Midterm Exam

Instructions

This exam is due on Sunday, February 6th at 10:30AM. No late submissions will be accepted.

You have the full 48 hours to work on the midterm. There are no time limits on the exam other than that it has to come in before the deadline. That means that it's fine to start working on the exam, take a break, go for a walk or hike in the area, get a good night's sleep, come back, pick up where you left off, etc. Although you have the full 48 hours to complete the exam, the exam itself is designed to take around three hours to complete.

The starter files for the exam are available here:

 Midterm Starter Files

Submission instructions are at the bottom of this page.

Honor Code Policies

This exam is open-book and open-note, so you are free to make use of all course materials on the course website and on Canvas, including lecture slides and lecture videos. You are also permitted to search online for conceptual information (for example, by visiting Wikipedia). However, ***this exam must be completed individually***. It is a violation of the Stanford Honor Code to communicate with any other humans about the exam, to solicit solutions to the exam, or to share your solutions with others.

You are not permitted to communicate with other humans about the exam or to solicit help from others. For example, you ***must not*** communicate with other students in the course about the exam, and you ***must not*** ask questions on sites like Chegg or Stack Overflow.

If you have questions about this exam, you are welcome to post them in EdStem. We will not be able to offer much support other than clarifying questions about what is being asked of you in each problem. If you do post in EdStem, ***you must post privately***; posting publicly violates the "do not communicate with other humans" rule, and ***you must not post code***.

All work done with the assistance of any material in any way (other than provided CS106B course materials) must include a detailed citation (e.g., "I visited the Wikipedia page for *X* on Problem 1 and made use of insights *A*, *B*, and *C*"). ***Copying solutions is never acceptable***, even with citation, and is always a violation of the Honor Code. If by chance you encounter solutions to a problem, navigate away from that page before you feel tempted to copy. If you're worried you've done something you probably shouldn't have and aren't sure what to do, email the course staff before you submit and we'll figure out how to proceed.

Words of Encouragement

Hello Wonderful CS106Bers!

We've covered a lot of ground in the first half of CS106B. Neel, the section leaders, and I are impressed with just how much you've learned and what you've accomplished. Treat this exam as a chance to show off what you've learned so far. ***You can do this.*** Best of luck on the exam!

-Keith

Problem One: Pseudotautonyms (4 Points)

What do the words "intestines," "pullup," "reappear," "redder," "teammate," and "signings" have in common? Each is a ***pseudotautonym***, an even-length string where the characters in the first half can be rearranged to form the back half. For example, with "teammate," we can rearrange the first half of the string ("team") to get the second half ("mate"), and with "pullup" we can rearrange the first half of the string ("pul") to get the second half ("lup"). The true tautonyms - words like "dikdik" and "caracara" that consist of the same string repeated twice - also count as pseudotautonyms because the letters in the first half perfectly match the letters in the back half.

Open the file **`Pseudotautonyms.cpp`** and implement a function

```
bool isPseudotautonym(const string& str);
```

that takes as input a string, then returns whether it's a pseudotautonym. We recommend that you add some **`STUDENT_TEST`**s at the bottom of **`Pseudotautonyms.cpp`** to check whether your solution works correctly, but you are not required to do so. We've placed our own tests in the separate file **`PseudotautonymsTests.cpp`** in case you're curious to see what we're already checking for.

Some notes on this problem:

- For full credit, your solution ***must not*** use recursion.
- Your function should be *case-insensitive*, so, for example, **`isPseudotautonym("TeAmMaTe")`** should return true.
- Although we've focused on words in the above examples, your function should work on strings that contain digits, punctuation, spaces, etc. For example, **`isPseudotautonym("(-: ** :-(")`** should return true.
- However, as you saw in Assignment 2, C++ strings don't play nicely with non-English letters. You aren't expected to handle inputs like "ካሜሩን", for example.
- Your solution should be fast enough to handle the "stress tests" in the starter files.
- The test cases we have provided are by no means exhaustive. You are encouraged to add your own, and we will run your program on a variety of test cases not included with the starter files.

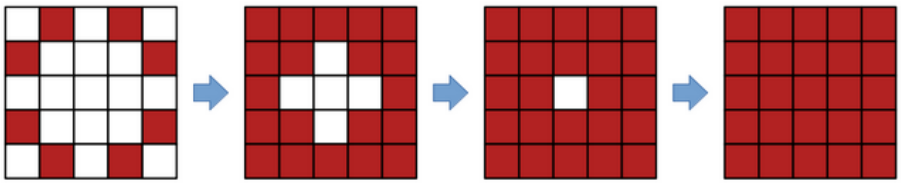
Problem Two: Amoebas (8 Points)

In this problem, you'll simulate the growth of amoebas living in a grid world. We'll represent our world using a **`Grid<bool>`**, where each cell is **`true`** if there's an amoeba present in the cell and **`false`** otherwise. Amoebas spread according to the following two rules:

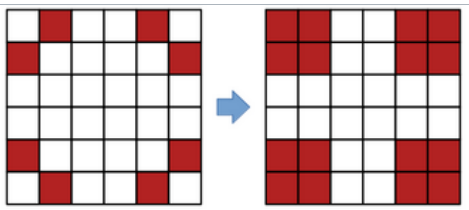
- Amoebas never die, so once an amoeba appears in a grid location, it will stay there forever.
- Whenever a cell is bordered by two or more amoebas along cardinal directions (up, down, left, and right), a new amoeba will appear there.

By repeatedly applying these two rules, we can see what happens as amoebas begin to grow and spread through the world. For example, here's the evolution of a 3 × 3 world containing three amoebas:

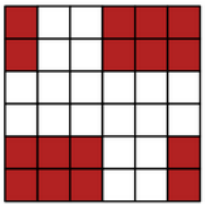
Here's a 5 × 5 world being filled in:



Here's a 6 × 6 world with some amoebas in it. While the amoebas spread, they never fill the full world:



And finally, here's a 6 × 6 world in which no amoebas spread at all:



Open the file **`Amoebas.cpp`** and implement a function

```
Vector<Grid<bool>> simulateAmoebas(const Grid<bool>& world);
```

that takes in a **`Grid<bool>`** representing the initial world, then returns a **`Vector<Grid<bool>>`** showing all the steps that happen as the amoebas spread, along the lines of the evolutions shown above. For example, the first **`Grid<bool>`** in the **`Vector`** should be the initial world, and the last **`Grid<bool>`** in the **`Vector`** should be the very last step in the amoebas spreading.

We recommend that you add your own **`STUDENT_TEST`**s at the bottom of **`Amoebas.cpp`** so that you can increase your confidence that your solution is correct, but you are not required to do so. Our own tests are in **`AmoebasTests.cpp`** in case you'd like to see what we're already checking for.

Some notes on this problem:

- For full credit, your solution ***must not*** use recursion.
- The input **`Grid`** may have any number of rows and any number of columns, and may be empty.
- Your solution should be fast enough to handle the "stress test" in the starter files.
- The test cases we have provided are by no means exhaustive. You are encouraged to add your own, and we will run your program on a variety of test cases not included with the starter files.

Problem Three: Care Teams (10 Points)

Residents are doctors-in-training who work at a hospital under the supervision of a more experienced doctor. Residents get their training on-the-job by working with patients, often with a doctor keeping tabs on them to make sure everything is done safely.

A ***triad*** is a group of three people: one doctor, one patient, and one resident. We can represent such a triad using the following **`struct`**:

```
struct Triad {
    string doctor;
    string resident;
    string patient;
};
```

If there's exactly the same number of doctors, residents, and patients (what a coincidence!), then it's always possible to split the doctors, residents, and patients into triads such that each doctor, each resident, and each patient belongs to exactly one triad. For example, suppose there are doctors *AA* and *BB*, residents *CC* and *DD*, and patients *EE* and *FF*. Then there are four ways we could assign them to triads:

- {*AA*, *CC*, *EE*} and {*BB*, *DD*, *FF*}
- {*AA*, *CC*, *FF*} and {*BB*, *DD*, *EE*}
- {*AA*, *DD*, *EE*} and {*BB*, *CC*, *FF*}
- {*AA*, *DD*, *FF*} and {*BB*, *CC*, *EE*}

If, on the other hand, there are three doctors *AA*, *BB*, and *CC*; three residents *DD*, *EE*, and *FF*; and three patients *GG*, *HH*, and *II*, then there are 36 ways to assign them to triads:

- {*AA*, *DD*, *GG*}, {*BB*, *EE*, *HH*} and {*CC*, *FF*, *II*}
- {*AA*, *DD*, *GG*}, {*BB*, *EE*, *II*} and {*CC*, *FF*, *HH*}
- {*AA*, *DD*, *GG*}, {*BB*, *FF*, *HH*} and {*CC*, *EE*, *II*}
- {*AA*, *DD*, *GG*}, {*BB*, *FF*, *II*} and {*CC*, *EE*, *HH*}
- {*AA*, *DD*, *HH*}, {*BB*, *EE*, *GG*} and {*CC*, *FF*, *II*}
- {*AA*, *DD*, *HH*}, {*BB*, *EE*, *II*} and {*CC*, *FF*, *GG*}
- {*AA*, *DD*, *HH*}, {*BB*, *FF*, *GG*} and {*CC*, *EE*, *II*}
- {*AA*, *DD*, *HH*}, {*BB*, *FF*, *II*} and {*CC*, *EE*, *GG*}
- {*AA*, *DD*, *II*}, {*BB*, *EE*, *GG*} and {*CC*, *FF*, *HH*}
- {*AA*, *DD*, *II*}, {*BB*, *EE*, *HH*} and {*CC*, *FF*, *GG*}
- {*AA*, *DD*, *II*}, {*BB*, *FF*, *GG*} and {*CC*, *EE*, *HH*}
- {*AA*, *DD*, *II*}, {*BB*, *FF*, *HH*} and {*CC*, *EE*, *GG*}
- {*AA*, *EE*, *GG*}, {*BB*, *DD*, *HH*} and {*CC*, *FF*, *II*}
- {*AA*, *EE*, *GG*}, {*BB*, *DD*, *II*} and {*CC*, *FF*, *HH*}
- {*AA*, *EE*, *GG*}, {*BB*, *FF*, *HH*} and {*CC*, *DD*, *II*}
- {*AA*, *EE*, *GG*}, {*BB*, *FF*, *II*} and {*CC*, *DD*, *HH*}
- {*AA*, *EE*, *HH*}, {*BB*, *DD*, *GG*} and {*CC*, *FF*, *II*}
- {*AA*, *EE*, *HH*}, {*BB*, *DD*, *II*} and {*CC*, *FF*, *GG*}
- {*AA*, *EE*, *HH*}, {*BB*, *FF*, *GG*} and {*CC*, *DD*, *II*}
- {*AA*, *EE*, *HH*}, {*BB*, *FF*, *II*} and {*CC*, *DD*, *GG*}
- {*AA*, *EE*, *II*}, {*BB*, *DD*, *GG*} and {*CC*, *FF*, *HH*}
- {*AA*, *EE*, *II*}, {*BB*, *DD*, *HH*} and {*CC*, *FF*, *GG*}
- {*AA*, *EE*, *II*}, {*BB*, *FF*, *GG*} and {*CC*, *DD*, *HH*}
- {*AA*, *EE*, *II*}, {*BB*, *FF*, *HH*} and {*CC*, *DD*, *GG*}
- {*AA*, *FF*, *GG*}, {*BB*, *DD*, *HH*} and {*CC*, *EE*, *II*}
- {*AA*, *FF*, *GG*}, {*BB*, *DD*, *II*} and {*CC*, *EE*, *HH*}
- {*AA*, *FF*, *GG*}, {*BB*, *EE*, *HH*} and {*CC*, *DD*, *II*}
- {*AA*, *FF*, *GG*}, {*BB*, *EE*, *II*} and {*CC*, *DD*, *HH*}
- {*AA*, *FF*, *HH*}, {*BB*, *DD*, *GG*} and {*CC*, *EE*, *II*}
- {*AA*, *FF*, *HH*}, {*BB*, *DD*, *II*} and {*CC*, *EE*, *GG*}
- {*AA*, *FF*, *HH*}, {*BB*, *EE*, *GG*} and {*CC*, *DD*, *II*}
- {*AA*, *FF*, *HH*}, {*BB*, *EE*, *II*} and {*CC*, *DD*, *GG*}
- {*AA*, *FF*, *II*}, {*BB*, *DD*, *GG*} and {*CC*, *EE*, *HH*}
- {*AA*, *FF*, *II*}, {*BB*, *DD*, *HH*} and {*CC*, *EE*, *GG*}
- {*AA*, *FF*, *II*}, {*BB*, *EE*, *GG*} and {*CC*, *DD*, *HH*}
- {*AA*, *FF*, *II*}, {*BB*, *EE*, *HH*} and {*CC*, *DD*, *GG*}

A **`Triad`** represents a single group of three people (one doctor, one resident, and one patient). Each of the items in the above bulleted lists are **`Set<Triad>`**s that represent a way of splitting everyone up into triads (notice that for each bullet point, each doctor, each resident, and each patient appear exactly once). The entire bulleted list, which consists of a group of **`Set<Triad>`**s, is a **`Set<Set<Triad>>`**: a collection of groups of **`Triad`**s.

Open the file **`CareTeams.cpp`** and implement a function

```
Set<Set<Triad>> allCareTeamsFrom(const Set<string>& doctors,
                                 const Set<string>& residents,
                                 const Set<string>& patients);
```

that takes as input three sets - one of doctors, one of residents, and one of patients. It then returns a **`Set<Set<Triad>>`** representing every possible way to split the doctors, residents, and patients into triads. If the three input sets don't all

have the same size, your function should call **error** to report an error.

We'd like you to use the following recursive insight when coding this function up. If there are no doctors left, then there's also no residents left and no patients left and you've grouped everyone into triads. Otherwise, there must be some doctor who isn't part of a triad. Pick any one of them, then try all possible ways of putting that doctor into a triad with a resident and a patient who aren't yet assigned a triad.

You are encouraged to write your own **STUDENT_TEST**s in **CareTeams.cpp**, though you are not required to do so. Our provided tests are located in **CareTeamsTests.cpp**.

Some notes on this problem:

- Your solution ***must*** be recursive. More specifically, your solution ***must*** use the recursive insight mentioned above.
- While your solution does not need to be as fast as possible, your solution should avoid unnecessary sources of inefficiency such as generating the same partition into triads multiple times. For reference, memoization will *not* help you here.
- The number of possible splits into triads grows *extremely* quickly as a function of the number of doctors, residents, and patients. It's okay if your function takes a very, *very* long time to run for the scenario where there are six or more doctors.
- Your solution should be fast enough to pass our provided stress test.
- The test cases we have provided are by no means exhaustive. You are encouraged to add your own, and we will run your program on a variety of test cases not included with the starter files.

Problem Four: Peake Sequences (12 Points)

Consider the following process. Begin with any positive integer (e.g. 1, 137, 106, etc.), then repeatedly take the number you have and either add a positive integer to it or multiply it by an integer greater than one. The sequences generated this way are called ***Peake sequences***. Here's some examples:

- 137
- ((106) + 4) * 3
- (((((2) + 2) * 2) + 2) * 2) + 2
- (((((2) + 1) + 2) + 1) + 2) + 1
- (((((2) * 2) * 2) * 2) * 2) + 2

Open the file **PeakeSequences.cpp** and write a ***recursive*** function

```
int numPeakeSequencesFor(int n);
```

that takes as input a number *n*, then returns the number of Peake sequences that evaluate to *n*. For example, calling **numPeakeSequencesFor(3)** would return 6 because there are six Peake sequences that evaluate to 3:

- ((1) + 1) + 1
- (1) + 2
- ((1) × 2) + 1
- (1) × 3
- (2) + 1
- 3

Notice that (1) + 2 is considered different from (2) + 1. The first sequence corresponds to starting wtih 1 and then adding 2, whereas the second corresponds to starting with 2 and adding 1.

Calling **numPeakeSequencesFor(5)** should return 27, because there are 27 such sequences:

- ((((1) + 1) + 1) + 1) + 1
- (((1) + 1) + 1) + 2
- (((1) + 1) + 2) + 1
- ((1) + 1) + 3
- (((1) + 1) × 2) + 1
- (((1) + 2) + 1) + 1
- ((1) + 2) + 2
- ((1) + 3) + 1
- (1) + 4
- ((((1) × 2) + 1) + 1) + 1
- (((1) × 2) + 1) + 2
- (((1) × 2) + 2) + 1
- ((1) × 2) + 3
- (((1) × 2) × 2) + 1
- (((1) × 3) + 1) + 1
- ((1) × 3) + 2
- ((1) × 4) + 1
- (1) × 5
- (((2) + 1) + 1) + 1
- ((2) + 1) + 2
- ((2) + 2) + 1
- (2) + 3
- ((2) × 2) + 1
- ((3) + 1) + 1
- (3) + 2
- (4) + 1
- 5

The number of Peake sequences for a number *n* grows very quickly. For example, calling **`numPeakeSequencesFor(10)`** should return 1,008. Mercifully, we won't list them all here.

You are encouraged to write your own **`STUDENT_TEST`**s at the bottom of **`PeakeSequences.cpp`**, though you are not required to do so. Our provided tests are in **`PeakeSequencesTests.cpp`**, and you're welcome to look there to see what we're already checking for.

Some notes on this problem:

- Your solution ***must*** be implemented recursively.
- Your solution must be fast enough to pass our provided stress test. ***This will require you to use memoization.*** As with Human Pyramids, we recommend first implementing the solution without memoization, then editing your solution to include memoization. And, as usual, you must not modify the signature of the **`numPeakeSequencesFor`** function.
- To make sure you didn't miss it, *order matters* in Peake sequences. For example, these sequences are all considered different from one another:
  - ((1) + 2) + 3
  - ((1) + 3) + 2
  - ((2) + 1) + 3
  - ((2) + 3) + 1
  - ((3) + 1) + 2
  - ((3) + 2) + 1
- Your function just needs to return how many Peake sequences add up to *n*. You don't need to - and, in fact shouldn't - actually construct what those sequences are.
- The number of Peake sequences grows extremely quickly, so much so that the number of Peake sequences for 32 is too large to hold in a C++ **`int`** variable. You don't need to worry about this - we won't test your code on any values of *n* above 31.
- It's impossible to form a Peake sequence for 0 or for negative numbers. Your function should return 0 in those cases.
- The test cases we have provided are by no means exhaustive. You are encouraged to add your own, and we will run your program on a variety of test cases not included with the starter files.

Submission Instructions

Before you call it done, run through our submit checklist to be sure all your **`t`**s are crossed and **`i`**s are dotted. Make sure

your code follows our style guide. Then upload your completed files to Paperless for grading.

Please submit these files to Paperless:

- **`Pseudotautonyms.cpp`**
- **`Amoebas.cpp`**
- **`CareTeams.cpp`**
- **`PeakeSequences.cpp`**

You don't need to - and, indeed, shouldn't - submit any of the other files in the project folder.

Submit to Paperless

Best of luck on the exam!