 3×3 matrix can be computed modularly. First, let's define a 3×3 matrix:

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

The formula for computing the determinant of M is as follows:

$$Det(M) = a \times Det(M_a) - b \times Det(M_b) + c \times Det(M_c)$$

The matrix M_a is a 2×2 matrix that can be obtained by eliminating the row and column that a belongs to in M. More specifically, since a is on the first row and first column, we eliminate the first row and first column from M:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

This gives us a 2×2 matrix for M_a:

$$M_a = \begin{bmatrix} e & f \\ h & i \end{bmatrix}$$

M_b can be computed similarly. Since b is on the first row and second column, we eliminate the first row and second column from M:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

This gives us a 2×2 matrix for M_b:

$$M_b = \begin{bmatrix} d & f \\ g & i \end{bmatrix}$$

M_c can be computed by removing the first row and the third column from M since c is on the first row and third column. Thus,

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

$$M_c = \begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

Finally, the formula for computing the determinant of M is:

$$Det(M) = a \times Det(\begin{bmatrix} e & f \\ h & i \end{bmatrix}) - b \times Det(\begin{bmatrix} d & f \\ g & i \end{bmatrix}) + c \times Det(\begin{bmatrix} d & e \\ g & h \end{bmatrix})$$

For example, we can compute the determinant of the following matrix,

$$M = \begin{bmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{bmatrix}$$

as follows:

$$Det(M) = 2 \times Det(\begin{bmatrix} 5 & 1 \\ 3 & 8 \end{bmatrix}) - 7 \times Det(\begin{bmatrix} 9 & 1 \\ 4 & 8 \end{bmatrix}) + 6 \times Det(\begin{bmatrix} 9 & 5 \\ 4 & 3 \end{bmatrix}) = 2(37) - 7(68) + 6(7) = -360$$

$n \times n$ **matrix** Computing the determinant of an $n \times n$ matrix can be considered as a scaled version of computing the determinant of a 3×3 matrix. First, let's say we're given an $n \times n$ matrix,

$$M = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \dots & x_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & x_{n,3} & \dots & x_{n,n} \end{bmatrix}$$

In essence, we have to pivot each element in the first row and create $(n-1) \times (n-1)$ matrix for each pivot element (in the case of computing the determinant of 3×3 matrix, we had M_a that corresponds to a, etc).

For example, when we pivot $x_{1,1}$, we create the corresponding $(n-1) \times (n-1)$ matrix for $x_{1,1}$ by deleting the 1^{st} row and 1^{st} column:

$$M_{1,1} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \dots & x_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & x_{n,3} & \dots & x_{n,n} \end{bmatrix} = \begin{bmatrix} x_{2,2} & x_{2,3} & \dots & x_{2,n} \\ x_{3,2} & x_{3,3} & \dots & x_{3,n} \\ \vdots & \vdots & & \vdots \\ x_{n,2} & x_{n,3} & \dots & x_{n,n} \end{bmatrix}$$

Similarly, we can create $M_{1,2}$, $M_{1,3}$, $\dots$ by pivoting $x_{1,2}$, $x_{1,3}$, and so on:

$$M_{1,2} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \dots & x_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & x_{n,3} & \dots & x_{n,n} \end{bmatrix} = \begin{bmatrix} x_{2,1} & x_{2,3} & \dots & x_{2,n} \\ x_{3,1} & x_{3,3} & \dots & x_{3,n} \\ \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,3} & \dots & x_{n,n} \end{bmatrix}$$

$$M_{1,3} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \dots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \dots & x_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & x_{n,3} & \dots & x_{n,n} \end{bmatrix} = \begin{bmatrix} x_{2,1} & x_{2,2} & x_{2,4} & \dots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,4} & \dots & x_{3,n} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & x_{n,4} & \dots & x_{n,n} \end{bmatrix}$$

Finally, you can compute the determinant of M using the following formula:

$Det(M) = x_{1,1} \times Det(M_{1,1}) - x_{1,2} \times Det(M_{1,2}) + x_{1,3} \times Det(M_{1,3}) - x_{1,4} \times Det(M_{1,4}) + x_{1,5} \times Det(M_{1,5}) \dots$

The above formula can be shortened to the following formula:

$$Det(M) = \Sigma_{i=1}^{n} (-1)^{i-1} x_{1,i} \times Det(M_{1,i})$$

This general formula for computing the determinant of $n \times n$ matrix applies to all n. The formula for computing the determinant of 2×2 and 3×3 matrix is exactly the same as this formula.

Input-Output format:

Your program should accept a file as command line input. The format of a sample file `test3.txt` is shown below:

```
3
2	7	6
9	5	1
4	3	8
```

The first number (3) corresponds to the size of the square matrix (n). The dimensions of the matrix will be n x n. You can assume that n will not be greater than 20. The rest of the file contains the content of the matrix. Each line contains a row of the matrix, where each element is separated by a tab. You can assume that there will be no malformed input and the matrices will always contain valid integers.

Your program should output the determinant of the $n \times n$ matrix provided by the file.

Example Execution

A sample execution with above input file `test3.txt` is shown below:

```
$./fifth test3.txt
-360
```

Structure of your submission folder

All files must be included in the `pa1` folder. The `pa1` directory in your tar file must contain 5 subdirectories, one each for each of the parts. The name of the directories should be named first through fifth (in lower case). Each directory should contain a c source file, a header file (if you use it) and a Makefile. For example, the subdirectory first will contain, first.c, first.h (if you create one) and Makefile (the names are case sensitive).

```
pa1
|- first
   |-- first.c
   |-- first.h (if used)
   |-- Makefile
|- second
   |-- second.c
   |-- second.h (if used)
   |-- Makefile
|- third
   |-- third.c
   |-- third.h (if used)
   |-- Makefile
|- fourth
   |-- fourth.c
   |-- fourth.h (if used)
   |-- Makefile
|- fifth
   |-- fifth.c
   |-- fifth.h (if used)
   |-- Makefile
```

Submission

You have to e-submit the assignment using Canvas. Your submission should be a tar file named `pa1.tar`. To create this file, put everything that you are submitting into a directory (folder) named `pa1`. Then, `cd` into the directory containing `pa1` (that is, `pa1`'s parent directory) and run the following command:

```
tar cvf pa1.tar pa1
```

To check that you have correctly created the tar file, you should copy it (`pa1.tar`) into an empty directory and run the following command:

```
tar xvf pa1.tar
```

This should create a directory named `pa1` in the (previously) empty directory.

The `pa1` directory in your tar file must contain 5 subdirectories, one each for each of the parts. The name of the directories should be named first through fifth (in lower case). Each directory should contain a c source file, a header file and a make file. For example, the subdirectory first will contain, first.c, first.h and Makefile (the names are case sensitive).

AutoGrader

We provide the AutoGrader to test your assignment. AutoGrader is provided as autograder.tar. Executing the following command will create the autograder folder.

```
$tar xvf autograder.tar
```

There are two modes available for testing your assignment with the AutoGrader.

First mode

Testing when you are writing code with a `pa1` folder

(1) Lets say you have a `pa1` folder with the directory structure as described in the assignment.

(2) Copy the folder to the directory of the autograder

(3) Run the autograder with the following command

$python auto_grader.py

It will run your programs and print your scores.

Second mode

This mode is to test your final submission (i.e, pa1.tar)

(1) Copy pa1.tar to the auto_grader directory

(2) Run the auto_grader with pa1.tar as the argument.

The command line is

$python auto_grader.py pa1.tar

Scoring

The autograder will print out information about the compilation and the testing process. At the end, if your assignment is completely correct, the score will something similar to what is given below.

```
You scored
2.5  in  second
```

```
5.0  in  fourth
5.0  in  third
5.0  in  fifth
7.5  in  first
Your TOTAL SCORE =  25.0 /25
Your assignment will be graded for another 25 points with test cases not given to you
```

Grading Guidelines

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- **You should not see or use your friend's code either partially or fully. We will run state of the art plagiarism detectors. We will report everything caught by the tool to Office of Student Conduct**.
- You should make sure that we can build your program by just running `make`.
- Your compilation command with gcc should include the following flags: **-Wall -Werror -fsanitize=address**
- You should test your code as thoroughly as you can. For example, programs should *not* crash with memory errors.
- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result **in up to 100% penalty**. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.
- **Your folder names in the path should have not have any spaces. Autograder will not work if any of the folder names have spaces.**

Be careful to follow all instructions. If something doesn't seem right, ask on discussion forum.