Lab 6: Recursion [noun]: a procedure that uses recursion.
CSCI 2101 – Fall 2021

Due (both sections): Wednesday, November 10, 11:59 pm
Collaboration Policy: Level 1 (review full policy for details)
Group Policy: Individual

This lab will give you more experience writing recursive algorithms – in particular, your task is simply to complete a set of recursive methods. We will start with some smaller problems that you would likely be able to easily solve using iteration, but solving them recursively will get you into the right mindset. We will then move onto a more interesting and complex problem that is quite well suited to a recursive solution. Good recursive solutions tend to be short, compact, and elegant – but they can be tricky to come up with! As you work through these methods, remember the essential rules of all recursive algorithms:

- You must have (at least one) base case, which does *not* use recursion.
- You must have (at least one) recursive case, which *does* use recursion.
- Each recursive call must move you *towards* a base case.
- You should not write any loops (if you do, you're probably writing an iterative solution rather than a recursive solution).

You will be provided with some starter code for this lab, which can be downloaded from Blackboard. The `Recursion` and `TestRecursion` classes will be used for Part 1 of the lab, while the `BoggleBoard` and `BoggleDictionary` classes will be used for Part 2 of the lab. More details are provided for each part below.

1 Basic Recursion

In this section, you will write four self-contained recursive methods within the provided `Recursion` class. Each of these methods will be a `static` method that is called directly from the `TestRecursion` class. The method declarations are given to you in the starter code and must not be changed. Additionally, **you may not use any of the following while implementing these methods:**

- Loops of any kind (`for`, `while`, etc.)
- Instance or static variables
- Any built-in methods except those specifically indicated

The provided `TestRecursion` class provides a `main` method that will run a test battery on all four recursive functions in the `Recursion` class. Executing the `main` method will run the tests and highlight any tests on which your functions are failing. You do not need to modify the `TestRecursion` class in any way.

The four methods in the `Recursion` class that you will implement are described below.

1.1 Cannonballs

Imagine stacking cannonballs in a pyramid – e.g., one cannonball at the top, sitting on top of a level of four cannonballs, sitting on top of a level of nine cannonballs, and so forth. Write a recursive method `countCannonballs` that is given the number of levels of the pyramid (≥ 0) and returns the number of cannonballs in the complete pyramid. The method definition is as follows:

```
public static int countCannonballs(int height)
```

You may find it helpful to observe that each level of the pyramid consists of a "square" of cannonballs. You should not need to call any methods except for your recursive calls.

1.2 Digit Sum

Write a recursive method `digitSum` that takes a non-negative integer and returns the sum of its digits. For example, `digitSum(1234)` returns $1 + 2 + 3 + 4 = 10$. You are *not* allowed to convert the number to a `String`, and instead must only work with the value numerically. The method definition is as follows:

```
public static int digitSum(int n)
```

You may find the mod (%) and division (/) operators useful here. You should not need to call any methods except for your recursive calls.

1.3 Binary Numbers (redux)

Recall from the Two Towers lab that a binary number (base-2) is a number made up of bits, each of which is 0 or 1 only, in which each bit represents a power of 2. A binary number like 10011 is converted to a decimal number (base-10) by multiplying the bits by the powers of 2 and adding them. In the case of 10011, reading from left to right, $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 19$.

Suppose we want to convert a series of bits to its decimal value. You could write an iterative algorithm that does this by looping over the bit values and powers of 2 and adding them together. You can also approach this problem recursively. Consider the effect of adding an extra bit to the right side of an existing binary number. For example, consider the binary number 11 (decimal value 3). Adding an extra zero bit simply doubles the value (binary 110 = decimal 6). Adding an extra one bit also doubles the value, but then adds one to the end result (binary 111, decimal 7). Try adding another bit or two to convince yourself that this property always holds.

Using this property, write a recursive method `bitsToDecimal` that is given a series of bits as a `String` and returns the decimal value of the binary number specified. For example, calling `bitsToDecimal("101")` should return 5. The method definition is as follows:

```
public static int bitsToDecimal(String bits)
```

To ensure that you write a recursive solution and do not rely on iterative `String` methods, **you may call the `String` methods `equals`, `length`, `substring`, and `charAt`, but no others.**

One trap to watch out for is how you check whether a specific `char` is a 1 bit or a 0 bit. In particular, Java will let you compare a `char` with an `int`, but the `char` of a particular number is *not* equal to its integer value. This property is illustrated below:

```
char bit = '1'; // a char: it has a numeric value, but its value is NOT 1
boolean test1 = (bit == '1'); // true, comparing two chars
boolean test2 = (bit == 1); // FALSE, the int 1 is not equal to the char '1'
```

It's easy to write the latter when you really mean to write the former (and the compiler won't warn you if you write the latter, since comparing a `char` to an `int` is permitted).

1.4 Palindromes

A *palindrome* is a word that is the same spelled forwards or backwards. For example, the word "racecar" is a palindrome, but the word "car" is not. Palindromes can also be multiple words, e.g., "rise to vote sir" (note that in multi-word palindromes, we normally ignore the spaces).

Write a recursive method `isPalindrome` that accepts a `String` and returns whether that string represents a palindrome. You may assume that the string consists only of letters and spaces. Your method may be case sensitive (so "racecar" would be considered a palindrome, but "Racecar" would not). However, you should *ignore* spaces; so for example, the string "rise to vote sir" would be considered a palindrome. The method definition is as follows:

```
public static boolean isPalindrome(String str)
```

You may call the `String` methods `equals`, `length`, `substring`, and `charAt`, but not any others. **In particular, you cannot use `replace` to remove spaces in the string.**

2 Boggle Words

In this part, you will complete a program that will be able to perfectly play the game Boggle. A Boggle board is a 4x4 grid of 16 randomly-rolled dice with letters on their faces. The objective of the game is to find as many words as possible in the board by tracing paths through neighboring dice faces. Paths must contain at least 3 letters and can move horizontally, vertically, or diagonally in any direction, but cannot re-use any dice within a single word. Below is a sample Boggle board:

E	B	I	S
L	D	R	E
K	A	Qu	T
C	W	H	I

Figure 1: A sample Boggle board.

Among many others, some of the words that are present in this board include "set", "quit", "warble", "rise", and "whack". In fact, counting words included in the Official Scrabble Player's Dictionary (aka OSPD), this board contains a total of 203 different valid words!

Your task is to complete a program that will be able to locate every possible word within any Boggle board. In particular, you will write a recursive method that provides the core logic for determining all words that are present in the board.

2.1 Program Structure

The lab starter code contains two classes for this task: `BoggleDictionary` and `BoggleBoard`. The `BoggleDictionary` class represents a list of valid Boggle words and is complete – do not modify the `BoggleDictionary` class in any way. The `BoggleBoard` class represents an actual Boggle board, and is complete except for the `findWords` method, which you will implement. You should not change any part of the `BoggleBoard` class other than implementing the `findWords` method. There is a `main` method already implemented in `BoggleBoard` that prompts you for a filename representing a Boggle method and then (once the program is finished) will find and print out all words contained within the board.

You are also given a set of data files along with the starter code: a word list file ("ospd2.txt") which is automatically used by `BoggleBoard`, and five example Boggle board files to use for testing ("b1.txt" through "b5.txt").

2.2 Finding Words

Your task is to implement the recursive `findWords` method within `BoggleBoard`, whose declaration is shown below and should not be changed:

```
private void findWords(Set<String> words, String prefix, int r, int c)
```

You will need to use the methods of the `BoggleDictionary` class and the instance variables of the `BoggleBoard` class in your method, so you should start by looking over the existing code (and documentation) to make sure you understand the structure of the existing code. In particular, make sure you understand the `isPrefix` method of `BoggleDictionary`, the instance variables of `BoggleBoard`, and how the `findWords` method is called from `findAllWords` in `BoggleBoard`.

Once you are comfortable with the existing code, you can turn to completing the `findWords` method. The basic process of looking for words consists of gradually tracing paths and building up word prefixes as you go (since each path leading towards a complete word is a prefix for that word). Once you find an actual word, it should be added to the set of all words.

One special case that must be handled is with respect to the letters "qu". Since the letter "q" on a die is mostly useless without a following "u", the two letters "qu" appear on a single die face instead of just "q". However, the board representation in `BoggleBoard` only stores `chars` – thus, when you encounter a "q", you will need to manually treat it as "qu" instead. The first board ("b1.txt") contains the word "quid", which should allow you to check that this case is being handled correctly.

Remember that you can change direction when tracing the path of a word, so at any given point in assembling a character sequence, there are eight directions you can go in to continue assembling the sequence. The only restriction is that a letter on the board cannot be used more

than once in the same word. The `BlobWorld` example from class should be useful in thinking about how to approach this problem. Note, however, that unlike in the blobs problem, you must put the character back after you've made your recursive calls, because that character might be part of a word in a different path.

It is okay to use loops in this method for the purpose of making multiple recursive subcalls, but otherwise the method should be recursive as usual.

To help you in determining whether your program is working correctly, the first board sample ("b1.txt") contains 21 words, and the last board sample ("b5.txt", which is the same board pictured in Figure 1) contains 203 words. Do your initial testing on "b1.txt", since it is the simplest board of the five examples. The full list of words for "b1.txt" is given in "b1-solution.txt".

2.3 Sets

The `findAllWords` method is defined to return a `Set` object, and one of the parameters of `findWords` is also a `Set`. A set is an abstract data type (and also a Java interface, like `List` or `Map`) that represents an unordered collection of elements in which duplicates are not allowed. The fundamental operations of a set are adding/removing an element and checking whether some element exists within the set. When working with a set, we often just want to iterate over its elements (which might be returned in some arbitrary order).

Although you should not create any new `Set` objects in your own code, the `findAllWords` method creates a `HashSet` object (which implements `Set`) to store the words being collected. It is not a coincidence that the standard `Map` implementation (`HashMap`) is named so similarly to the standard `Set` implementation (`HashSet`) – they use the same underlying technique (hashing), which we will learn about towards the end of the semester. For now, the only `Set` method that you need to use in `findWords` is the `add` method, which adds an element to the set if it doesn't already exist in the set (and if it does already exist, adding it again does nothing).

Lastly, note that one place we have seen `Sets` before (even if we didn't realize it at the time) is when we were using `Maps` – the collection of all keys for a given map (e.g., as returned by the `keySet` method) is a `Set`. This makes sense when we note that a map cannot have duplicate keys, and the keys exist in no particular order.

3 Evaluation

As usual, your program will be evaluated for correctness, design, and style. Make sure that your program is documented appropriately and is in compliance with the coding and style guide. Lastly, make sure that your name is included in all of your Java files.

4 Submitting Your Program

Submit your program on Blackboard in the usual way. Remember to create a zip file named with your username and lab number, e.g., `sbowdoin-lab6.zip`, and upload that file.