PAiC++ March 28, 2023

**Assignment 1.2**

# Problem 1 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**ea**st**,” 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 you.

To form a puzzle string like this, you simply delete all characters 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("I'm the bad guy. (Duh!)") returns "MTHBDGDH", and
* onlyConnectize("Annie Mae, My Sea Anemone Enemy!") returns "NNMMSNMNNM".

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 test 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 two expressions. If those expressions are equal, great! Nothing happens. Otherwise, EXPECT reports an error. 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, if you like, leave an *Only Connect puzzle* of your own choosing! To do so, edit the file comments at the top of the file with the consonant string, along with a hint.

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

* Implement the onlyConnectize function in OnlyConnect.cpp. This function must be implemented recursively. It takes as input a string. The output should be that same string, in upper case, with all characters except consonants deleted. Feel free to write helper functions if you’d like.
* Add at least one test case using the **ADD\_TEST** macro. Your test should go in the file OnlyConnect.cpp, preferably with all the other test cases. For full credit, your test case should check some style of input that wasn’t previously addressed by the provided tests.

As you’re writing up your solution to this problem, remember that coding style is important. We have a style guide available on the BB website. Take a few minutes to read over it, then review your code to make sure it conforms to those style expectations.

Some notes on this problem:

* All C++ programs begin inside the main() function, and we’ve already written this function for you in one of the starter files. You just need to implement onlyConnectize and are not responsible for writing main(). (In fact, if you did try to write your own main() function here, you’d get an error because there would be two different versions of main() and C++ wouldn’t know which one to pick!)
* 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 or vowel, though.
* If you’re coming from Python, note that C++ doesn’t have an in operator like the one in Python and that the ***and*** and ***or*** operators are denoted by ***&&*** and **||**, respectively. Check out the online “Python-to-C++ Guide” on the course website for more information.
* Remember that C++ treats individual characters differently than strings. Individual characters have type char. To talk about a specific single character, you must use single quotes (e.g. 'a' rather than "a"). Strings have type string. To talk about a specific string, you must use double- quotes (e.g. "hello" rather than 'hello').
* You can convert a ***char*** to a ***string*** by using the ***charToString*** function from "***strlib.h***".
* You are welcome to add your own helper functions when solving this problem. Those functions must obey the same rules of this problem as the main function (e.g. no loops) .
* Just to make sure you didn’t miss this, we are treating the letter y as a vowel.
* You shouldn’t need to edit ***OnlyConnect.h*** in the course of solving this problem.

# Problem 2 Playing Fair (Optional)

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 games as white and as black. However, if one player gets too many games 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 pos- session 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?

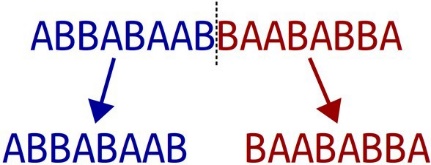
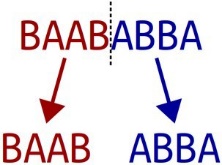
These scenarios all have a core setup in common. There are two parties (we’ll call them A and B) who take turns at an activity that confers an advantage to whoever performs it. The goal is to determine the order in which A and B should perform that activity so as to make it as close to fair as possible.

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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *A-sequence:* | *Order 0*  **A** | *Order 1*  **AB** | *Order 2*  **ABBA** | *Order 3*  **ABBABAAB** | *Order 4*  **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 some of 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 se- quences **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 fol- lowed 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 ≥ 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 from the header "***error.h***" to report an error. You call this function by writing

**error("a string containing your error message");**

The error function immediately jumps out of the function to say that something terrible has happened.

We’ve included some tests you can use to check your solution, but they aren’t exhaustive. You’ll need to add at least one custom test case, and ideally more. 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 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, 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.

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

* Implement the **aSequenceOfOrder** and **bSequenceOfOrder** functions in the file **Playing**- **Fair.cpp**. These functions should return an A-sequence and B-sequence of order n, respec- tively. These functions must be implemented recursively. If either function receives a nega- tive number as an input, they should use the **error()** function to report an error. As you go, test your code by using the “Run Tests” button in the provided program.
* Add at least one test case using the **STUDENT\_TEST** macro. Your test should go in the file **PlayingFair.cpp**, preferably with all the other test cases. For full credit, your test case should try to check some case that wasn’t previously addressed by the bundled tests.
* Once everything works, click the “Playing Fair” button in the demo program to see a cool visualization of the sequences you’ve generated.

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 an easier solution route and might want to check in with your TA for advice.
* An A-sequence of order n has length 2n, 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 ≤ 20.
* If your code isn’t working, step through it in the debugger! You saw how to step through code in Assignment 0.
* 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?
* You shouldn’t need to edit **PlayingFair.h** in the course of solving this problem.