Code Jam 2016 - Round 2

Analysis: The Gardener of Seville

The Gardener of Seville: Analysis

Small dataset

For the Small dataset, there are at most 16 cells in the courtyard that we must assign hedge directions to, a total of at most 65536 different hedge mazes. Note that because every cell must have a diagonal hedge, any maze creates a bijection between courtiers; it is impossible for the paths used by two different pairs of lovers to intersect, so we don't need to worry about that. We can use brute force to generate and check all possible hedge mazes, as long as we can efficiently work out which outer cells are paired up through the hedge maze. This can be done in various ways, including:

- Consider hedges to be mirrors, and imagine shining a beam of light into the maze from an outer cell, perpendicular to the edge it's on. At each cell the light will reflect off the cell's mirror at a 90 degree angle and continue to the next cell in its new direction. It repeatedly bounces off mirrors until it exits the maze at the cell it's paired with.
- Imagine drawing both hedges in each cell, splitting each cell into four quadrants. We can represent the maze as a graph in which the nodes are these quadrants, and there are edges between adjacent quadrants that are not blocked by part of a hedge. We can find which courtiers are paired by starting at the node corresponding to that courtier's starting edge of the maze, and traversing the graph until we reach another edge. Equivalently, we can find the connected components of the graph.

Large dataset

The Large dataset has test cases with up to 100 cells, which is too large for our brute force approach. A more fruitful approach is to take the given pairs and install hedges so that those pairs are connected. The easiest cases would be connecting two cells which are adjacent (either along an edge or around a corner). For example, to connect the two outer cells adjacent to the upper left corner, a single `/` hedge is sufficient. Does it ever make sense to connect them any other way? The illustration for test case 3 from the sample input connects the two outer cells adjacent to the upper right corner via a longer winding path, but it would also work to make this connection direct and leave the center of the garden unreachable. The direct connection covers just two triangular quadrants (as defined in the Small dataset section) in the corner, and it is easy to see that any other possible path between the two cells also covers these two quadrants. Thus there is no reason not to use this direct path if we need to connect the cells at a corner.

How about outer cells next to each other along an edge? These can be easily connected using two hedges, which covers four triangular quadrants. All possible connections will necessarily cover the two quadrants at the edge of the board, but we can construct paths which do not cover the other two quadrants. However, any path that does not use those two quadrants is guaranteed to block them off and make them inaccessible from other edges. Thus, any path between the two cells will either cover the four quadrants forming the simplest path, or render some of them unusable; there is no reason to use anything more complex than the simplest path.

If every pair to connect has a similarly optimal path which we can easily determine, then we can solve the problem by installing hedges so each pair is connected via its respective optimal path, and if any of the paths intersect then there is no solution. Consider, however, a pair between outer cells on the left and right sides of the garden. Depending on the other connections we need to make, we may be able to freely choose between (for example) having the path go through the top half of the garden and the bottom half of the garden. As such, there isn't a clear single optimal path for connecting this pair. However, we can consider uppermost and lowermost paths, which leave the most space for paths below and above them, respectively. For an uppermost path, for example, we want to take the least space possible to connect the pair, and all the pairs above it. turns out there is a optimal way to connect such pairs.

For the rest of the analysis, we will assume that there is a solution to the problem. If there is a solution, then our strategy will provide a way to find it. If there is no solution, our strategy may not be correct but we can easily detect that it fails by checking the hedge maze as we did in the Small solution.

Define a 'group of pairs' as a non-zero number of pairs where all the outer cells used form a fully contiguous section around the perimeter (but not the full perimeter). For a pair connecting the left and right sides of the garden, we can consider the group of pairs above the path connecting this pair, and the group of pairs above and including this pair. Every group of pairs has an optimal set of triangular quadrants to join all pairs in that section. Similar to before, optimality here means that we can install hedges to connect each pair in the group without covering quadrants outside the optimal set (this property is sufficiency), and if all pairs in the group are connected, no paths from other pairs can ever cover quadrants inside the set (this property is necessity).

We already know the optimal sets for the groups with a single pair of adjacent outer cells. If we have two groups of pairs, which together would form a larger group of pairs, the optimal set will be the union of the optimal sets for the two smaller groups. We can prove that this meets both the sufficiency and necessity properties (as stated before, this requires the assumption that there is a solution).

Consider again the case of a pair connecting the left and right sides of the garden. If we have the optimal set for the group of pairs above this pair, then we can try and extend this to the optimal set for the group of pairs above and including this pair. It makes sense to try and make the path for this pair as high up as possible, staying as close as possible to the paths above it. It can be proven that including this path makes a new optimal set. In general, this works for any non-adjacent pair. If we have the optimal set of quadrants for the group of pairs on one side of a pair, we can extend it by adding a path that stays as close to those quadrants as possible. This means we can inductively find optimal sets of quadrants until we cover all the pairs (note that we did not define all pairs as a valid group of pairs, as the definition of optimality doesn't work for that case).

These ideas give us the following algorithm:

- Start with no hedges in the garden
- Iterate over pairs, in increasing order of distance (along the perimeter) between the two cells. Ties can be broken arbitrarily.
    - Let the two outer cells be A and B, such that A→B clockwise around the edge is shorter than counterclockwise. Due to the chosen iteration order, we've already built paths for all points on the left side of the A→B path we're going to construct.
    - Walk through maze starting at A (the mirror analogy is useful here). We want to stay to the left as much as possible, so if we get to a cell without a hedge installed we pick one so that we turn left. Once we exit the maze, check if we actually made it to B. (If there is no solution we might end up somewhere else.)
- Fill in remaining cells arbitrarily

A sample implementation of this in Python is provided below. We encode directions with integers, which allows us to rotate direction and calculation movement easily using bitwise operations and array lookups.

```
def position(v, R, C):
    # Map from outer cell number to a direction facing into the maze
    # and the position of the outer cell
    # 0->downwards, 1->leftwards, 2->upwards, 3->rightwards
    if v <= C: return 0, v-1, -1
    v -= C
    if v <= R: return 1, C, v-1
    v -= R
    if v <= C: return 2, C-v, R
    v -= C
    return 3, -1, R-v

def move(x, y, direction):
    return x + [0,-1,0,1][direction], y + [1,0,-1,0][direction]

def solve(R, C, permutation):
    board = [[None] * C for _ in range(R)]
    size = 2*(R+C)
    permutation = zip(permutation[::2], permutation[1::2])
    permutation.sort(key=lambda(a,b): min((b-a)%size, (a-b)%size))
    for start, end in permutation:
        if (start-end) % size > R+C:
            start, end = end, start
        direction, x, y = position(start, R, C)
        x, y = move(x, y, direction)
        while 0<=x<C and 0<=y<R:
            if board[y][x] is None:
                board[y][x] = "/\\"[direction & 1]
            direction ^= {"/": 1, "\\": 3}[board[y][x]]
            x, y = move(x, y, direction)
        if (x, y) != position(end, R, C)[1:]:
            return "IMPOSSIBLE"
    return "\n".join("".join(c or "/" for c in row) for row in board)

if __name__ == "__main__":
    for t in range(1, input() + 1):
        R, C = map(int, raw_input().split())
        permutation = map(int, raw_input().split())
        print "Case #%d:" % t
        print solve(R, C, permutation)
```