CSCI-1200 Data Structures — Spring 2020
Homework 6 — Ricochet Robots Recursion

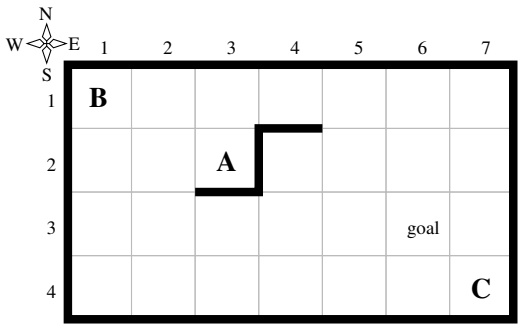
Your task for this homework is to solve robot movement puzzles using the technique of recursion. This homework is inspired by the board game "Ricochet Robots":

http://en.wikipedia.org/wiki/Ricochet_Robot
http://riograndegames.com/Game/163-Ricochet-Robots

Understanding the non-linear word search program from Lecture 12 will be helpful in thinking about how you will solve this type of problem. We strongly urge you to study and play with that program, including tracing through its behavior using a debugger or `cout` statements or both. *Please carefully read the entire assignment and study the provided code before beginning your implementation.*

You will be given a two-dimensional play space and one or more robots that move around within the space. Each robot has a starting position on the board and moves by sliding in the 4 primary directions: north, east, south, or west. The input file will also specify one or more goal locations that must be simultaneously occupied by robots to "solve" the puzzle. The challenge is to navigate the robots to the goal position(s) in the fewest total moves. The catch is that when a robot is directed to move, it *continues to slide until it bumps into either a wall or another robot.* To solve the puzzle, the robot *must come to rest on the goal position.* It does not count as a solution if the robot simply slides over and past the goal location. The input file will indicate that either a specific robot must reach the goal, or that the puzzle can be solved with any robot at that goal. First, let's look at a sample input file, `puzzle1.txt`.

```
4 7
robot A 2 3
robot B 1 1
robot C 4 7
vertical_wall 2 3.5
horizontal_wall 1.5 4
horizontal_wall 2.5 3
goal any 3 6
```



First note that grid locations begin with (1,1) in the upper left corner. The input file specifies the overall size of the grid. The grid is always bounded by walls (thick black lines in the diagram above) on the outer edges of the grid. Interior horizontal and vertical walls are specified with half grid cell coordinates. The goal of this particular puzzle is to navigate *any* robot to the specified goal location at row=3, column=6.

Your program should expect one or more command line arguments, e.g.:

```
./richochet_robots puzzle1.txt -max_moves <#> -visualize <which_robot>
./richochet_robots puzzle1.txt -max_moves <#>
./richochet_robots puzzle1.txt -max_moves <#> -all_solutions
./richochet_robots puzzle1.txt -all_solutions
```

The first argument specifies the name of the input puzzle file. To begin, we recommend that you use the optional argument `-max_moves`, which places a cap on the maximum number of moves that will be searched to find solutions. We also recommend that you first work on a visualization of the reachability — the possible places the specified robot could land in less than or equal to the specified number of moves.

For example, if we run these 3 separate commands:

```
./richochet_robots puzzle1.txt -max_moves 3 -visualize A
./richochet_robots puzzle1.txt -max_moves 3 -visualize B
./richochet_robots puzzle1.txt -max_moves 3 -visualize C
```

The program outputs reachability data for each of the robots in the scene:

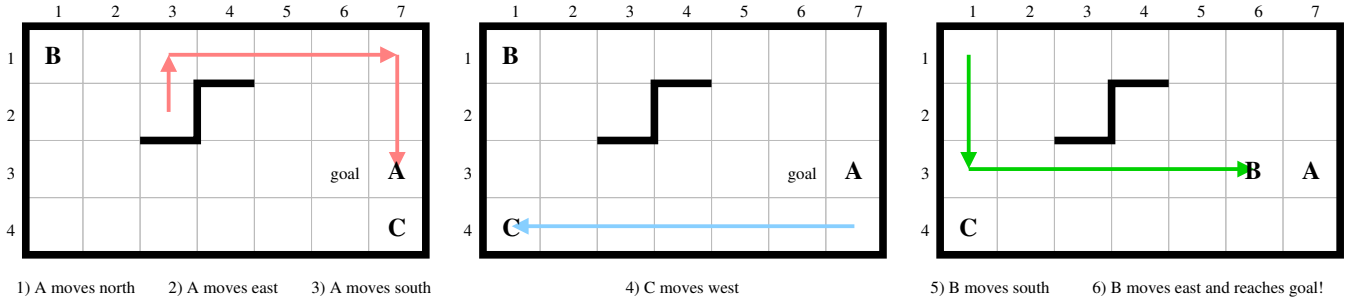
```
Reachable by robot A:
  3  2  1  .  .  3  2
  1  3  0  .  .  .  .
  3  .  .  .  .  .  3
  2  3  .  .  .  3  .
```

```
Reachable by robot B:
  0  2  .  3  .  2  1
  .  .  .  .  .  .  .
  2  .  .  .  .  .  2
  1  3  .  .  .  2  3
```

```
Reachable by robot C:
  3  2  .  3  .  .  1
  2  3  .  3  .  .  2
  3  .  .  .  .  .  .
  1  2  .  .  .  .  0
```

The starting point of the robot is labeled 0, because no moves are necessary to position the robot at this location. All locations that can be accessed with just one robot move are labeled 1, all locations that can be reached in 2 moves (by the same or different robots) are labeled 2, etc. If the location cannot be reached within the specified number of moves, the period character is placed at that location.

When your movement visualization is working, the next step is to search for the one or more solutions to the puzzle. The sample puzzle shown earlier can be solved in 6 moves:



There are actually 24 different 6 move solutions to this puzzle, and there are no solutions with fewer than 6 moves. When the `-all_solutions` command line argument *is not* specified, any of the shortest legal solutions may be output, and the board should be printed after each move. When the `-all_solutions` command line argument *is* specified, the number of solutions is output, and then all of the shortest solutions should be output, in any order. The intermediate board states are not printed when all solutions are output.

All program output should be sent to `std::cout`. If the puzzle is impossible your program should output "`no solutions with XX or fewer moves`", where XX is the specified maximum number of moves to be searched, or "`no solutions`", if a maximum number of moves was not specified. Please match the sample output formatting exactly (except for choice of single output solution and order of all solutions).

Provided Code, Additional Requirements, and Homework Submission

We provide the `Board` class, and code to parse the command line arguments and load the puzzle input file. You may use or modify any or all of the provided code in your solution.

You must use recursion in a non-trivial way in your solution to this homework. As always, we recommend you work on this program in logical steps. Partial credit will be awarded for each component of the assignment, first for the visualization of reachability, then for finding a correct solution (even if it is not the shortest) with a single goal, and a bit more if that solution is one of the shortest solutions. Finding a correct solution when multiple goals are specified is the next logical step. Finding *all* unique shortest solutions is worth nearly full credit, and finding all of the shortest solutions when the maximum number of moves is *not* specified is worth

full credit. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles with a few robots! Be sure to create your own simple test cases as you debug your program.*

Once you have finished your implementation, analyze the performance of your algorithm using order notation. What important variables control the complexity of a particular problem? The dimensions of the board (i and j), the number of robots on the board (r), the number of goal locations specified (g), the number of interior walls (w), the maximum total number of moves allowed (m)? In your `README.txt` file write a concise paragraph (< 200 words) justifying your answer. Also include a summary of the running time and success of your program on each of the provided examples.

To measure running time, you may find it useful to run your program by *redirecting* `std::cout` to a file and measure performance with the UNIX `time` command. For example:

```
time ricochet_robots puzzle1.txt -max_moves 10 -all_solutions > output.txt
```

This will produce output that looks like:

```
real    0m0.155s
user    0m0.156s
sys     0m0.000s
```

The "real" time is the actual time (sometimes called "wall clock time") that your program spent. The other two numbers give information beyond the scope of this class. As a note, "user" and "sys" may add up to be larger than "real" on multiprocessor/multicore systems. Redirecting output to a file will both make it easy to read the results from `time`, and may speed things up significantly for later test cases where there is a lot of output.