ECOO 2014

Programming Contest Solutions and Notes

Regional Competition (Round 2)

April 26, 2014

Sam Scott, Sheridan College (sam.scott@sheridancollege.ca)

Problem 1: Scratch and Win

Recommended Approach

The mechanics of this question are simple – just make a counter for each type of prize and one for the blanks. Then it's all about anticipating all the possibilities. First of all, if any card is showing a win already, then that is the only possible win (no card can win twice). If there is no win showing, then there are different cases depending on the number of blanks. If the card has no blanks, it's a loser. If there is one blank, you are looking for doubles. If there are two blanks, then any symbol that is showing could be a winner. Finally, if there are three or more blanks, any prize is possible.

Test Cases

The 10 test cases in each set were constructed to cover all the possibilities described above.

Problem 2: Black and White

Recommended Approach

First, you have to find all the factors that divide evenly into the board size, then simulate the tile flipping starting with the smallest factor and ending with the largest. But if you try to represent the entire board, you will run out of time or heap space on the larger boards. Since you only have to output the result for 5 tiles, you can avoid this by figuring out at each step whether each of the tiles should be flipped.

Suppose you are looking at the tile at row 4 and column 7 (rows and columns numbered from 0) on a 10x10 board, and you are at the stage where you are using 2x2 checkerboard squares. This checkerboard is 5x5 and you can figure out which row and column your tile is in using integer division.

4 div 2 = 2 7 div 2 = 3

So now you know that on the 5x5 checkerboard, your tile is in row 2 and column 3. Is that a black or a white checkerboard square? There are a number of ways to determine this, but one cute trick is to add up the row and column. If the result is even, the square is black. Since 2+3 = 5, the tile is in a white checkerboard square and should not be flipped on this round.

Problem 4: What Lies Ahead

Recommended Approach

These puzzles are small enough that they can be solved with any kind of search algorithm (breadth-first, depth-first, etc.). We performed a depth-first search using a recursive backtracker, marking squares visited when we passed them (to avoid loops) and unvisited when we backtracked. The wrinkle here is that you have to keep track of your orientation to get the legal set of moves, and you also have to keep track of the orientation when marking a square as visited. For example, if you visit square 2,2 facing upwards, that is different than if you visit it facing right.

There are a number of ways you can keep track of this visited list, but the most efficient will make use of a grid where each grid has some way of setting a flag or marker for each of the four orientations.

One way to do this is to have each grid square be an int that starts off as zero. Then you can designate one bit in the underlying binary representation for each direction. For example, the least significant bit (the "1" bit) could be up, then the next ("2") could be down, then the next two ("4", "8") could be the left and right.

If you are at location (2,2) facing down and you want to check if you've been here before, use the "bitwise" AND operation available in most languages (this would be `visited[2][2] & 2` in Java). To mark the square as visited, facing up, add 2. To unmark it, subtract 2.

Example: You're at square (2,2) and you've already been here facing left and right. This means that `visited[2][2]` must be 12 (4+8) which is 1100 in binary. You do a bitwise and with 2 to see if you've been here before facing down. Down is 2 which is 0010 in binary. Bitwise and does an AND operation on every bit in the two operands, so the result is 0, which means you haven't been here before. You add 2 to `visited[2][2]` and now you get 14, which is 1110 in binary. If you did bitwise AND again, the result would be 2 instead of 0.

http://en.wikipedia.org/wiki/Bitwise_operation