Assignment 3: Chess Paths

Due date: *Wednesday, December 18, 2019 at 22:00*

> *This is an individual assignment. You may discuss it with others, but your code and documentation must be written on your own.*

In a file called `chess_paths.c` write a C library that implements the three functions `increasing_path_len`, `decreasing_path_len`, and `simple_path_len` declared in the following header file `chess_paths.h`:

```
#ifndef CHESS_PATHS_H_INCLUDED
#define CHESS_PATHS_H_INCLUDED

enum chess_piece { KING, QUEEN, ROOK, BISHOP, KNIGHT, PAWN };

/* Implemented by the application: */
struct chessboard;

int value_at(const struct chessboard * c, unsigned column, unsigned row);
unsigned int columns(const struct chessboard * c);
unsigned int rows(const struct chessboard * c);

struct piece_position {
   enum chess_piece piece;
   unsigned int column;
   unsigned int row;
};

/* Implemented by you: */
unsigned int increasing_path_len(const struct chessboard * c, struct piece_position * p);
unsigned int decreasing_path_len(const struct chessboard * c, struct piece_position * p);
unsigned int simple_path_len(const struct chessboard * c, struct piece_position * p);

#endif
```

Notice that you must implement only the last three path-len functions. In other words, your implementation must simply *use* the functions `value_at`, `columns`, and `rows`, which will be provided by the application or test. In particular, with those three functions, and with an appropriate definition of a **`struct`** `chessboard`, the application defines a chess board with `columns()` columns and `rows()` rows, and where each square has a unique numeric value that can be read using `value_at()`.

Your library must simulate the *path* taken by a chess piece—a king, a queen, a rook, a bishop, a knight, or a pawn—within the given chess board. The pieces move according to the normal rules of Chess. In particular, pawns move in the up direction (see the examples below). The path starts at a given position, and consists of moves that follow a greedy rule depending on the type of path, as detailed below. All the path-len functions must return the length of the path, and must also update the position of the piece to the last position in the path.

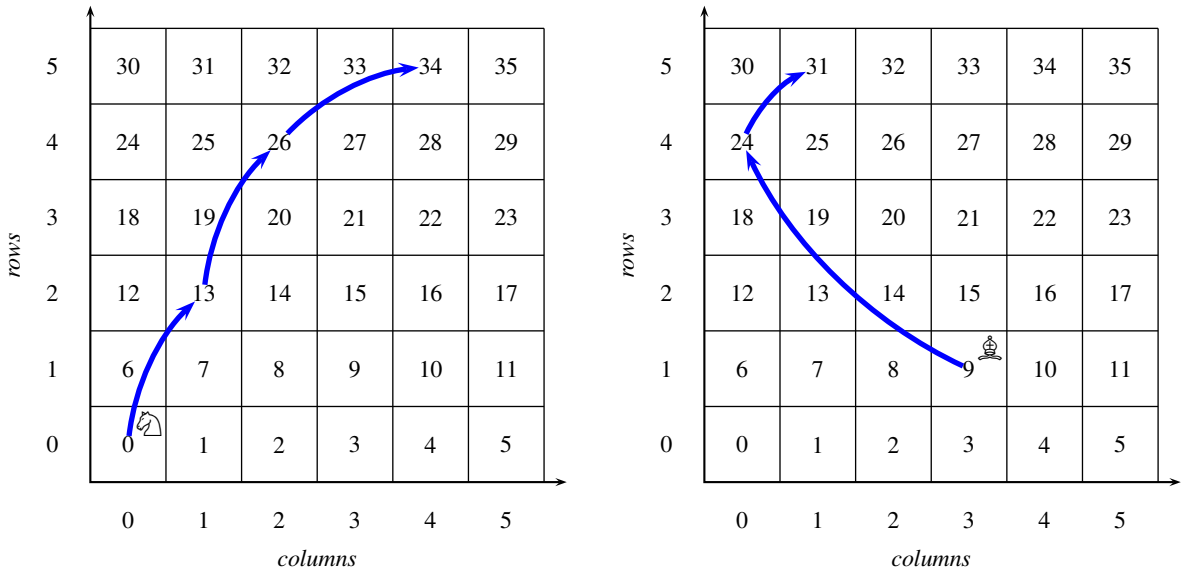
- `increasing_path_len` simulates a greedy value-increasing path. At each step, the chess piece moves to the square reachable from its current position that has the highest numeric value (as determined by `value_at()`). The path ends when the piece has no more legal moves (e.g., a pawn reaching the top row of the board) or when there are no moves that take to a higher value than the current one.

- `decreasing_path_len` simulates a greedy, value-decreasing path. The rules are similar to those given above for `increasing_path_len`, except that the chess piece moves to the square reachable from its current position that has the *lowest* numeric value.
- `simple_path_len` simulates a *simple* path. At each step, the chess piece moves to the square reachable from its current position that the piece has not yet visited in this path. Among the available cells, the piece moves to one with the highest numeric value, even if this is lower than the value at the current position. The path ends when the piece has no more legal moves to squares that were not previously visited.

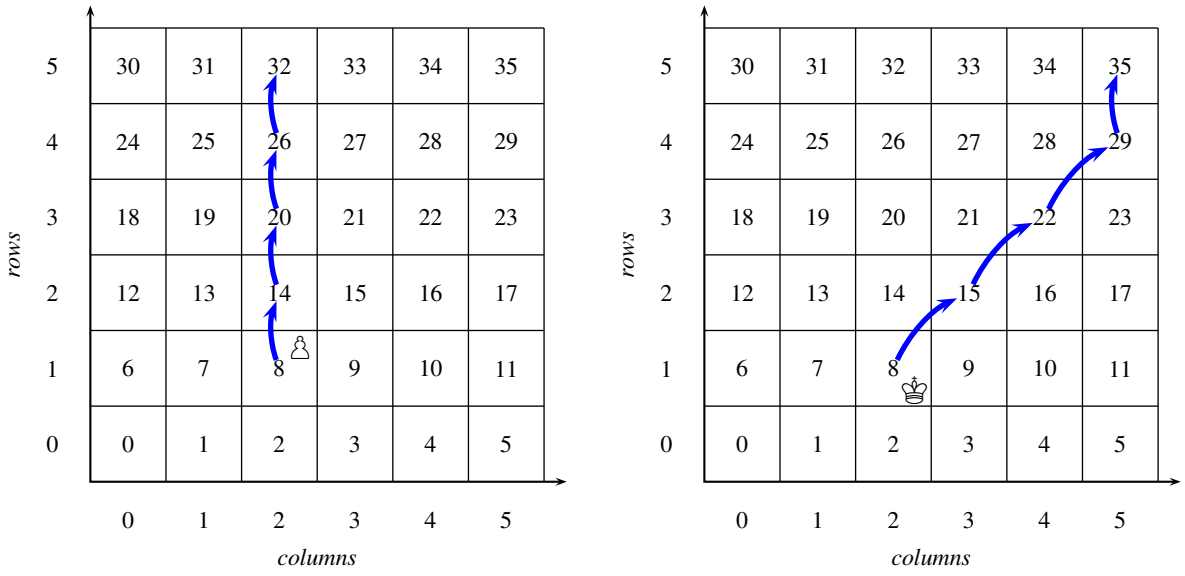
Notice that since the values for each square are *unique,* the choice of move is unambiguous for all types of paths.

Examples

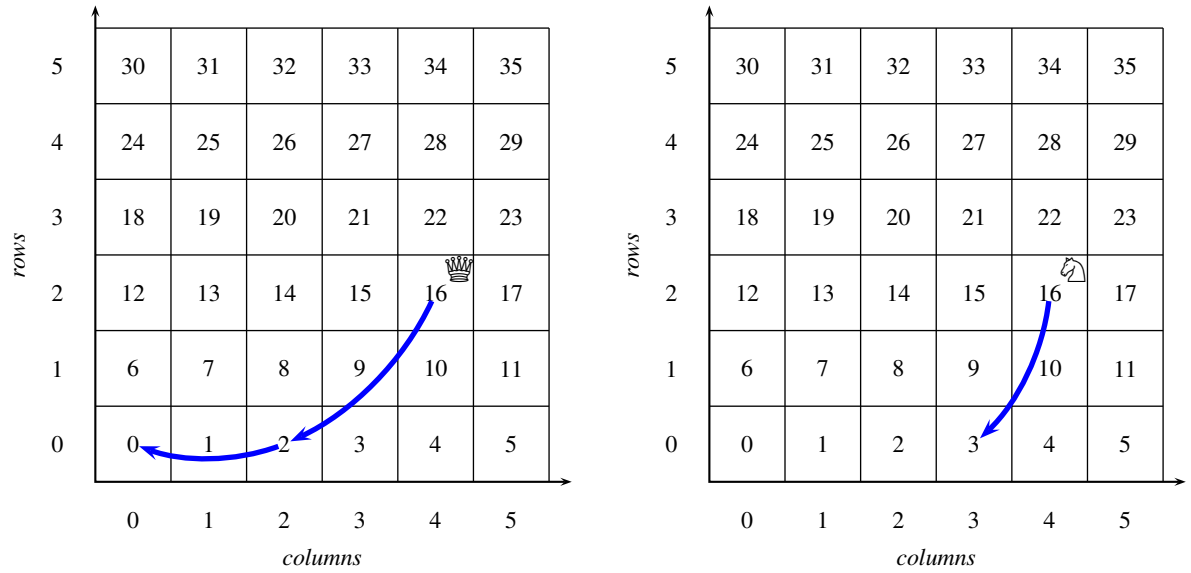
The following are two *increasing* paths on a 6×6 board. On the left is the length-3 path of a Knight starting from column 0 and row 0. On the right is the length-2 path of a Bishop starting from column 3 and row 1



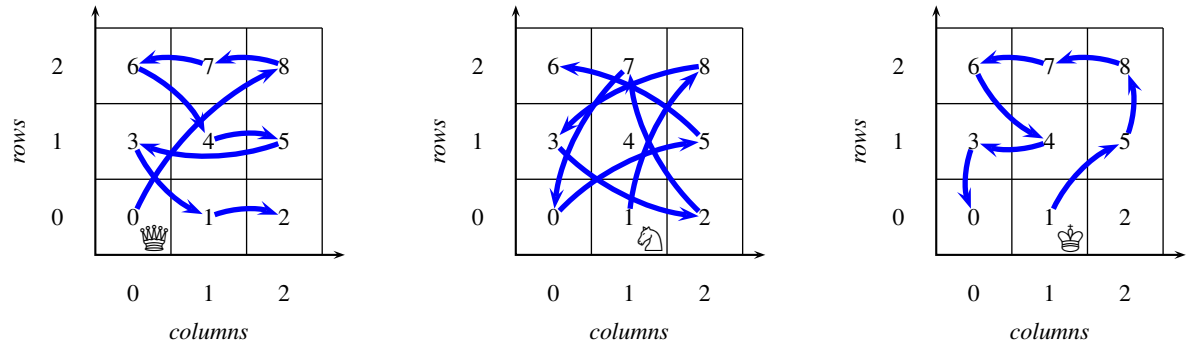
The following are two more *increasing* paths on the same board. On the left is the length-4 path of a Pawn starting from column 2 and row 1. On the right is the length-4 path of a King starting from column 2 and row 1.



Below are two *decreasing* paths. On the left is the length-2 path of a Queen starting from column 4 and row 2. On the right is a one-hop path of a Knight starting from column 4 and row 2.



Below are three *simple* paths. On the left is the path of a Queen starting from column 0 and row 0. In the middle is a Knight starting from column 1 and row 0, and on the right is a King starting from column 1 and row 0. Notice how a move in a simple path is chosen by maximizing the value among those moves that are still available.



Notice that, unlike in increasing paths, in simple paths might go from a higher value to a lower one. In fact, in simple paths, the value is used to choose one among a number of available moves.

Submission Instructions

Submit one source file named `chess_paths.c` through the iCorsi system.. Add comments to your code to explain sections of the code that might not be clear. You must also add comments at the beginning of the source file to properly acknowledge any and all external sources of information you may have used, including code, suggestions, and comments from other students. If your implementation has limitations and errors you are aware of (and were unable to fix), then list those as well in the initial comments.

Use the tests provided with the on-line assignment package. You may also use the program `paths_io` provided in the assignment package to create tests of your own interactively. The input format for `paths_io` is exemplified in a small test file called `test1.in`.

You may use an integrated development environment (IDE) of your choice. However, *do not submit any IDE-specific file,* such as project description files. Also, *make absolutely sure that the file you submit can be compiled and tested with a simple invocation of the standard C compiler.*