Assignment 1

Due Jan 29 by 4pm **Points** 5

Jan 22nd: For life2D, the functions `print_state` and `update_state` take a pointer to a 1D array of integers. The format of that 1D array for a board:

$x_{0,0}$	$x_{0,1}$	...	$x_{0,n}$
$x_{1,0}$	$x_{1,1}$	...	$x_{1,n}$
...	...	...	...
$x_{m,0}$	$x_{m,1}$	...	$x_{m,n}$

will be {$x_{0,0}$, $x_{0,1}$, ... ,$x_{0,n}$, $x_{1,0}$, $x_{1,1}$ ..., $x_{1,n}$, ..., $x_{m,n}$}. In other words, the 1D array will contain the integers from row 0, followed by the integers from row 1, and so on.

Jan 18th: Revised Benford wording about the base and added some examples with a base other than 10. See the coloured highlights.

Jan 18th: You'll learn more about files and file reading later in the course, but for this assignment there are two functions you'll need: fopen (**video** **(https://www.youtube.com/watch?v=RXbnIZcCoPA&feature=youtu.be)**) and fscanf (**video (https://www.youtube.com/watch?v=C9ihXyPhKJE)**). In addition to the videos, use the man pages to learn more about the functions. At the end of the fscanf video, it mentions fgets, which is a function we'll use for file reading once you learn more about strings, but for this assignment fscanf is sufficient.

Assignment 1: Intro C

Introduction

For this assignment, you will be writing two command-line utilities: a program that explores Benford's Law and a 2D game of life. The assignment involves C basics, including command-line arguments, arrays, and processing data from standard input (using scanf).

Please remember that we are using testing scripts to evaluate your work. As a result, it's very important that (a) all of your files are named correctly and located in the specified locations in your repository and (b) the output of your programs match the expected output precisely.

Part 0: Getting Started: starter code and how to compile (0%)

- **Starter code**: Your first step should be to log into MarkUs and navigate to the **a1: Intro C** assignment. Like for the labs, this triggers the starter code for this assignment to be committed to your repository. Pull your git repository. There should be a new directory named a1 with subdirectories benford and life2D. All of your code for this assignment should be located in the benford and life2D directories. There is starter code for each program.
- **How to compile (and clean up)**: To compile your program, you should use the Makefile provided by issuing the command "*make*". To remove the .o files and executables, type "*make clean*".
- **Running the checker**: We have provided a single test case for each program. Type "*make check_life2D*" or "*make check_benford*". That will compile the corresponding program and run our basic check.
- **Checking your submission**: Once you have pushed files to your repository, you can use MarkUs to verify that what you intended to submit was actually submitted. The Submissions tab for the assignment will show you what is in the repository, and will let you know if you named the files correctly.

Part 1: Exploring Benford's Law (benford.c and benford_helpers.c) (2.5%)

What treasures are hidden in large data sets? What do the number of posts by students to a Piazza forum and the populations of towns in Quebec have in common? To find out, read about Benford's Law in: **Looking out for number one (https://plus.maths.org/content/os/issue9/features/benford/index)** .

For this part of the assignment, you will write a program to find the distribution of digits in a set of data. The program takes a position `i` and a set of numbers, and outputs a list of values: the frequency with which each digit appears in position `i` of one of the input numbers. For example, here is example output from `benford` for position 0 of a set of base 10 numbers (there are 0 numbers whose 0th digit is 0, 123 numbers whose 0th digit is 1, 46 numbers whose 0th digit is 2, and so on):

```
0s: 0
1s: 123
2s: 46
3s: 34
4s: 21
5s: 6
6s: 9
7s: 14
8s: 8
9s: 7
```

Your first task is to complete a program, `benford.c`, and its helper functions from `benford_helpers.c`:

- `BASE` is a constant representing the number of unique digits. You may assume `BASE` will be between 2 and 10 inclusive.
- The program `benford` takes up to two arguments. The first represents the position (a non-negative integer) ~~between 0 and `BASE` -10 inclusive~~. The second argument is optional and, if given, represents the name of a data set file: a file containing one non-negative integer per line. If not given, the data set will be redirected as input to the program.
- Assume that the input file is formatted correctly and you do not need to do any error checking on the file format.
- Assume that the integers in the file are represented in base 10.
- Assume that the position is a valid position in every integer given as input (e.g., if the position is 2, then every integer will be at least three digits long when represented in base BASE).
- You must use the printf statements specified in the starter code, so that the output is correctly formatted.

Implement the following functions in `benford_helpers.c`:

- **int count_digits(int num)**: Return the number of digits needed to represent the non-negative base 10 integer `num` in base `BASE`. For example, `count_digits` should return 1 for inputs 0-9, 2 for 10-99, 3 for 100-999, and so on if BASE is 10. For another example, if `num` is 4 and `BASE` is 2, it should return 3 (i.e., it takes three digits to represent 4 in base 2: 100). Tip: use repeated division by `BASE`.
- **int get_ith_from_right(int num, int i)**: Return the digit `i` positions from the right-hand side of the base `BASE` representation of base 10 number `num` . For example, for `num` 12345 and `BASE` 10, `get_ith_back` returns 5 when `i` is 0, 4 when `i` is 1, 1 when `i` is 4 and so on. ~~You may assume that `i` is between 0 and `BASE` -1~~. For example, for `num` 5 and `BASE` 2, `get_ith_back` returns 1 when `i` is 0, 0 when `i` is 1, and 1 when `i` is 2.
- **int get_ith_from_left(int num, int i)**: Return the digit `i` positions from the left-hand side of the base `BASE` representation of base 10 number `num`
- **void add_to_tally(int num, int i, int *tally):** Assume parameter `tally` points to an array of non-negative integers that contains `BASE` items, where the left-most item represents digit 0, the next represents digit 1, and so on. Update `tally` to include the digit `i` positions from the left-hand side of `num`. For example, for `num` 2365, `BASE` equal to 10, `i` equal to 2, and the tally array `[3, 2, 3, 0, 4, 2, 2, 0, 0, 1]`, the tally array would be updated to `[3, 2, 3, 0, 4, 2, 3, 0, 0, 1]`.

Next, complete the program `benford.c`, calling on the functions from `benford_helpers.c` when appropriate.

Note that you must use **exactly** the format shown above in your output. The `printf` statements are given to you in the starter code to make this easy for you.

Command line arguments and return codes

For the programs in part 1, you may assume that if the correct number of command-line arguments are provided, they will have the correct format. If the user calls the program with too few or too many arguments, the program should not print anything to stdout (only the provided message to stderr), but should return from main with return code 1. Following the standard C conventions, main should return 0 when the program runs successfully.

Part 2: 2D Game of Life (life2D.c and life2D_helpers.c) (2.5%)

You will write a C program called life2D.c that implements a variant of the **Game of Life (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)** . You'll also write two helper functions in life2D_helpers.c.

Details

Your program reads three command line arguments: the first represents the height of the board (the number of rows), the second represents the width of the board (the number of columns), and the third represents the number of states to print. Your program will read the initial state from standard input. Your program will then print the number of states specified, each followed by a blank line, starting with the initial state. For example, given the initial state (stored in a file called sample_input.txt):

```
wolf:~$ cat sample_input.txt
1 0 0 0
0 1 0 0
0 1 1 0
0 0 0 1
0 1 0 0
```

the following invocation of the program would print:

```
wolf:~$ ./life2D 5 4 3 < sample_input.txt
1000
0100
0110
0001
0100

1000
0110
0110
0101
0100

1000
0010
0000
0101
0100

wolf:~$
```

Requirements

You may assume that the initial state given is a rectangular grid of 0s and 1s with each pair of digits in a row separated by a single space and each row ending with a newline. You may assume that the width and height given match the state, and that the number of states is a positive integer greater or equal to 1.

Along with the main function, you are required to write one helper function (`update_state`) and use another provided helper function (`print_state`). The main function is in life2D.c and the helper function is to be implemented in the file life2D_helpers.c:

- void update_board(int *board, int num_rows, int num_cols): Given an array representing the board, along with the number of rows and columns the board has, update the state of the array according to the following rules:
- the cells around the edge (first and last row, and first and last column) never change
- a cell's neighbours are the cells adjacent to it (a non-edge cell has 8 neighbours)
- a cell that is currently 1, becomes 0 if it has too few (< 2) or too many (> 3) neighbours

- a cell that is currently 0, becomes 1 if it has exactly 2 or 3 neighbours

Command line arguments and return codes

For the program in part 2, you may assume that if exactly three command-line arguments are provided, they will have the correct format. If the user calls the program with too few or too many arguments, the program should not print anything to stdout (only the provided message to stderr), but should return from main with return code 1. Following the standard C conventions, main should return 0 when the program runs successfully.

Reminder

Your programs must compile on teach.cs using gcc with the -Wall option and should not produce any warning or error messages. Programs that do not compile, will get 0. You can still get part marks by submitting something that doesn't completely work but does some of the job -- but it **must at least compile** to get any marks at all. Also check that your output messages are **exactly** as specified in this handout.

Submission

We will be looking for the following files in the a1 directory of your repository:

- benford.c
- benford_helpers.c
- life2D.c
- life2D_helpers.c

Do not commit .o files or executables to your repository.