CSCI-1200 Data Structures — Fall 2015
Homework 6 — Battleship Recursion

In this homework we will solve ship placement puzzles inspired by the pencil & paper "Battleship" game that was later made into a board game by Milton Bradley and then a puzzle that is a regular feature in Games magazine. You can read more about the history of the game and see examples here:

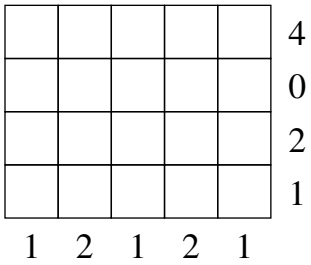
https://en.wikipedia.org/wiki/Battleship_(game)
https://en.wikipedia.org/wiki/Battleship_(puzzle)

This is a popular game with lots of material available online. You may not search for, study, or use any code related to the Battleship game or puzzle. *Please carefully read the entire assignment and study the examples before beginning your implementation.*

Battleship Puzzles - How to Play

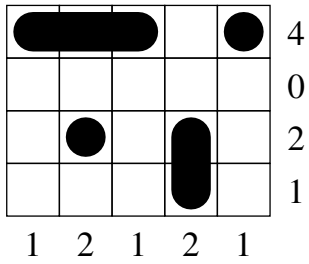
Your program will accept one or two command line arguments. The first argument is the name of a battleship puzzle board file similar to the file shown below. This sample file begins with the dimensions of the board, in this case 4 rows and 5 columns. Next, we give the number of cells in each row and each column that are occupied by a ship. The other cells in the row are open water. Then, we have a simple list of the ships that must be placed on that board. All ships are 1 cell wide, but each ship type has a different length (# of cells): submarine = 1, destroyer = 2, cruiser = 3, battleship = 4, carrier = 5, cargo = 6, and tanker = 7.

```
board 4 5
rows 4 0 2 1
cols 1 2 1 2 1
cruiser
destroyer
submarine
submarine
```



Your task is to place the ships on the board satisfying the counts for each row. One important rule in placing the ships is that no two ships may occupy adjacent cells (including the diagonal). The sample puzzle above actually has two solutions. The diagram and sample output below show one of the solutions. Can you manually find the other?

```
Solution:
cruiser    0 0 horizontal
submarine  0 4
submarine  2 1
destroyer  2 3 vertical
+-----+
|<X> o|4
|     |0
| o ^ |2
|   v |1
+-----+
 12121
```



Output Formatting

To ensure full credit on the homework server, please format your solution exactly as shown above. The solution must begin with the keyword "`Solution:`", followed by a line for each ship beginning with the ship type, the row and column of the upperleftmost cell occupied by the ship, and for non-submarine ships the orientation of the ship ("`horizontal`" or "`vertical`"). The ships may be listed in any order. After the ship

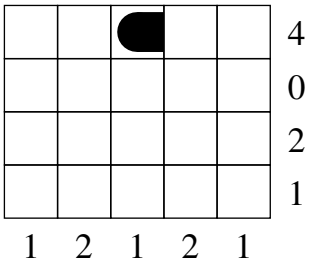
placement details, you should make an ASCII art diagram of the solved board (this will help in debugging). However, this will be graded by the TAs not the automated grading on the homework server, so you may format your ASCII art diagram somewhat differently than the sample above.

If the optional second argument `find_all_solutions` is not specified, your program should output to `std::cout` any single valid solution to the puzzle. If the optional argument `find_all_solutions` is specified, your program should output all valid, unique solutions (in any order) and then also print at the bottom the number of solutions found, e.g., "`Found 2 solution(s)`". If the puzzle has no solutions, your program should print "`No solutions`". When searching for all solutions, make sure you do not double count or duplicate the same solution. For example, if a puzzle has two submarines, swapping the submarines does not make a "new" solution.

Puzzles with Cell Constraints

Some input puzzle files have one or more additional constraints placed on some of the cells. These will be listed in the input file after the ships. Here is an example:

```
board 4 5
rows 4 0 2 1
cols 1 2 1 2 1
cruiser
destroyer
submarine
submarine
constraint 0 2 <
```

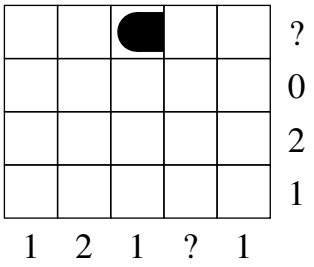


Each constraint line begins with the keyword "`constraint`", then the row and column, then one of 7 characters: '`o`' to represent a submarine; '`<`' or '`>`', to represent the left or right cells of a horizontal ship (length ≥ 2); '`^`', or '`v`', to represent the top or bottom cells of a vertical ship (length ≥ 2); '`X`' to represent a middle cell (not either end) of a ship with length ≥ 3; or '`_`' to represent open water. Your task is to limit the output to solutions that match these constraints.

Puzzles with Unknown Sums and/or Unknown Ship Types

The final twist for this assignment is that the input file may have some unspecified row and/or column sums (listed as '`?`') and/or some ships of unspecified type (listed as "`unknown`"), which may be any length from 1 to 7 cells. Here is an example input:

```
board 4 5
rows ? 0 2 1
cols 1 2 1 ? 1
cruiser
destroyer
submarine
unknown
constraint 0 2 <
```



Additional Requirements: Recursion, Order Notation, & Contest Submission

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. Your program should do some error checking when reading in the input to make sure you understand the file format. *IMPORTANT NOTE: This problem is computationally expensive, even for medium-sized puzzles! 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 (w

and h)? The number of ships (s)? The total number of occupied cells (o) or open water (w)? The number of constraints (c)? The number of unknown sums (u) or unspecified ship types (t)? Etc. In your `README.txt` file write a concise paragraph (< 200 words) justifying your answer. Also include a simple table summarizing the running time and number of solutions found by your program on each of the provided examples.

All students are required to submit their program to the Homework 6 contest (see below). Extra credit will be awarded for programs that have a strong performance in the contest.

You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss this assignment, problem solving techniques, or error messages, etc. with anyone, please list their names in your `README.txt` file.

Battleship Contest Rules

- Contest submissions are a separate homework submission. Contest submissions are due Sunday Nov 1st at 11:59pm. You may not use late days for the contest. (The regular homework deadline is Thursday Oct 29th at 11:59pm and late days are allowed for the regular homework submissions.)
- You may submit the same code for both the regular homework submission and the contest. Or you may make a small or significant change for the contest.
- Contest submissions *do not* need to use recursion.
- Contest submissions must follow the output specifications and match the formatting of the examples posted on the course webpage – except that contest submissions may include or omit the ASCII art board representation.
- We will compile your code with optimizations enabled (`g++ -O3 *.cpp`) and run all submitted entries on the homework server. Programs that do not compile, or do not complete the basic tests in a reasonable amount of time with correct output, will not receive extra credit.
- Programs must be single-threaded and single-process.
- We will run your program by *redirecting* `std::cout` to a file and measure performance with the UNIX `time` command. For example:
  ```
  time battleship.out puzzle_sample.txt > out_puzzle_sample.txt
  time battleship.out puzzle_sample.txt find_all_solutions > out_puzzle_sample_all.txt
  ```
- You may want to use a *C++ code profiler* to measure the efficiency of your program and identify the portions of your code that consume most of the running time. A profiler can confirm your suspicions about what is slow, uncover unexpected problems, and focus your optimization efforts on the most inefficient portions of the code.
- We will be testing with and without the optional command line argument `find_all_solutions` and will highlight the most correct and the fastest programs.
- You may submit up to two interesting new test cases for possible inclusion in the contest. Name these tests `smithj_1.txt` and `smithj_2.txt` (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!
- 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.
- Extra credit will be awarded based on overall performance in the contest.