Assignment 2: Fun with Collections

This assignment is all about the amazing things you can do with collections. It's a two-parter. The first exercise is a program that models rising sea levels on a sampler of realistic terrains. The second is a program that administers one of those viral online personality quizzes to tell you which fictional character you're most similar to. By the time you've completed this assignment, you'll have a much better handle on the container classes and how to use different types like queues, maps, vectors, and sets to model and solve problems. Plus, you'll have some things we think you'd love to share with your friends and family.

Due Friday, January 24ᵗʰ at the start of class.

This assignment must be completed individually.
Working in pairs is not permitted.

This assignment has two parts. It will be quite a lot to do if you start this assignment the night before it's due, but if you make slow and steady progress on this assignment each day you should be in great shape. Here's our recommended timetable:

- Aim to complete Rising Tides within three days of this assignment going out.

- Aim to complete You Got Hufflepuff! within seven days of the assignment going out.

As always, feel free to reach out to us if you have questions. Feel free to contact us on Piazza, to email your section leader, or to stop by the LaIR (Sunday through Thursday, 7:00PM – 11:00PM in the Tresidder first floor area).

Problem One: Rising Tides

Global sea levels have been rising, and the most recent data suggest that the rate at which sea levels are rising is increasing. This means that city planners in coastal areas need to start designing developments so that an extra meter of water doesn't flood people out of their homes.

Your task in this part of the assignment is to build a tool that models flooding due to sea level rise. To do so, we're going to model terrains as grids of `double`s, where each `double` represents the altitude of a particular square region on Earth. Higher values indicate higher elevations, while lower values indicate lower elevations. For example, take a look at the three grids to the right. Before moving on, take a minute to think over the following questions, which you don't need to submit. Which picture represents a small hill? Which one represents a long, sloping incline? Which one represents a lowland area surrounded by levees?

0	1	2	3	4	2	1
1	2	3	4	5	4	2
3	4	5	6	6	5	4
2	4	5	7	5	3	2
1	2	4	5	3	2	1
0	1	2	3	1	1	1
0	0	1	2	1	1	1

-1	0	0	4	0	0	1
0	0	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

0	1	2	3	4	5	6
0	1	2	3	4	5	5
0	1	2	3	3	4	4
0	0	1	2	3	3	3
0	0	1	1	2	2	2
-1	-1	0	0	1	1	1
-2	-1	0	0	0	0	0

We can model the flow of water as follows. We'll imagine that there's a water source somewhere in the world and that we have a known height for the water. Water will then flow anywhere it can reach by moving in the four cardinal directions (up/down/left/right) without moving to a location at a higher elevation than the initial water height. For example, suppose that the upper-left corner of each of the three above worlds is the water source. Here's what would be underwater given several different water heights:

Water source at top-left corner

Height: 0m

Height: 1m

Height: 2m

A few things to notice here. First, notice that the water height is independent of the height of the terrain at its starting point. For example, in the bottom row, the water height is always two meters, even though the terrain height of the upper-left corner is either 0m or -1m, depending on the world. Second, in the terrain used in the middle column, notice that the water stays above the upper diagonal line of 4's, since we assume water can only move up, down, left, and right and therefore can't move diagonally through the gaps. Although there's a lot of terrain below the water height, it doesn't end up under water until the height reaches that of the barrier.

It's possible that a particular grid has multiple different water sources. This might happen, for example, if we were looking at a zoomed-in region of the San Francisco Peninsula, we might have water to both the east and west of the region of land in the middle, and so we'd need to account for the possibility that the water level is rising on both sides. Here's another set of images, this time showing where the water would be in the sample worlds above assume that both the top-left and bottom-right corner are water sources. (We'll assume each water source has the same height.)

Water sources at top-left and bottom-right corners

Height: 0m

0	1	2	3	4	2	1	-1	0	0	4	0	0	1	0	1	2	3	4	5	6
1	2	3	4	5	4	2	0	0	4	0	-1	-1	0	0	1	2	3	4	5	5
3	4	5	6	6	5	4	0	4	0	0	0	0	0	0	1	2	3	3	4	4
2	4	5	7	5	3	2	4	0	-1	-1	0	0	3	0	0	1	2	3	3	3
1	2	4	5	3	2	1	0	-1	-2	-1	0	3	0	0	0	1	1	2	2	2
0	1	2	3	1	1	1	0	0	-1	0	3	0	0	-1	-1	0	0	1	1	1
0	0	1	2	1	1	1	0	0	0	3	0	0	-1	-2	-1	0	0	0	0	0

Height: 1m

0	1	2	3	4	2	1	-1	0	0	4	0	0	1	0	1	2	3	4	5	6
1	2	3	4	5	4	2	0	0	4	0	-1	-1	0	0	1	2	3	4	5	5
3	4	5	6	6	5	4	0	4	0	0	0	0	0	0	1	2	3	3	4	4
2	4	5	7	5	3	2	4	0	-1	-1	0	0	3	0	0	1	2	3	3	3
1	2	4	5	3	2	1	0	-1	-2	-1	0	3	0	0	0	1	1	2	2	2
0	1	2	3	1	1	1	0	0	-1	0	3	0	0	-1	-1	0	0	1	1	1
0	0	1	2	1	1	1	0	0	0	3	0	0	-1	-2	-1	0	0	0	0	0

Height: 2m

0	1	2	3	4	2	1	-1	0	0	4	0	0	1	0	1	2	3	4	5	6
1	2	3	4	5	4	2	0	0	4	0	-1	-1	0	0	1	2	3	4	5	5
3	4	5	6	6	5	4	0	4	0	0	0	0	0	0	1	2	3	3	4	4
2	4	5	7	5	3	2	4	0	-1	-1	0	0	3	0	0	1	2	3	3	3
1	2	4	5	3	2	1	0	-1	-2	-1	0	3	0	0	0	1	1	2	2	2
0	1	2	3	1	1	1	0	0	-1	0	3	0	0	-1	-1	0	0	1	1	1
0	0	1	2	1	1	1	0	0	0	3	0	0	-1	-2	-1	0	0	0	0	0

Height: 3m

0	1	2	3	4	2	1	-1	0	0	4	0	0	1	0	1	2	3	4	5	6
1	2	3	4	5	4	2	0	0	4	0	-1	-1	0	0	1	2	3	4	5	5
3	4	5	6	6	5	4	0	4	0	0	0	0	0	0	1	2	3	3	4	4
2	4	5	7	5	3	2	4	0	-1	-1	0	0	3	0	0	1	2	3	3	3
1	2	4	5	3	2	1	0	-1	-2	-1	0	3	0	0	0	1	1	2	2	2
0	1	2	3	1	1	1	0	0	-1	0	3	0	0	-1	-1	0	0	1	1	1
0	0	1	2	1	1	1	0	0	0	3	0	0	-1	-2	-1	0	0	0	0	0

Notice that the water overtops the levees in the central world, completely flooding the area, as soon as the water height reaches three meters. The water line never changes, regardless of the current elevation. As such, water will never flood across cells at a higher elevation than the water line, but will flood across cells at the same height or below the water line

Your task is to implement a function

```
Grid<bool> floodedRegionsIn(const Grid<double>& terrain,
                            const Vector<GridLocation>& sources,
                            double height);
```

that takes as input a terrain (given as a `Grid<double>`), a list of locations of water sources (represented as a `Vector<GridLocation>`; more on `GridLocation` later), and the height of the water level, then returns a `Grid<bool>` indicating, for each spot in the terrain, whether it's under water (**true**) or above the water (**false**).

You may have noticed that we're making use of the `GridLocation` type. This is a type representing a position in a `Grid`. You can create a `GridLocation` and access its row and column using this syntax:

```
GridLocation location;
location.row = 137;
location.col = 42;

GridLocation otherLocation = { 106, 103 }; // Row 106, column 103
otherLocation.row++; // Increment the row.
cout << otherLocation.col << endl; // Prints 103
```

Now that we've talked about the types involved here, let's address how to solve this problem. How, exactly, do you determine what's going to be underwater? Doing so requires you to determine which grid locations are both (1) below the water level and (2) places water can flow to from one of the sources.

Fortunately, there's a beautiful algorithm you can use to solve this problem called ***breadth-first search***. The idea is to simulate having the water flow out from each of the sources at greater and greater distances. First, you consider the water sources themselves. Then, you visit each location one step away from the water sources. Then, you visit each location two steps away from the water sources, then three steps, four steps, etc. In that way, the algorithm ends up eventually finding all places the water can flow to, and it does so fairly quickly!

Breadth-first search is typically implemented by using a ***queue*** that will process every flooded location. The idea is the following: we begin by enqueuing each water source, or at least the sources that aren't above the water line. That means that the queue ends up holding all the flooded locations zero steps away from the sources. We'll then enqueue a next group of elements, corresponding to all the flooded locations one step away from the sources. Then we'll enqueue all flooded locations two steps away from the sources, then three steps away, etc. Eventually, this process will end up visiting every flooded location in the map.

Let's make this a bit more concrete. To get the process started, you add to the queue each of the individual water sources that happen to be at least as high as the grid cell they're located in. From then on, the algorithm operates by dequeuing a location from the front of the queue. Once you have that location, you look at each of the location's neighbors in the four cardinal directions. For each of those neighbors, if that neighbor is already flooded, you don't need to do anything because the search has already considered that square. Similarly, if that neighbor is above the water line, there's nothing to do because that square shouldn't end up under water. However, if neither of those conditions hold, you then add the neighbor to the queue, meaning "I'm going to process this one later on." By delaying processing that neighbor, you get to incrementally explore at greater and greater distances from the water sources. The process stops once the queue is empty; when that happens, every location that needs to be considered will have been visited.

At each step in this process, you're removing the location from the queue that's been sitting there the longest. Take a minute to convince yourself that this means that you'll first visit everything zero steps from a water source, then one step from a water source, then two steps, etc.

To spell out the individual steps of the algorithm, here's some ***pseudocode*** for breadth-first search:

```
create an empty queue;
for (each water source at or below the water level) {
    flood that square;
    add that square to the queue;
}

while (the queue is not empty) {
    dequeue a position from the front of the queue;

    for (each square adjacent to the position in a cardinal direction) {
        if (that square at or below the water level and isn't yet flooded) {
            flood that square;
            add that square to the queue;
        }
    }
}
```

As is generally the case with pseudocode, several of the operations that are expressed in English need to be fleshed out a bit. For example, the loop that reads

for (*each square adjacent to the position in a cardinal direction*)

is a conceptual description of the code that belongs there. It's up to you to determine how to code this up; this might be a loop, or it might be several loops, or it may be something totally different. The basic idea, however, should still make sense. What you need to do is iterate over all the locations that are one position away from the current position. How you do that is up to you.

It's important to ensure that the code you write works correctly here, and to help you with that we've provided a suite of test cases with the starter files. These test cases look at a few examples, but they aren't exhaustive. You'll need to add at least one custom test case – and, preferably, more than one – using the handy `ADD_TEST` macro that you saw in the first programming assignment. We don't recommend copying one of the test cases from the handout, since (1) that's really tedious and time-consuming and (2) it's better to more directly test particular tricky behaviors with smaller, more focused examples.

To summarize, here's what you need to do for this assignment:

---

1. Implement the `floodedRegionsIn` function in `RisingTides.cpp`. Your code should use the breadth-first search algorithm outlined above in pseudocode to determine which regions are under water. Water flows up, down, left, and right and will submerge any region whose height is less than or equal to the global water level passed in as a parameter. Test as you go.

2. Add in at least one custom test case into `RisingTides.cpp` – preferably, not one from the assignment handout – and see how things go.

Some notes on this problem:

- ***The provided demo will crash if you try running it without first having implemented the `floodedRegionsIn` function. This is normal, and happens because the default implementation of `floodedRegionsIn` returns an empty grid.*** Generally speaking, our demos are designed to show off your code once it's working correctly and aren't really designed to be used for testing purposes. To test whether your code is working, use the "Run Tests" button and the `ADD_TEST` macro.

- Need a refresher on `Grid` type? Check the [Stanford C++ Library Documentation](#).

- The initial height of the water at a source may be below the level of the terrain there. If that happens, that water source doesn't flood anything, including its initial terrain cell. (This is an edge case where both the "flood it" and "don't flood it" options end up being weird, and for consistency we decided to tiebreak in the "don't flood it" direction.)

- The heights of adjacent cells in the grid may have no relation to one another. For example, if you have the topography of Preikestolen in Norway, you might have a cell of altitude 0m immediately adjacent to a cell of altitude 604m. If you have a low-resolution scan of Mount Whitney and Death Valley, you might have a cell of altitude 4,421m next to a cell of altitude -85m.

- Curious about an edge case? Check the examples on the previous pages.

Once you're passing all the tests, click the "Rising Tides" option to see your code in action! When you pull up the "Rising Tides" demo, it will use your code to flood-fill the sample world you've chosen. Some of these sample worlds are large, and it might take a while for that flood-fill to finish. However, it should probably take no more than 30 seconds for the program to finish running your code. If you find that it's taking more than 30 seconds to load a terrain, it may mean that you have an inefficiency somewhere in your code. Make sure that you aren't passing large objects into functions by value (that is, if you're passing around a `Grid` or `Queue` or something like that, it should be done by reference or `const` reference).

The demo app we've included will let you simulate changes to the sea levels for several geographical regions of the United States. The data here is taken from the National Oceanographic and Atmospheric Administration (NOAA), which provides detailed topological maps for the US. Play around with the data sets – what do you find?

Part Two: You Got Hufflepuff!

Have you ever been browsing through social media and seen a post like this one?

"I got Hufflepuff! Which Hogwarts house are you? Click here to find out!"

If you've ever clicked on one of those links, you've probably been presented with a series of statements and asked how much you agree with each of them. At the end, you're given your classification – whether that's which Hogwarts house you're in, which Game of Thrones character you most resemble, or even weirder things like which sandwich best represents you.

These sorts of viral quizzes spread as widely as they do for many reasons. Partially, it's because people like to answer questions about themselves, and this gives people a fun outlet for doing so. It's also because people get really invested in the books, movies, music, shows, etc. that they consume. And in large part this happens because people just plain enjoy sharing the results with friends and talking about the results.

Your task in this assignment is to build one of these personality quizzes. Specifically, you'll implement a function

```
void administerQuiz(const HashSet<Question>& questions,
                    int numQuestions,
                    const HashSet<Person>& people);
```

that takes as input a set of possible personality questions, a number of questions to ask in the quiz, and a set of people. Your function should then administer the personality quiz by choosing `numQuestions` random questions from `questions`, and then it should determine which person out of `people` is most similar to the user. The set of people in `people` could be characters from a movie, houses from Hogwarts, etc.

Let's break this down into smaller pieces to talk about what, specifically, you need to do.

First, let's talk about how these sorts of quizzes usually work. Many personality quizzes are based on a what's called the *five-factor model*. In that model, personalities are described by giving numbers (positive or negative) in five different categories with the handy acronym "OCEAN:" *o*penness, *c*onscientiousness, *e*xtraversion, *a*greeableness, and *n*euroticism. (It may seem reductionist to distill entire personalities down to numbers in five categories, but hey, the program you're writing is for entertainment purposes only.) As you take the personality quiz and answer questions, the computer updates your scores in these five categories based on how you answer each question. Your score in each category begins at zero and is then adjusted in response to your answers. At the end of the quiz, the program reports the fictional character whose scores are most similar to yours.

Each quiz question is represented as a statement, along with what categories the question asks about and in what way those questions impact those categories. For example, you might have a question like

```
+A +O I love to reflect on things.
```

If you got this question on a personality quiz, it might be given to you like this:

How much do you agree with this statement?
I love to reflect on things.

    ○ Strongly disagree
    ○ Disagree
    ○ Neutral
    ○ Agree
    ○ Strongly agree

As indicated by the +A +O here. this particular statement is aligned in the positive direction of categories A (**a**greeableness) and O (**o**penness), so the way that you answer this question will adjust your scores in categories A and O. For example, suppose you choose "Agree." Since you agreed with the statement, you'd get +1 point in category A and +1 point in category O. Had you chosen "Strongly Agree", you'd instead get +2 points in category A and +2 points in category 0. Choosing "Neutral" here would leave your score unchanged.

On the other hand, if you chose "Disagree," then you'd get -1 point in category A and -1 point in category O. Finally, if you chose "Strongly Disagree," then you'd get -2 points in category A and -2 points in category O. (Negative numbers don't represent negative personality traits. Positive scores in category O indicate a low barrier to trying new things, while negative scores indicate more caution in trying new things. Neither of these is the "right" way to do things.)

Here's another sample question:

```
-N +C I try to impress others.
```

If your program chooses to ask this question, it would do so like this:

How much do you agree with this statement?
I try to impress others.

○ Strongly disagree
○ Disagree
○ Neutral
○ Agree
○ Strongly agree

The -N +C here means that this question is designed to count negatively in category N (**n**euroticism) and positively in category C (**c**onscientiousness). So, for example, answering with "Strongly Agree" would add -2 to your N score and +2 to your C score, and answering with "Strongly Disagree" would add +2 to your N score and -2 to your C score. Answering with "Neutral" wouldn't change your score.

As a last example, consider this question:

```
+E I am the life of the party.
```

The +E on this question means that it counts positively in factor E ("extraversion"). An answer of "Neutral" wouldn't impact your score in the E category. An answer of "Agree" would add +1 to your E score, and an answer of "Strongly Disagree" would add -2 to your E score.

Let's drop a bit lower to the level of actual C++ code. The function you're writing works with a type called `Question`, which is represented as follows:

```
struct Question {
    string questionText;
    HashMap<char, int> factors;
};
```

This `struct` has two fields. First, there's `questionText`, which is a string representing the text of the question. The second field, `factors`, is a map from characters to integers. This `factors` map is our way of representing the information about which categories the question asks about, and whether those categories count positively or negatively. For example, the question

```
-A +C I listen to my brain rather than my heart.
```

would be represented as `Question` whose `question` field is `"I listen to my brain rather than my heart"` and whose `factors` map would associate `'A'` with the number -1 and `'C'` with the number +1. Similarly, the question

```
+C -O I do things by the book.
```

would be represented as a `Question` whose `question` field is `"I do things by the book"` and whose `factors` map associates `'C'` with +1 and `'O'` with -1.

By asking a series of questions and updating the O, C, E, A, and N scores for the user based on how they respond to the questions, you can administer a personality quiz and get five numbers representing the user's personality. In fact, that's what we'd like you to do as the first milestone for this part of the assignment.

Step One: Administer a Personality Quiz

Begin your implementation of `administerQuiz`. Your function should do the following:

1. Ask a series of `numQuestions` questions chosen randomly from `questions` to the user. You should not ask the same question twice. To ask a question to the user, use this helper function that we've provided to you:

   `int askPersonalityQuestion(const Question& question);`

   This function takes as input a `Question` object, then asks the user to choose one of the five options mentioned earlier. The function returns a number between 1 and 5, inclusive, where 1 means "strongly disagree" and 5 means "strongly agree."

   You may also want to use `randomElement`, which is provided in `"hashset.h"` and returns (but does not remove) a random element of a `HashSet`. You can use it by calling

   `Type var = randomElement(your-hash-set);`

2. As you ask questions in Step 1, keep track of the user's O, C, E, A, and N scores, updating them based on the factors each question asks and the strengths of the user's answers.

3. Print out the user's O, C, E, A, and N scores. To do so, use the handy function

   `void displayScores(const HashMap<char, int>& scores);`

   which takes in a `HashMap` from categories to scores, then displays them in the window. To aid with testing, we recommend that you check the "Show Factors on Questions" checkbox; this will show which categories each question assesses, along with whether they count positively or negatively in those categories.

At this point you have five numbers – your scores for each of O, C, E, A, and N – that make up your results. That might be interesting in and of itself, but usually the online quizzes won't directly show you these numbers and will instead make a pronouncement like "your Marvel Superhero is Black Panther!" or "your Disney character is Elsa!" How exactly do these quizzes do this?

Once you have a five-number score for the user, you can comparing the user's scores against scores for a bunch of fictional characters. You'd then report the fictional character whose scores were "closest" to the user's score, for some definition of "closest." We've provided a collection of data files, each of which has a collection of characters with their associated O, C, E, A, and N scores. Each person is represented by the following handy `struct`:

```
struct Person {
    string name;
    HashMap<char, int> scores;
};
```

Here, `name` represents the person's name, and `scores` is a `HashMap` from the five categories to the user's score in those categories.

Once we have these scores, we need to find character is "closest" to the user's scores. There are many ways to define "closest," but one of the most common ways to do this uses something called the ***cosine similarity***, which is described below.

First, an observation. Suppose one person takes a personality quiz and answers ten questions, and a second, similar person takes a personality quiz and answers twenty questions. Suppose they get these scores:

	O	*C*	*E*	*A*	*N*
Person 1	6	-7	8	1	-3
Person 2	11	-14	17	2	-2

Notice that, generally speaking, the numbers for Person 2 have higher absolute value than the numbers for Person 1. There's a simple reason for this: since Person 2 answered more questions, they had more opportunities to have numbers added or subtracted from each category. This means that we'd expect the second person's numbers to have higher magnitudes than the first person's numbers.

To correct for this, you'll need to *normalize* the user's scores. Borrowing a technique from linear algebra, assuming the user's scores are o, c, e, a, and n, you should compute the value

$$\sqrt{o^2 + c^2 + e^2 + a^2 + n^2}$$

and then divide each of the user's five scores by this number. (For those of you coming from Python, note that C++ doesn't have a built-in exponentiation operator like `**`. Instead, to compute square roots, include the header `<cmath>` and use the `sqrt` function.) Watch your types: the above expression isn't guaranteed to be an integer, and there are several variables that might be `int`s in these expressions.

As an example, normalizing the two above scores gives these values:

	O	*C*	*E*	*A*	*N*
Person 1	0.476	-0.555	0.634	0.079	-0.238
Person 2	0.444	-0.565	0.686	0.081	-0.081

As you can see, the normalized scores of these two people are much closer to one another, indicating that their personalities are pretty similar.

We as humans can look at these numbers and say "yep, they're pretty close," but how could a computer do this? Suppose we have two different five-number scores $(o_1, c_1, e_1, a_1, n_1)$ and $(o_2, c_2, e_2, a_2, n_2)$ that have already been normalized. One measure we can use to determine how similar they are is to compute their *cosine similarity*, which is defined here:

$$similarity = o_1 o_2 + c_1 c_2 + e_1 e_2 + a_1 a_2 + n_1 n_2.$$

That is, you multiply the corresponding O scores, the corresponding C scores, the corresponding A scores, etc., then add them all together. The number you get back from the cosine similarity ranges from -1 to +1, inclusive. A similarity of -1 means "these people are about as diametrically opposite from one another as you could possibly get." A similarity of +1 means "these two groups of scores are as aligned as possible." If you run this calculation using the above numbers, you'll get a score of roughly 0.9855; these two people are remarkably similar!

With that in mind, here's the second step of your program:

Step Two: Find the Best Match

Update your implementation of `administerQuiz` to do the following:

1. There's a quick edge case to handle: if the user's scores in all categories are 0's, then you can't normalize their score (do you see why?) and you should report a message indicating that you don't have enough information to make a determination of who they're most similar to. (How sad!) If so, skip these next steps. You can report a message with this function:

    ```
    void displayMessage(const string& message);
    ```

2. Find the person from the `people HashSet` whose five-number score has the highest cosine similarity with the user's scores. Don't forget to normalize both the character's scores and the user's scores before computing cosine similarity! If multiple characters are tied, you can choose any one of them.

3. Report who the closest match is, along with the similarity of that match. To do so, use the `displayMessage` helper function, passing in whatever string you'd like, as long as it contains the name of the best match and the similarity. (We recommend something like `"You Got Hufflepuff! (Similarity: 0.38233)"`, but you can pick whatever you'd like.)

Once you've done this, all that's left is final polish and having some fun. Test your code extensively to make sure that it works correctly. You might want to consider, as a debug feature, having your program print out the categories associated with each question so that you can make sure that the final computed scores match what you expect to find. You may want to show the similarities of all the characters in the file and check them by hand to make sure they're computed correctly.

Then, feel free to create or edit our lists of characters! Pretend to be some fictional character (or someone else you know, or something inanimate, etc.) and take a quiz. You can then stash the numbers you got into your own sample file. In fact, that's how we generated the sample files shipped with the program!

Some notes on this part of the assignment:

- If there are many characters tied with the user for most similar, you can break ties in whatever way seems easiest to you.

- Our provided starter code already converts questions from the form where there are pluses and minuses at the front of the question into `Question struct`s, where the `factors HashMap` has that information stored in an easier-to-access way. You don't need to – and in fact shouldn't – need to do any file or string parsing in the course of completing this assignment.

- You can assume that `numQuestions` is at least as large as the number of questions in the `questions HashSet`.

- Make sure not to ask the same question twice.

- If you'd like, you can assume that no question will assign a number other than +1 or -1 to a particular category, though there's no fundamental reason why this would necessarily be the case.

- It's possible for questions to adjust scores in any number of the categories, not just one or two.

- The number produced by the formula for cosine similarity **_only works with normalized scores_**. Don't forget to normalize both the user's scores and the scores associated with each fictional person before computing their cosine similarity; if you do, the similarity scores you'll get back will be way off.

A note on testing: the function `administerQuiz` is not one that you can directly test via ADD_TEST, both because `administerQuiz` requires user interaction and because `administerQuiz` doesn't actually return a value. However, we expect that in the course of writing `administerQuiz`, you'll need to add in helper functions that perform various calculations. You are encouraged to add your own custom tests for *those* functions by using ADD_TEST, though you aren't required to. If you do write those custom tests, note that the test suite will not let you call `askPersonalityQuestion`, `displayScores`, or `displayMessage`, since those functions require either graphics or user interaction.

Part Three: (Optional) Extensions

If you'd like to run wild with these assignments, go for it! Here are some suggestions.

- **_Rising Tides:_** The sample worlds here have their data taken from the NOAA, which is specific to the United States, but we'd love to see what other data sets you can find. Get some topographical data from other sources, modify it to fit our file format, and load it into the program. (You can find information about the file format in the `res/CreatingYourOwnTerrains.txt` file.) What information – either for good or for ill – does that forecast?

  Or, consider the following question: suppose you have a terrain map of your area and know your home's position within that area. Can you determine how much higher the water can rise without flooding your home?

- **_You Got Hufflepuff:_** Once you have a five-factor score for someone, you can do all sorts of things. For example, what happens if you have five-factor scores for a bunch of different people? Could you play matchmaker in the style of the Stanford Marriage Pact? Could you explore ways in which different groups of people (Stanford students, professional bricklayers, airline pilots, etc.) are similar to or different from one another?

  Cosine similarities have uses way beyond personality quizzes. More generally, given a collection of data points, you can use cosine similarity as a way of determining how closely aligned they are. What other data sets could you apply this idea to?

  You can also create your own groups of characters! We'd love to see what you come up with.

  The five-factor model is only one of many different ways of computing a "personality score" for a person. Research some other way to do this, and see if you can design a better personality quiz than the one we gave here.

Submission Instructions

Once you're sure you've done everything correctly, including going through the Assignment Submission Checklist, submit the files `RisingTides.cpp` and `YouGotHufflepuff.cpp`, plus any other files that you modified in the course of coding up your solutions, at https://paperless.stanford.edu/. And that's it! You're done!

Good luck, and have fun!