Computer Science 352 Spring 2023
Programming Assignment 2
Due February 1st 2023 by 7pm

This second PA for CS 352 has two parts. The first is easier, and is a program for calculating how many positions a chess piece could be in on an infinite chessboard given a number of moves. The second is a program to encrypt standard input using a Beaufort cipher.

```
pa2
├── infinite_chess
│   └── infinite_chess.c
└── beaufort
    └── beaufort.c
```

Remember to use at least the **`-Wall -Werror -std=c11`** compilation flags, and ensure that you compile, run, and test everything on lectura before you turn it in.

Part A - Infinite Chessboard

For this program, you should write a C program named **`infinite_chess.c`**. This program will accept two inputs from standard in. The first will be one character (used to identify which type of chess piece we are dealing with) and the second will be an integer number, representing the number of moves to simulate. You should get both of these input values using the **`scanf`** function.

For this program, the only two standard library imports you are allowed to use are **`<stdlib.h>`** and **`<stdio.h>`**. You may *not* use **`math.h`**.

A chess board is typically an 8x8 2D grid. However, for this assignment, you should assume we are working with an infinitely large 2D grid / board. The goal of the program is to determine how many possible board locations a particular piece could be at if it started from some location on the board and was able to make at most N moves. You can expect that the first input will be one of three characters: k (for king), p (for pawn) and b (for brigadier).

In real chess, pawn movement is a bit more complex, but for our purposes a **Pawn** can only move 1 location, and can only move in 1 direction (forward). Thus, this is the easiest one to simulate. If the pawn had zero moves, there is only 1 location it could be at. If it had 1 move, it could be in 2 possible locations. If it had 2 moves, it could be in 3 possible locations, and so on.

When a **King** moves in chess, it can move to any of the neighboring pieces on the board, including diagonals. For this problem, if the king had zero moves, there is only 1 location it could be at. However, if it had 1 move, it could be in 9 possible locations. If it had 2 moves, it could be in any of 25 possible locations, and so on.

The **Brigadier** is a new chess piece that I made up for this PA. A brigadier can only move to any of its 4 neighboring diagonal locations. It cannot move directly forward / backward / left / right. Thus, if the brigadier had zero moves, there is only 1 location it could be at. However, if it had 1 move, it could be in 5 possible locations. If it had 2 moves, it could be in any of 13 possible locations, and so on.

Here are a few examples, with standard input highlighted in red:

```
$ ./a.out
Enter piece type (k, b, p):
k
Enter number of moves:
5
possible locations: 121
```

```
$ ./a.out
Enter piece type (k, b, p):
p
Enter number of moves:
20
possible locations: 21
```

```
$ ./a.out
Enter piece type (k, b, p):
b
Enter number of moves:
4
possible locations: 41
```

```
$ ./a.out
Enter piece type (k, b, p):
k
Enter number of moves:
20
possible locations: 1681
```

You should spend some time thinking through formulas to model the possible locations these pieces could be after N moves, and incorporate these into the program. Test thoroughly.

Part B - Beaufort Cipher

In this problem, you will be writing a program named **beaufort.c**. In this program, you will create an implementation of the Beaufort cipher for encrypting messages (you do not have to implement decryption). Before you jump into implementing this program, you should read up on how this cipher works. Rather than trying to explain it here, I will link to an article which provides an explanation. Read through this, and then return to the spec:

http://practicalcryptography.com/ciphers/classical-era/beaufort/

As you should have read about, the Beaufort cipher requires an encryption **key** in order to function. Once you have the key, you can encrypt any incoming message. **`beaufort.c`** should expect that the first line it gets on standard input will be the value of the key (a char sequence with only upper-case alphabetical letters). All remaining lines it can expect will be input strings to encrypt. The program can end when it encounters its first empty input line. The program can also expect that all of the text coming from standard input will be upper-case. If it sees a space in a message to encrypt, it should be left-as-is.

The article that I linked shows that this algorithm uses a large, 2-dimensional tableau for converting from input text to encrypted text. This *could* be implemented using a large 2D array in our code, but that is actually not necessary, and not allowed for this PA. The algorithm can be implemented with strings (1D char arrays). Study the algorithm and take some time to think through how this can be done.

For example, let's say that we have a file named **`input.txt`** in the same directory as the compiled **`beaufort.c`** and **`a.out`** file with these contents:

```
FORTIFICATION
DEFENDTHEEASTWALLOFTHECASTLE
WALLS MUST BE DEFENDED
DEFEND THE EAST WALL OF THE CASTLE
```

We should be able to get the following:

```
$ cat input.txt | ./a.out
CKMPVCPVWPIWUJOGIUAPVWRIWUUK
VTXUQ QGBP NJ CKMPVCEZ
XPDKAC VKP EFQJ ETXD ZA VKP GFQJPP
```

Another example, if this is the input file:

```
ELEPHANTS
AAAAAAAAAAAAAAAAAAAAA
ZZZZZZZZZZZZZZZZZZZZZ
WWGW
```

We should be able to get the following:

```
$ cat input.txt | ./beaufort
ELEPHANTSELEPHANTSELEP
IBOUTFMFQIBOUTFMFQIBOU
WIFI
```

You may assume that no line, including the key line, will be longer than 128 characters (bytes) of data. However, you should not assume any maximum number of lines. The program should be able to operate on however many lines of input it is given. The last line of output should end with a newline.

The index / offset into the key used for encrypting should carry-over for each encrypted line.

Submitting Your PA

After you have completed the two parts of this assignment, double check that the file structure matches what is shown in the project overview on the first page. Though you are encouraged to create and run test cases, you should remove them from your final submission before you turn this in. Before you submit, you should ensure that both programs compile, run, and produce the correct outputs on lectura and that you don't have any extra files other than the c files and directories.

Once you are ready to submit, zip up the **`pa2`** directory by running:

`$ zip -r pa2.zip ./pa2`

Then, turn this file in to the PA 2 dropbox on gradescope.

There will be some public test cases on lectura, and you should ensure you pass those. There will also be some hidden test cases that will not be revealed until after grades are published. Therefore, you should test your programs thoroughly!