Homework 6: Boggle

Introduction & Setup

In this homework, you will be an application of Tries.

Pull the homework skeleton from Github.

```
git pull skeleton master
```

You will be working with command-line arguments, which you've seen already. You will also be using standard in and standard out. These are **unix streams**, and correspond to `System.in` and `System.out` respectively at runtime. A few pieces of recommended reading, if you're not familiar with them already: 1, 2, 3.

Boggle

Description

The game of Boggle involves finding valid words on a 4x4 board of letters.

A brief description of the rules: Each player searches for words that can be constructed from the letters of sequentially adjacent cubes, where "adjacent" cubes are those horizontally, vertically, and diagonally neighboring. Words must be at least three letters long, may include singular and plural (or other derived forms) separately, but may not use the same letter cube more than once per word.

In homework, you will implement a generalized Boggle solver with a few modifications:

- You must support rectangular boards of arbitrary dimensions

The design choice of data structures and algorithms is up to you. However, we will impose a runtime requirements, which are discussed in a following section.

Boggle reads an `N x M` newline separated letter grid from `stdin`, where the dimensions of the board (N and M) are given by the size of the input, unless the `-r` option is provided, in which case you should generate a NxM random board, where the characters are selected uniformly at random from the lowercase English alphabet and N and M can be specified by command line options. The `-n` and `-m` arguments specify the size of the board if randomly generating one. The default values, if not given, are N=4 and M=4. The `-r`, `-n`, and `-m` flags are for your fun and convenience and are not tested.

The `-d` option takes a file path to a newline separated dictionary. Otherwise, use the default dictionary, the file `words` in the current directory. This is provided in the skeleton. This file can also be found, on most Unix machines, in `/usr/share/dict/words`.

Prints the `K` longest unique words, where `K=1` by default, and can be set with command line flag `-k K`, sorted in descending order of length. If multiple words have the same length, print them in ascending alphabetical order.

An input command to boggle should look like:

```
$ java Boggle (-k [number of words])
              (-n [width])
              (-m [height])
              (-d [path to dictionary])
              (-r) < [input board file]
```

Here we use the input redirection symbol `<` to represent opening the input board file as standard in. This is not a

type the board manually.

Example

For input file `test`:

```
ness
tack
bmuh
esft
```

we expect:

```
$ java Boggle -k 7 < test
thumbtacks
thumbtack
setbacks
setback
ascent
humane
smacks
```

For input files `test` and `testDict`:

`test`:

```
baa
aba
aab
baa
```

`dict`:

```
aaaa
aaaaa
```

Output:

```
$ java Boggle -d testDict -k 20 < test
aaaaa
aaaa
```

Warning: Do not use the StdIn class from the Princeton Standard Library, or keep a static `Scanner` or anything else static and uncleared in your class. Doing so will fail the autograder without much meaningful explanation. We will be calling `main()` multiple times in one JVM instance to simulate running from the command line. I recommend using

using the Files library to read the dictionary file (for example, to get all words, you could do: `Files.readAllLines(Paths.get(dictionaryFile))` and then handle them). Scanners may have issues with input files.

Timing and Runtime

You will be graded on runtime. You should be able to handle large dictionaries and boards efficiently. For a dictionary of fixed size, and a random board, you should have runtime expected linear in the size of the board - that is, for an NxM board and getting the top K words, your runtime should be expected O(MN log K). Don't think too hard about this expected runtime though - the analysis for this is a little complex and we can certainly bound it tighter. If you have an efficient solution that behaves and grows linearly with the size of the board, you should pass the autograder.

For example, on my computer, one solve on testsmallboard (100x100) takes 209ms and one solve on testlargeboard (500x500, 25x larger) takes 4957ms. Linearly extrapolating from the testsmallboard runtime, we would expect a runtime of 209*25=5225, which is close to our achieved runtime.

Some tips: you cannot inspect all possible permutations of words (in other words, you cannot submit a brute force solution). Your solution should utilize pruning - if you cannot continue constructing a word from a certain letter onwards, you should not explore that letter's neighbor nodes. As a warning: if you have a recursive solution, it is possible that it is slower by a nontrivial constant factor than an equivalent iterative solution.

Error Cases

For Boggle, throw an `IllegalArgumentException` (with some informative message of your choice) if:

1. The input board is not rectangular.

3. N, M, or K is non-positive.

Do not call `System.exit()`.

Submission

Submit the `Boggle.java` file and any of its dependencies to Gradescope. It is not a part of a package so you do not need to zip anything.

Homework 6 Navigation