M4PA
Enter the Matrix

Synopsis

We want to implement two functions which are part of a program meant to operate on a two-dimensional integer matrix. Other programmers will work on the GUI while we have been tasked to develop the functions. In order to help us test the functions before they are used by the other programmers, we write a *main* which will use our two functions on some pre-defined data in order to make sure they operate as expected by the requirements.

Our first function will test whether a given matrix is in order or not while the other will sort it. We will have other functions in our project, but they will be there to help us structure the testing of the two above-mentioned functions. The *main* will simply invoke the test functions so we display the results when we run our program.

We're getting one tiny step nearer to how testing is actually done "in the real world".

Files you will be working with

You will be provided with several files to get you started working on this assignment.

- You must not alter the file names, remove or add files to the project
- You must only modify the ones marked below with a yes in "Modify it?"

Here are the files;

File name	Modify it?	Role
tools.c	Yes	Implementation of your solution to the assignment
tests.c	Yes	Implementation of your test functions
tools.h	No	Header file for *tools.c*
main.c	No	Implementation of the main function starting your tests
testlib.h	No	Definition of the *TEST* function you must use in your tests
testlib.c	No	Implementation of the above

Task #1 – Implementing and testing your isMatrixSorted

Some reading before you start

Before to start working on implementing anything, you'll have to read the entire source and make sure you understand what it is doing so that you are able to add to it without breaking everything.

Pay specific attention to the way we implemented some test functions in *tests.c*. These will make use of your *isMatrixSorted* implementation to make sure it is behaving as expected from the requirements.

In addition, spend some time reading the requirements in this document and adding your own test functions in *tests.c* to reflect the tests you think your program ought to pass to meet these requirements.

To give you an idea of what tests look like, you will find examples in *tests.c*. In addition, think about the following;

- Right off the bat, you want *isMatrixSorted* to return false when you pass to it an unsorted matrix. This will actually succeed without you implementing anything since the default implementation of *isMatrixSorted* returns false all the time.
- Then, you want *isMatrixSorted* to return true when you pass an already sorted matrix. This one is also a very important test but it will fail with the default implementation. No worries, this gives you a goal to meet as you start working on your solution
- Even though we are not yet working on *matrixSort*, you might want to throw in some tests about it too – knowing they will mostly fail until you actually get started. How does one test *matrixSort* you ask? Well, use *isMatrixSorted* and assume for now it will work one day ☺
- Here is an example;
    - Provide *matrixSort* with an already sorted matrix
    - Use *isMatrixSorted* afterward to make sure it's still sorted – i.e. your implementation of *matrixSort* didn't mess up the matrix
- Here is another example;
    - Provide *matrixSort* with a non-sorted matrix
    - Use *isMatrixSorted* afterward to make sure it is now sorted

Implementing your solution

Next step is implementing the **isMatrixSorted** function in the *tools.c* file so that it detects whether a matrix passed as parameter is already sorted or not and returns an appropriate value. Refer to the comments in the code for the specific details.

To give you an example, a two dimensional 3x5 matrix is sorted when its elements are ordered as follows;

3	5	9	20	20
42	50	55	99	142
200	209	500	900	999

Testing your solution

Now is time to add more tests based on what you learned when you actually started implementing the solution and what you learned when you had to fend off your first bugs.

Revise the tests you previously implemented in *tests.c*.

Bit of refactoring

Do not attempt the rest of this assignment until your first function is working on a good quality suite of tests. See the PDF document explaining how to design a good “test harness” which was uploaded as part of this week’s module.

Now that things are working, what about refactoring / simplifying your program and tests?

- Refactoring means rewriting some parts to improve them without adding any new features or altering the way your program behaves.
- Improving sometimes means making your program run faster. In this offering we are not really going for top performance due to the simplicity of the tasks we tackle but this doesn’t mean there are no other forms or improvement we are not after;
    - Simplify your solution; if you are able to do the same job in 3 lines than you are doing now with 3 nested loops, do it.
    - Make your program easier to understand; fix your indentation, add comments for all variables / parameters / functions or even every non-trivial step, rename variables / parameters / functions so they better reflect their role...

Every time you improve something, no matter how small, make sure you recompile, relink, re-run your project against the tests you have. This is to make sure you did not just break something which was working before. Never move to the next refactoring until you have passed what is referred to as your *regression testing*.

Task #2 – Implementing and Testing matrixSort

When we are going to test your 2nd function, **matrixSort**, we are going to rely on **isMatrixSorted**. This means you should not attack this part of the practice assignment before to be fairly sure your **isMatrixSorted** function works.

Some reading before you start

Read the requirements and the comments in the program itself to make sure you understand what's expected of this function. Then, add more tests focused on ensuring it works. This should be relatively easy and maybe you won't be able to think about too many more tests to add to those you used to validate **isMatrixSorted**.

Implementing your solution

The **matrixSort** function will take a two dimensional integer matrix which might or might not be already ordered and order it in a manner which **isMatrixSorted** will recognize as ordered. There are different algorithms you might implement.

Most students like to use the "bubble sort" from the textbook and adapt it to a two dimensional matrix. I personally think it's a bit too much work.

Others go with the following logic; we need to look at each element of the matrix in order, one after the other. For each such element, I use loops to go other all the elements between it and the end of the matrix. Each time, I compare the two elements and swap them if they are out of order. This result is me looking for the smallest element and putting it first in the matrix, then looking for the next smallest one and putting it second in the matrix. For a one dimensional array, the algorithm would look something like;

```
// my array name is data which has indexes between 0 and N-1
int i,j;
for(i=0 ; i < N-1; i++){
    for(j=i+1; j < N ; j++){
        if( data[i] > data[j]) SWAP THEM
    }
}
```

The hard part is to adapt this to a two dimensional matrix. Feel free to explore.

Testing your solution

Again, you might have learned a bit more about the problem by trying to implement it or found some bugs which you may detect with a test; E.g. I had a bug when only the first and last elements were not in order; I was pretty much off by one in one of my loops. I’m now adding a test to make sure I detect this specific bug from now on. Revise your tests.

Bit of refactoring

When your program is working on task #2, take time to improve its implementation. Do not try to improve things like indentation before it works but do not move to the next tasks until you’ve improved it.