Additional Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of March 7, 2023

This document contains the specifications for additional Python problems created by Ilkka Kokkarinen starting from March 2023, to augment the original set of 109 Python Problems for the course **CCPS 109 Computer Science I** for Chang School of Continuing Education, Toronto Metropolitan University, Canada.

These problems are intended for first year students of various levels. Same as in the original collection, these problems are listed roughly in their order of difficulty. Complexity of their solutions ranges from straightforward loops to rather convoluted branching recursive searches that allow all kinds of clever optimizations to prune the branches of the search.

Rules for these problems are exactly the same as they were for the original 109 problems. You must implement all your functions in a single source code file named `labs109.py`, along with the your solutions to the original 109 problems. This setup allows you to run the `tester109.py` script at any time to validate the functions that you have completed so far. These automated tests will be executed in the same order that your functions appear inside the `labs109.py` file.

The Fischer King

```
def is_chess_960(row):
```

Tired of seeing his beloved game devolve into rote memorization of openings, the famous chess grandmaster and overall colourful character Bobby Fischer generalized chess into a more exciting variation known as "Chess 960" that plays otherwise the same except that the home rank pieces are randomly permuted before each match begins, to render any memorization of openings moot. So that neither player gains an unfair random advantage from the start, these pieces are permuted the exact same way for both black and white. However, to maintain the spirit of chess, the two bishops must be placed on squares of opposite colour (that is, different **parity** of their column numbers), and the king must be positioned between the two rooks to enable **castling**.

Given the home rank as some permutation of the string `'KQrrbbkk'` with these letters denoting the pieces of **K**ing, **Q**ueen, **r**ook, **b**ishop and **k**night, this function should determine whether that string constitutes a legal initial position in Chess 960.

row	Expected result
`'rbrkbkQK'`	`False`
`'rkbQKbkr'`	`True`
`'rrkkbKQb'`	`False`
`'rbbkKQkr'`	`True`
`'bbrQrkkK'`	`False`
`'rKrQkkbb'`	`True`
`'kkQbKbrr'`	`False`
`'bQrKkrkb'`	`True`

Discrete math enthusiasts can convince themselves that out of the 8! / (2! × 2! × 2!) = 5040 possible permutations of these chess pieces, exactly 960 permutations satisfy the above constraints.

Top of the swops

```
def topswops(perm):
```

The late great John Horton Conway invented all sorts whimsical mathematical puzzles and games that are simple enough even for wee children to understand, yet hide far deeper mathematical truths beneath the superficial veil. This problem deals with topswops, an automaton puzzle played with cards that are numbered with positive integers from 1 to *n.*

These cards are initially laid out in a row in some `permutation`, given to your function as a tuple. Each move in topdrops reverses the order the first *k* cards counted from the beginning, where *k* is the card at the head of the row. For example, if the current permutation is `(3, 1, 5, 2, 4)`, reversing the first three cards gives the permutation `(5, 1, 3, 2, 4)`, from which reversing the first five cards gives the permutation `(4, 2, 3, 1, 5)`, from which reversing the first four cards gives the permutation `(1, 3, 2, 4, 5)`, where the game will remain stuck in a fixed state.

It can be proven that for every initial permutation of these *n* cards, the card that carries the number one must eventually end up the first card in the row to terminate this process. Your function should count how many moves are needed to reach this fixed state from the given initial `permutation` of cards, and return that count.

`perm`	Expected result
`(1, 2)`	0
`(2, 1)`	1
`(3, 2, 5, 1, 4)`	5
`(7, 1, 2, 4, 6, 3, 5)`	7
`(4, 1, 8, 5, 6, 9, 2, 7, 3)`	13
`(4, 10, 7, 6, 15, 8, 14, 19, 2, 17, 3, 16, 12, 13, 5, 1, 9, 11, 18)`	20
`(9, 1, 3, 17, 8, 7, 4, 16, 15, 11, 2, 19, 6, 14, 12, 10, 5, 18, 13)`	28

An interesting question is to come up with a permutation for the given *n* that needs the largest possible number of moves to terminate. Exact solutions for this are known for values of *n* up to 17.

Soundex good to me

```
def soundex(word):
```

With all those homophones easily confused with each other, spoken words in the English language are not always the clearest to decipher. This would be problematic especially in the field of genealogy, since our ancestors may have spelled their names with different variations of spelling. Patented back in 1918, the ingenious Soundex encoding encodes each English `word` into a short code of a single letter followed by three numerical digits, so that words that sound similar end up getting the same code, which is important when searching through paper records by hand.

The rules to construct the Soundex encoding for the given `word` are listed on the linked Wikipedia page for the reader to consult and convert to working Python code. When implementing these rules, be careful to distinguish between the vowels *aeiou* and the letters *yhw* so that unlike in some implementations of Soundex, we never use *y* as a vowel. Even though both types of letters are dropped from the word for constructing the code, having one or more vowels between two consonants with the same digit code causes that digit code to be duplicated, whereas having one of the ignored letters *yhw* between those consonants squeezes these duplicate digit codes into a single digit.

`word`	Expected result
`'robert'`	`'r163'`
`'rupert'`	`'r163'`
`'ashcroft'`	`'a261'`
`'tymczak'`	`'t522'`
`'pfister'`	`'p236'`
`'honeyman'`	`'h555'`
`'stewart'`	`'s363'`
`'stuart'`	`'s363'`
`'gutierrez'`	`'g362'`
`'carwruth'`	`'c630'`
`'leonardo'`	`'l563'`
`'leotardo'`	`'l363'`

Sum of consecutive squares

```
def sum_of_consecutive_squares(n):
```

The original collection of *109 Python problems* included problems to determine whether the given positive integer `n` can be expressed as a sum of exactly two squares of integers, or as a sum of cubes of one or more distinct integers. Continuing on this same spirit, the problem "Sum of Consecutive Squares" that appeared recently in the Stack Overflow Code Golf problem collection site asks for a function that checks whether the given positive integer `n` could be expressed as a sum of squares of one or more **consecutive** positive integers. For example, since $77 = 4^2 + 5^2 + 6^2$, the integer 77 can thusly be expressed as a sum of consecutive squares.

This problem is best to solve with the classic **two pointers** approach, using two indices `lo` and `hi` to as the **point man** and **rear guard** who delimit the range of integers whose sum we want to make equal to `n`. Initialize both indices `lo` and `hi` to the largest integer whose square is less than equal to `n`, and then initialize a third local variable `s` to keep track of the sum of squares of integers from `lo` to `hi`, inclusive. If `s` is equal to `n`, return `True`. Otherwise, depending on whether `s` is smaller or larger than `n`, expand or contract this range by decrementing either index `lo` or `hi` (or both) as appropriate, update `s` to give the sum of squares in the new range, and keep going the same way.

`n`	Expected result
`9`	`True`
`30`	`True`
`294`	`True`
`3043`	`False`
`4770038`	`True`
`24015042`	`False`
`736683194`	`False`

When implemented as explained above, this function maintains a **loop invariant** that says that if `n` can be expressed as a sum of squares of consecutive integers, the largest integer used in this sum cannot be greater than `hi`. This invariant is initially true, due to the initial choice of `hi`. Maintenance of this invariant during a single round of the loop body can then be proven for both possible branches of `s > n` and `s < n`. Since at least one of the positive indices `hi` and `lo` will decrease each round, this loop must necessarily terminate after at most `2*hi` rounds, a massive improvement over the "Shlemiel" approach of iterating through all possibilities in quadratic time with two nested loops… or if the coder is especially clumsy, cubic time for three levels of nested loops.

Scatter her enemies

```
def queen_captures_all(queen, pawns):
```

This cute little problem was inspired by a tweet by chess grandmaster Maurice Ashley. On a generalized *n*-by-*n* chessboard, the positions of a lone `queen` and the opposing `pawns` are given as tuples `(row, col)`. This function should determine whether the `queen` can capture all enemy `pawns` in an unbroken sequence of moves where each move captures exactly one enemy pawn. The pawns stay put while the queen executes her sequence of moves. Note that the queen cannot teleport through pawns, but must always capture the nearest pawn in her chosen current direction of move.

This problem is best solved with recursion. The base case is when zero pawns remain, so the answer is trivially `True`. Otherwise, when *m* pawns remain, find the nearest pawn to the queen for each of the eight compass directions. For each direction where there exists a pawn to be captured, recursively solve the smaller version of the problem with the queen, having captured that particular pawn, attempts to capture the *m* – 1 remaining pawns in the same regal fashion. If any one of the eight possible starting directions yields a working solution for the smaller problem with *m* – 1 pawns, that gives a working solution for the original problem for *m* pawns.

queen	pawns	Expected result
`(4, 4)`	`[(0, 2), (4, 1), (1, 4)]`	`False`
`(1, 3)`	`[(0, 3), (2, 0), (2, 2)]`	`True`
`(2, 1)`	`[(1, 1), (5, 4), (2, 0), (5, 3)]`	`True`
`(0, 0)`	`[(3, 1), (4, 3), (2, 0), (6, 4), (0, 5)]`	`False`
`(11, 7)`	`[(0, 4), (10, 8), (5, 8), (6, 9), (9, 6), (11, 9), (2, 13), (11, 6), (5, 0), (9, 7), (11, 4)]`	`True`

The bottleneck of this recursion is to quickly find the nearest pawn in each direction, along with the ability to realize as soon as possible that the current sequence of initial captures cannot be extended to capture the remaining pawns. If you preprocess the pawns to encode the adjacency information as a graph whose nodes are the positions of each pawn and the initial position of the queen, you need to note that the neighbourhood relation between these positions changes dynamically as the queen captures pawns along her route, causing pawns initially separated by those captured pawns to become each other's neighbours. As the recursive calls return without finding a solution, you need to downdate your data structures to undo the updates that were made to these data structures to reflect the new situation before commencing that recursive call.

Complete a Costas array

```
def costas_array(rows):
```

Over the past decade, this author has learned so much from John D. Cook's excellent blog "Endeavour" that he can't even begin to count that high. A recent blog post "Costas arrays" examines a variation of the famous N queens problem where queens have been replaced by rooks. Of course, placing *n* rooks on the chessboard so that no two rooks threaten each other is trivial; in absence of diagonal threats, any permutation of column numbers whatsoever works as a legal placement for rooks, one rook per row same as for queens.

To make this problem interesting, we require that all direction vectors between pairs of rooks are unique. A placement of *n* rooks in this manner is called a Costas array, useful in certain radar and sonar applications. Recognizing a legal Costas array is straightforward, but other than the trivial enumeration of going through all permutations, there are no efficient algorithms to generate all possible Costas arrays.

In this problem, your function is given a list of `rows` where each element is either `None` to indicate that that row does not yet contain a rook, or is the column number of the rook that has already been nailed in that row. Your function should determine whether it is possible to place rooks on the unfilled rows to complete the permutation into a Costas array.

`rows`	Expected result
`[3, 1, None, None]`	`True`
`[5, None, 1, 0, 3, None]`	`False`
`[4, None, None, 0, 5, None]`	`True`
`[4, 6, 3, 0, 2, None, 7, None, None]`	`False`
`[4, None, None, 6, None, 5, None, 0, 9, None]`	`True`
`[9, 0, 7, 6, None, 15, 11, None, 1, None, None, None, 4, 10, 8, None]`	`False`
`[2, None, 9, None, 4, 1, 10, 0, None, None, 6, 12, 11, None, 5, None]`	`True`