Ryan Helmlinger

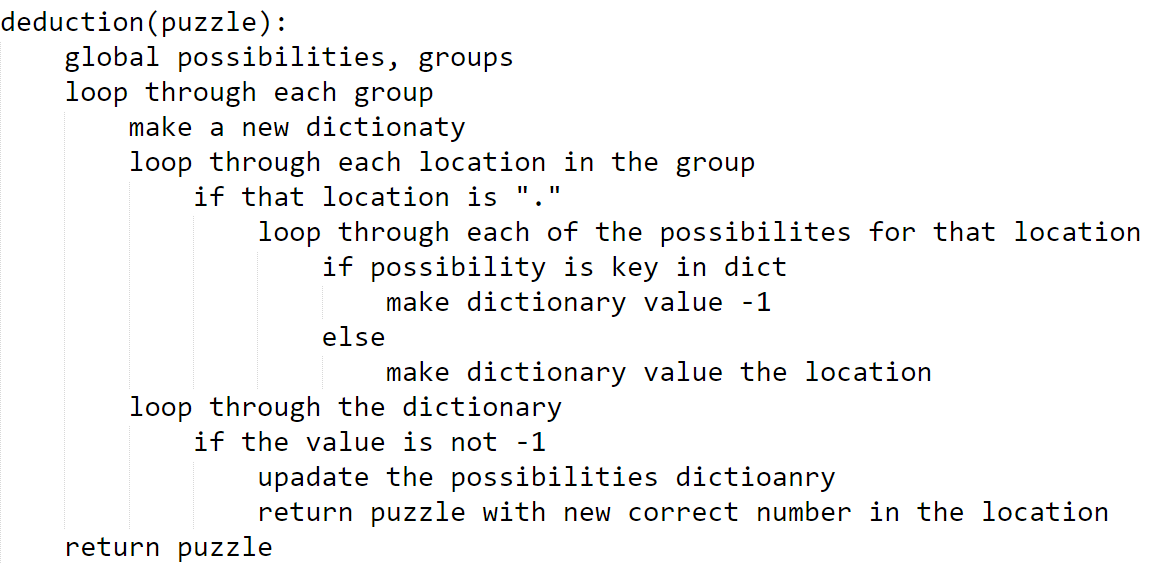
11/24/15

Period 3

Sudoku is a popular puzzle that involves logic and the combination of numbers. The puzzle is most commonly found in a 9x9 grid. The final goal is to fill each of the groups with one of each symbol, which in a 9x9 puzzle all of the digits from 1 to 9. The groups in each puzzle are the horizontal rows, the vertical columns, and the nine 3x3 sub-blocks that compose the grid. The completed puzzle never contains the same number in the same row, column, or 3x3 sub-blocks. I wrote a python program that solves any 9x9 Sudoku puzzle.

My Sudoku python code runs on the TJ servers in about 2-3 seconds and takes around 6000-7000 guesses. I did this by using deductions and by implementing better methods and data structures. One initial data structure I used was syms, which was a set of all of the possible symbols, which for a 9x9 was 1 through 9. Another data structure I used was groups, which was a list of sets in which each index in the list represented the set of all indices for each separate group (either a row, column, or 3x3 sub-region). Another data structure was cellNeighbors, which is a list of sets in which the list contains every spot on the grid and the set contains the intersecting indices of the point. The intersecting indices are the indices that appear in the row, column, and 3x3 sub-region of the certain spot in the grid. And finally, possibilities was a dictionary where the key is the index of puzzle and the value is a set of the possible symbols that can validly fit in that index.

Each puzzle the code solves is sent to my recursive brute force method, which calls helper methods and guesses parts of the puzzle until it is solved. The brute force method initially calls the global variable possibilities. Next I call a deduction method. This method essentially makes a deduction where if a symbol can’t go anywhere in a group except for one spot, then it must go in that one spot. This is the pseudo-code for the deduction method:



This method goes through each of the groups and finds the places where there is a dot. Earlier in the method there also a new dictionary made which will eventually contain the symbol as the key and the index as its value. The code then goes through the possibilities for each of the "." If the possibility is already in the new dictionary, then the value becomes negative one, but if it has not, then the value becomes the location of that symbol. The code does this so that when it finishes all of the loops (except for the loop that goes through the groups), the dictionary has keys with all the possible symbols that can fit into the group, but it also makes the value negative one for all of the symbols that are possible more than once. This leaves only the symbols that appear once, with their index as the value. Then it updates the possibilities according to the new insertion, and then it inserts the symbol into the puzzle. Then my code calls deduction again with the new puzzle as the argument. If the dictionary never had only one possibility, thus could never make a deduction, then it returns the puzzle again, so the method ends.

After the deduction is called, I check to see if the puzzle is completely solved. If it is, I return the puzzle, otherwise I continue through the brute force method. Next, the code loops through the possibilities dictionary and finds the key with the smallest length of the value set. Then I use the possibilities set with the smallest length and guess each of the values in the set. In simpler problems, often times the length of the set is only one, so then it isn’t actually guessing, because it knows the only possible solution. When it guesses each number, it updates the puzzle and the possibilities dictionary and then recursively calls the brute force method, which either returns the correctly updated puzzle, or a blank string meaning that the guess didn’t work. If the puzzle is returned correctly, then all the method does is return the puzzle. But if it isn’t, it keeps looping through the guesses. If it reaches the end of the loop without guessing any spots correctly, then the method returns a blank string to show that the current puzzle isn’t possible to solve.

All of this, using sets, dictionaries, and deductions as mentioned, took about 10-12 seconds to run on the TJ servers. However, one other change made my code speed up about 4 times faster. I have a findPossible method which makes the dictionary of possibilities. It basically loops through every open space in the puzzle and checks its neighbors to find all of the possible values. However, this takes a while after being called every deduction and brute force call, so instead I created a separate method that updated the possibilities index based on the new value entered into the puzzle. These data structures, deductions, and efficiencies have led to my code’s 2 to 3 second runtime.