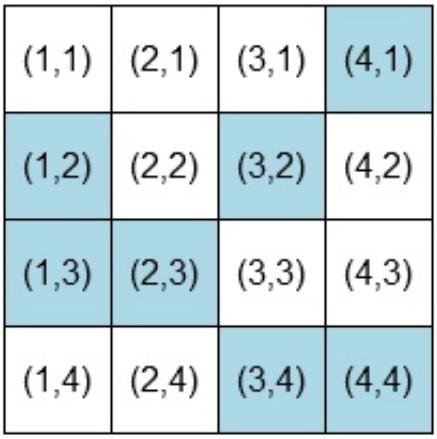
1. Imagine an infinite chessboard. The board begins at (1,1) and streaches out infinitely in two directions. You can think of the positions on the board as pairs of positive integers.

   We say that two positions on the board are *adjacent* iff they share a corner, but not an edge. So positions (1,2) and (2,3) are adjacent, and so are (1,2) and (2,1), but positions (1,2) and (1,3) are not adjacent because they share an edge. (See the power of examples!). We say a position p is adjacent to some set of positions S iff it is adjacent to some position q in S.

   The board also contains some *blocks*. Each block sits at exactly one position on the chessboard, and completely fills that position. Here's a picture of a board with blocks on positions (1,2),(1,3),(2,3),(3,2),(3,4),(4,1),(4,4).



   Here we've represented the positions using computer-graphics coordinates, so all the positions with x = 1 are in the first column.

   A position that contains a block is said to be *occupied*; otherwise it is *vacant*.

   A set of positions is an *obstacle* iff it has the following properties:

   a. The set is non-empty
   b. Every position in the set is occupied
   c. Any two positions in the set are linked by a chain of adjacent blocks in the set.
   d. Every position that is adjacent to any position in the set is either in the set or is vacant.

   Clearly each occupied position on the chessboard is part of a unique obstacle. There are 3 obstacles in the picture: {(1,2),(2,3),(3,2),(4,1),(3,4)}, {(1,3)}, and {(4,4)}.

   The problem is: given the list of the positions on the board that contain blocks, to find the obstacles on the board

   You will have to select suitable data structures and algorithms, in addition to designing the functions. For example, you may or may not choose to represent the board as a list of blocks.

   The API below uses the following data definitions:

```
;; A Position is a (list PosInt PosInt)
;; (x y) represents the position x, y.
;; Note: this is not to be confused with the built-in data type Posn.

;; A PositionSet is a list of positions without duplication.
```

```
;; A PositionSetSet is a list of PositionSets without duplication,
;; that is, no two position-sets denote the same set of positions.
```

For example, `(list (list 1 2) (list 1 3))` and `(list (list 1 3) (list 1 2))` are both PositionSets, but

```
(list
  (list (list 1 2) (list 1 3))
  (list (list 1 3) (list 1 2)))
```

is NOT a PositionSetSet, because both of these position sets denote the same set of positions, namely {(1,2),(1,3)}.

Write a file "obstacles.rkt" that provides the following functions

```
position-set-equal? : PositionSet PositionSet -> Boolean
GIVEN: two PositionSets
RETURNS: true iff they denote the same set of positions.

obstacle? : PositionSet -> Boolean
GIVEN: a PositionSet
RETURNS: true iff the set of positions would be an obstacle if they
were all occupied and all other positions were vacant.

blocks-to-obstacles : PositionSet -> PositionSetSet
GIVEN: the set of occupied positions on some chessboard
RETURNS: the set of obstacles on that chessboard.
```

Before you turn in your solution, make sure it passes the tests in [ps07-obstacles-qualification.rkt](). As before, download this file, save it in your set07 directory, and run it to qualify your program for grading. Be sure to commit this file to repository so we know that you've done this correctly.

For what it's worth, my solution to this problem was 103 lines + tests, and the median time-on-task reported by students the last time I assigned this problem was about 10 hours.

2. Let's go back to that chessboard. On the chessboard, we have a robot and some blocks. The robot occupies a single square on the chessboard, as do the blocks. The robot can move any number of squares north, east, south, or west.

   You are to write a file called `robot.rkt` that provides the following function:

   ```
   path : Position Position ListOf<Position> -> Maybe<Plan>>
   GIVEN:
   1. the starting position of the robot,
   2. the target position that robot is supposed to reach
   3. A list of the blocks on the board
   RETURNS: a plan that, when executed, will take the robot from
   the starting position to the target position without passing over any
   of the blocks, or false if no such sequence of moves exists.

   This API uses the following data definitions:

   ;; A Position is a (list PosInt PosInt)
   ```

```
;; (x y) represents the position at position x, y.
;; Note: this is not to be confused with the built-in data type Posn.

;; A Move is a (list Direction PosInt)
;; Interp: a move of the specified number of steps in the indicated
;; direction.

;; A Direction is one of
;; -- "north"
;; -- "east"
;; -- "south"
;; -- "west"

;; A Plan is a ListOf<Move>
;; WHERE: the list does not contain two consecutive moves in the same
;; direction.
```