

# Iverson computing competition 2016 may 31

name \_\_\_\_\_

school \_\_\_\_\_

city \_\_\_\_\_

grade \_\_\_\_\_

cs teacher \_\_\_\_\_

are you taking AP computer science? (yes/no) \_\_\_\_\_

are you taking IB computer science? (yes/no) \_\_\_\_\_

have you taken advanced level courses? (3000 level, e.g. CSE3110 iterative algorithms I.) (yes/no/currently taking) \_\_\_\_\_

**illegible answers will not be marked**

question	- - - -	marks	your score
1	tiling	11	
2	ascii maze	12	
3	blorks	10	
4	hex	7	
total	- - - -	40	

## Exam Format

This is a two-hour paper and pencil exam. There are four questions, each with multiple parts. Some part(s) might be easy. Solve as many parts of as many questions as you can.

## Programming Language

Questions that require programming can be answered using any language (e.g. C/C++, Java, Python, ...) or pseudo-code. **Pseudo-code should be detailed enough to allow for a near direct translation into a programming language.** Clarify your code with appropriate comments. For full marks, an answer must be correct, well-explained, and as simple as possible.

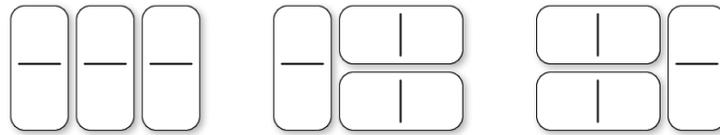
Our primary interest is in thinking skill rather than coding wizardry, so logical thinking and systematic problem solving count for more than programming language knowledge.

## Suggestions

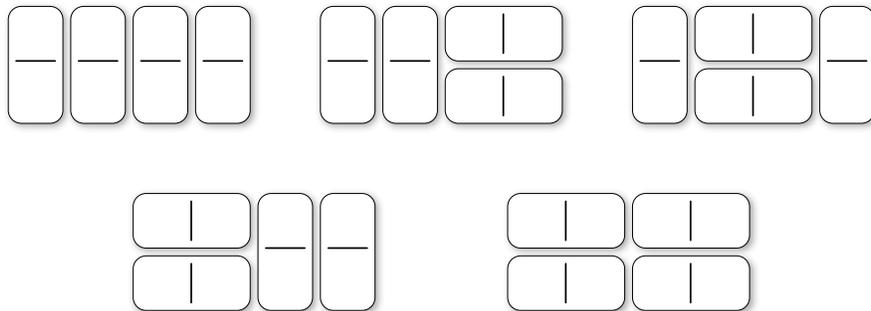
1. You can assume that the user enters only valid input in the coding questions.
2. In some cases, sample executions of the desired program are shown. Review the samples carefully to make sure you understand the specifications. The samples may give hints.
3. Design your algorithm before writing any code. Use any format (pseudo-code, diagrams, tables) or aid to assist your design plan. We may give part marks for legible rough work, especially if your final answer is lacking. We are looking for key computing ideas, not specific coding details, so you can invent your own “built-in” functions for simple subtasks such as reading the next number, or the next character in a string, or loading an array. Make sure to specify such functions by giving a relationship between their inputs and outputs.
4. Read all questions before deciding which ones to attempt, and in which order. Start with the easiest parts of each question.

**question 1: tiling**

We are interested in tiling  $2 \times n$  grids using dominos ( $2 \times 1$  tiles that can be rotated). Here are all ways to tile a  $2 \times 3$  grid.



(a) [1 mark] Draw all ways to tile a  $2 \times 4$  grid.

**Solution**

(b) [2 marks] Let  $f(n)$  denote the number of ways to tile a  $2 \times n$  grid. So  $f(0) = 1$ ,  $f(1) = 1$ ,  $f(2) = 2$ , and  $f(3) = 3$ . Calculate  $f(5)$  and  $f(6)$  (you don't have to draw the actual tilings, but you can if it helps).

**Solution**

$f(5) = 8$ ,  $f(6) = 13$ .

(c) [2 marks] Let  $n \geq 2$ . Suppose we know  $f(k)$  for all values  $0 \leq k \leq n - 1$ . Give a simple expression that easily calculates  $f(n)$  using these known  $f(k)$  values. Justify your answer using at most **two sentences**.

**Solution**

$$f(n) = f(n - 1) + f(n - 2)$$

If the first column is covered by a vertical domino then there are  $f(n - 1)$  different ways to tile the remaining  $2 \times (n - 1)$  grid. Otherwise, the first two columns are covered by two horizontal dominoes and the remaining  $2 \times (n - 2)$  grid can be tiled  $f(n - 2)$  ways.

(d) [2 marks] Write a function `tiling(n)` that returns the value  $f(n)$  for a given integer  $n \geq 0$ . For full marks, it should run very fast: an implementation should finish in a fraction of a second even for values of  $n$  in the 1000s.

### Solution

Two possible solutions are below.

```
def tiling_sol1(n):
    a, b = 1, 1

    # invariant: just before iteration i
    # we have a = f(i), b = f(i+1)
    for i in range(n):
        a, b = b, a+b

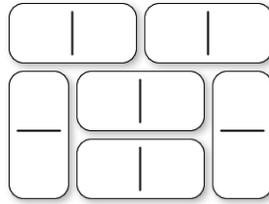
    # postcondition: a = f(n), b = f(n+1)
    return a

def tiling_sol2(n):
    f = [1, 1]
    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
```

Some tried the following approach, but it is very slow. It cannot even compute  $f(50)$  in any reasonable time.

```
def tiling_slow(n):
    if n <= 1:
        return 1
    else:
        return f(n-1) + f(n-2)
```

(e) [2 mark] Now let  $g(n)$  denote the number of ways to tile a  $3 \times n$  grid using  $2 \times 1$  dominoes. Here is a tiling of a  $3 \times 4$  grid.



Some values:  $g(0) = 1$ ,  $g(2) = 3$ ,  $g(4) = 11$ , and  $g(6) = 41$ .

What is  $g(n)$  when  $n$  is odd? Justify your answer.

### Solution

Each domino covers exactly 2 squares so any tiling by dominos can only cover an even number of squares. When  $n$  is odd, a  $3 \times n$  grid contains an odd number of squares. So it cannot be tiled by dominos.

(f) [3 marks] Let  $n \geq 2$  be an even integer. Suppose we know  $g(k)$  for all  $0 \leq k \leq n-1$ . Give an expression that easily calculates  $g(n)$  using these known  $g(k)$  values. Justify your answer.

### Solution

Here are two possible solution.

- $g(n) = 3 \cdot g(n-2) + 2 \cdot g(n-4) + 2 \cdot g(n-6) + \dots + 2 \cdot g(2) + 2 \cdot g(0)$ .

To see this, any tiling either begins with three horizontal dominos which is followed by one of  $g(n-2)$  possibilities. Or it begins with a “wall” (see Figure 1). There are two such walls of length  $k$  for every even  $2 \leq k \leq n$ , and each possible wall is followed by one of  $g(n-k)$  possibilities.

- $g(2) = 3$  and  $g(n) = 4 \cdot g(n-2) - g(n-4)$  for  $n \geq 4$ .

To see this, the number of tilings beginning with three horizontal dominos or a length 2 wall is  $3 \cdot g(n-2)$ . Every other tiling begins with a wall of length at least 4. Such tilings corresponds to tilings of the  $3 \times (n-2)$  grid that starts with a wall (see Figure 2). The number of tilings of the  $3 \times (n-2)$  grid that begin with a wall is exactly  $g(n-2) - g(n-4)$ .

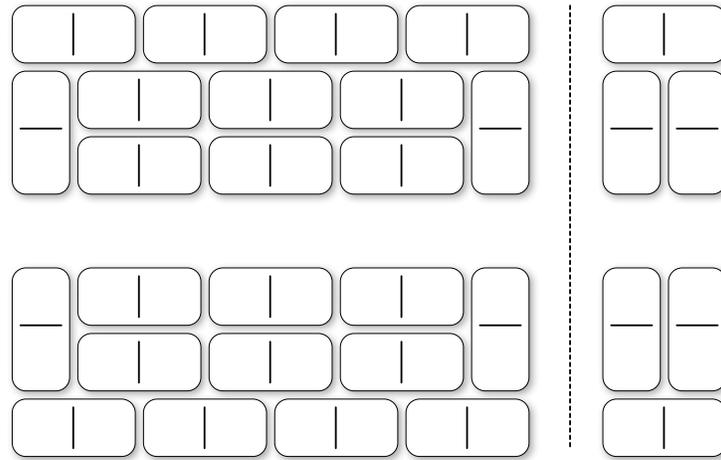


Figure 1: Left: the two walls of length 8. Right: the two walls of length 2.

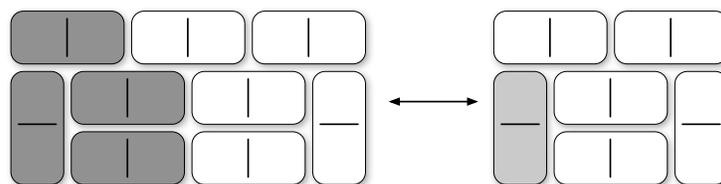


Figure 2: Swapping the dark grey dominos on the left with the single light grey domino shows how to convert between a length  $k$  wall and a length  $k - 2$  wall.

## question 2: ascii mazes

One way to represent a maze using ASCII characters is *|-format*, which uses vertical bars |, underscores \_, and spaces. A rectangular grid using only these three characters is a *valid maze* if

- there are at least 2 rows and at least 3 columns, and
- the top row has only \_ characters, and
- the first character in the second row is a space, indicating the entrance, and
- the last character in the last row is \_, indicating the exit, and
- except for the entrance and exit, the first and last character on each row is |, and
- the bottom row has no spaces.

**Example i**

```

      | _ _ _ |
      | | | | |
      | _ _ _ _ | _ _
    
```

In *|-format*, a grid cell is *vacant* if it is a space, or if it is not on the top row and is \_ . Two vacant cells *join each other* if one is beside the other (up, down, left, or right). Neighbouring vacant cells join each other if and only if no wall separates them.

(a) [1 marks] For this maze, put a dot in each vacant cell, and draw a line between each pair of vacant cells that join each other.

```

      _ _ _ _
      _ |
      | _ |
      | _ _ _
    
```

Another way to represent a maze is with *X-format*. *X-format* uses a space for vacant cells and an X otherwise. The first character of the second row is a space, for the entrance; the last character of the second-last row is a space, for the exit; every other character on the rectangular boundary of this grid is X.

**Example ii**

```

XXXXXXXXXX
  X      X
X X XXX X
X X X X X
X X X X X
  X      X
XXXXXXXXXX
    
```

We can convert from *|-format* to *X-format* by adding extra rows, and placing a space or X between two cells that are on top of each other indicating whether they join each other. The *X-format* example above is what we get by converting from the *|-format* example. Notice that the number of vacant cells can change during this conversion. The maze in (b) is what you would get by converting from the maze in (a).

(b) [1 marks] For this maze, put a dot in each vacant cell, and draw a line between each pair of vacant cells that join each other.

```

      XXXX
      X
      X XX
      X X
      X XX
      X
      XXXX
    
```

(c) [4 marks] Write a function `convert(maze)` that takes an array or list of strings representing a valid maze in |-format and prints the corresponding maze in X-format. For example, `maze` is from (a) then `convert(maze)` prints the maze from (b).

(d) [6 marks] Now write a function `search(maze)` that takes an array of strings representing a maze in X-format. It should print a path from the entrance to the exit using \* characters. You may assume that there is exactly one way to travel from the entrance to the exit using a path that does not visit a cell more than once.

**Example:** calling `search(maze)` with X-format example ii gives this output:

```
XXXXXXXXXX
**X*****X
X*X*XXX*X
X*X*X X*X
X*X*X X*X
X*** X**
XXXXXXXXXX
```

### question 3: borks

A *binary string* contains only 0s and 1. A *bork* is a string containing only characters 0, 1, and \*. A binary string *str* *matches* a bork *blk* if each \* character in *blk* can be replaced with a binary string (possibly empty, and the binary strings do not all have to be the same) so that the resulting string equals *str*. We show such a matching by starting with the bork and then replacing each \* with (b) where b is the needed binary string for that \*.

#### Example

- `str = "01001"`, `blk = "01*01"`, **match** `"01(0)01"`
- `str = "101"`, `blk = "10**1"`, **match** `"10()()1"`
- `str = "10101101"`, `blk = "10*10*"`, **match** `"10(101)10(1)"` and `"10()10(1101)"`
- `str = ""`, `blk = ""` (both strings are empty), **match** `""`
- `str = "11101"`, `blk = "101*"`, **no match possible**

(a) [2 marks] For each of the following, indicate whether it matches. If yes, give one replacement, in the same form as the example.

- `str = "1001"`, `blk = "*1*0*1*"`  
**match:** `"()1()0(0)1()"`
- `str = "110110001010100"`, `blk = "1101*100*10*01*"`  
**match:** `"1101()100(0)10(1)01(00)"`
- `str = "10001101001"`, `blk = "100*11*101"`  
**no match**
- `str = ""`, `blk = "*"`  
**match:** `"()"`
- `str = "100101010100001010001011101101"`, `blk = "*1001*001*11*101"`  
**match:** `"()1001(01010100)001(0100010)11(101)101"`
- `str = "000110101101100010101010101101"`, `blk = "*1101*101010*110"`  
**no match**

(b) [3 marks] Write a function `extract(blrk)` that takes a bork `blrk` as parameter and returns an array or list of strings with the binary substring *pieces* of `blrk`, namely the nonempty binary substrings left over if each `*` is replaced with a space.

Example: for `blrk = "*10**1110*110"` the three pieces are "10", "1110", "110".

### Solution:

You could use a `split` function that is built in to most programming languages. However, you must take care to remove empty strings from the list of substrings.

```
def extract(blrk):
    # split into substrings delimited by '*' character
    # the list comprehension will only use those that are nonempty
    substrings = blrk.split('*')
    return [str for str in substrings if str != '']
```

Without the `split` function, you could add the characters of `blrk` to a buffer, one character at a time. If you encounter the `*` character then the buffer string could be added to the output list (provided the buffer string is not empty).

```
def extract_v2(blrk):
    buf = ''
    substrings = []

    # ensures the last substring of blrk will be processed
    blrk += '*'

    for c in blrk:
        if c == '*':
            if buf != '':
                substrings.append(buf)
            buf = ''
        else:
            buf += c

    return substrings
```

(c) [5 marks] Write a function `match(str, blrk)` that takes a binary string `str` and bork `blrk` as parameters and returns `true` if `str` matches `blrk` and `false` otherwise. If it is helpful, you may use the function described in part (b) even if you did not answer that question.

### Solution

The idea is to find the earliest occurrence of each binary substring of `blrk` that appeared after the occurrence we recorded for the previous binary substring. If we do not find all binary substrings this way, there is no match. We also have to make sure the first substring of `blrk` occurs as a prefix of `str` if `blrk` does not begin with `*` (and similarly for the last substring). Care must be taken if `blrk` contains no `*`.

```
def match(str, blk):
    if '*' not in blk:
        return str == blk

    # here blk has at least one *
    substrings = extract(blk)
    if substrings == []:
        return True # blk has only *, and at least one, so any string matches

    # if required, ensure the start and/or end of str is the appropriate binary string
    if blk[0] != '*' and str[:len(substrings[0])]:
        return False
    if blk[-1] != '*' and str[-len(substrings[-1])]:
        return False

    for sub in substrings:
        # will find the least index of str where sub appears as a substring
        loc = str.find(sub)
        if loc == -1:
            return False
        else:
            # remove all characters of str up to the end of the occurrence of sub
            str = str[loc + len(sub):]

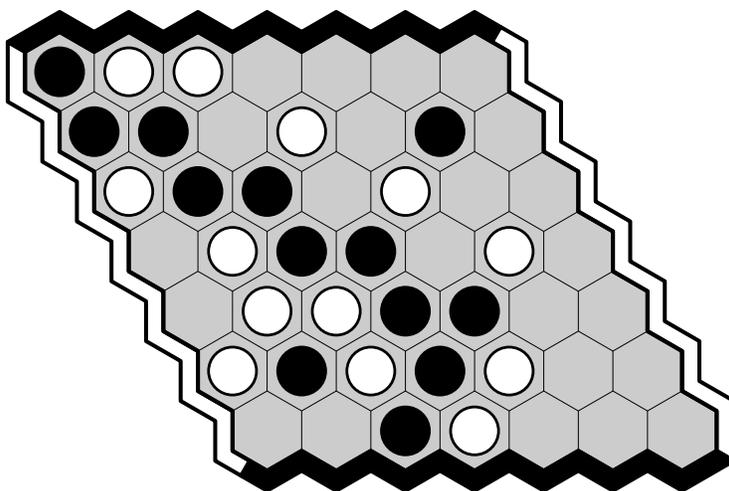
    return True
```

## question 4: hex

**Warning:** this question can take time. Budget your time wisely.

The two-player game of Hex is played on an  $n \times n$  board with hexagonal cells. Below is a  $7 \times 7$  board. Players alternate turns. On a turn, a player puts a stone on an empty cell.

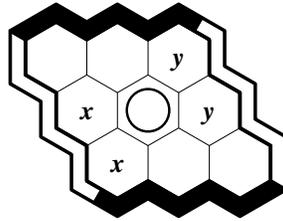
The first player is called *white* and places white stones. The second player is called *black* and places black stones. Two opposing sides of the board are white and the other two opposing sides are black. The winner is whoever connects their two sides with a connected path of their stones. In the example below, black has won.



An amazing property of Hex is that if the board is completely covered with stones then exactly one player has joined their two sides. So draws are not possible.

This question is about playing perfectly in Hex. We say that a player plays *perfectly* if, on each move, if there is some move that is part of a winning strategy for that player, then the player makes such a move. So, whoever can win always makes a winning move; whoever cannot win can play anywhere.

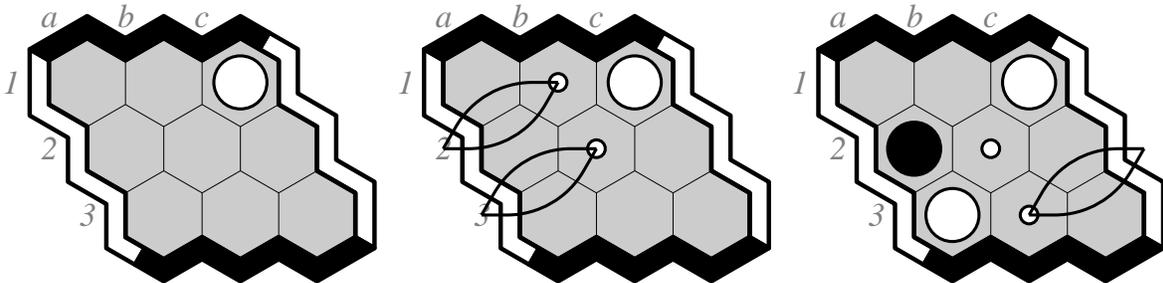
(a) [3 marks] On this  $3 \times 3$  board white played her first move in the middle. (The letters are for the analysis below.) Now it is black's turn. We claim that white can win.



White can guarantee she gets at least one of the cells labelled  $x$  by placing a stone on one of these two cells if black ever places a stone on the other. Similarly, white can guarantee she gets at least one of the cells labelled  $y$ . But each  $x$  cell touches the left side and the middle, and each  $y$  cell touches the middle and the right side. So white will join her two sides if she follows this strategy.

Now assume that white starts a game by by placing her first stone as shown below. It is black's turn. Who will win the game if both players play perfectly from now on? Justify your answer by giving the winning player's strategy **as concisely as possible**.

(We have included a page with blank  $3 \times 3$  hex grids on the second last page of this exam. You can use these to help describe the strategy, just make sure that we can follow your reasoning.)



**Solution**

(above middle) If black's 1st move is not in the set  $\{a1, a2, b1\}$  then white wins by playing  $b1$ , and then on the next move one of  $\{a1, a2\}$ . Similarly, if black's 1st move is not in  $\{a2, a3, b2\}$  then white wins by playing  $b2$  and then on the next move one of  $\{a2, a3\}$ . So white wins unless black's 1st move is in *both* sets, so  $a2$ .

(above right) Now white wins by playing  $a3$ , and then on the next move either  $b2$  or (if black is already there)  $b3$ , and then on the next move either  $c2$  or  $c3$ .

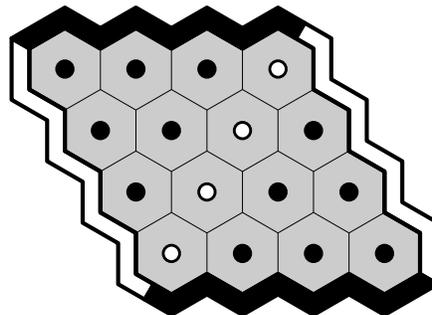
(b) [4 marks] For **each** cell on the  $4 \times 4$  board below, show who wins (white or black) if that cell is white's first move, and both players play perfectly from then on: if white wins, then on that cell draw an empty circle (or the letter W); if black wins, then on that cell draw a filled circle (or the letter B).

You do not need to justify your answer: your score will be calculated as follows:

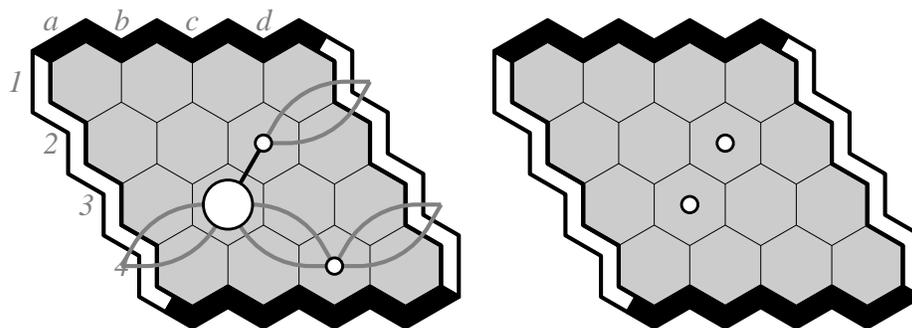
$$\max \left\{ \frac{(\# \text{ of correct answers}) - (\# \text{ of incorrect answers})}{4}, 0 \right\}.$$

Cells left blank will be counted neither correct nor incorrect. Illegible entries will be counted incorrect. (Some empty  $4 \times 4$  grids can be found at the end of this exam. These are for your scratch notes only and will not be looked at when we grade.)

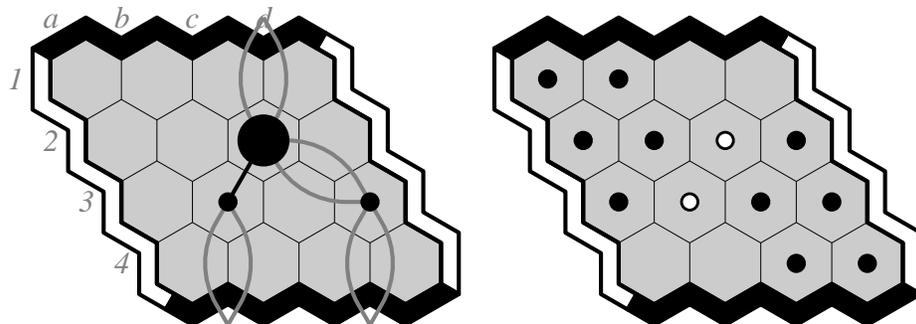
**Solution**



White wins by playing b3, as shown (below left). By a symmetric strategy (reflect the board through the centermost point) white also wins by playing c2. So far we have two winning openings.

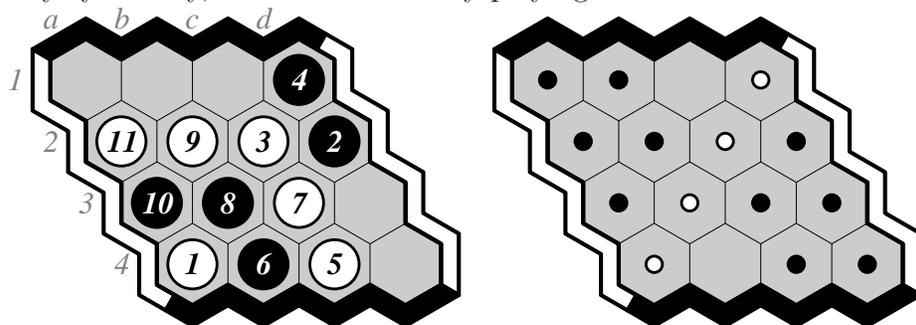


If white plays first in any of the unmarked cells below left, then black can win. By also considering the symmetric black strategy, we now have 10 losing openings (below right).

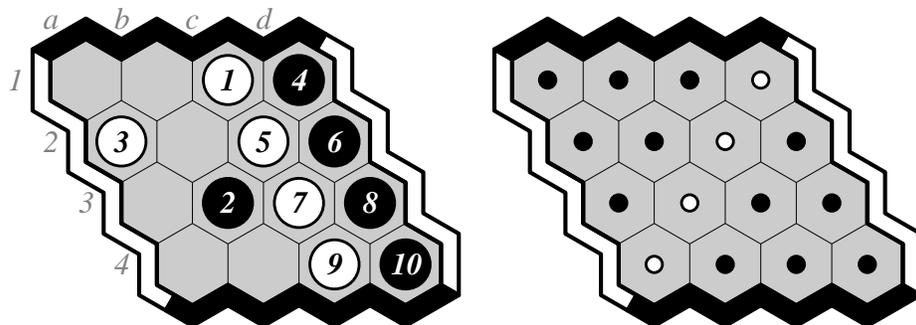


White can win by playing first at a4. If black's reply is not in  $\{b3, b4, c3, d2, d3\}$  then white wins by next playing c3. If black's reply is in one of these 5 cells, then white wins by playing c2. This is easy to see if black played in  $\{b3, c3, d3\}$ , and not too hard to see if black played in  $\{b4, d2\}$ . One such line of play is shown below.

By symmetry, White also wins by playing first at d1.



There are only two opening moves left, and they are symmetric. If white plays first at c1 (below left), black can win: the main variation is shown, this is not a complete proof. By symmetry, if white plays first at b4, black wins. So we are done.



PS. Ryan Hayward is writing a book on Hex. If you would like to be notified when it is published, send him email: [hayward@ualberta.ca](mailto:hayward@ualberta.ca) .



this page will not be graded

