CSCI-1200 Data Structures — Spring 2013
Homework 6 — Robo Rally 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 "Robo Rally":

http://en.wikipedia.org/wiki/RoboRally
http://www.wizards.com/default.asp?x=ah/prod/roborally

Understanding the non-linear word search program from Lecture 12 will be helpful in thinking about how you will solve this 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.

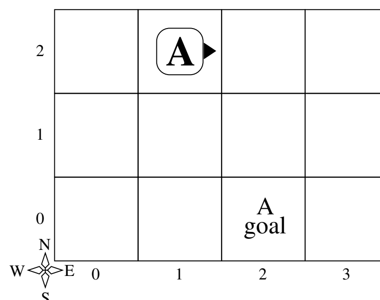
You will be given a two-dimensional play space and one or more robots that move around within the space. Each robot has a starting configuration (position and direction/orientation), goal position (direction/orientation not specified), and a set of sequential robot instructions that must be rearranged to solve the overall puzzle and navigate the robot to its goal. When the robot programs are run simultaneously, the robots should move around the board, avoiding collisions with the outer walls and each other. If, at the end of the last instruction, all robots are in their goal positions, the puzzle is solved. *Please carefully read the entire assignment and study the provided code before beginning your implementation.*

Your program should expect one or more command line arguments:

```
robo_rally puzzle1.txt
robo_rally puzzle1.txt -find_all_solutions
robo_rally puzzle1.txt -verbose -find_all_solutions -allow_pushing
```

The first argument specifies the name of the input file. By default the program stops after finding a valid solution to the robot movement programming puzzle. But when the additional argument `-find_all_solutions` is specified, your program should enumerate all unique solutions (in any order). Here's an example of the input file format and a diagram of the corresponding robot puzzle board:

```
board 4 3
num_steps 4
num_robots 1
robot A
start 1 2 east
goal 2 0
forward_1 101
rotate_right
forward_2 101
rotate_left
```



The first few lines describe the dimensions of the board (# cols = 4, # rows = 3), the number of sequential movement steps each robot will execute, and the number of robots on the board for this puzzle. The details of each robot follow, including: the unique `char` symbol that represents the robot in the ASCII art board output, the start position and direction (`north`, `east`, `south`, or `west`), the goal position, and a list of the instructions that the robot will execute in this game *in some order*. The robot is restricted to this set of instructions/moves. All moves will be executed *exactly once*. This is similar to the board game, in which each player/robot is dealt a set of cards and must play all of the cards exactly once. There may be duplicates in the collection of moves for a robot, and the robot must execute all of these actions. Note that swapping the order of two identical moves is not a *different* solution.

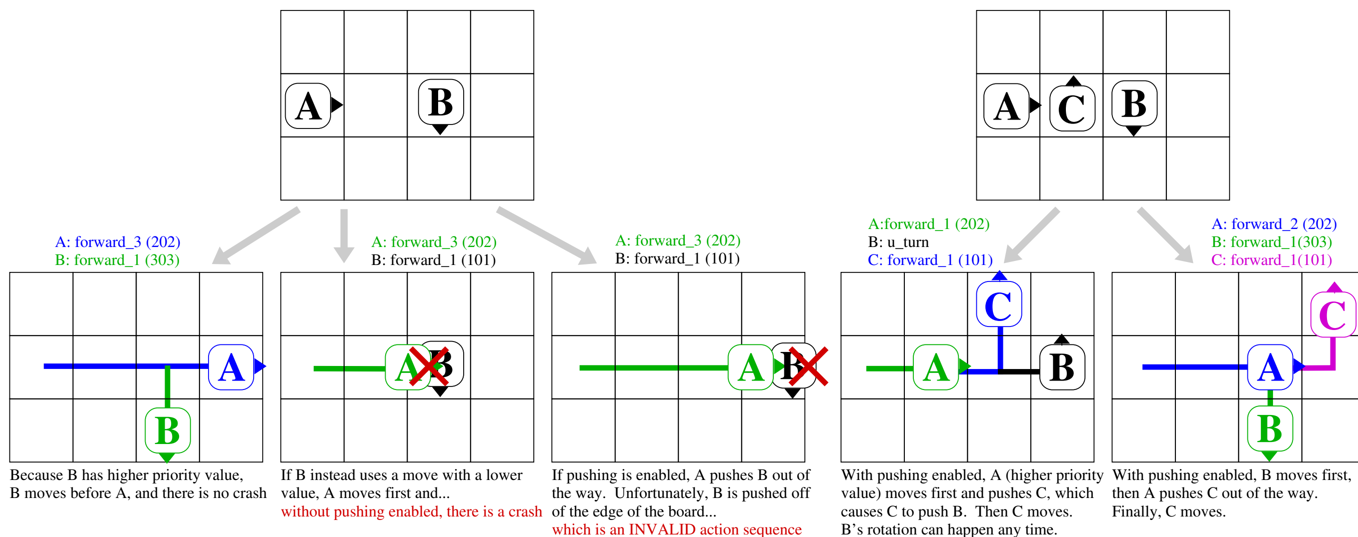
Legal Moves & Collisions

The different movement instructions for a robot are: `forward_1`, `forward_2`, `forward_3`, `backward_1`, `rotate_left`, `rotate_right`, `u_turn`, and `do_nothing`. The first four instructions also include an non-negative integer *priority* value. This value is used to determine which robot moves first (more below).

If a robot attempts to move off the edge of the board, it crashes into the wall and that sequence fails as a solution. Similarly, if a robot moves into a grid square currently occupied by another robot, those robots crash and the sequence fails (unless the optional command line argument `-allow_pushing` is specified).

When we have multiple robots, the robots execute their moves step by step as a group. Step 1: all robots will execute their first move, step 2: then all robots will execute their second move, etc. Within a single step, the order that two robots execute their moves *may* affect the overall outcome. To resolve this, we will refer to the priority value on the action each robot will perform. Within that step, we allow the robots with higher priority values to complete their action before robots with lower priority values start their action. In the examples below, the priority value is in parentheses. Note: The input file may have equal priority values for actions available to one robot, but the same priority value will not appear in different robots.

If the optional command line argument `-allow_pushing` is specified, robots within the path of the moving robot are pushed into the next empty cell. Note that this can cause a chain reaction. Also, note that the timing of robot rotation actions is not significant.



Output

Your program should output the solutions it found to cout. If the puzzle is impossible your program should output "`Found no solutions`". If the user has specified `-find_all_solutions` you should print the total number of solutions found at the bottom of the output. Here's the all solutions output for our earlier example:

```
Searching for all solutions...
Solution:
Robot A: forward_1(101), rotate_right, forward_2(101), rotate_left
+---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+
|               |   |               |   |               |   |               |   |               |
|     A>        |   |         A>    |   |         A     |   |               |   |               |
|               |   |               |   |         v     |   |               |   |               |
+   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +
|               |   |               |   |               |   |               |   |               |
|               |   |               |   |               |   |               |   |               |
|               |   |               |   |               |   |               |   |               |
+   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +
|               |   |               |   |               |   |               |   |               |
|               |   |               |   |               |   |         A     |   |         A>    |
|               |   |               |   |               |   |         v     |   |               |
+---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+
Solution:
Robot A: rotate_right, forward_2(101), rotate_left, forward_1(101)
+---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+
|               |   |               |   |               |   |               |   |               |
|     A>        |   |     A         |   |               |   |               |   |               |
|               |   |     v         |   |               |   |               |   |               |
+   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +
|               |   |               |   |               |   |               |   |               |
|               |   |               |   |               |   |               |   |               |
|               |   |               |   |               |   |               |   |               |
+   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +   +
|               |   |               |   |               |   |               |   |               |
|               |   |               |   |     A         |   |     A>        |   |         A>    |
|               |   |               |   |     v         |   |               |   |               |
+---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+   +---+---+---+---+
Found 2 solution(s)
```

Use the command line argument `-verbose` to help in debugging, especially to aid in examining robot collisions and pushing interactions. The content and formatting of the debugging output is not specified.

Homework Submission

You must use recursion in a non-trivial way in your solution to this homework. First, you should tackle the problem of finding and outputting one legal solution to the puzzle without pushing (if one exists). Significant partial credit will be given for submissions that do this correctly. Full credit will be given to programs that find *all* unique solutions to the puzzle.

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 (n and m), the number of robots on the board (r), the number of steps taken by each robot (s), etc.? In your `README.txt` file write a concise paragraph ($<$ 200 words) justifying your answer. Also include a summary of the running time of your program on each of the provided examples. For extra credit you may implement techniques to improve the performance of the basic algorithm and submit your code to the HW6 contest. Describe these improvements in your `README_contest.txt` file. Extra credit will also be awarded for interesting new robot puzzles – but don't make the puzzles so difficult your program can't solve them!

We have provided basic `Robot`, `Board`, `Configuration`, and `Command` classes, including most of the necessary I/O functionality. You may use or modify any or all of the provided code in your solution. Use good coding style when you design and implement your program. Be sure to make up new test cases and don't forget to comment your code! When you are finished please zip up your folder exactly as instructed for the previous assignments and submit it through the course webpage.

Robo Rally Contest Rules

Contest submissions are a separate homework submission. Contest submissions are due Saturday March 30th at 11:59pm. You may not use late days for the contest.

- Contest submissions *do not* need to use recursion. Programs must follow the output specifications and match the formatting of the examples posted on the course webpage.
- We will recompile (`g++ -O3 *.cpp`) and run all submitted entries on one of our machines. Programs that do not compile, or do not complete the basic tests in a reasonable amount of time with correct output, will be disqualified and will not receive extra credit.
- Programs must be single-threaded and single process.
- We will run your program by *redirecting* `cout` to a file and measure performance with the UNIX `time` command:

  ```
  time robo_rally puzzle1.txt -find_all_solutions > output.txt
  ```
- We will be testing with and without the optional command line arguments `-find_all_solutions` and `-allow_pushing`. We will separately judge the fastest single solution solver, the fastest all solutions solver, and the fastest solver with pushing allowed. You are encouraged to submit to the contest even if your implementation for the optional arguments is incomplete.
- You may submit one or more interesting new test cases for possible inclusion in the contest. Name these tests `smithj_1.txt`, `smithj_2.txt`, etc. (where `smithj` is your RCS username). Extra credit will be awarded for interesting test cases that are used in the contest. Caution: Don't make the test cases so difficult that your program cannot solve them in a reasonable amount of time!
- Extra credit will be awarded based on performance in the contest.
- In your `README_contest.txt` file, describe the optimizations you implemented for the contest, describe your new test cases, and summarize the performance of your program on all test cases.