Additional Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of March 6, 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.

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 developed a variation known as "Chess 960" where 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 advantage from the start, the pieces are permuted the exact same way for both black and white. However, to maintain the spirit of chess, not every permutation of pieces is allowed, but the two bishops must be placed on squares of opposite colour, and the king must be positioned between the two rooks.

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 the game of 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 $8! / (2! \times 2! \times 2!) = 5040$ possible permutations of the chess pieces, exactly 960 permutations satisfy the above constraints.

Soundex

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

With all its 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 the ancestors of people may have spelled their names using slightly 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 digits, to allow similar names to be indexed together.

The rules to construct the Soundex encoding for the given `word` are given 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*; even though both types of letters are dropped from the word during the construction of the code, having one or more vowels between two consonants with the same digit code causes that consonant digit to be duplicated, whereas having any one of the latter type of ignored letters *yhw* causes such duplicate digits to be squeezed 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'`

Sum of consecutive squares

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

The original collection of *109 Python problems* included problems about testing whether the given positive integer `n` could 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 be expressed as such a sum of consecutive squares.

This problem is handiest to solve with the classic **two pointers** approach, using two indices `lo` and `hi` to as the **point man** and **rear guard** that 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.

`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 `n` cannot be expressed as a sum of squares of consecutive integers so that the largest integer in this sequence is 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` must decrease each round, this loop will necessarily terminate after at most `2*hi` rounds.

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 all of 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 do not move 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 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 a similar fashion. If any one of the eight possible directions for the first move yields a working solution, this 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, 1)`	`[(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 quickly finding the nearest pawn in each compass direction, along with the ability to realize as soon as possible that the current sequence of captures has already painted itself in a corner by making it impossible 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 may change dynamically as the queen captures pawns along her route, making 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.