Instructions

Deliverables

There are three deliverables for this assignment: C programs named `stats.c`, `bitboard.c`, and `cardRecovery.c`. Please submit the files using the submission link on Moodle. Only one member of each pair needs to do this, though do remember to add the other person's name to the submission.

Pair Programming

You are required to work with your assigned partner on this assignment. You must observe the pair programming guidelines outlined in the course syllabus — failure to do so will result in both team members receiving a score of 0 for the assignment. *Collaboration across teams is prohibited and at no point should you be in possession of any work that was completed by a person other than you or your partner.*

Getting Started

On this assignment, you will be solving three different problems by writing medium-sized C programs. These problems will require the use of many common programming constructs — `if`-statements, looping structures, functions etc. It may be worth reviewing the "C for Python Programmers" tutorial again (linked to on Moodle) to remind yourselves of C syntax rules. One problem requires extensive use of bitwise manipulations, so I also recommend reviewing the handout on this topic. There is a style guide at the top of the course Moodle page that outlines the conventions to follow when writing C programs for this class. Finally, use the `-Wall` compiler flag and `valgrind` as your first line of defense against programming errors, and `gdb` to help you debug your programs. A cheatsheet outlining how to use these programs is also available at the top of the class Moodle page.

[5pts] Statistics

A common task when working with data is to compute the mean (average) and variance of a set of numeric values. Given a sequence of numbers $x_1, x_2, \ldots, x_n$, the formula for computing the mean (μ) and variance (σ^2) are as follows:

$$\mu = \frac{1}{n} \sum_{i=1}^{i=n} x_i$$

$$\sigma^2 = \frac{1}{n} \left(\sum_{i=1}^{i=n} x_i^2 \right) - \mu^2$$

In a file named `stats.c`, write a function with the following signature:

```
void computeStats()
```

that when called:

- repeatedly prompts the user to enter a single floating-point number (ignoring invalid values), until the user types in a negative value, and
- prints out the mean and the variance of the entered values with two decimal points of precision.

Here is an example of how your function is expected to behave when called (for clarity, user input is highlighted in red):

```
Please enter a number:  2
Please enter a number:  4
Please enter a number:  6
Please enter a number:  8
Please enter a number:  foo
Sorry, that is not a valid number - input ignored
Please enter a number:  10
Please enter a number:  -1
Mean:  6.00, Variance:  8.00
```

Notes and Constraints

- To pass my tests, ensure that your output to the screen precisely matches the formatting shown in the example above, including spacing, capitalization, etc.

- Use the `scanf` function to read values from the user. Here's an example of how to use `scanf` to read a single float from the console and store it in a variable named `num` (of type `float`).

  ```
  scanf("%f", &num);
  ```

  Note the use of the `&` before the name of the variable. For now, ignore the purpose of this symbol — we'll examine it in greater detail next week when we discuss memory and pointers.

- Your program should be robust to user-input errors and handle them gracefully as shown in the example above. You can determine whether the user entered a valid floating-point value by examining the return value of the `scanf` function. For example, suppose we had the following line of code:

  ```
  int numRead = scanf("%f", &num);
  ```

  The value of the variable `numRead` will be set to the number of values that `scanf` was able to successfully read. In this case, if `scanf` successfully reads a `float` from the console, then `numRead` will be 1; otherwise, it will be set to 0. (The value that is read from the console itself will still be stored in `num` — don't confuse this with the return value of `scanf`!)

- Related to the previous point: when a `scanf` call fails, the illegal user input is left in the input stream. This is problematic, since this means that future calls to `scanf` will also fail, since the illegal input is not "flushed". You are provided a file named `io.c` as part of the supplementary zip archive that contains a utility function named `flushInputBuffer` that will consume any leftover data in the input stream. You can open `io.c` to read the documentation about how to call `flushInputBuffer`, though you don't have to understand how the function works (but you're welcome to have a look!). To use this function in your `stats.c` program, you'll first need to "import" it — place the following statement at the very top of your `stats.c` file:

  ```
  #include "io.h"
  ```

  Note the use of the quotes around the name of the library `io.h`, instead of the angled brackets (like in `<stdio.h>`) — this is to indicate to the compiler that the `io.h` file can be found in the current folder, and not in the default location where the rest of the

standard C library files live. Since your program is now split up across multiple files, you will also need to invoke `gcc` in a slightly different way, by providing the names of *all* the source files like so:

```
gcc stats.c io.c
```

This will jointly compile the two C files to produce a single executable named `a.out`. **You should not modify `io.h` or `io.c` — these files are meant to be used as is.**

- Note that a negative value acts as a *sentinel* to indicate "end-of-input". It is not a part of the data and should not be included in your mean/variance calculation.
- If the user enters no data points, then the mean and variance should be computed to be `0.00`.

[10pts] Generating Pawn Moves

For many decades, building a world championship caliber Chess-playing program was the holy grail of Artificial Intelligence research — indeed, Alan Turing himself worked on this problem and wrote the world's first Chess-playing program in 1948[1]. One of the key bottlenecks in a high-performing Chess engine is *move generation* — the rules defining how pieces may move in Chess are complex, and if implemented naïvely, can dominate the runtime of the program. Most modern Chess engines use a special data structure called a *bitboard* to speed up this computation. On this problem, your task is to write a move generator for the pawn piece on a Chess board using the bitboard representation.

First, some background: a Chess board is an 8×8 grid with alternating white and black squares. The columns (called *files*) are labeled `a-h`, while the rows (called *ranks*) are labeled 1-8, as shown in figure 1. This figure also shows the starting configuration of the pieces of the two players, white and black. The *pawns* are the pieces that appear in ranks 2 and 7. The rules governing the movement of pawns are as follows:

- A pawn may *advance* along its own file by one square, if the destination square is unoccupied. So, for example, in the Chess position shown in the right panel of figure

[1]Interestingly, Turing designed his Chess-playing algorithm *before* the first general-purpose digital computers had been built! Lacking an actual machine on which he could run his program, Turing played the role of computer himself, faithfully executing the instructions one-by-one, by hand. This "program" won its first game against a human amateur (the spouse of Turing's friend), but then lost its second game to Turing's colleague, Alick Glennie, an avid Chess player.

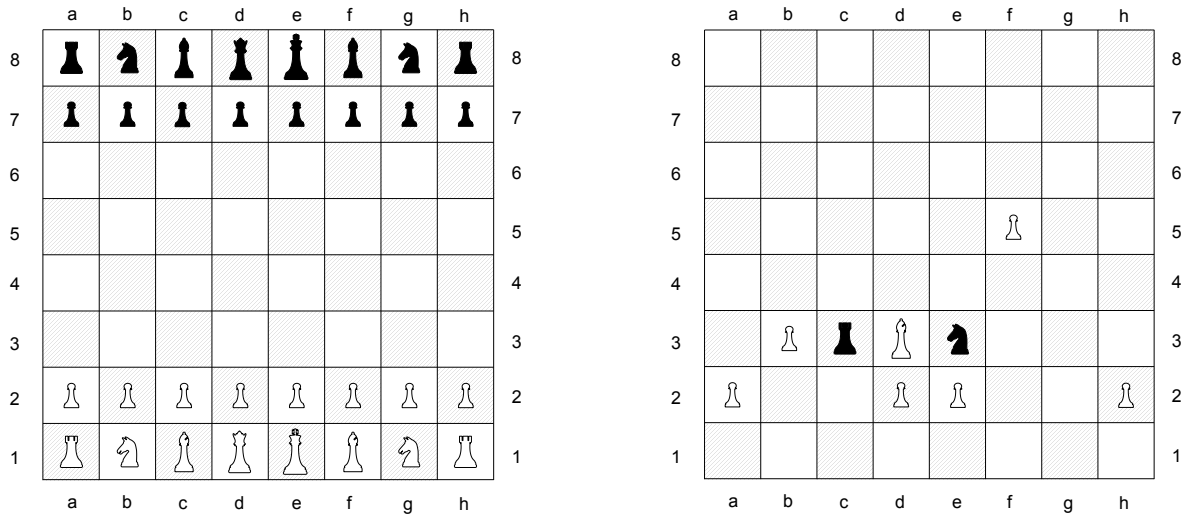

Figure 1: Left: the initial configuration of a chess board. Right: an arbitrary arrangement of pieces on a board, to help illustrate legal pawn moves. Image credit: `https://www.presentationmagazine.com/`.

1, the pawn at position `a2` can advance one square to position `a3`. Similarly, `b3` can move to `b4`, `f5` to `f6`, and `h2` to `h3`. Note that the pawns at `d2` and `e2` are blocked by the pieces immediately in front of them, so neither can advance along their own rank. Pawns can never move backwards.

- A pawn on rank 2 may also advance along its file by two squares so long as its path is not blocked by another piece — however, this special rule only applies to pawns on rank 2. In the position in the right panel of figure 1, the pawn at `a2` can thus also move to `a4`, and `h2` can also move to `h4`. Note that `b3` and `f5` may only advance by one square since they're not rank 2 pawns. Pawns `d2` and `e2` are blocked by pieces on `d3` and `e3`, and cannot advance to `d4` and `e4`, respectively, either.

- A pawn on rank 8 cannot advance any further. (Technically, the rules of the game state that such a pawn can be promoted to behave like any other piece on the board, but we will ignore this rule.)

- Finally, a pawn may *capture* a piece of the opposite color that resides on a square that is one step away diagonally. In the right panel of figure 1, the pawn on `d2` can capture the black piece either to its left (`c3`) or to its right (`e3`). Captures *must* occur diagonally; thus, the pawn on `e2` may not capture `e3`. Also, only pieces of the opposite color may be captured — thus `e2` cannot move to `d3` either. The pawn on `e2` is thus stuck and cannot move at all. Figure 2 summarizes all possible moves that the pawns

in the right panel of figure 1 can make. Blue arrows indicate advances and red arrows indicate captures.

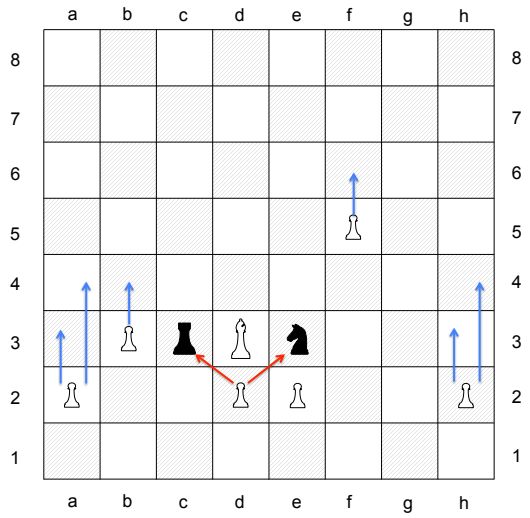

Figure 2: All possible pawn moves from the position depicted in the right panel of figure 1.

A straightforward way to represent a Chess board would be as an 8×8 two-dimensional array. One could then generate all possible white pawn moves, given a specific board configuration, by writing a set of `for` loops, with appropriately phrased `if` conditions to enforce the movement rules. However, a strong Chess engine often analyzes millions of chess positions in order to make a single move and this naïve approach to move generation is often too slow. A bitboard representation will allow us to compute the possible moves of *all* the pawns in parallel, using only fast bitwise operators, and speed up the move generation process by orders of magnitude.

To create a bitboard, we number the squares of the Chess board from 0 to 63 as shown in the left panel of figure 3. We then indicate the occupancy of each square using a 0 (unoccupied) or 1 (occupied). Finally, we treat the sequence of 64 bits as a single 64-bit integer (in C, this is the type `unsigned long`), where square 63 is the most significant bit and square 0 is the least-significant bit. For example, consider the sample configuration of white pawn pieces in the right panel of figure 3. In the bitboard representation, we would fill in six ones in all: in bit positions 34 (`f5`), 23 (`b3`), 15 (`a2`), 12 (`d2`), 11 (`e2`), and 8 (`h2`). As a 64-bit number, this would be the hex value `0x0000000400409900`.

What if our board contained pieces of more than one type? The solution is to create multiple bitboards, one for each kind of piece — for example, a bitboard for all the white

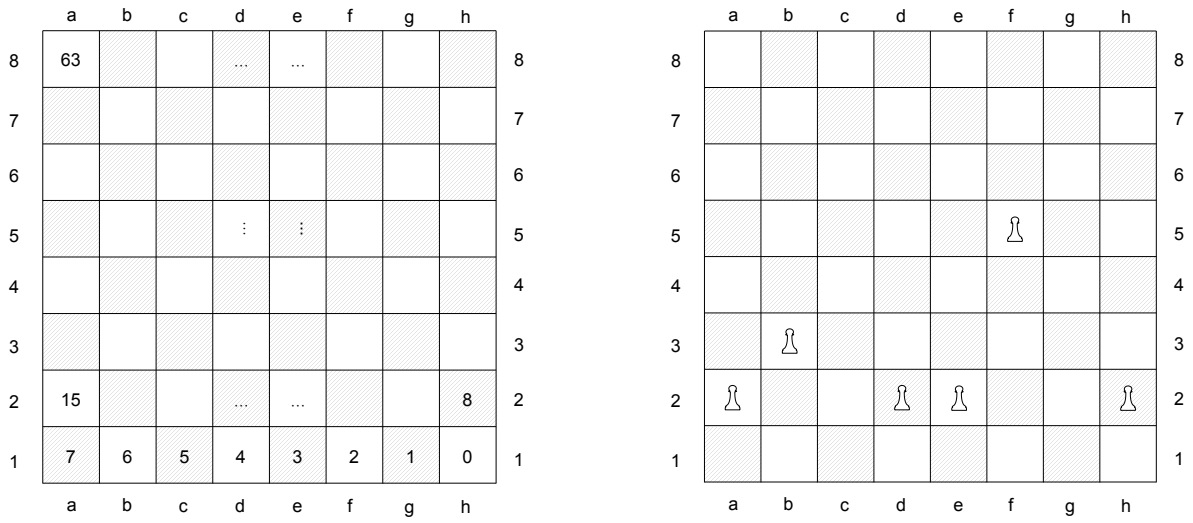

Figure 3: Left: a numbering of the squares of a Chess board. Right: a possible configuration of white pawn pieces on a Chess board.

pawns, a bitboard for all the black pawns, a bitboard for the white knights, and so on. In this way, the entire state of the Chess board can be described with just a handful of 64-bit integers and piece movements can be defined using bitwise operations on these different bitboards.

In a file named `bitboard.c`, write a function with the following signature:

```
unsigned long computePawnMoves(unsigned long pawnPositions,
                               unsigned long whitePieces,
                               unsigned long blackPieces)
```

that:

- accepts three bitboard parameters representing the positions of the white pawn pieces, all the non-pawn white pieces, and all the black pieces on the board, respectively, and
- returns a bitboard representing all the positions that could be occupied after performing some legal white pawn move.

For example, let's return to our original Chess position depicted in the right panel of figure 1. As bitboards, this position would be encoded by `pawnPositions=0x0000000400409900`, `whitePieces=0x0000000000100000`, and `blackPieces=0x0000000000280000`. As we've already seen, figure 2 depicts all possible positions that could be occupied by white pawns after white's turn: `f6`, `a4`, `b4`, `h4`, `a3`, `c3`, `e3`, and `h3`. Thus, in this example, the function

`computePawnMoves` should return the bitboard `0x00000400c1a90000`. Note that the pawn on `e2` is not included in the result, since what we're generating here are *new* white pawn positions after a move — since the pawn `e2` is immobile, we exclude it from our result.

Notes and Constraints

- **This is important:** Your `computePawnMoves` function should be implemented entirely using bit-manipulation operations. In particular, the only types of statements allowed are variable and constant declarations, assignment statements and `return` statements. The only permitted operators are `<<`, `>>`, `~`, `&`, and `|`. Arithmetic operators (`+`, `*`, etc.), loops and `if`-statements are *not* allowed!
- The `io.c` file provides a `prettyPrint` convenience function that accepts a bitboard as a parameter, and prints it out in human readable form (formatted like a Chess board, and indicating occupied and unoccupied squares). You may find this useful when testing your solution.

[15pts] Digital Forensics[2]

I was recently working on a photo-essay when I somehow managed to corrupt the Compact-Flash (CF) card on which all my pictures were stored. Both my laptop and desktop refuse to recognize the card now as having any photos, even though I'm pretty sure I took them. Any computer I connect the card to wants to format it, but, thus far, I've refused to let that happen, hoping instead that someone can come to the rescue. On this problem, your task is to write a program that recovers these photos. (Please!)

How could one accomplish this? The photos on the card are stored in the JPEG format and JPEG files have "signatures", i.e., patterns of bytes that distinguish them from other file formats. In fact, most JPEGs begin with one of two sequences of bytes. Specifically, the first four bytes of most JPEGs are either `0xffd8ffe0` or `0xffd8ffe1`. Odds are, if you find one of these patterns of bytes on a disk known to store photos (e.g., my CF card), it demarcates the start of a JPEG.

Fortunately, digital cameras tend to store photographs contiguously on CF cards, whereby each photo is stored immediately after the previously taken photo. Accordingly, the start of a JPEG usually marks the end of another. The implication of this is that you, my knight

[2]Credit for the assignment idea goes to David Malan, Harvard University

in shining armor, can write a program that iterates over a copy of my CF card, looking for JPEG signatures. Each time you find a signature, you can open a new file for writing and start filling that file with bytes from my CF card, closing that file once you encounter another signature (or the end of the data stream).

Since I have but one CF card, I've created a "forensic image" of the card, storing its contents, byte after byte, in a file called `card.raw`. I've only imaged the first few MB of the CF card, the portion that contains all my photos, so that you don't waste time iterating over parts of the card that contain no data. When executed, your program should recover every one of the JPEGs from `card.raw`, storing each as a separate file in your current folder. Your program should number the files it outputs sequentially, naming them `1.jpg`, `2.jpg`, and so on. You need not try to recover the JPEGs' original names. To check whether the JPEGs your program generates are correct, simply double-click on them and take a look. If each photo appears intact, your operation was likely a success! Make sure to check the corners and boundaries of each image — this will help you catch any off-by-one errors.

The zip archive supplied on Moodle contains the disk image file `card.raw`, in addition to the `io.c` and `io.h` files mentioned earlier. Create a new C file named `cardRecovery.c`. In it, write a function with the following signature:

```
int recoverFiles()
```

that when called:

- opens a binary forensic file named `card.raw`,
- recovers each individual JPEG image from the binary data in the disk image, and
- returns the total number of images that were recovered.

Your function should not print any output to the console.

Notes

- Read the following tutorial to familiarize yourself with how file I/O in C works. In particular, read sections 2, 3, and 5 closely, and skim section 4.

`https://www.cs.bu.edu/teaching/c/file-io/intro/`

You will find the `fopen`, `fclose`, and `feof` functions described in that tutorial particularly useful. When specifying the "mode" for `fopen`, you will want to use `"rb"` or `"wb"`, to indicate that you would like the file opened in "read (binary)"/"write (binary)" mode. Using just `"r"` or `"w"` would attempt to treat the files as regular text (ASCII) files.

- The `fscanf` and `fprintf` functions described in the tutorial are useful when one is dealing with ASCII text files. In this assignment, however, we are dealing with binary files, and that requires a little more care. The supplied `io.c` file abstracts away some of the details of doing byte-level read and write operations, and offers a simpler I/O interface via the `readByte` and `writeByte` functions. Read the documentation attached to these functions to learn how to use them.

- You will be using the `fopen` function to open files — make sure that you also close these files when you're done with them by calling `fclose`. File handles are a scarce resource and having too many files open at once can cause future `fopen` calls to fail. On a related note, always check the return value of `fopen` (as specified in the style guide) to ensure that your attempt to open a file succeeded; if the attempt failed (say, because, the `card.raw` file couldn't be found), your program should print an error message and exit gracefully. You can terminate your program by calling the `exit` function — you'll need to `#include <stdlib.h>` to import this function.

- You should use the `snprintf` function to generate the names of the output JPEG files. This function behaves similarly to the `printf` function — except where the latter prints a specified formatted string to the console, the former "prints" to a string (Aside: strings aren't a special type in C, they're implemented as an array of `char`s. See Section 2.5 from "C for Python Programmers" for more on arrays). Here's an example:

```
char str[10]; // allocates space for a string of length 10
int x = 42;
float f = 3.14;
snprintf(str, 10, "foo %d %f", x, f);
```

In order, the arguments to `snprintf` are: the name of the string (character array) in which you would like to store the result, the length of the string (this should match your array size, and 10 should be sufficiently large for this assignment), a format string (like in `printf`), and the values for the formatting placeholders (again like in `printf`).

Think about how you would use this function to generate file names like `1.jpg`, `2.jpg`, and so on.

- If you'd like to examine the contents of the `card.raw` file via the terminal (say, for debugging purposes), use the `hexdump` command as follows:

  ```
  hexdump card.raw
  ```

  This will print out the file contents byte-by-byte. (Contrast this against the behavior of `cat card.raw` — this will attempt to print out the image file as though it were a text file, and produce illegible results)

- Remember the code style guidelines — your program should contain no magic numbers (like the JPEG signature byte patterns) or other hard-coded constants (like the name of the binary file `card.raw`). Create named constants instead, using the `#define` directive.

Code Style Guidelines

Your code should abide by the guidelines outlined in the *C Style Guide* document that is available at the top of the CSC 250 Moodle page. The following (non-exhaustive) checklist may be handy when preparing your code for submission.

Checklist

- Correctness and Defensive Programming
  - ☐ The files `stats.c`, `bitboard.c` and `cardRecovery.c` implement the specified functions.
  - ☐ The source files compile with no error messages when using the `-Wall` flag.
  - ☐ The programs run without generating any `valgrind` errors.
  - ☐ The programs check the return values of all calls to the library functions listed in the style guide.
  - ☐ All open files are closed before each program terminates.
  - ☐ The behavior of the functions in the programs meets the specifications laid out in this document.
- Documentation
  - ☐ A comment header at the top of each program has a brief description of your program, the names of you and your partner, and the time you spent on the assignment.
  - ☐ Every function has its own comment header that briefly describes its purpose, parameters, and return values.

- ☐ Comments inside functions describe the sub-tasks solved by the function, when appropriate.

- Readability
  - ☐ Variable and function names are meaningful and use camelCase.
  - ☐ Local variables are used to the maximum extent possible, without resorting to unnecessary global variables.
  - ☐ The code is indented neatly and consistently.
  - ☐ Long lines of code (longer than 79 columns) are wrapped to the next line.
  - ☐ "Magic numbers" have been eliminated using the `#define` directive.
  - ☐ Boolean expressions are used in the simplest form possible.
- Structure and Parsimony
  - ☐ The code is free of unnecessary complications and redundant fragments.
  - ☐ The program displays evidence of thoughtful design, with appropriate decomposition and helper functions.

Submission

Submit the files `stats.c`, `bitboard.c` and `cardRecovery.c` via the submission link on Moodle. Only one member of your team needs to do this, though you should remember to attach your partner's name to the submission. If you wish to resubmit a file, simply upload a new version — this will replace any previous submissions.