Assignment 2: Fun with Collections

An assignment similar to the "You Got Hufflepuff!" assignment was independently developed and used by Stuart Reges in 2001.
A huge thanks to all the SLs who preflighted this assignment and worked to design the data files. Check the attributions files for details!

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 29th 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 EdStem, to email your section leader, or to stop by the LaIR.

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

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**

Height: 1m

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**

Height: 2m

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

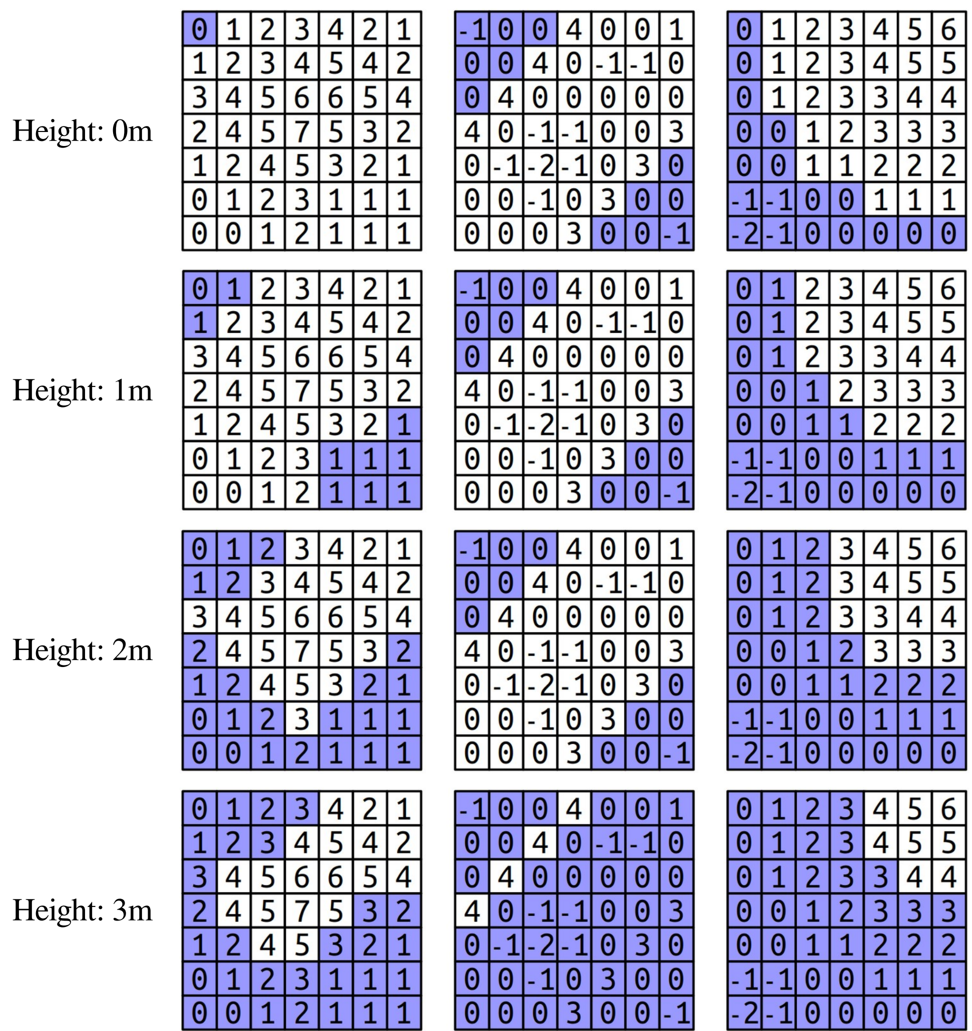
-1	**0**	**0**	4	0	0	1
0	**0**	4	0	-1	-1	0
0	4	0	0	0	0	0
4	0	-1	-1	0	0	3
0	-1	-2	-1	0	3	0
0	0	-1	0	3	0	0
0	0	0	3	0	0	-1

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**

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 for a grid to have multiple 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



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 enqueing 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 is 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 `STUDENT_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:

- 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.
- Make sure, when passing large objects around, to do so by reference (or `const` reference, if appropriate) rather than by value. Copying large grids can take a very long time and make your code slow down appreciably.

Once you're passing all the tests, click the "Rising Tides" option to see your code in action! This demo uses 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.

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 mostly 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?

We also have some maps from Mars, supplied by a former CS106B student as an extension to their assignment! The Martian terrain includes many regions way above and way below 0 elevation, so try out some large positive and large negative altitudes to get the full experience here.

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.

Many personality quizzes – including the one you'll be building – 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.
```

This would be presented to the user in the following way:

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.

The Assignment at a Glance

We've written all the logic needed to load in the bank of quiz questions, ask questions to the user, and save the results. (This code mostly involves working with our graphical interface, which we haven't discussed in lecture, and so we're handling this for you.)

Your task is to implement the data processing framework that works behind the scenes to tabulate scores from the personality quiz and determines who the user is most similar to. We'll use your code to select which questions to ask the user, to determine the user's score from their quiz answers, to determine the similarity between the user and a fictional character, and to select the fictional character who is most similar to the user.

Milestone One: Select Random Questions

In order to give a personality quiz, you'll need some way of selecting which questions to ask the user. To do this, we'd like you to write a function

```
Question randomQuestionFrom(Set<Question>& unaskedQuestions);
```

that works as follows. We provide as input to this function a `Set` containing objects of type `Question`, where `Question` is a custom type we'll detail a bit later in the assignment. This set represents a bank of personality questions that haven't yet been asked. Your should implement the function to do the following: choose a random question from the `Set`, remove it from the `Set`, then return it. The idea is that someone could call this function multiple times to pick personality quiz questions to ask the user.

We recommend that you use the `randomElement` function, which is provided in `"set.h"`. This function takes in a `Set` and returns (but does not remove) a random element of that `Set`. You can use it by calling

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

Where *`Type`* is the type of element stored in the `Set`, *`var`* is whatever variable name you'd like, and *`your-set`* is the name of the set in question.

If the input set provided to `randomQuestionFrom` is empty, then you should use the `error()` function you saw in Assignment 1 to report an error – after all, it's not possible to pick anything out of an empty set.

To recap, here's what you need to do in the first step:

> ***Milestone One***
>
> 1. Implement the `randomQuestionFrom` function in `YouGotHufflepuff.cpp`.
> 2. Use the provided tests to confirm that your code works correctly.
> 3. Optionally, but not required: use `STUDENT_TEST` to add a test case or two of your own!

Milestone Two: Compute Scores From Question/Answer Pairs

In the previous part of this question, you worked with variables of type `Question` without looking too closely at how the `Question` type works. For the next part of this problem, you will need to manipulate `Question`s, so let's dive deeper into this type. We've defined the `Question` type in the header file `YouGotHufflepuff.h`, and it looks like this:

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

Here, `questionText` contains the text of the personality quiz question, and `factors` is a map encoding the question's assessed OCEAN factors. For example, the question

```
-E +N I let others finish what they are saying
```

Would be represented as a `Question` where `questionText` is "`I let others finish what they are saying`" and where `factors` is a map that associates `'E'` with -1 and and `'N'` with +1.

Here's a quick sample of a variable of type `Question` in use:

```
Question q = /* ... get a question from somewhere ... */
cout << q.questionText << endl;
if (q.factors.containsKey('O')) {
    cout << "O score for this question: " << q.factors['O'] << endl;
}
```

Now, on to what you need to do for this milestone. We've written a piece of code that administers a personality quiz, recording the user's answers in a variable of type `Map<Question, int>`. Here, the keys represent individual personality quiz questions, and the values represent the result the user typed in. As a refresher, a response of 1 means "strongly disagree," while a response of 5 means "strongly agree."

Your task is to write code that uses the data in that `Map` to determine the user's OCEAN scores. Specifically, we'd like you to write a function

```
Map<char, int> scoresFrom(const Map<Question, int>& answers);
```

that takes as input a `Map<Question, int>` containing the user's answers to their personality quiz questions, then returns a `Map<char, int>` representing their OCEAN scores. You should compute scores using the approach described earlier: for each question, weight the factors of that question based on the user's answer (5 means "strongly agree" and adds doubles the weight of each factor; 4 means "agree" and adds the weight of each factor; 3 means "neutral" and adds zero to the weight of each factor, etc.).

As an example, suppose you were provided these questions and the indicated responses:

Question	*Factors*					*Answer*
	O	***C***	***E***	***A***	***N***	
I am quick to understand things.	+1				+1	5
I go my own way.	+1		-1			4
I know no limits.			1	-1		2
I become overwhelmed by events.	-1				-1	1

You should then produce a `Map` where O maps to +5, E maps to -2, A maps to +1, and N maps to +4. To see where this comes from, let's focus on the O component. The answer of 5 to "I am quick to understand things" adds $2 \times (+1) = +2$ to the O score, the answer of 4 to "I go my own way" adds $1 \times (+1) = 1$ to the O score, and the answer of 1 to "I become overwhelmed by events" adds $(-2) \times (-1) = +2$ to the O score. Running similar calculations for the other factors accounts for the other numbers – do you see why?

Also, notice that C wouldn't be a key in the resulting `Map`, since none of the questions have C as a factor.

Similarly, suppose you were given these question/response pairs:

Question	*Factors*					*Answer*
	O	***C***	***E***	***A***	***N***	
I keep my thoughts to myself.			-1			4
I show my gratitude.			1	1		3
I put off unpleasant tasks.		-1				4
I take an interest in other people's lives.		-1		1		4

You should produce a `Map` where C maps to -2, E maps to -1, and A maps to 1. Explaining the E score this time, the answer of 4 to “I keep my thoughts to myself” adds 1 × (-1) = -1 to the E category, and the answer of 3 to “I show my gratitude” adds 0 × 1 = 0 to the E category for the net total of -1.

None of the input questions assess the O or N factors, your resulting map should not have those as keys.

One last example. Suppose you have these question/response pairs:

Question	*Factors*					*Answer*
	O	***C***	***E***	***A***	***N***	
I reassure others.		-1		1		3
I feel others' emotions.			1	1		3

The resulting map should have C mappings to 0, E mapping to 0, and A mapping to 0. Exploring the A score here, the answer of 3 to “I reassure others” adds 0 × 1 = 0 to the A score, and the answer of 3 to “I feel others’ emotions” adds a second 0 × 1 = 0 to the A score, for a net total of 0.

Since none of the questions addressed factors O or N, you should leave those keys out.

To summarize, here’s what you need to do:

Milestone Two

1. Implement the `scoresFrom` function in `YouGotHufflepuff.cpp`.
2. Use the provided tests to confirm that your code works correctly.
3. Optionally, but not required: use `STUDENT_TEST` to add a test case or two of your own!

Some notes on this problem:

- In order to implement this function, you’ll need to iterate over the keys in the `Map`. To do so, use this syntax:

```
for (KeyType var: map) {
    /* ... do something with var, which represents a key in the map. ... */
}
```

- Remember that the `Map`’s square bracket operator does automatic insertion if you look up a key that isn’t there. In the context of a `Map<char, int>`, this means that if you look up a character that isn’t present in the map, it will automatically insert that key with the value 0.
- Although in the context of a personality quiz the keys in a `Map<char, int>` will be some subset of O, C, E, A, and N, your function should work even if we have other keys in the factors maps. (For example, we might repurpose this code to work with a different set of factors than OCEAN.)
- Do not use the `+` or `+=` operators to add two `Map`s. Adding them this way “form a new `Map` that uses all the key/value pairs from the two input maps, with duplicates resolved arbitrarily.” It does not mean “form a new `Map` by adding the two `Map`s’ elements pairwise.”

Milestone Three: Normalize Scores

We now have a `Map<char, int>` representing the user's OCEAN scores. 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.

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.

Your next task is to implement a function

```
Map<char, double> normalize(const Map<char, int>& scores);
```

that takes as input a set of raw scores, then returns a normalized version of the scores.

There is an important edge case you need to handle. If the input `Map` does not contain any nonzero values, then the calculation we've instructed you to do above would divide by zero. (Do you see why?) To address this, if the map doesn't contain at least one nonzero value, you should use the `error()` function to report an error.

Milestone Three

1. Implement the `normalize` function in `YouGotHufflepuff.cpp`.
2. Use the provided tests to confirm that your code works correctly.
3. Optionally, but not required: use `STUDENT_TEST` to add a test case or two of your own!

(Continued on the next page…)

Some notes on this problem:

- There are no restrictions on what the keys in the map can be. Although you'll ultimately be using this function in the context of a personality quiz, you should not assume that the keys will be the letters O, C, E, A, and N. It might be that the keys are some subset of those five letters, or perhaps there will be other letters represented. In each case, sum up the squares of all the values, then divide each of the values by the square root of that sum.
- The keys in the resulting map should be the same as the keys in the input map. In other words, you should not add or remove keys.
- Be mindful of the types involved in this function. The input `Map` has integer keys representing whole-numbered scores. The output `Map` has keys that are arbitrary real numbers (hence the use of the type double).
- Remember that dividing two `int`s in C++ always results in an `int`. Check the textbook for details about how to divide two `int`s and get back a double.
- For those of you coming from Python: C++ doesn't have a built-in exponentiation operator like `**`. Instead, to compute square roots, include the header `<cmath>` and use the `sqrt` function.
- We've sequenced the milestones in this assignment so that they flow in logical order, not by their difficulty. As a result, don't worry if the code you write here ends up being a bit shorter than the code you wrote for Milestone Two.

Milestone Four: Implement Cosine Similarity

At the end of the previous milestone, we saw this example of two similar OCEAN scores:

	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

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:

$$similarity = o_1o_2 + c_1c_2 + e_1e_2 + a_1a_2 + n_1n_2.$$

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 people 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!

Your next task is to write a function

```
double cosineSimilarityOf(const Map<char, double>& lhs,
                          const Map<char, double>& rhs);
```

that takes as input two sets of ***normalized*** scores, then returns their cosine similarity using the formula given above.

> ***Milestone Four***
>
> 1. Implement the `cosineSimilarityOf` function in `YouGotHufflepuff.cpp`.
> 2. Use the provided tests to confirm that your code works correctly.
> 3. Optionally, but not required: use `STUDENT_TEST` to add a test case or two of your own!

Some notes:

- The two input maps are not guaranteed to have the same keys. For example, one map might have the keys A, C, and N, and the other might have the keys O and N. You should treat missing keys as if they're associated with zero weight.
- You can assume that the two sets of input scores are normalized and don't need to handle the case where this isn't true.
- As above, while in the finished product this code will be used on OCEAN scores, this function should work equally well if the keys are arbitrary characters.

Milestone Five: Find the Best Match

At this point, you have the user's scores, and you have the ability to determine the similarity between pairs of scores. All that's left to do now is to tell the user which fictional character their scores are most similar to!

We've defined a type called `Person` in `YouGotHufflepuff.h`. It looks like this:

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

Here, `name` represents the person's name, and `scores` represents their five-factor OCEAN scores. (The scores are not normalized – can you tell why that is simply by looking at the type of the `Map`?). Your final task is to write a function

```
Person mostSimilarTo(const Map<char, int>& scores, const Set<Person>& people);
```

that takes as input the user's raw OCEAN scores and a `Set` of fictional people. This function then returns the `Person` whose scores have the highest cosine similarity to the user's scores. As an edge case, if the people set is empty, you should call `error()` to report an error, since there's no one to be similar to in that case. Similarly, if the user's scores can't be normalized – or if any of the people in the input set have scores that can't be normalized – you should call `error()` to report an error.

With that in mind, here's your final task.

> ***Milestone Five***
>
> 1. Implement the `mostSimilarTo` function in `YouGotHufflepuff.cpp`.
> 2. Use the provided tests to confirm that your code works correctly.
> 3. Optionally, but not required: use `STUDENT_TEST` to add a test case or two of your own!

Some notes on this part of the problem:

- Remember that you can only compute cosine similarities of normalized scores.
- If there is a tie between two or more people being the most similar, you can break the tie in whatever way you'd like.
- The most similar person to the user may have a negative similarity.
- As before, while we're ultimately going to be using this code on OCEAN scores, your function should work regardless of what the keys in the different maps are.

(Optional) Milestone Six: Enjoy Your Creation!

Once you've done this, all that's left is final polish and having some fun! Our provided starter code contains logic to administer a personality quiz (using random questions selected by your implementation of the `randomQuestionFrom` function), then report who the user is most similar to (using your implementations of the other functions you wrote). Take a minute or two to play around with the program and see what you find!

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!

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 terrains 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!