Fun with Collections
Welcome back to YEAH!

- A few logistics:
Welcome back to YEAH!

- A few logistics:
  - Check the YEAH A2 Ed post for the slides. For future YEAH sessions, I’ll try to get the slides up before the session so that you can follow along :)


Welcome back to YEAH!

- A few logistics:
  - Check the YEAH A2 Ed post for the slides. For future YEAH sessions, I’ll try to get the slides up before the session so that you can follow along :)
  - We will have all of our remaining YEAH hours at this time, every Saturday!
Welcome back to YEAH!

- A few logistics:
  - Check the YEAH A2 Ed post for the slides. For future YEAH sessions, I’ll try to get the slides up before the session so that you can follow along :)
  - We will have all of our remaining YEAH hours at this time, every Saturday!
  - This assignment will be due **Friday, January 29th at the start of class. Partners are not permitted.**
This assignment only consists of two parts! (But they’ll be more involved 😊)
This assignment *only* consists of two parts! (But they’ll be more involved 😛)

- Part 1 - Rising Tides
  - Implement a Breadth-First Search to examine rising sea levels!
This assignment *only* consists of two parts! (But they’ll be more involved 😊)

- **Part 1 - Rising Tides**
  - Implement a Breadth-First Search to examine rising sea levels!
- **Part 2 - You Got Hufflepuff!**
  - Ever want to be a BuzzFeed quiz maker? Me neither, but you need to do this to complete the assignment.
Are you ready?
Are you ready?

This is where the "Fun with Collections" begins
Part 1: Rising Tides

- In this part of the assignment, you’ll be simulating water flooding over a terrain with topography represented as a Grid<double>.
Part 1: Rising Tides

- In this part of the assignment, you’ll be simulating **water flooding** over a terrain with topography represented as a `Grid<double>`.

![Grid representations of topography]

Take a look at these grid representations of topography. Can you intuit anything about the landscapes?
Part 1: Rising Tides

- For any given landscape, assume that a water source exists somewhere in the world.

Here, it’s in the top-left corner!
Part 1: Rising Tides

- For any given landscape, assume that a water source exists somewhere in the world.
- Water can flow to adjacent squares in the four cardinal directions (N, W, E, S), as long as the height of a square is \textit{less than or equal} to the predefined water level.

A different water level can change how far water can go in the same topography!
Part 1: Rising Tides

- For any given landscape, assume that a water source exists somewhere in the world.
- Water can flow to adjacent squares in the four cardinal directions (N, W, E, S), as long as the height of a square is less than or equal to the predefined water level.

Depending on the water level and the topography, the spread can go nuts!
Part 1: Rising Tides

- For any given landscape, assume that a water source exists somewhere in the world.
- Water can flow to adjacent squares in the four cardinal directions (N, W, E, S), as long as the height of a square is **less than or equal** to the predefined water level.
- Keep in mind that there can be multiple water sources!

<table>
<thead>
<tr>
<th>Water sources at top-left and bottom-right corners</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Water sources grid" /></td>
</tr>
<tr>
<td><strong>Height: 0m</strong></td>
</tr>
</tbody>
</table>
Part 1: Rising Tides

- Your job is to implement this function:

```cpp
Grid<bool> floodedRegionsIn(const Grid<double>& terrain,
                           const Vector<GridLocation>& sources,
                           double height);
```
Part 1: Rising Tides

- Your job is to implement this function:

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

- Here, `terrain` is the grid representing our topography, `sources` contains
  the locations of our water sources, and `height` represents the highest value
  that water can still spread to.
Part 1: Rising Tides

- Your job is to implement this function:

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

- Here, `terrain` is the grid representing our topography, `sources` contains the locations of our water sources, and `height` represents the highest value that water can still spread to.

- A `GridLocation` is a helpful struct that stores a `{row, col}` pair! Here’s how you can use it:

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

You can also use `GridLocation` to access into a grid like so:
```cpp
GridLocation gl = {1,2};
Terrain[gl] = 0.0;
```
Part 1: Rising Tides

- Your job is to implement this function:

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

- You will be filling in `terrain` accordingly so that water in `sources` will flow to all reachable cells at or under `height`. 
Part 1: Rising Tides

● Your job is to implement this function:

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

● You will be filling in `terrain` accordingly so that water in `sources` will flow to all reachable cells at or under `height`.
  ○ To fill a cell in the grid, set its value to `True` (cells above land are set to `False` by default)
Part 1: Rising Tides

● Your job is to implement this function:

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

● You will be filling in `terrain` accordingly so that water in `sources` will flow to all reachable cells at or under `height`.
  ○ To fill a cell in the grid, set its value to `True` (cells above land are set to `False` by default)

● To do this, you’ll need to implement a Breadth First Search (BFS) algorithm.
Part 1: Rising Tides

● Your job is to implement this function:

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

● You will be filling in `terrain` accordingly so that water in `sources` will flow to all reachable cells at or under `height`.
  ○ To fill a cell in the grid, set its value to `True` (cells above land are set to `False` by default)

● To do this, you’ll need to implement a Breadth First Search (BFS) algorithm.
  ○ Let’s talk a little more about this one.
It’s time for...
Search Algorithms with Al Gore: BFS edition

Al Gore, Former Vice-President and Algorithmic fiend, pictured grappling with “Fun with Collections”
Breadth First Search

- A good way to think about BFS is with a literal flood or spill, where a source expands outwards to all reachable locations.
Breadth First Search

- A good way to think about BFS is with a literal flood or spill, where a source expands outwards to all reachable locations.
  - Here’s an example of a BFS “exploring”, or “flooding” a maze
    - Furthermore, you can imagine how this would be a valid way to find an exit in a maze!
Here is **very good** pseudocode for how to write up a BFS to solve this problem:

```java
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;
    }
  }
}
```
Here is very good pseudocode for how to write up a BFS to solve this problem:

```plaintext
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;
        }
    }
}
```
Breadth First Search

- Here is very good pseudocode for how to write up a BFS to solve this problem:

```python
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;
        }
    }
}
```

Here, you’ll need to figure out how to write a loop that loops through only the 4 cardinal neighbors of the current location! You’ll have to be creative here if you don’t want to write redundant code :)

This is the tricky part!
Here is very good pseudocode for how to write up a BFS to solve this problem:

```plaintext
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;
        }
    }
}
```

Can I make anything clearer on this slide? Any questions?
Part 1: Rising Tides

• Once you get that BFS up and running, you should be good to go!
Part 1: Rising Tides

- Once you get that BFS up and running, you should be good to go!
  - Please add at least ONE custom test case to the test harness before you move on, however. We won’t test for everything with the provided tests!
Part 1: Rising Tides

- Once you get that BFS up and running, you should be good to go!
  - Please add at least ONE custom test case to the test harness before you move on, however. We won’t test for everything with the provided tests!
- A few final thoughts about the problem:
Part 1: Rising Tides

- Once you get that BFS up and running, you should be good to go!
  - Please add at least ONE custom test case to the test harness before you move on, however. We won’t test for everything with the provided tests!

- A few final thoughts about the problem:
  - There doesn’t need to be a correlation between cells in the terrain. Neighboring cells can go from large positive values to negative values!
Part 1: Rising Tides

- Once you get that BFS up and running, you should be good to go!
  - Please add at least ONE custom test case to the test harness before you move on, however. We won’t test for everything with the provided tests!

- A few final thoughts about the problem:
  - There doesn’t need to be a correlation between cells in the terrain. Neighboring cells can go from large positive values to negative values!
  - Remember to use the `grid.inBounds()` function so that you don’t go off the grid during your BFS!
Part 1: Rising Tides

- Once you get that BFS up and running, you should be good to go!
  - Please add at least ONE custom test case to the test harness before you move on, however. We won’t test for everything with the provided tests!
- A few final thoughts about the problem:
  - There doesn’t need to be a correlation between cells in the terrain. Neighboring cells can go from large positive values to negative values!
  - Remember to use the `grid.inBounds()` function so that you don’t go off the grid during your BFS!
  - If the initial height of a source block is higher than the flood level, you shouldn’t flood anything.
Questions about part 1?

What are things like ten years from now in 2020? We have this new "Bitcoin" thing—does it ever catch on and become normal?

It's still around. I just bought a bottle of hand sanitizer for one bitcoin.

Cool, that sounds pretty normal.

Well, here's the thing...
Part 2: You Got Hufflepuff!

- You know these quizzes?
Part 2: You Got Hufflepuff!

- You know these quizzes?
Part 2: You Got Hufflepuff!

- You know these quizzes?
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on BuzzFeed, or other wastes of time popular websites.
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on Buzzfeed, or other wastes of time popular websites.
- More specifically, you’ll be responsible for writing the functionality to do a few things:
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on Buzzfeed, or other wastes of time popular websites.
- More specifically, you’ll be responsible for writing the functionality to do a few things:
  - Pick a random question to ask out of a \textit{Set} of questions
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on Buzzfeed, or other wastes of time popular websites.
- More specifically, you’ll be responsible for writing the functionality to do a few things:
  - Pick a random question to ask out of a Set of questions
  - Turn a collection of question/answer pairs into a “personality score”
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on Buzzfeed, or other wastes of time popular websites.
- More specifically, you’ll be responsible for writing the functionality to do a few things:
  - Pick a random question to ask out of a Set of questions
  - Turn a collection of question/answer pairs into a “personality score”
  - Normalize those scores to account for sampling differences
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on Buzzfeed, or other wastes of time popular websites.
- More specifically, you’ll be responsible for writing the functionality to do a few things:
  - Pick a random question to ask out of a set of questions
  - Turn a collection of question/answer pairs into a “personality score”
  - Normalize those scores to account for sampling differences
  - Take the cosine similarity of two personality scores to determine their closeness
Part 2: You Got Hufflepuff!

- In this last part of the assignment, you’ll be implementing a personality quiz, like the ones you see on Buzzfeed, or other wastes of time popular websites.
- More specifically, you’ll be responsible for writing the functionality to do a few things:
  - Pick a random question to ask out of a set of questions
  - Turn a collection of question/answer pairs into a “personality score”
  - Normalize those scores to account for sampling differences
  - Take the cosine similarity of two personality scores to determine their closeness
  - Find the best match between a user’s personality score and the personality scores of fictional characters
Milestone 1: Select Random Questions

- In this first milestone, you need to implement the following function:

    Question randomQuestionFrom(Set<Question>& unaskedQuestions);
Milestone 1: Select Random Questions

- In this first milestone, you need to implement the following function:
  ```
  Question randomQuestionFrom(Set<Question>& unaskedQuestions);
  ```
- For now, don’t worry about what a Question struct holds.
Milestone 1: Select Random Questions

- In this first milestone, you need to implement the following function:
  
  ```
  Question randomQuestionFrom(Set<Question>& unaskedQuestions);
  ```

- For now, don’t worry about what a Question struct holds.
  
  - An important part of Computer Science is being able to work with objects without necessarily knowing the underlying implementations! This is the key behind abstraction.
Milestone 1: Select Random Questions

- In this first milestone, you need to implement the following function:

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

- For now, don’t worry about what a `Question` struct holds.
  - An important part of Computer Science is being able to work with objects without necessarily knowing the underlying implementations! This is the key behind *abstraction*.

- Your job is to pick a random element from `unaskedQuestions`, remove it from the set, and return it!
Milestone 1: Select Random Questions

- In this first milestone, you need to implement the following function:

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

- For now, don’t worry about what a `Question` struct holds.
  - An important part of Computer Science is being able to work with objects without necessarily knowing the underlying implementations! This is the key behind abstraction.

- Your job is to pick a random element from `unaskedQuestions`, remove it from the set, and return it!
  - The function `randomElement(someSet)` might be helpful here :)
Milestone 1: Select Random Questions

- In this first milestone, you need to implement the following function:

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

- For now, don’t worry about what a `Question` struct holds.
  - An important part of Computer Science is being able to work with objects without necessarily knowing the underlying implementations! This is the key behind abstraction.

- Your job is to pick a random element from `unaskedQuestions`, remove it from the set, and return it!
  - The function `randomElement(someSet)` might be helpful here :)

- Any questions? This first part isn’t meant to trip you up : ]
Milestone 2: Compute Scores from Question/Answer Pairs

- Now we can take a closer look at the `Question` struct!

```c++
struct Question {
    string questionText;
    Map<char, int> factors;
};
```
Now we can take a closer look at the `Question` struct!

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

We don’t care too much about the body of the question, but we do care about the `factors` map, because it stores both a question’s personality factors, and their weights!
Milestone 2: Compute Scores from Question/Answer Pairs

- Now we can take a closer look at the `Question` struct!
  ```c
  struct Question {
    string questionText;
    Map<char, int> factors;
  }
  ```

- We don’t care too much about the body of the question, but we do care about the `factors` map, because it stores **both** a question’s personality factors, and their weights!
  
  - For your assignment, you’ll be using a 5-factor personality score called OCEAN (openness, conscientiousness, extraversion, agreeableness, and neuroticism)
Milestone 2: Compute Scores from Question/Answer Pairs

- Now we can take a closer look at the `Question` struct!

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

- We don’t care too much about the body of the question, but we do care about the `factors` map, because it stores both a question’s personality factors, and their weights!
  - For your assignment, you’ll be using a 5-factor personality score called OCEAN (openness, conscientiousness, extraversion, agreeableness, and neuroticism)
  - Weights are either +1 or -1. A weight of zero will simply not manifest a key in the map.
Milestone 2: Compute Scores from Question/Answer Pairs

- Here’s what the contents of factors might look like:

<table>
<thead>
<tr>
<th>O</th>
<th>C</th>
<th>E</th>
<th>A</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td></td>
<td></td>
<td>+1</td>
<td></td>
</tr>
</tbody>
</table>
Milestone 2: Compute Scores from Question/Answer Pairs

- Here’s what the contents of factors might look like:

<table>
<thead>
<tr>
<th>O</th>
<th>C</th>
<th>E</th>
<th>A</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
</tr>
</tbody>
</table>

- This signifies that the given question would attribute +1 to both ‘O’ and ‘N’ categories.
Milestone 2: Compute Scores from Question/Answer Pairs

- Here’s what the contents of factors might look like:

<table>
<thead>
<tr>
<th>O</th>
<th>C</th>
<th>E</th>
<th>A</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td></td>
<td></td>
<td>+1</td>
<td></td>
</tr>
</tbody>
</table>

- This signifies that the given question would attribute +1 to both ‘O’ and ‘N’ categories.
  - Notice how the other 3 factors have no value in the map! This means this question didn’t address those factors!
Milestone 2: Compute Scores from Question/Answer Pairs

- In this part, you’ll need to implement the following function:

  ```
  Map<char, int> scoresFrom(const Map<Question, int>& answers);
  ```
Milestone 2: Compute Scores from Question/Answer Pairs

- In this part, you’ll need to implement the following function:

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

- Where, given a map of user responses to questions, returns an aggregate of the user’s personality scores.
Milestone 2: Compute Scores from Question/Answer Pairs

- In this part, you’ll need to implement the following function:

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

- Where, given a map of user responses to questions, returns an aggregate of the user’s personality scores.
- Here’s what the map `answers` might look like:

<table>
<thead>
<tr>
<th>Question</th>
<th>Factors</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am quick to understand things.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>I go my own way.</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>I know no limits.</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>I become overwhelmed by events.</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Remember that inside of each Question struct is the question text AND the factors map!
Milestone 2: Compute Scores from Question/Answer Pairs

- Some more notes about this diagram:

<table>
<thead>
<tr>
<th>Question</th>
<th>Factors</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am quick to understand things.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>I go my own way.</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>I know no limits.</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>I become overwhelmed by events.</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Some more notes about this diagram:
  ○ Each **answer** integer corresponds to a different weight. 5 corresponds to “**strongly agree**” and 1 corresponds to “**strongly disagree**”

<table>
<thead>
<tr>
<th>Question</th>
<th>Factors</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am quick to understand things.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>I go my own way.</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>I know no limits.</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>I become overwhelmed by events.</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Milestone 2: Compute Scores from Question/Answer Pairs

- Some more notes about this diagram:
  - Each answer integer corresponds to a different weight. 5 corresponds to “strongly agree” and 1 corresponds to “strongly disagree”
  - An answer of 3 corresponds to indifference, and doesn’t contribute any values to one’s personality score. This set of question/answers simply didn’t have any responses of 3.

<table>
<thead>
<tr>
<th>Question</th>
<th>Factors</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am quick to understand things</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>I go my own way.</td>
<td>+1 -1</td>
<td>4</td>
</tr>
<tr>
<td>I know no limits.</td>
<td>1 -1</td>
<td>2</td>
</tr>
<tr>
<td>I become overwhelmed by events</td>
<td>-1 -1</td>
<td>1</td>
</tr>
</tbody>
</table>
Milestone 2: Compute Scores from Question/Answer Pairs

- Some more notes about this diagram:
  - Each answer integer corresponds to a different weight. 5 corresponds to “strongly agree” and 1 corresponds to “strongly disagree”
  - An answer of 3 corresponds to indifference, and doesn’t contribute any values to one’s personality score. This set of question/answers simply didn’t have any responses of 3.

<table>
<thead>
<tr>
<th>Question</th>
<th>Factors</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am quick to understand things.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>I go my own way.</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>I know no limits.</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>I become overwhelmed by events.</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Milestone 2: Compute Scores from Question/Answer Pairs

- Some more notes about this diagram:
  - Each answer integer corresponds to a different weight. 5 corresponds to “strongly agree” and 1 corresponds to “strongly disagree”
  - An answer of 3 corresponds to indifference, and doesn’t contribute any values to one’s personality score. This set of question/answers simply didn’t have any responses of 3.
  - This diagram shows a sampler of only 4 question/answer pairs. You should expect to see multiple of each answer (i.e. more than one of each answer number) in the maps that you have to process!

<table>
<thead>
<tr>
<th>Question</th>
<th>Factors</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am quick to understand things.</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>I go my own way.</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>I know no limits.</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>I become overwhelmed by events.</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Milestone 2: Compute Scores from Question/Answer Pairs

- Here’s how you can aggregate the user’s score:
Milestone 2: Compute Scores from Question/Answer Pairs

- Here’s how you can aggregate the user’s score:
  - If a question has an answer of 3, you can disregard the question. You **should not** make keys in the map for this question’s factors.
Milestone 2: Compute Scores from Question/Answer Pairs

Here’s how you can aggregate the user’s score:

- If a question has an answer of 3, you can disregard the question. You **should not** make keys in the map for this question’s factors.
- When a question has a non-3 answer, for each of the char factors for a question, you’ll need to multiply the corresponding integer value according to this table:

<table>
<thead>
<tr>
<th>Response</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*-2</td>
</tr>
<tr>
<td>2</td>
<td>*-1</td>
</tr>
<tr>
<td>4</td>
<td>*1</td>
</tr>
<tr>
<td>5</td>
<td>*2</td>
</tr>
</tbody>
</table>
Milestone 2: Compute Scores from Question/Answer Pairs

Here’s how you can aggregate the user’s score:

- If a question has an answer of 3, you can disregard the question. You **should not** make keys in the map for this question’s factors.
- When a question has a non-3 answer, for each of the **char** factors for a question, you’ll need to multiply the corresponding **integer** value according to this table:

<table>
<thead>
<tr>
<th>Response</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-2</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Applying this, if a question has factors ‘E’ → +1, ‘A’ → -1, and the user has an answer ‘1’, you should add (-2) to the aggregate ‘E’ score and (+2) to the aggregate ‘A’ score in the map<char,int> that you’ll return.
Milestone 2: Compute Scores from Question/Answer Pairs

- Here’s how you can aggregate the user’s score:
  - If a question has an answer of 3, you can disregard the question. You **should not** make keys in the map for this question’s factors.
  - When a question has a non-3 answer, for each of the char factors for a question, you’ll need to multiply the corresponding integer value according to this table:

<table>
<thead>
<tr>
<th>Response</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*-2</td>
</tr>
<tr>
<td>2</td>
<td>*-1</td>
</tr>
<tr>
<td>4</td>
<td>*1</td>
</tr>
<tr>
<td>5</td>
<td>*2</td>
</tr>
</tbody>
</table>

Applying this, if a question has factors ‘E’ \(\rightarrow+1\), ‘A’ \(\rightarrow-1\), and the user has an answer ‘1’, you should add \((-2)\) to the aggregate ‘E’ score and \((+2)\) to the aggregate ‘A’ score in the map<\text{char, int}> that you’ll return.

Once you’ve done this for all of the provided question/answers, return the map!
Milestone 2: Compute Scores from Question/Answer Pairs

- Some final notes about this problem:
Milestone 2: Compute Scores from Question/Answer Pairs

- Some final notes about this problem:
  - You can assume that the answers you get are always 1-5. That being said, if you wanted to write a more robust program, you could introduce some error handling for inputs outside of that range!
Milestone 2: Compute Scores from Question/Answer Pairs

- Some final notes about this problem:
  - You can assume that the answers you get are always 1-5. That being said, if you wanted to write a more robust program, you could introduce some error handling for inputs outside of that range!
  - You **cannot** assume that the characters you encounter in your questions use the “OCEAN” paradigm!
Milestone 2: Compute Scores from Question/Answer Pairs

● Some final notes about this problem:
  ○ You can assume that the answers you get are always 1-5. That being said, if you wanted to write a more robust program, you could introduce some error handling for inputs outside of that range!
  ○ You **cannot** assume that the characters you encounter in your questions use the “OCEAN” paradigm!
    ■ This means you can’t pre-define a map with 5 keys in it!
Milestone 2: Compute Scores from Question/Answer Pairs

- Some final notes about this problem:
  - You can assume that the answers you get are always 1-5. That being said, if you wanted to write a more robust program, you could introduce some error handling for inputs outside of that range!
  - You **cannot** assume that the characters you encounter in your questions use the “OCEAN” paradigm!
    - This means you can’t pre-define a map with 5 keys in it!
    - To get around this, you could use something called map **auto-insertion**. Basically, when you write code like this:
      ```
      Map<char, int> myMap;
      myMap['Q'] += 1;
      ```
      The map will attempt to look for key ‘Q’ and add to its value, but if the key is not present, it will **automatically insert** the key with a default value (for integers, it’s zero!)
Milestone 2: Compute Scores from Question/Answer Pairs

- A few more reminders:
Milestone 2: Compute Scores from Question/Answer Pairs

- A few more reminders:
  - Remember to skip questions that have answer ‘3’! This will help you keep unused keys out of the map!
Milestone 2: Compute Scores from Question/Answer Pairs

- A few more reminders:
  - [UPDATED] Even if a factor is only represented through answers of ‘3’, you shouldn’t exclude its key from the map. Just include it with a value ‘0’.
  - Here’s an easy way to loop through a map:

```java
for (char key : myMap) {
    // myMap[key] is the particular value
}
```
Milestone 2: Compute Scores from Question/Answer Pairs

- A few more reminders:
  - [UPDATED] Even if a factor is only represented through answers of '3', you shouldn’t exclude its key from the map. Just include it with a value ‘0’.
  - Here’s an easy way to loop through a map:

```java
for (char key : myMap) {
    // myMap[key] is the particular value
}
```

This is probably the most challenging milestone, so plan accordingly!

Any questions?
Milestone 3: Normalize Scores

- Now it’s time to do some math! You’ll be implementing this function:

```cpp
Map<char, double> normalize(const Map<char, int>& scores);
```
Milestone 3: Normalize Scores

- Now it’s time to do some math! You’ll be implementing this function:

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

- Your job is to loop through the `scores` map and divide each score by the following value: 
  
  \[ \sqrt{o^2 + c^2 + e^2 + a^2 + n^2} \]
Milestone 3: Normalize Scores

- Now it's time to do some math! You'll be implementing this function:

\[
\text{Map<}\text{char, double> normalize(const Map<}\text{char, int>}& \text{scores);}
\]

- Your job is to loop through the scores map and divide each score by the following value:

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

  - Recall that not all maps will contain the OCEAN keys -- in general form, this is simply the square root of the sum of all values in the map individually squared.
Milestone 3: Normalize Scores

Now it’s time to do some math! You’ll be implementing this function:

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

Your job is to loop through the `scores` map and divide each score by the following value:

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

- Recall that not all maps will contain the OCEAN keys -- in general form, this is simply the square root of the sum of all values in the map individually squared.

You can then store these values in a new map with identical keys to `scores`. The values will be the old integer values divided by the above calculation.
Milestone 3: Normalize Scores

- A few notes about this milestone:
  - You should compute the square root sum first, and *then* populate the resultant map!
Milestone 3: Normalize Scores

- A few notes about this milestone:
  - You should compute the square root sum first, and *then* populate the resultant map!
  - The keys in the resulting map should be exactly the same as the original map.
Milestone 3: Normalize Scores

- A few notes about this milestone:
  - You should compute the square root sum first, and *then* populate the resultant map!
  - The keys in the resulting map should be exactly the same as the original map.
  - Another reminder that the keys do **not** need to be OCEAN!
Milestone 3: Normalize Scores

- A few notes about this milestone:
  - You should compute the square root sum first, and then populate the resultant map!
  - The keys in the resulting map should be exactly the same as the original map.
  - Another reminder that the keys do not need to be OCEAN!
  - If you include the `<cmath>` header, you can use the `sqrt()` function! Be aware that it returns a double!
Milestone 3: Normalize Scores

- A few notes about this milestone:
  - You should compute the square root sum first, and *then* populate the resultant map!
  - The keys in the resulting map should be exactly the same as the original map.
  - Another reminder that the keys do **not** need to be OCEAN!
  - If you `#include` the `<cmath>` header, you can use the `sqrt()` function! Be aware that it returns a double!
  - **IMPORTANT:** If the input map contains ONLY zero values, you should raise an error(). This is because we’d be forced to divide by zero!

Any questions?
Milestone 4: Implement Cosine Similarity

- Don’t worry! This part isn’t as bad as it sounds :)


Milestone 4: Implement Cosine Similarity

- You’ll need to implement this function:

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

which returns a single double signifying how similar two normalized personality scores are! (This value is bounded between -1 and 1)
Milestone 4: Implement Cosine Similarity

- You’ll need to implement this function:

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

which returns a single double signifying how similar two normalized personality scores are! (This value is bounded between -1 and 1)

- Here is how you do this calculation (remember that this is OCEAN specific, and you’ll need to generalize yours! (What do you do if the maps have different keys?)

\[
similarity = o_1 o_2 + c_1 c_2 + e_1 e_2 + a_1 a_2 + n_1 n_2.
\]

You can assume that these scores are normalized :). Any questions?
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
- You finally get to find a given user’s best match by implementing this function:

```cpp
Person mostSimilarTo(const Map<char, int>& scores, const Set<Person>& people);
```
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
- You finally get to find a given user’s best match by implementing this function:

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

- You’ll need to normalize the user’s scores, and then loop through the Set of people. Here’s what a Person looks like
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
- You finally get to find a given user’s best match by implementing this function:

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

- You’ll need to normalize the user’s scores, and then loop through the Set of people. Here’s what a Person looks like
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
- You finally get to find a given user’s best match by implementing this function:

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

- You’ll need to normalize the user’s scores, and then loop through the Set of people. Here’s what a Person looks like

  oops.
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
- You finally get to find a given user’s best match by implementing this function:

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

- You’ll need to normalize the user’s scores, and then loop through the Set of people. Here’s what a `Person` looks like

```cpp
struct Person {
    string name;
    Map<char, int> scores;
};
```
Milestone 5: Find The Best Match!

- Nice work! You’ve made it to the last milestone!
- You finally get to find a given user’s best match by implementing this function:

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

- You’ll need to normalize the user’s scores, and then loop through the Set of people. Here’s what a Person looks like:

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

As you can see, each Person has a scores Map! For each person, you’ll need to normalize their score and then find the person who has the highest cosine similarity to the original user’s score. This is the Person you should return!
Milestone 5: Find The Best Match!

- Some final notes on this problem:
  - Don’t try and take the cosine similarity of a score before you normalize it!
Milestone 5: Find The Best Match!

- Some final notes on this problem:
  - Don’t try and take the cosine similarity of a score before you normalize it!
  - If the People Set is empty, you should throw an error!
Milestone 5: Find The Best Match!

- Some final notes on this problem:
  - Don’t try and take the cosine similarity of a score before you normalize it!
  - If the People \( \text{Set} \) is empty, you should throw an error!
  - You can break ties however you’d like!
Milestone 5: Find The Best Match!

- Some final notes on this problem:
  - Don’t try and take the cosine similarity of a score before you normalize it!
  - If the People Set is empty, you should throw an error!
  - You can break ties however you’d like!
  - Be aware that cosine similarities can be positive or negative. This means that a user can match with someone with a negative cosine similarity!

Questions about this last part?
Part 3: Revel in your Creations!

- Once all of your tests pass, you can run the GUI portions for your work! This is **optional**, but I’d recommend it :)

Choose which demo to run.
That's it!

● Congrats! We hope you had fun with collections :)

Stack Efron, High School Musician and CS106B alum, congratulates you on your hard work!