Julian Galbraith-Paul

Sherif Khattab

CS 1501

09/26/2020

A1: My Approach

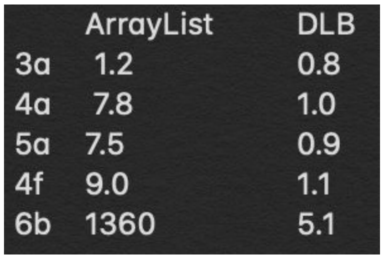
In backtracking there is no room for error, everything must be clear and concise, so I started the project right away...At first, I gave it a go with two stringbuilder arrays, one for row and one for column. This allows our access and checks for rows and columns to be 0(1). However the mechanics were confusing, and after hours of debugging, I had spaghetti code EVERYWHERE (not what you want for backtracking!). So I scratched my entire project and built from the ground up. Rather than using the global StringBuilder and a direction class, I used a global 2-d char array to represent my partial solution and StringBuilders to represent my current rows and columns.

In *isValid* I added a new letter to my partial solution, then converted the corresponding rows and columns into stringbuilders (curRow, curCol). In this step I performed multiple checks to ensure curRow/Col were correctly *formatted* for my prefix and word checks to work properly. This is where debugging was ESSENTIAL! For example, what if any of the strings “ab-”, “b-a”, or “aa+” were inputted as our current row or column. Printing out my partial solution allowed me to visualize what was actually happening under the hood. Then I could adjust my logic accurately to account for the different possible cases, instead of blindly guessing in the dark. After a letter is accepted by isValid, I append it to theBoard and recurse on the choice. If isValid rejects, it simply passes the next letter through at the same index until it accepts. Finally, if theBoard does not contain any ‘+’s, theBoard must be a full solution.

Note: extra credit / more detailed run-time measurements included in last section

After implementing the DLB, I immediately noticed a jurassic enhancement in my program. Rather than hanging on the command line after running with ArrayLists, with the DLB, a solution appeared right away. See chart below for **approximate** runtimes (*more detailed* analysis following...)

*Time in seconds
*Time = avg. of 3 stopwatch



	ArrayList	DLB
3a	1.2	0.8
4a	7.8	1.0
5a	7.5	0.9
4f	9.0	1.1
6b	1360	5.1

As the *size* and the *complexity* of the board increase the DLB outperforms the Arraylist by larger and larger margins. Look specifically at example input 4f, a board containing both irremovable letters as well as blocked off spots. In this case, an addition in *complexity* causes my DLB runtime to hardly fluctuate. However, the ArrayList implementation increases more notably compared to its simple blank 4x4 board as seen in 4a. Furthermore, in example input 6b, as the size of the board increases, our ArrayList runtime increases almost **100X** while the DLB remains somewhat consistent. The DLB is therefore the clear winner!

For part1, the implementation containing Arraylist, the search algorithm is not that efficient. The worst-case runtime is O(LK), where L is the length of the dictionary and K is the key length(largest word in our dictionary, which is 8 plus the null character so 9), therefore the **ArrayList search runtime is O(17273*9).** After accounting for other major factors in my program, the total worst-case runtime is approximately the following...

Note: extra credit / more detailed run-time measurements included in last section

O(boardSize^2 + 26letters*boardSize(boardSize + 2(17273*9)))**

**note I did implement stringbuilder arrays, so for loop in isValid loads curRow/curCol

w/ stringbuilder[]s = O(boardSize^2 + 26letters*boardSize(1 + 2(17273*9)))

For part2, assuming we have S valid characters in our alphabet, and our key contains K characters, the worst-case runtime for my DLB search is O(KS). In this case, S is the number of letters in the english dictionary, 26, and K is the key length(largest word in our dictionary, which is 8 plus the null character so 9), therefore my **DLB search runtime is O(26*9).** After accounting for other major factors in my program, the total worst-case runtime is approximately the following…

O(boardSize^2 + 26letters*boardSize(boardSize + 2(26*9)))**

** w/stringbuilder[]s = O(boardSize^2 + 26letters*boardSize(1 + 2(26*9)))

Once we include the search for the best score, our runtime changes to the following…

O(boardSize^2(boardSize^2 + 26letters*boardSize(boardSize + 2(26*9))))**

** w/stringbuilder[]s = O(boardSize^2(boardSize^2 + 26letters*boardSize(1+2(26*9))))

Finally let's plug in values 3 and 6 for board size and compare the approximate runtimes of the DLB and the ArrayList…

Note: extra credit / more detailed run-time measurements included in last section

Board size = 3

ArrayList = O(3^2 + 26*3(3 + 2(17273*9))) = 24,251,535

DLB = O(3^2 + 26*3(3 + 2(26*9))) = 37,647

ArrayList / DLB = 24,251,535 / 37,647 = 644.2

For board size 3, ArrayList runtime is **644.2x** the DLB runtime

Board size = 6

ArrayList = O(6^2 + 26*6(6 + 2(17273*9))) = 48,503,556

DLB = O(6^2 + 26*6(6 + 2(26*9))) = 73,980

ArrayList / DLB = 48,503,556 / 73,980 = 655.6

For board size 6, ArrayList runtime is **655.6x** the DLB runtime

As we can see from the comparisons above, the DLB is astonishingly efficient in comparison to the ArrayList. This is due to the fact that the ArrayList is reliant on the size of the dictionary while the DLB is restricted by the numbers of characters in the alphabet instead. This is the difference between making 17273 and 26 comparisons in the worst case. At the end of the day, it all boils down to what the problem is at hand. In this problem, DLB’s outperform ArrayLists, but given a different issue, that may not be the case. Analysis is crucial to understanding these divisions and facilitates a programmers ability to accurately select data structures in any context.