1. Start by defining your `Location` type. Take care to design an appropriate type. Then write your `toLocation` and `fromLocation` functions to convert between a a `Location` to a two-character `String`.

   Next write your `feedback` function and test it very carefully. If your `feedback` function is erroneous, correct guessing code can easily go wrong.

   Finally, write your `initialGuess` and `nextGuess` functions. I suggest starting with a simple implementation, and get it working, before trying to reduce the number of guesses. Below are several hints for that.

2. A very simple approach to this program is to simply guess every possible combination of locations until you guess right. There are only 4960 possible targets, so on average it should only take about 2480 guesses, making it perfectly feasible to do in 5 seconds. However, this will give a very poor score for guess quality.

3. A better approach would be to only make guesses that are consistent with the answers you have received for previous guesses. You can do this by computing the list of possible targets, and removing elements that are inconsistent with any answers you have received to previous guesses. A possible target is inconsistent with an answer you have received for a previous guess if the answer you would receive for that guess and that (possible) target is different from the answer you actually received for that guess.

   You can use your `GameState` type to store your previous guesses and the corresponding answers. Or, more efficient and just as easy, store the list of remaining possible targets in your `GameState`, and pare it down each time you receive feedback for a guess. That way you don't need to remember past guesses or feedback.

4. The best results can be had by carefully choosing each guess so that it is most likely to leave a small remaining list of possible targets. You can do this by computing for each remaining possible target the *average* number of possible targets that will remain after each guess, giving the *expected* number of remaining possible targets for each guess, and choose the guess with the smallest expected number of remaining possible targets.

5. Unfortunately, this is much more expensive to compute, and you will need to be careful to make it efficient enough to use. One thing you can do to speed it up is to laboriously (somehow) find the best first guess and hard code that into your program. After the first guess, there are much fewer possible targets remaining, and your implementation may be fast enough then.

The choice of a good first guess is quite important, and the best first guess might not be what you'd intuitively expect. It turns out you get more information from feedback like (0,0,0) (which tells you there are no ships within 2 spaces of any of the guessed locations) than from feedback like (1,1,1), which says there are ships near all your guesses, but not where they are.

6. You can also remove *symmetry* in the problem space. The key insight needed for this is that given any *guess* and an answer returned for it, the set of remaining possibilities after receiving that answer for that guess will be the same regardless of which target yielded that answer. In other words, all the guesses that yield the same feedback will leave you with the same set of remaining possibilities — specifically, the set of guesses that yield that feedback.

   For example, suppose there are ten remaining candidate targets, and one guess gives the answer (3,0,0), three others give (1,0,2), and the remaining six give the answer (2,0,1). In this case, if you make that guess, there is a 1 in 10 chance of that being the right answer (so you are left with that as the only remaining candidate), 3 in 10 of being left with three candidates, and a 6 in 10 chance of being left with six candidates. This means on average you would expect this answer to leave you with

   $$\frac{1}{10} \times 1 + \frac{3}{10} \times 3 + \frac{6}{10} \times 6 = 4.6$$

   remaining candidates. In general, the formula is:

   $$\sum_{f \in F} \frac{\text{count}(f)^2}{T}$$

   where *F* is the set of all distinct feedbacks ((3,0,0), (1,0,2), and (2,0,1) in the example above), *count*(*f*) is the number of occurrences of the feedback *f* (1 for (3,0,0), 3 for (1,0,2), and 6 for (2,0,1) in the example), and *T* is the total number of tests (10 in the example).

   Once you've computed this for each possible guess, you can just pick one that gives the minimum expected number of remaining candidates.

   Also note that if you do this incorrectly, the worst consequence is that your program takes more guesses than necessary to find the target. As long as you only ever guess a possible target, every guess other than the right one removes at least one possible target, so you will eventually guess the right target.

7. Note that these are just hints; you are welcome to use any approach you like to solve this, as long as it is correct and runs within the allowed time.