Judicious brute force

There is a class of puzzles where each position of the puzzle is to be filled in with a symbol. This could be as simple as a 0/1 symbol (such as in the *n*-queens problem where a 0 means no queen at that position and a 1 means there is a queen at that position), or numbers (such as 1-9 in most Sudoku problems), or arbitrary labels (such as 0-9 and A-F in 16 x 16 sudokus). Often, as with a standard Sudoku puzzle, the numbers are acting as only labels, meaning that the numeric qualities of the number are not being used. In those cases, letters or other symbols could just as easily be used.

A possible (horrible) approach to solving this type of puzzle might be

```
def bruteForce(pzl):
  # returns a solved pzl or the empty string on failure
  if pzl is completely filled out:
    return "" if isInvalid(pzl) else pzl

  find a setOfChoices that is collectively exhaustive
  for each possibleChoice in the setOfChoices:
    subPzl = pzl with possibleChoice applied
    bF = bruteForce(subPzl)
    if bF: return bF
  return ""
```

The reason this is so horrible is that it fills each puzzle out completely before testing. For all but the most trivial of problems, the routine will hang because there are way too many possibilities to recurse through. However, we can gain considerable traction with only a slight modification. By testing the validity of a potential solution at each step of the process, it is the hope that large subtrees of invalid possibilities may be immediately pruned. For example, in the *n*-queens problem, if two queens attacked each other, there is no reason to continue filling in the puzzle since it's already unsolvable. The following is what was shown in class as our starting out point for the month:

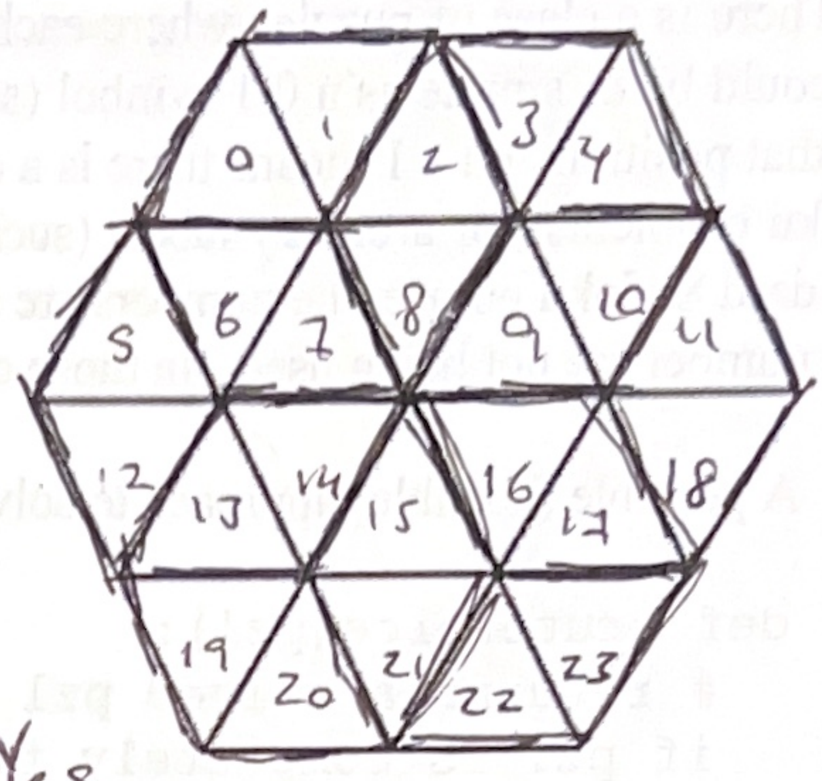
```
def bruteForce(pzl):
  # returns a solved pzl or the empty string on failure
  if isInvalid(pzl): return ""
  if isSolved(pzl):  return pzl

  find a setOfChoices that is collectively exhaustive
  for each possibleChoice in the setOfChoices:
    subPzl = pzl with possibleChoice applied
    bF = bruteForce(subPzl)
    if bF: return bF
  return ""
```

Your homework (to be ready by the start of class on T, Oct. 24) is to use the above *judicious* `bruteForce` code (or small variation on it) to solve the following three problems:

Q1. Given an $n \times n$ board and n distinct types of tiles, where there are exactly n of each type of tile (for a total of n^2 tiles), is it possible to place these tiles so that no row, no column, and no main diagonal has more than one of the same type of tile? Run for $n = 7$.

For Q2 and Q3, you are given an inner hexagon (made up of 6 small triangles), and around each of the 6 outer vertices of this inner hexagon you construct another hexagon to wind up with a total of 7 complete hexagons utilizing 24 of the small triangles in total. In other words, the outer perimeter has 18 triangles.



In addition, for each of the three main directions for this amalgamation of triangles, there are 4 rows of either 5 or 7 triangles. That is, there are 12 distinct rows of triangles.

Q2) Is there a way to label each small triangle with an integer from 1-6 where none of the seven hexagons has two triangles with the same number? Yes

Q3) Is there a way to label each small triangle with an integer from 1-7 where none of the seven hexagons and none of the twelve rows of triangles has two triangles with the same number?

If the answer to any of the question is yes, you should provide a solution.

Q2: 1 2 3 1 2
1 4 5 6 4 5 1
2 6 3 1 2 3 6
2 4 5 4 6

Q3: No solution possible

Specifics: You are to write a command line scripts that takes up to one argument. If an argument is provided, it acts as a partially filled out board, and if no argument is provided, the default should be an empty board (ie. `"."*sizeOfBoard`). The output should either be the text `"No solution possible"`, or you should show a solution. Note well: **In this unit, the period (and not an underscore) will indicate a position that is to be filled out.**

The solution that you show should be a square for Q1. As an example, if you were solving the 4x4 version you might display the following (of course you may wish to use different symbols such as 1-4 or A-D):

```
*=+/
/+=*
=*/+
+/*=
```

For Q2 & Q3, we would like a straightforward output. Our goal here is not to create beautiful hexagons but to output something that is straightforward to interpret. To that end, we will use a scheme that a former student came up with: `ACDEFGHIJKLMNOPQRSTUVWX` =>

```
  A B C D E        (5 triangles)
F G H I J K L      (7 triangles)
M N O P Q R S      (7 triangles)
  T U V W X        (5 triangles)
```

Considerations for Q2 and Q3:

Your program is not expected to draw triangles or hexagons. However, it must have some way of identifying each of the triangles. So, you will have to devise some indexing scheme to decide which is the 0 triangle, which is triangle with index 1, and so forth. As long as each triangle has a unique index, it would be a valid indexing (you could even give each triangle a name, like "Fred" and "Bill" and "Martha", but it's not clear to me what purpose this would serve).

As discussed in class, isFinished() is trivial, so at first glance, it looks like you have 3 scripts to write where you have three functions to implement per script (isInvalid(), find a collectively exhaustive set of choices, and outputSolution()). However, the usual way to find an exhaustive set of choices is to find the location of a period in the puzzle (which means that position needs to be filled in), so this is the same function common to all three scripts.

Now, the most important point is following (ie. the next month is built around this point – it is vital to understand it). It looks like `isInvalid()` must be customized for each problem. However, in all three problems, it is the case that you don't want to have a duplicate symbol appear in a certain set of positions, and this is true for several sets of positions. For example in Q1, no symbol should appear twice in any column. In Q2 and Q3, no symbol should appear twice in any hexagon. In Q1 and Q3, no symbol should appear twice in any "row". In all of these cases, we have a geometrical idea of what a row, column, hexagon, or diagonal is. However, by the time you implement it in code, you are concerned about which positions (in your puzzle string) constitute the geometrical item of interest. That is, in all cases you are interested in a set of positions where no symbol at those positions may be repeated. This set of positions is known as a **constraint set**.

Therefore, for each of the three problems, you should identify a list of constraint sets. How you identify them is up to you. For Q1, it is straightforward to use a script to generate the constraint sets. For Q2 and Q3, you may wish to code the constraint sets by hand. In the end, you should have a list of sets (of positions), which will be unique to each of your scripts. However, `isInvalid()` can be the same code for all of them. The number of constraint sets for Q1 is $2n + 2$ – note that there are exactly two main diagonals. The number of constraint sets for Q2 is 7, while the number of constraint sets for Q3 is $7 + 12 = 19$.

In the end, you will write `isInvalid(pzl)` once to take a puzzle and go through the list of constraint sets looking for a violation. Your `bruteForce()` code will be the same. What will be different is that for each problem, you will have a different list of constraint sets, and for Q1 you will have one output routine (for displaying a square puzzle), while for Q2 and Q3 you should use the above scheme to display a solved puzzle.

Here is another example for a Q2/Q3 puzzle:

```
   1 2 3 4 5
 6 . . . . . 6
 1 . . . . . 1
   5 4 3 2 1
```

where the top row represents the first 5 triangles going across, the 2nd row represents the next 7, and so on. If the above were to be input as a single string, it would look like:

```
hexy.py 123456.....61.....154321
```

with at most 10 characters to be filled out (or a determination made that no solution is possible). The original Q2 and Q3 just want a string of 24 dots.