Assignment 1: Welcome to C++!

Thanks to Sophia Westwood for suggesting Flesch-Kincaid readability as an assignment.

Here it is – the first programming assignment of the quarter! This assignment is designed to get you comfortable designing and building software in C++. It consists of five problems that collectively play around with control structures, string processing, recursion, and problem decomposition in C++. By the time you've completed this assignment, you'll be a lot more comfortable working in C++ and will be ready to start building larger projects.

Good luck, and have fun!

This assignment must be completed individually. Working in groups is not permitted.
Due Friday, January 18th at the start of class.

This assignment consists of five parts, of which four require writing code. This 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 Stack Overflows the day this assignment is released. You don't need to write any code for it; it's just about working the debugger, something you practiced in Assignment Zero.
- Aim to complete the three recursion problems within five days of this assignment being released. We recommend that you spend a day each on Only Connect, Playing Fair, and Sandpiles, leaving the rest as buffer time. Recursion as a concept can take a bit of time to adjust to, and that's perfectly normal. Allocating some extra buffer time here will give you a chance to tinker around and ask for help if you need it.
- Aim to complete Flesch-Kincaid readability within seven days of this assignment being released. This part of the assignment doesn't involve recursion and is more about learning to break down larger problems into smaller pieces.

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

Problem One: Stack Overflows

Whenever a program calls a function, the computer sets aside memory called a ***stack frame*** for that function call in a region called the ***call stack***. Whenever a function is called, it creates a new stack frame, and whenever a function returns the space for that stack frame is recycled.

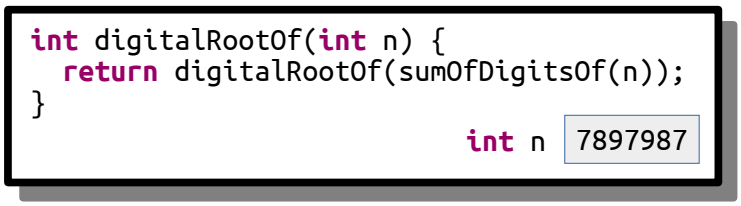
As you saw on Wednesday, whenever a recursive function makes a recursive call, it creates a new stack frame, which in turn might make more stack frames, which in turn might make even more stack frames, etc. For example, when we computed `factorial(5)`, we ended up creating a net of six stack frames: one for each of `factorial(5)`, `factorial(4)`, `factorial(3)`, `factorial(2)`, `factorial(1)`, and `factorial(0)`. They were automatically cleaned up as soon as those functions returned.

But what would happen if you called `factorial` on a very large number, say, `factorial(7897987)`? This would create a *lot* of stack frames; specifically, it will make 7,897,988 of them (one for `factorial(7987987)`, one for `factorial(7987986)`, …, and one for `factorial(0)`). This is such a large number of stack frames that the call stack might not have space to store them. When too many stack frames need to be created at the same time, the result is a ***stack overflow*** and the program will crash.

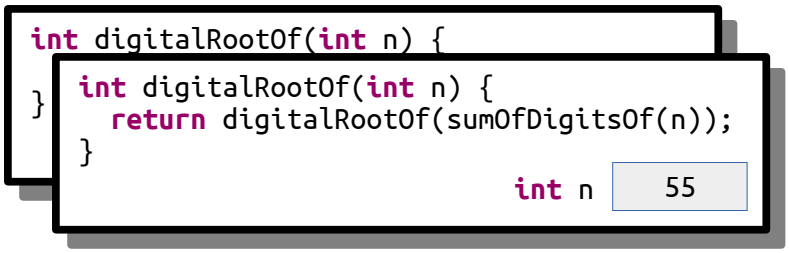
In the previous example, a stack overflow occurs because we need a large but still finite number of stack frames. However, you often see stack overflows resulting due to programming errors. For example, consider the following (buggy!) implementation of the `digitalRootOf` function from Friday's lecture:

```
int digitalRootOf(int n) {
    return digitalRootOf(sumOfDigitsOf(n));
}
```

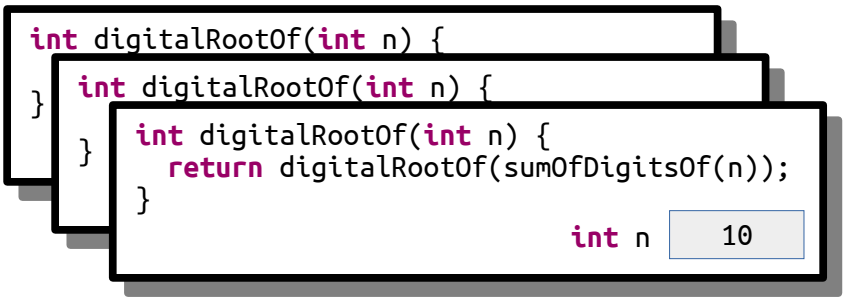
Let's imagine that you try calling `digitalRootOf(7897987)`. The initial stack frame looks like this:



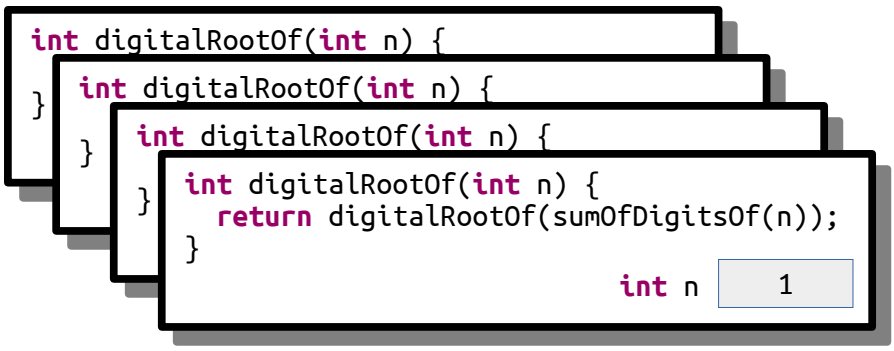
This call now tries to call `digitialRootOf(sumOfDigitsOf(7897987))`. The sum of the digits in the number is 7 + 8 + 9 + 7 + 9 + 8 + 7 = 55, so this fires off a call to `digitalRootOf(55)`, as shown here:



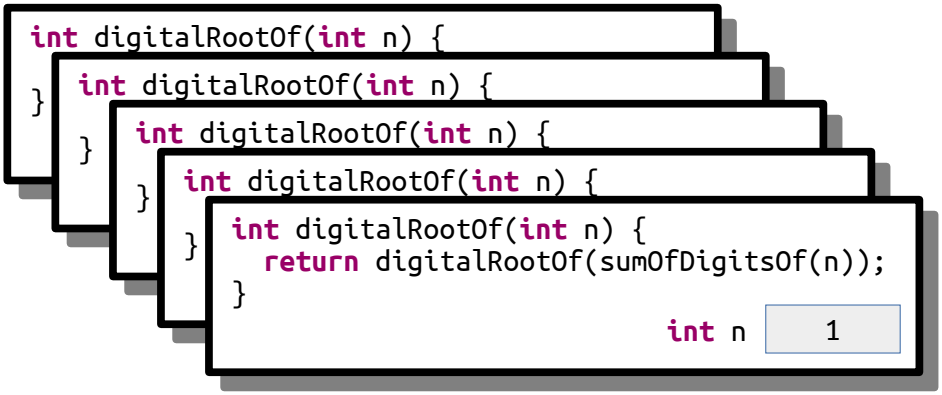
This now calls `digitalRootOf(sumOfDigitsOf(55))`, which ends up calling `digitalRootOf(10)`:



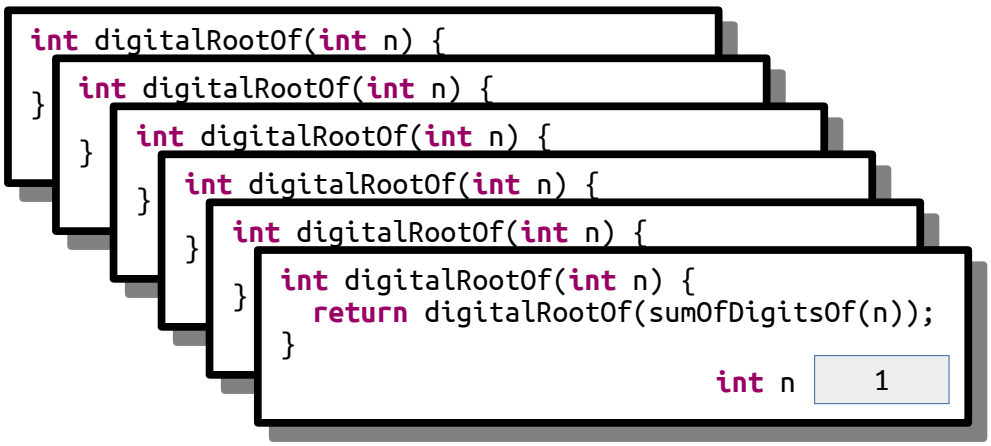
This now calls `digitalRootOf(sumOfDigitsOf(10))`, which ends up calling `digitalRootOf(1)`:



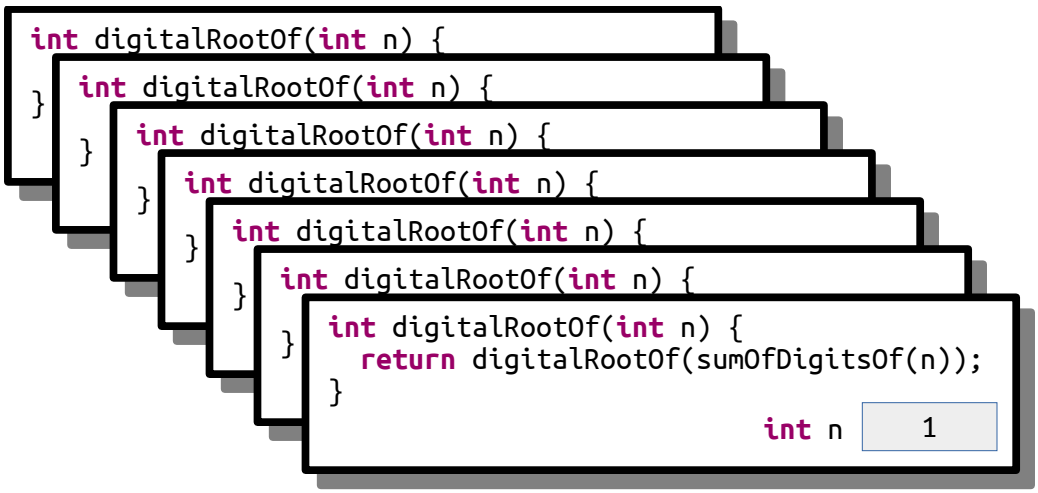
This now call `digitalRootOf(sumOfDigitsOf(1))`, which ends up calling `digitalRootOf(1)` again, as shown here:



This call makes yet another call to `digitalRootOf(1)` for the same reason:



And *that* call makes yet *another* call to `digitalRootOf(1)`:

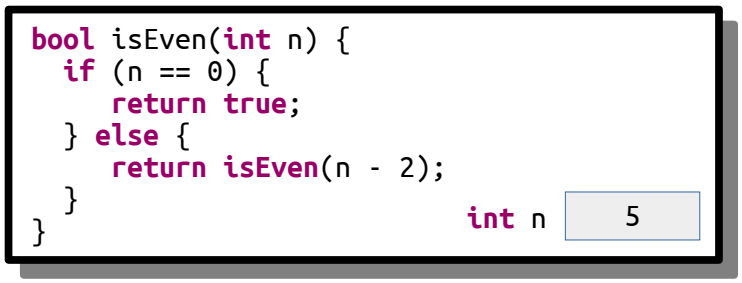


As you can see, this recursion is off to the races. It's like an infinite loop, but with function calls. This code will trigger a stack overflow because at some point it will exhaust the memory in the call stack.

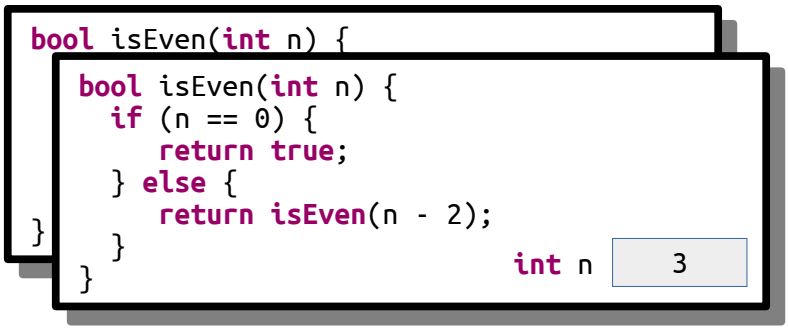
Another place you'll see stack overflows is when you have a recursive function that, for some reason, misses or skips over its base case. For example, let's suppose you want to write a function `isEven` that takes as input a number n and returns whether it's an even number. You note that 0 is even (trust us, it is; take CS103 for details!) and, more generally, a number n is even precisely if $n - 2$ is even. For example, 2 is even because $2 - 2 = 0$ is even, 4 is even because $4 - 2 = 2$ is even, and 6 is even because $6 - 2 = 4$ is even, etc. Based on this (correct) insight, you decide to write this (incorrect) recursive function:

```
bool isEven(int n) {
    if (n == 0) {
        return true;
    } else {
        return isEven(n - 2);
    }
}
```

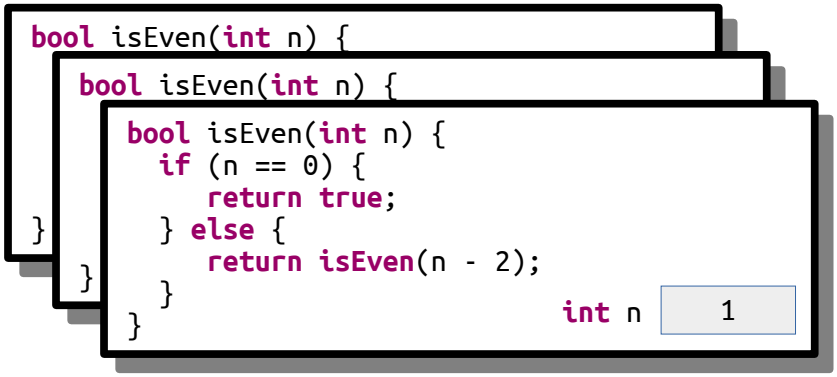
Now, what happens if you call `isEven(5)`? This initially looks like this:



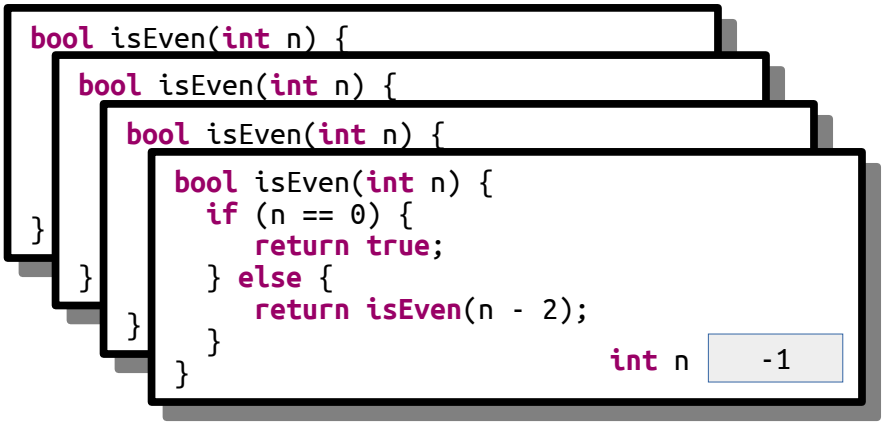
This call will go and call `isEven(3)`, as shown here:



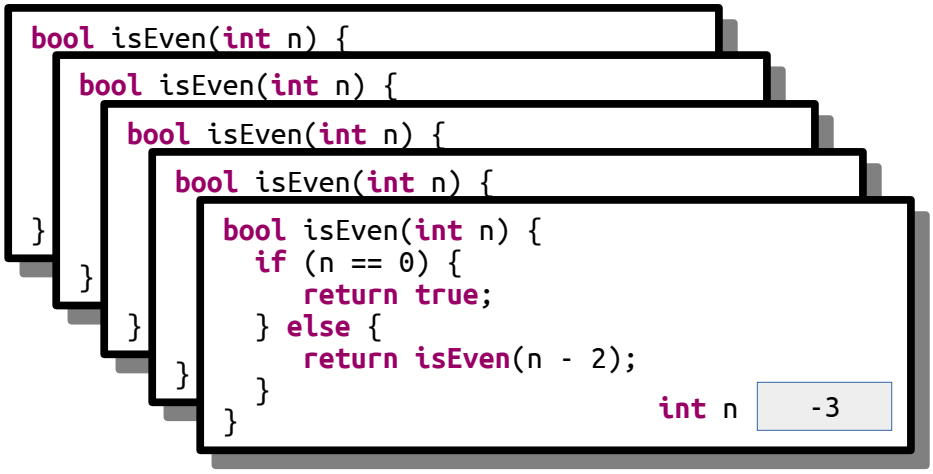
And that call then goes and calls `isEven(1)`:



And here's where things go wrong. This function now calls `isEven(-1)`, skipping over the base case:



This call then calls `isEven(-3)`:



And again we're off to the Stack Overflow Races because this is just going to keep getting more and more negative until we run out of call stack space.

It's important to know about stack overflows because as you start writing your own recursive functions you're likely to run into them in practice. When that happens, don't panic! Instead, pull out your debugger. Look at the call stack – it will be really, really full – and move up and down it, looking at the local variables. As you do, look at the arguments to the functions on the call stack. Are they repeating over and over again? Are they getting more and more negative or bigger and bigger? Based on what you find, you can make progress on debugging your code.

In this part of the assignment, we've included a recursive function that looks like this:

```
void triggerStackOverflow(int index) {
    triggerStackOverflow(kGotoTable[index]);
}
```

Here, `kGotoTable` is a giant (1024-element) array of the numbers 0 through 1023 that have been randomly permuted (see next week's section handout for how to do this!) This function looks up its argument in the table, then makes a recursive call using that argument. As a result, the series of recursive calls made is extremely hard to predict by hand, and since the recursion never stops (do you see why?) this code will always trigger a stack overflow.

Our starter code includes the option to call this function passing in 137 as an initial value (click "Stack Overflows"). Run the provided starter code in debug mode (hey, you learned how to do that in Assignment Zero!) and trigger the stack overflow. You'll get an error message that depends on your OS (it could be something like "segmentation fault," "access violation," "stack overflow," or something like that) and the debugger should pop up. Walk up and down the call stack and see if you can see the sequence of values passed in as parameters to `triggerStackOverflow`.

We've specifically crafted the numbers in `kGotoTable` so that the calls in `triggerStackOverflow` form a cycle. Specifically, `triggerStackOverflow(137)` calls `triggerStackOverflow(`*x*`)` for some number *x*, and that calls `triggerStackOverflow(`*y*`)` for some number *y*, and that calls `triggerStackOverflow(`*z*`)` for some number *z*, etc. until eventually there's some number *w* where `triggerStackOverflow(`*w*`)` calls `triggerStackOverflow(137)`, starting the cycle anew.

Your task in this part of the assignment is to tell us the sequence of the numbers in the cycle. For example, if the sequence was

`triggerStackOverflow(137)` calls
`triggerStackOverflow(106)`, which calls
`triggerStackOverflow(271)`, which calls
`triggerStackOverflow(137)`, which calls
`triggerStackOverflow(106)`, which calls
`triggerStackOverflow(271)`, which calls
`triggerStackOverflow(137)`, which calls
`triggerStackOverflow(106)`, which calls
…

Then you would give us the cycle 137, 106, 271, 137.

Update the file comments at the top of `StackOverflow.cpp` to list the cycle you found.

Some notes on this part of the assignment:

- The topmost entry on the call stack might be corrupted and either not show a value or show the wrong number. Don't worry if that's the case – just move down an entry in the stack.
- Remember that if function *A* calls function *B*, then function *B* will appear higher on the call stack than function *A* because function *B* was called more recently than function *A*. Make sure you don't report the cycle in reverse order!
- When you run the program in Debug mode, Qt Creator will switch to Debug View, which has a bunch of windows and side panels to view information about the running program. That's great when you're debugging, and not so great when you're just trying to write code. You can switch back to the regular code-writing view by clicking the "Edit" button in the left side pane.

Problem Two: Only Connect

The last round of the British quiz show *Only Connect* consists of puzzles of the following form: can you identify these movie titles, given that all characters except consonants have been deleted?

`BTNDTHBST` `MN` `CTCHMFCN` `SRSMN`

The first is "**B**eau**t**y a**nd** **th**e **B**eas**t**," the second is "**M**oa**n**a," the third is "**C**a**tch** **M**e **I**f You **C**a**n**," and we'll leave the last one as an exercise to the reader. 😀

To form a puzzle string like this, you simply delete all the letters from the original word or phrase except for consonants, then convert the remaining letters to ALL-CAPS.

Your task is to write a ***recursive*** function

```
string onlyConnectize(string phrase);
```

that takes as input a string, then transforms it into an *Only Connect* puzzle. For example:

- `onlyConnectize("Elena Kagan")` returns `"LNKGN"`,
- `onlyConnectize("Antonin Scalia")` returns `"NTNNSCL"`,
- `onlyConnectize("EE 364A")` returns `""`,
- `onlyConnectize("For sale: baby shoes, never worn.")` returns `"FRSLBBSHSNVRWRN"`,
- `onlyConnectize("Thank you, next (next)")` returns `"THNKNXTNXT"`, and
- `onlyConnectize("Annie Mae, My Sea Anemone Enemy!")` returns `"NNMMSNMNNM"`.

Some notes on this problem:

- Your solution must be recursive. You may not use loops (`while`, `for`, `do`…`while`, or `goto`).
- Make sure that you're always returning a value from your recursive function. It's easy to accidentally forget to do this when you're getting started with recursion.
- You can use `toUpperCase` from the `"strlib.h"` header to convert a single character to upper case. It takes in a `char`, then returns the upper-case version of that letter. If you call `toUpperCase` on a non-letter character like `'$'` or `'*'`, it will return the original character unmodified.
- The `isalpha` function from the `<cctype>` header takes in a character and returns whether it's a letter. There is no library function that checks if a letter is a consonant, though.
- Just to make sure you didn't miss this, we are treating the letter `y` as a vowel.

The starter code that we've provided contains code to test your function on certain inputs. These tests check a few sample strings and are not designed to be comprehensive. In addition to implementing the `onlyConnectize` function, you will need to add in at least one new testing function of your own using the `ADD_TEST` macro. To do so, use this syntax:

```
ADD_TEST("description of the test") {
    /* Put your testing code here. */
}
```

Take a look at the other tests provided to get a sense of how to write a test case. The `EXPECT` macro takes in an expression and evaluates it. If it evaluates to true, great! Nothing happens. Otherwise, it reports that the test failed. You can run the tests by choosing the "Run Tests" button from the demo app.

When you're writing tests, be strategic about the tests you add. What are some tricky cases you might want to confirm work correctly? Are there any edge cases (extremely small cases, highly unusual cases, etc.) that would be worth testing?

Once you have everything working, run our demo program to play around with your code interactively. Then, leave an *Only Connect* puzzle of your own choosing for your section leader! To do so, edit the file comments at the top of the file with the consonant string, along with a hint.

All the code you'll need to write for this part of the assignment should go in the `OnlyConnect.cpp` file.

Problem Three: Playing Fair

Consider the following scenarios:

- Ten people want to play pick-up basketball. They select one person to captain team A and one to captain team B. The two captains then take turns choosing players for their team. One option would be to alternate between captains A and B, but that would give the team that picked first a noticeable advantage over the other. In what order should the captains pick players?
- The World Chess Championship is a multi-game chess match held every two years between the reigning world champion and a challenger. In chess, white has a slight advantage over black, so both players should have an equal number of turns as white and as black. However, if one player gets too many turns as white early on, they might accumulate a score advantage early on that puts pressure on the other player. In what order should the players play as white and black?
- In old-school NFL rules, if a postseason football game went to overtime, one team would get possession of the ball and be given a chance to score. If they scored, they'd instantly win the game. If they didn't, the other team would get possession and a chance to score. If they didn't, the first team would get another try, etc. This gives an advantage to whoever gets possession first. What's a better way to decide which team gets a chance to score to make this as fair as possible?

These scenarios all have a core setup in common. There are two parties (we'll call them A and B) who take turns at an activity that confers an advantage to whoever performs it. The goal is to determine the order in which A and B should perform that activity so as to make it as close to fair as possible.

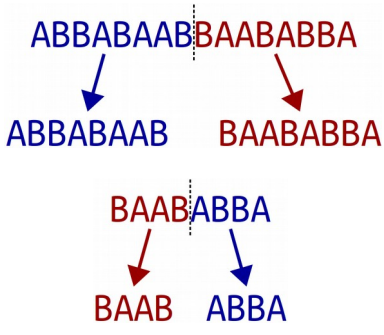
There's a clever recursive technique for addressing this problem that keeps the advantage of going first to a minimum. We're going to consider two kinds of sequences: ***A-sequences*** and ***B-sequences***, which each give a slight advantage to players A and B, respectively.

There are different A-sequences and B-sequences of different lengths. Each sequence is given a number called its ***order***. The higher the order of the sequence, the more games are played. For example, here are the A and B sequences of orders 0, 1, 2, 3, and 4:

	Order 0	*Order 1*	*Order 2*	*Order 3*	*Order 4*
A-sequence:	A	AB	ABBA	ABBABAAB	ABBABAABBAABABBA
B-sequence:	B	BA	BAAB	BAABABBA	BAABABBAABBABAAB

We can interpret these sequences as instructions of who gets to play when. For example, the A-sequence of order 2, ABBA, can be thought of as saying "A plays first, then B, then B again, then A again." There's a slight advantage to A going first, but it's mitigated because B gets two turns in a row. The B-sequence of order three, BAABABBA, means "B takes a turn, then A, then A again, then B, then A, then B, then B again, then A." If you think about what this looks like in practice, it means that B has a little advantage from going first, but the other alternations ensure that A gets to recoup that disadvantage later on.

Right now these sequences might look mysterious, but there's a nice pattern. Take the A-sequence of order 4, ABBABAABBAABABBA, and split it in half down the middle, as shown to the right. That gives back the two sequences ABBABAAB and BAABABBA. If we look in the table shown above, we can see that this first sequence is the A-sequence of order 3, and the second sequence is the B-sequence of order 3. Interesting!



Similarly, look at what happens when you split the B-sequence of order 3, BAABABBA, in half. That gives back BAAB and ABBA. And again, if we look in our table, we see that this first string is the B-sequence of order 2 and the second is the A-sequence of order 2. Nifty!

More generally, the pattern looks like this: if you take an A-sequence of order n (where $n > 0$) and split it in half, you get back an A-sequence of order $n - 1$ followed by a B-sequence of order $n - 1$. Similarly, if you take a B-sequence of order n (with $n > 0$) and split it in half, you get a B-sequence of order $n - 1$ followed by an A-sequence of order $n - 1$. This process stops when you need to form the A-sequence or B-sequence of order 0, which are just A and B, respectively.

Using these observations, implement a pair of functions

```
string aSequenceOfOrder(int n);
string bSequenceOfOrder(int n);
```

that take as input a number $n \geq 0$, then return the A-sequence and B-sequence of order n, respectively. As with the Only Connect problem, these functions must be recursive and must not use loops.

If someone calls either function passing in a negative value of n, you should use the `error()` function to report an error. That function has the signature

```
void error(string message);
```

and, when called, immediately jumps out of the function to say that something terrible has happened.

We've included some testing code you can use to check your solution, but those tests aren't exhaustive. As part of this assignment, add at least one own custom test case, and ideally more. All the code you'll need to write for this part of the assignment should go in `PlayingFair.cpp`.

You might notice that we've used the `EXPECT_ERROR` macro in one of our test cases. This macro evaluates the expression and sees whether it calls the `error()` function to report an error. If so, great! It prevents the error from propagating and makes a note that an error was indeed generated. If not, it causes the test to fail and reports that the expression failed to trigger an error.

The starter code contains a demo of another property of these sequences. Imagine you're in an open field. You then read the characters of an A-sequence from left to right. Whenever you read an A, you take a step forward, place down a marker, then rotate 60°. Every time you read a B, you turn around without moving. Repeating this process gives an intricate and complex result. Once your code is working, run our demo app to see what it is! The slider at the bottom controls the order of the sequence.

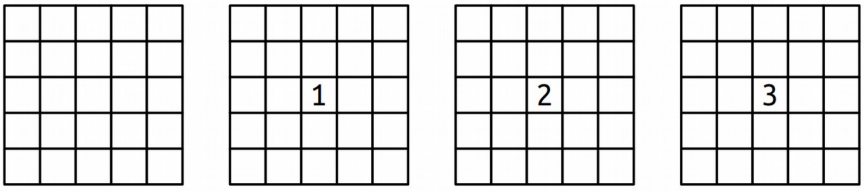
Some notes on this problem:

- As with most recursive functions, you probably shouldn't need to write much total code here. If you've written thirty or more lines of code and are struggling to get things working, you may be missing a slightly easier solution route and might want to check in with your section leader for advice.
- An A-sequence of order n has length 2^n, which means that the lengths of these sequences grow extremely rapidly as a function of n. For reference, an order-30 A-sequence will be over a billion characters long, and an order-300 A-sequence has more characters than there are atoms in the observable universe. Don't worry about generating sequences of those sorts of orders; stick with low-order sequences, say, with $n \leq 20$.
- If your code isn't working, step through it in the debugger! You saw how to step through recursive code in Part One of this assignment and in Assignment Zero.
- Testing is key here. The tests we've provided aren't sufficient to cover all the cases that might come up. You're required to add at least one test case, and really do take that seriously. If you're trying to test something recursive, what sorts of cases might be good to check?

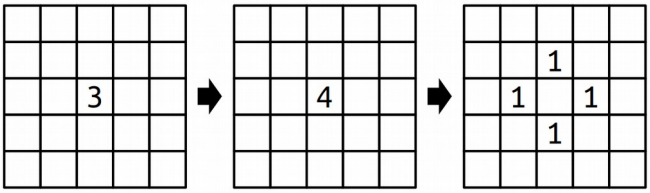
Problem Four: Sandpiles

One of the most beautiful ideas in computer science is that *simple rules give rise to complex behavior*. You can have straightforward principles that govern how a particular system operates, and yet discover that repeated applications of those rules gives intricate and surprising patterns. The best way to appreciate this is to see it for yourself.

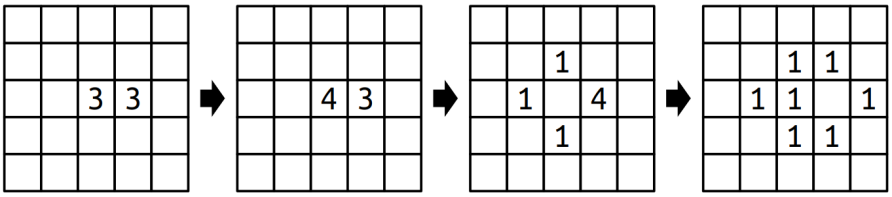
This problem concerns a mathematical model of a sandpile called the ***Bak-Tang-Wiesenfeld model*** after the three physicists who first devised it. They modeled a pile of sand as follows. We begin with a two-dimensional grid subdivided into units called *cells*. Each cell is initially empty. We then pick a cell and start dropping grains of sand into it. Each cell is allowed to hold between 0 and 3 grains of sand. So, for example, if we had a 5 × 5 grid and started dropping sand into the center square, the first few steps would look like this:



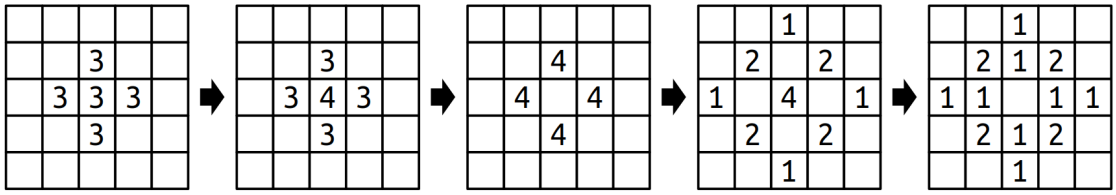
Now, what happens when we drop a fourth grain of sand in the center? As we mentioned earlier, each cell can hold between 0 and 3 grains of sand, so we can't have four grains of sand there. In this case, the cell ***topples***, sending one grain of sand to each of its four neighbors in the cardinal directions (up, down, left, and right), ending up empty. This is shown here:



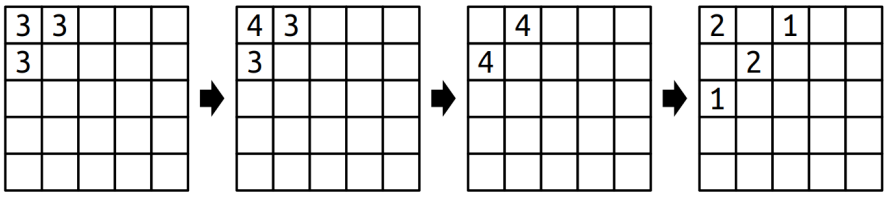
That might be the end of the story, or it might just be the beginning. Imagine, for example, that there are two adjacent cells, each of which contains three grains of sand. What happens when a grain of sand is dropped into one of those full cells? That initial cell topples, sending one grain of sand in each direction. One of those grains lands inside another cell holding three grains, boosting it up to four grains, causing *that* cell to topple. This is illustrated below:



A single grain of sand can cause a large cascade. For example, here's what happens if five full cells in a plus shape are disturbed by one sand grain:



There's one additional case to consider, and that's what happens at the border of the world. We can think of the cells in our grid as the surface of a table. If sand were to fall outside of the the boundaries of the world, it just falls off the table and is never seen again. For example, here's what happens if a cell in the corner were to topple:



If you start with an empty world and repeatedly drop grains of sand into the center square, the resulting sand patterns are *extremely* surprising. But that's something that you'll need to see for yourself. 😀

Before moving on, we recommend that you take a few minutes to answer the following question: what will the world to the right look like if we drop a single grain of sand onto the center square? The chain reaction here is quite interesting – work this out with a pencil and paper!

3	3	2
2	3	3
	2	3

Your job is to implement a ***recursive*** function

```
void dropSandOn(Grid<int>& world, int row, int col);
```

that takes as input a row and a column index, along with a `Grid<int>` representing the number of grains of sand in each cell (more on this later on), then adds a single grain of sand to the specified location. As a reminder, here's what happens when you drop sand on a cell:

- If the cell ends up with three or fewer grains of sand in it, you're done.
- Otherwise, empty that cell, then drop one grain of sand on each of the four neighbors.

You might have noticed that the first argument is a `Grid<int>`, a type we haven't encountered before. This is a type representing a fixed-size, two-dimensional grid of integers, like the grids that we showed on the previous pages. You can access and manipulate the individual cells using statements like these:

```
world[row][col] += 137;
if (world[row][col] == 0) { … }
```

Row and column indices start at zero, and row 0, column 0 is in the upper-left corner of the screen. The rows increase from top to bottom and columns increase from left to right.

You may be given coordinates in `dropSandOn` that are outside the boundaries of the grid, either because whoever is giving you the coordinates just wants to drop sand on the floor or because something toppled off the edge. If that happens, `dropSandOn` should have no effect and should leave the grid unmodified. You can test whether a coordinate is in bound by using the grid's `inBounds()` member function:

```
if (world.inBounds(row, col)) { … }
```

Make sure not to read or write to a grid in an out-of-bounds location. Doing so will cause the program to crash with an error message.

Some notes on this problem:

- As before, your solution should be purely recursive and should not involve any loops.
- We advise against writing any helper functions here. It's actually easiest to write this as a single function that drops a grain of sand and handles toppling rather than splitting those bits apart.
- When a cell topples, make sure to empty that cell out before you start dropping grains of sand on other squares. Otherwise, weird things can happen when sand falls back into that cell.
- It actually doesn't matter what order you visit the neighbors of a toppling cell when that cell topples. The result is always the same. (This isn't supposed to be obvious, by the way; it's something that requires a creative mathematical proof.)

As before, you are required to also add in some of your own test cases. Take this seriously! It can be hard to tell whether you've gotten your solution working purely by looking at the output because the system evolves in such amazing and complex ways. Writing your own targeted, focused tests is a great way to make sure that what you *think* is going to happen matches what *actually* happens.

All the code you need to write for this part of the assignment goes in the `Sandpiles.cpp` file. Once you've gotten things working, take some time to play around with the visualizer. Isn't that beautiful?

Problem Five: The Flesch-Kincaid Grade-Level Test

Many word processing programs (such as Microsoft Word) employ heuristics to estimate how sophisticated a piece of text is. This makes it possible for authors to evaluate whether their works are too complex for their target audience. It also makes it possible for high school English teachers to save lots of time grading essays. 😄

One automated test for estimating the complexity of a text is the ***Flesch-Kincaid grade level test***. The resulting number gives an estimate of the grade level necessary to understand the text. For example, something with grade level 5.15 could be read by a typical fifth-grader, while something with grade level 15.1 would be appropriate for a typical college junior.

This Flesch-Kincaid test test makes no attempt to actually understand the *meaning* of the text. Instead, it focuses purely on the complexity of the sentences and words used within that text. Specifically, Flesch-Kincaid counts up the total number of words and sentences within the text, along with the total number of syllables within each of those words, then computes an approximate grade level for the text according to the following formula:

$$Grade = C_0 + C_1\left(\frac{num\ words}{num\ sentences}\right) + C_2\left(\frac{num\ syllables}{num\ words}\right)$$

Here, the constants C_0, C_1, and C_2 are chosen as follows:

$$C_0 = -15.59 \qquad C_1 = 0.39 \qquad C_2 = 11.8$$

(I honestly have no idea where these values came from, but they're the standard values used whenever this test is performed!)

In this last part of the assignment, you'll create a program that lets you compute Flesch-Kincaid grade levels of pieces of text. Specifically, your job is to write a function

```
DocumentInfo statisticsFor(istream& source);
```

that takes as input a stream containing the contents of a file (described later), then returns a `DocumentInfo` object (described later) containing the number of sentences, words, and syllables contained in that file. Our provided starter code will call your function and use it to compute the Flesch-Kincaid score for the file. The rest of this section talks about what, specifically, you'll need to do.

Before going over the code specifics, we should start off by noting that

☞ ***you don't need to use recursion for this part of the assignment,*** ☜

since we expect that the previous three parts of this problem will have gotten you quite used to thinking recursively.

With that said, let's talk about what this assignment is all about. First, what exactly is this `DocumentInfo` thing you're returning? If you look in the starter code (look at `Headers/src/FleschKincaid.h`), you'll see that it's defined like this:

```
struct DocumentInfo {
    int numSentences;
    int numWords;
    int numSyllables;
};
```

This is a *structure*, a type representing a bunch of different objects all packaged together as one. Here, this structure type groups together three `int`s named `numSentences`, `numWords`, and `numSyllables`. The name `DocumentInfo` refers to a type, just like `int` or `string`. You can create variables of type `DocumentInfo` just as you can variables of any other type. For example, you could declare a variable of type `DocumentInfo` named `info` like this:

```
DocumentInfo info;
```

One you have a variable of type `DocumentInfo`, you can access the constituent elements of the `struct` by using the dot operator. For example, you can write code like this:

```
DocumentInfo info;
info.numSentences = 137;
info.numWords++;
cout << info.numSyllables << endl;
```

As a note, just as regular `int` variables in C++ hold garbage values if you don't specify otherwise, the `int` variables inside of a `DocumentInfo` will be set to garbage if you don't initialize them. Therefore, you might want to initialize your variables like this:

```
DocumentInfo info = { 0, 0, 0 }; // Everything is zeroed out!
```

With that introduction to `struct`s done, let's talk about what exactly you need to get this function to do.

Step One: Tokenize the Input File

The `statisticsFor` function takes in an `istream&`, a reference to a stream. Streams are C++'s way of getting data from an external source (the keyboard, a file, the network, etc.), and in this case lets you read the contents of a file. You'll need to read in the original text and break it apart into a bunch of individual words and punctuation symbols so that you can count up how many words and sentences there are. For this purpose, we provide you a `TokenScanner` class that lets you read a file one "piece" at a time, where a "piece" is either a punctuation symbol or word. To start off this assignment, see if you can write a program that will open a file and read it one piece at a time using the `TokenScanner`.

We did not cover how to use the `TokenScanner` class or the `istream` type in lecture. However, we've provided documentation on the `TokenScanner` class on the course website (http://cs106b.stanford.edu) via the link "Stanford C++ Library Docs." Learning to read documentation is an important skill as a programmer, since you'll often find yourself working with new libraries! Read over the docs for the `TokenScanner` type. What header file is it declared in? Take a look at the sample code at the top of the documentation see how to use `TokenScanner` to read all the tokens out of a file. Find a way to get the `TokenScanner` to

- read directly from an `istream`, which dramatically simplifies the logic, and
- skip over whitespace tokens, so you don't have to handle them later on.

To test whether you've set things up properly, put together an implementation of `statisticsFor` that simply prints out all the tokens that come out of the `TokenScanner` without doing anything else. (You'll need to return something, so just create a dummy `DocumentInfo` and return it without filling in any of the fields.) Try running that code on our provided test cases. One of the test cases will run your code on this input:

```
Mr. Ford's first day as President was August 9, 1974.
```

If you're properly skipping over whitespace, you should get back the following sequence of tokens:

⟨`Mr`⟩ ⟨`.`⟩ ⟨`Ford`⟩ ⟨`'`⟩ ⟨`s`⟩ ⟨`first`⟩ ⟨`day`⟩ ⟨`as`⟩ ⟨`President`⟩ ⟨`was`⟩ ⟨`August`⟩ ⟨`9`⟩ ⟨`,`⟩ ⟨`1974`⟩ ⟨`.`⟩

We've surrounded each token in angle braces for the purposes of this handout so that it's easier for you to see what they are; you won't see them in the output of your program.

As you can see, the `TokenScanner` will sometimes split words apart in the wrong place. The most notable example of this is that `TokenScanner` doesn't consider single quotes to be parts of words, which is why `Ford's` was split apart into three tokens. For the purposes of this assignment, don't worry about this; after all, the Flesch-Kincaid score is already an approximation. That said, if you were trying to implement Flesch-Kincaid inside of a major commercial product like, say, Microsoft Word or Google Docs, you'd probably want to handle things more intelligently than this. 😄

Step Two: Estimate Words and Sentences

Update your program so that it estimates the number of words and sentences in the file. To determine what counts as a word, treat any token that starts with a letter as a word, so `apple` and `ICICLE` would both be considered words. As an approximation of the number of sentences, count the number of punctuation symbols that typically appear at the ends of sentences (periods, exclamation points, and question marks). This isn't entirely accurate, but it's good enough for our purposes.

As an edge case, if a file doesn't have any words or sentences, pretend it contains a single word or a single sentence. This prevents a division-by-zero error when evaluating Flesch-Kincaid formula.

For example, we'd estimate that the quote about Gerald Ford from above would have nine words. (This is an overestimate; the tokenization splits `Ford's` apart into two tokens beginning with a letter.) Additionally, we'd estimate there were two sentences. This overestimate occurs because we don't differentiate between the period after `Mr` and the period at the end of the sentence.

Before moving onward, take some time to test that your code is working as intended. Run our provided tests, many of which will check to make sure that you're counting words and sentences correctly. It's okay if you fail some test cases that are designed to check for syllable counts – you haven't written that code yet – but you should be passing all the others. Then add some tests of your own; we haven't covered all possible cases with our test suite.

Step Three: Estimate Syllables in Words

Your last task is to count the number of syllables in all word tokens. To approximate the number of syllables in a word, count up the number of vowels in the word (including `y`), *except* for

- vowels that have vowels directly before them, and
- the letter `e`, if it appears by itself at the end of a word.

For example, the word `program` would be counted as having two syllables; the word `peach` would have one syllable; the word `deduce` would have two syllables, since the final `e` does not count as a syllable; the word `oboe` would have two syllables (although it ends in an `e`, that `e` is preceded by another vowel); the word `why` would have one syllable, since `y` counts as a vowel; and the word `enqueue` would have two syllables. Notice that under this definition, the word `me` would have zero syllables in it, since the final `e` by itself doesn't contribute to the total. To address this, ***you should always report that any given word has at least one syllable***, even if this heuristic would normally report otherwise.

This approximation of syllable counts isn't always accurate. In fact, it incorrectly says that there are just two syllables in the word `syllable`. However, for our purposes, this is totally fine.

The syllable-counting logic is probably the trickiest code to write in this part of the assignment, so before you declare victory and move on, we strongly recommend writing test cases to stress-test it. Add in some test cases to the starter files that just test the function you wrote to count syllables (you did decompose that out into its own function, right?). A test that assesses just one part of a larger program is called a ***unit test***, and unit testing is standard throughout the software industry.

The capitalization of a word should have no effect on how many syllables it contains. The strings `Unite`, `UNITE`, `unite`, `unItE`, and `UnitE` all should report two syllables. And don't forget that `y` counts as a vowel: the word `unity` should have three syllables in it.

Once your code is passing all your unit tests, complete the implementation of `statisticsFor` by integrating in syllable counts. Test your final implementation thoroughly, adding some of your own tests in if need be.

Step Four: Play With Your Creation!

And that's it – you're done! All that's left to do is play around with your function and see what you find. Try running your function on some of the files we've provided. Do any of the results surprise you? Try finding a piece of text from one of your favorite books, songs, or articles. How do those scores compare to our sample files? If you find anything interesting, let us know about it in your submission!

Some general notes and advice on this part of the assignment:

- Chapter 4 of the textbook goes into detail about stream classes and provides examples about how to read from a file. However, for this part of the assignment, the `TokenScanner` class is powerful enough to do all the file reading for you. Read the docs and see if you can find a way to have `TokenScanner` read directly from the stream. If you do so correctly, you should never need to do any explicit reading from the file; `TokenScanner` will do it for you.
- The `TokenScanner` type is a "one-pass" scanner. Once you've read all the tokens out of a file, you can't "go back" and reread them again. If you create multiple `TokenScanner`s to read the same file, you still only get one pass over the file. This means that you can't, for example, make two passes over the file, once counting sentences and once counting words and syllables.
- If a file has no sentences, you should report that it has one sentence. If a file has no words, you should report that it has one word. However, if a file has no syllables (because there are no words), you should report that it has no syllables.
- If you read off the end of a string or before the beginning of a string, you trigger what C++ calls *undefined behavior*. Your program might crash and pull up a stack trace, or it might silently fail and return garbage data. Good testing should be able to nip this in the bud.
- You'll need to be able to determine whether a letter is a vowel in this part of the assignment, and that sounds a lot like something you'll have previously done in the Only Connect problem. This is an opportunity for you to reuse code! Based on how C++ splits apart code between header files and source files, what files might you want to edit in order to let you reuse your older vowel-counting code in this part of the assignment?

All the code you need to write for this part of the assignment should go in the `FleschKincaid.cpp` file.

(Optional) Problem Six: Extensions!

You are welcome to add extensions to your programs beyond what's required for the assignment, and if you do, we're happy to give extra credit! If you do, please submit two sets of source files – a set of originals meeting the specifications we've set, plus a set of modified files with whatever changes you'd like. Here are some ideas to help get you started:

- ***Only Connect:*** The puzzles given in the show *Only Connect* are made even more difficult because the show's creators don't just delete non-consonants; they also insert spaces into the result that may or may not align with the original phrase. For example, the string "O say, can you see?" might be rendered as `"SC NS"`, giving the illusion that it's only two words rather than the five from the original. Write a function that inserts spaces in random places into your resulting string.

  Another option: find a data set online containing a bunch of related phrases (for example, a list of movies, or historical figures, or geographic locations) and try building this into a legit trivia game. We're curious to see what you choose to do here!

- ***Playing Fair:*** Once you have an A-sequence or B-sequence, you could ask exactly how fair it is under various assumptions. For example, in the NFL overtime example, what would happen if you knew that one team had a 15% chance of scoring on each possession and the other had a 13% chance? What's the likelihood of each team winning as you take longer and longer sequences? Alternatively, suppose you have *three* people, A, B, and C, who need to choose players for three teams. Could you find a way to generalize these sequences to find fair systems for those scenarios? Or what about sequences whose lengths aren't powers of two?

  These sequences are related to a mathematical object called the *Thue-Morse sequence*, which has a ton of other properties. Consider reading up on the Thue-Morse sequence and seeing whether there are any other interesting things you can do with it.

  The two interrelated recursive functions you wrote in this section are an example of *mutual recursion*, which has applications in linguistics and game theory. Consider implementing another set of mutually recursive functions. For some fun leads about places to look for this, search for *probabilistic context-free grammar*, which can be used to synthesize text, and *minimax*, an algorithm that can be used to play certain games perfectly.

- ***Sandpiles:*** There are many ways you could change up the behavior of this system. What happens if you decide to shift sand grains around when a cell topples not equally to each neighbor, but instead putting one sand grain into two randomly-chosen neighbors and two sand grains into a third? Or what happens if you allow cells to hold four grains of sand stably, then topple only at five? There are many variations you could explore here – try them out and see what you find!

  The sandpile model you implemented is sometimes called the *Abelian sandpile* and is an example of a *chip-firing game*. Consider reading more about these systems and their properties. What cool things can you do with them that we didn't think of?

- ***Flesch-Kincaid Readability:*** Can you make the program better at tokenizing input files? Can you make the program better at counting syllables? Could you be more intelligent about how sentences are handled? Could you try measuring some other property of a piece of text in order to determine its complexity?

  There are other readability indices like the *Dale-Chall readability score* that compute difficulty in different ways. Try implementing one of those and comparing the results. Which one do you think works better, and why?

  Alternatively, could you use your program to reveal something about the human condition? In the past, we've had students use this program to investigate the disparity between the reading level of US patents and the average education level of a jury after *voir dire*, to look at the evolution of the English language by comparing older texts (Shakespeare, the US Constitution) to newer ones (Mark Zuckerberg's Facebook posts), to observe changes in State of the Union addresses over the years, and even to comment on the complexity of lyrics in rap music over time. If you find anything interesting, we'd love to see what you come up with!

Submission Instructions

To recap, here's what you're expected to submit:

- ***Stack Overflows:*** The sequence of looping values that repeat in the stack overflow.
- ***Only Connect:*** Your implementation of `onlyConnectize`, along with at least one custom test case and a puzzle for your section leader to think through.
- ***Playing Fair:*** Your implementations of the `aSequenceOfOrder` and `bSequenceOfOrder` functions and at least one custom test case.
- ***Sandpiles:*** Your implementation of `dropSandOn`, along with at least one custom test case.
- ***Flesch-Kincaid:*** Your implementation of `statisticsFor` and at least one custom test case.

Before submitting your assignment, make sure that you've read through our Assignment Submission Checklist to make sure your code obeys our style conventions. A few specific things to watch for:

- Make sure that you've commented your code. All functions should have a comment describing what they do.
- Make sure you've properly indented your code.
- Make sure you aren't using the `goto` keyword or global variables.

To submit this assignment, upload the following files to paperless.stanford.edu:

- `StackOverflow.cpp`
- `OnlyConnect.h` and `OnlyConnect.cpp`.
- `PlayingFair.h` and `PlayingFair.cpp`.
- `Sandpiles.h` and `Sandpiles.cpp`.
- `FleschKincaid.h` and `FleschKincaid.cpp`.

If you modified any other of the starter files, be sure to include those as well.

And that's it! You're done!

Good luck, and have fun!