Take the Right Turn!

created by Lisa Wang

Source: http://commons.wikimedia.org/wiki/File:No_Left_Turn.JPG.

Intro:

The concert of your favorite band starts in three minutes and you are still caught in your car, nervously waiting for your left turn. Filled with envy, you are counting the cars passing by to your right. You get the feeling that waiting for a left turn takes a hundred times longer than waiting for a right turn.

In midst of your desperation, you suddenly have an idea: What if you preferred right turns over left turns, thus allowing a sometimes longer, but faster route?

After the concert you feel so energized that you decide to write the first version of your personal route planner that will eventually beat your old GPS. Since you've just learned how to solve a maze using recursive backtracking, this problem comes at the right time. Green light for you!

Main Idea:

Instead of finding the shortest path from A to B, we will try to find the cheapest path. We assign costs[1] to each of the following steps:

- turn left : 15
- turn right : 2
- go straight : 1

[1] This is based on the assumption that taking a left turn takes on average 15 times longer than going straight, and a right turn takes on average twice as long as going straight. Of course, you could also assign other costs to the steps.

Today's Challenge:

You are given a world, represented as a char grid where '0' stands for a cell that is part of a street, and 'X' stands for a cell that is part of a building.

Take a look at the world to the right. Your car is in the start cell, facing West.

X	X	0	0	0
X	X	0	X	0
0	0	0	0	0
0	X	0	X	X
0 END	X	0	0	0 START

a) Find the cheapest path from start to end.
b) Find the shortest path from start to end.

	Cost	Length
Cheapest Path		
Shortest Path		

c) Finding the cheapest path in a world of streets and buildings is reminiscent of the the maze problem. Discuss with your partner in what ways these two problems are similar or different.
Similarities:_____

_____

Differences:

Maze Problem	**Cheapest Path Problem**

d) Implement the function `findCheapestPath`.
(Pages 5 and 6 contain useful data structures, enum types, constants and helper functions for reference. Feel free to tear those out)

```
/* Function: findCheapestPath
 * Usage: int cost = findCheapestPath(start, end, curOrientation, path, world, markers);
 * ------------------------------------------------------------------------------------
 * returns the cost of the cheapest path, or -1 if no path exists.
 * start : the cell the car is currently at.
 * end : the cell we would like to drive to.
 * curOrientation : specifies which way the car is currently facing (NORTH, EAST, ...).
 * path : used to "return" the path associated with the least cost, therefore
 * passed by reference. path is a vector that contains the directions, e.g.
 * [STRAIGHT, RIGHT, RIGHT, LEFT]
 * world is the grid that contains 'X' for a building or '0' for a street.
 * markers is a grid of ints that keep track how often we have passed a cell. The
 * coordinates of the markers grid correspond to the world grid.
 */
int findCheapestPath(Cell start, Cell end, Orientation curOrientation,
                     Vector<NextDirection> & path, Grid<char> & world, Grid<int> & markers)
```

Craving for more?

http://compass.ups.com/UPS-driver-avoid-left-turns/

http://www.businessinsider.com/save-money-gas-avoid-left-turns-2012-3

(Page intentionally left blank so you can tear out reference pages 5 and 6)

```
/* Cell
 * ----
 * like a Point struct, summarizes the row and the column of a cell
 * inside the grid world.
 */
struct Cell {
    int row;
    int col;
};

/* Orientation
 * -----------
 * The current orientation of the car (driving direction)
 */
enum Orientation {
    NORTH = 0,
    EAST,
    SOUTH,
    WEST
};

/* NextDirection
 * -------------
 * note that we only consider three possible directions
 * as it does not make sense to go back.
 */
enum NextDirection {
    LEFT = 0,
    STRAIGHT,
    RIGHT
};


/* NextDirection operator++
 * Usage: for (NextDirection dir = LEFT; dir <= RIGHT; ++dir) {...}
 * ----------------------------------------------------------------
 * We overload the ++ operator so we can easily loop through all
 * possible directions.
 */
NextDirection operator++(NextDirection & dir) {
    dir = NextDirection(dir + 1);
    return dir;
}


/* Constants */
const int leftTurnCost = 15;
const int straightCost = 1;
const int rightTurnCost = 2;
```

```
/* Function: computeNewOrientation
 * Usage: Orientation newOrientation = computeNewOrientation(SOUTH, LEFT);
 * ----------------------------------------------------------------------
 * Returns the new orientation (NORTH, ...) after the car has taken the next
 * action specified by dir.
 */
Orientation computeNewOrientation(Orientation curOrientation, NextDirection dir) {
    Orientation newOrientation;
    switch (dir) {
        case LEFT:
            newOrientation = Orientation((curOrientation+3)%4);
            break;
        case STRAIGHT:
            newOrientation = curOrientation;
            break;
        case RIGHT:
            newOrientation = Orientation((curOrientation+1)%4);
            break;
    }
    return newOrientation;
}

/* Function: computeAdjacentCell
 * Usage: Cell adjacentCell = computeAdjacentCell(curCell, SOUTH);
 * ---------------------------------------------------------------
 * returns the next Cell after the car moves one step further in the new
 * orientation. E.g. if the new orientation is NORTH, it moves one step further
 * to NORTH and the returned Cell specifies the new position of the car.
 */
Cell computeAdjacentCell(Cell start, Orientation newOrientation) {
    Cell newCell;
    switch (newOrientation) {
        case NORTH:
            newCell.row = start.row - 1;
            newCell.col = start.col;
            break;
        case EAST:
            newCell.row = start.row;
            newCell.col = start.col + 1;
            break;
        case SOUTH:
            newCell.row = start.row + 1;
            newCell.col = start.col;
            break;
        case WEST:
            newCell.row = start.row;
            newCell.col = start.col - 1;
            break;
    }
    return newCell;
}

/* Function: computeActionCost
 * Usage: int cost = computeActionCost(RIGHT);
 * -------------------------------------------
 * returns the cost of the specified action.
 * The costs are declared as constants at the top of the file.
 */
int computeActionCost(NextDirection dir) {
    switch (dir) {
        case LEFT: return leftTurnCost;
        case STRAIGHT: return straightCost;
        case RIGHT: return rightTurnCost;
    }
}
```