Log of how I developed my Wordsquare app

=================================

Thought around the problem and decided not to Google for a ready-made algorithm. Thinking it out from scratch felt like a more interesting challenge and still presented a chance to engineer it well.

Wondered what is the significance of the “dictionary” file? It’s used in chapter 14 of a book about data manipulation etc. Reminded me of my Corpus Linguistics studies, so I knew how complex things could get.

Decided to use the dictionary as a source of candidate words, ready-made, rather than attempt to construct valid words and validate them against the dictionary.

Dictionary is big and would either tie up a lot of memory if cached as a whole, or a lot of bandwidth if read a word at a time from the web, so I opted to pull it into the app as a resource and then cache in memory the subset of words that are relevant, given the dimension of the word square and the letters available.

I considered what the rules are for a word being a suitable next row/column, given which words are in place already (e.g. 2nd word beginning with 2nd letter of 1st word etc). Decided to elucidate that more clearly during coding.

Began coding at the file-handling end, building capability to open a read buffer and read a line from it.

Built up test that reads lines and mimics reaching the end of the file and returning null to signify there are no more lines. Nothing about relevance or word squares here. Just serving up lines from a file. Lack of file closure, now that I look at it. I would want to be tidier there.

Created a new class to handle the initial sifting of potentially relevant words. This read lines using the reader, testing each for whether it was the correct length and whether its letters were in the provided set to work with. The test just tests the length of the list. I would strengthen that to at least confirm that each filtered word conformed to those criteria.

Given the ability to get a list of potentially useable words, I drove out development of a class to construct a word square.

Started with a test to get 3x3 only, and with a particular easy word square in mind (letters “aaccrrtty”, to produce ‘act’, ‘car’ and ‘try’). These are in alphabetical order, as were the candidate words, hence easily findable quickly.

This resulted in selectWords(), collectWords(), findNextWord() and derivePrefix(). So selectWords() evaluates what is collected by collectWords(). That finds each next word by finding a word that starts with the letters (prefix) dictated by the words selected already.

Some head-scratching to get to the realisation (line 73) that the number of words gathered so far was an effective index for picking the right letter from each word when looking for the next word. It was more of a passing test moment than a cognitive eureka moment. Nice all the same!

Tried with a harder test that targeted words not in alphabetical order. Failed, despite only being a 3x3 square.

Made selectWords() iterate by starting at successive first word starting points. This didn’t solve the basic issue of an attempted square scuppering itself by producing unmatchable search prefixes.

Had the notion of being able to back out of these situations by popping words from the stack, hence the Deques that are in there. Started out with Stacks but found they were discouraged in favour of LinkedLists etc.

How many words to pop? Not at all obvious. Potentially all of them, given it’s a failed square. Any of the words gathered so far could be a culprit in making the attempt fail.

But the alphabetical set passed first go. That could just have been luck, but it made clear that ordering made a difference.

Switched to a tactic of shuffling the Deque and trying again from scratch after a failure to build. That works, if there are enough iterations and the proposed word square is smallish.

We are in “monkeys on typewriters eventually typing out Hamlet” territory with this approach. It mirrors brute force attacks on passwords of increasing complexity.

I added a main class that does what the tests do, getting the two parameters from args. That formed the entry point for a fat jar to execute outside the IDE.