Additional Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of March 14, 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 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 of kings 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 get-go, both black and white pieces are permuted the exact same way.

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 `'KQrrbbkk'` with these letters denoting the **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 under this discipline.

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

Discrete math enthusiasts can then 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 of whimsical mathematical puzzles and games that are simple enough even for wee children to understand, yet hide far deeper mathematical truths underneath 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 topswops 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 same 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

A still open problem in combinatorics is to find a permutation for the given *n* that requires the largest possible number of moves to terminate. Exact solutions are known only for *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'`

Ten pins, not six, Dolores

```
def bowling_score(rolls):
```

Even if we post-postmodern people will end up bowling alone as our society becomes more alienated and atomized, calculating the score in ten-pin bowling should still make for a fine exercise in Python programming. This function should compute the score for the given `rolls`, given as a list with each of the ten frames encoded as a string. Strikes are encoded as `'X'`, misses as `'-'`, and spares as `'\'`. For simplicity, we assume that no fouls take place during the game.

Rules for scoring a game of bowling can be found on several sites online for the readers not familiar with the notation. For example, see the interactive bowling score calculator at Bowling Genius. Note also the special three-character form of the last frame when it is either a strike or a spare.

`rolls`	Expected result
`['X', '4-', '54', '-2', '72', 'X', 'X', '5/', 'X', '--']`	113
`['2/', '34', '9/', '52', '4-', '11', '81', '8/', 'X', '6-']`	99
`['X', '7/', '-/', '31', '18', '11', '4/', '33', '43', '72']`	93
`['X', '2/', '9/', 'X', '8/', '6/', '25', '8/', '1/', '71']`	150
`['X', 'X', 'X', '6/', '2/', 'X', 'X', '2/', '4/', 'XXX']`	214
`['8/', '7/', '8/', '9/', 'X', 'X', '-9', '3/', 'X', 'XX7']`	199
`['42', '8/', '8/', '-8', 'X', 'X', '6/', '31', '5/', 'X51']`	141
`['21', '11', '8/', 'X', '61', '7/', '4/', 'X', 'X', '4/X']`	147
`['3/', '45', '2/', '7/', '2/', '8/', '11', '4/', '3/', '61']`	119

Mian–Chowla sequence

```
def mian_chowla(n):
```

The Mian–Chowla sequence is another interesting sequence of increasing positive integers determined with a simple rule from which a chaotic complexity organically emerges. As usual, the sequence starts with the element 1 in the first position. After that, the Mian–Chowla sequence can be defined in two equivalent ways, of which we here use the one that makes the implementation easier: **all differences between two distinct elements in the sequence must be pairwise distinct**. That is, if the sequence contains two distinct elements *a* and *b*, it cannot contain any other two distinct elements *c* and *d* so that $b - a = d - c$. The sequence begins 1, 2, 4, 8, …, giving the first impression of being the boring powers of two, but then bursts out as 13, 21, 31, 45, 66, 81, 97, 123, 148, …

This function should return the element in the position `n` in the sequence, using zero-based indexing. The automated tester is guaranteed to give your function the values of `n` in strictly increasing order, so your function should store its progress in generating this sequence in some global variables outside the function, so that your progress doesn't always get erased between the function calls and force you to again and again generate the entire sequence from the beginning like some modern day Sisyphus. Use a Python `set` to remember all the differences you have already encountered in the sequence, aided with a variable that keeps track of the smallest positive integer that you haven't seen as a difference of two elements so far.

`n`	Expected result
`0`	`1`
`1`	`2`
`19`	`475`
`46`	`4297`
`99`	`27219`
`300`	`524307`
`400`	`1145152`
`500`	`2107005`

'Tis but a scratch

```
def knight_survival(n, x, y, k):
```

A lone chess knight starts at the position `(x, y)` on the generalized *n*-by-*n* chessboard, position indexing again zero-based. The knight must make `k` random moves, each move done with the same 1/8 probability to one of the eight possible directions of the knight's move. Should a move end up outside the bounds of the board, the knight falls to his doom. This function should compute the exact probability that the knight survives on the board for the duration of the entire series of `k` moves, and return that exact probability as a `Fraction` object.

The formula for the survival probability $S(n, x, y, k)$ can be expressed recursively. The base cases of the recursion are when the coordinates (x, y) are outside the board and the survival probability is zero, and when $k = 0$ so the survival probability is one. Otherwise, the desired probability is the sum of the eight survival probabilities $S(n, x', y', k - 1)$, multiplied by the uniform transition probability 1/8 to make these probabilities add up to one, where x' and y' go through all the possible squares reachable by single knight's move from the coordinates (x, y).

This recursion can be expressed as recursion aided with some handy `@lru_cache` magic. Alternatively, you can solve this with **dynamic programming** by creating an *n*-by-*n* table of probabilities, all initialized to the same value 1. From this table that contains the survival probabilities for the current *k*, initially 0, you can compute the next higher table of survival probabilities for $k + 1$.

n	x	y	k	Expected result
4	3	0	3	5/128
9	8	2	1	1/2
9	6	5	1	1
10	1	6	3	119/256
11	3	5	11	124974217/536870912
13	1	12	13	25928405477/549755813888

This problem is "Knight Probability in Chessboard" at LeetCode.

Do or die

```
def accumulate_dice(d, goal):
```

Right on the heels of the previous problem about recursively computed probabilities, a fair `d`-sided die is repeatedly rolled and the results are added together until the total becomes greater or equal to the given `goal`. The final total can therefore end up being anything from `goal` to `goal+(d-1)`, inclusive. This function should compute the exact probabilities of the final total ending up at each of those values, and return the answer as a list of `Fraction` objects with exactly `d` elements.

Calculating the exact probabilities correctly requires a bit more finesse in this problem. To achieve this, we need to define the function $P(s, k)$ for the probability that the total sum of dice equals exactly s after k rolls. The base cases of this recursion are $P(0, 0) = 1$ and $P(s, 0) = 0$ for $s > 0$. The general probability $P(s, k)$ can be computed from the probabilities $P(s', k - 1)$ where s' goes through all sums that a single roll could turn into the total of s. Note that the sums greater than or equal to `goal` are **absorbing states** where further rolls do not change the achieved total. (As you can see in the third row in the table below, it is also possible for `d` to be greater than `goal`.)

`d`	`goal`	Expected result
2	4	`[Fraction(11, 16), Fraction(5, 16)]`
4	5	`[Fraction(369, 1024), Fraction(305, 1024), Fraction(225, 1024), Fraction(125, 1024)]`
3	8	`[Fraction(3289, 6561), Fraction(2200, 6561), Fraction(1072, 6561)]`
6	11	`[Fraction(106442161, 362797056), Fraction(87771985, 362797056), Fraction(65990113, 362797056), Fraction(50655625, 362797056), Fraction(34445005, 362797056), Fraction(17492167, 362797056)]`
8	19	An eight-element list whose first element equals `Fraction(31945752545707729, 144115188075855872)`

This problem was inspired by a problem titled “Rolling a Die” in the collection “Mathematical Morsels” by Ross Hensberger, asking the reader to prove that `goal` is always the most probable total to achieve with the rolls, regardless of `d`. It's always nice to have numerical confirmation for things that you have only proven to be so, but have not actually tried out.

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 popular Stack Overflow Code Golf coding 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 thus be expressed as a sum of some number of consecutive integer 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 one 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 to 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 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.

Boggles the mind

```
def word_board(board, words):
```

In the spirit of the game of Boggle, you are given an *n*-by-*n* `board` of letters encoded as a list of lists of characters, and a list of acceptable `words` listed in sorted dictionary order. This function should return the sorted list of all words that can be found on the `board` by starting from some square and repeatedly moving into one of the four neighbouring squares that has not yet been visited during the traversal of that word. To keep these results manageable even for the humongous wordlist used in our word problems, we are interested only in words that are at least five characters long, and unlike what is allowed in Boggle, do not move diagonally on the board.

This problem is best solved with a nested recursive function whose parameters are the current coordinates on the board, along with the word constructed so far. Maintain two sets on the side; the first set to keep track of the words that have already been found during the recursive search, and the second set to keep track of which squares have already been visited for the current word. Inside the recursive search function, use the `bisect_left` function from the Python standard library to look for the current word in the list of sorted `words`. If the current word appears in that list, add it to the set of found words. If the current word could theoretically still be extended into a longer legal word, continue the recursive search from each of the still unvisited neighbouring squares.

`board`	Expected result (using `words_sorted.txt`)

`[['e', 'c', 'a', 'l'],`
` ['d', 'd', 'i', 'p'],`
` ['i', 's', 'c', 'o'],`
` ['d', 'n', 'u', 'l']]`
	`['alpid', 'caids', 'cedis',`
	`'copia', 'decal', 'disci',`
	`'disco', 'eddic', 'laced',`
	`'laics', 'lucia', 'lucid',`
	`'picul', 'place', 'placed',`
	`'plaid', 'plaids', 'pocul',`
	`'undid']`

`[['u', 'o', 'h', 'a', 'r'],`
` ['s', 'a', 'c', 'o', 'r'],`
` ['e', 'e', 'n', 'v', 'e'],`
` ['b', 'k', 'n', 's', 'n'],`
` ['r', 'o', 'e', 's', 'i']]`
	(a list of 26 words, the longest of which is `'housebrokenness'`)

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 the *n* column numbers whatsoever works as a legal placement for rooks, one rook placed in each row same as was done with queens.

To make this problem with rooks worth your while, we impose an additional requirement that all direction vectors between pairs of rooks must be unique. A placement of *n* rooks in this manner is called a Costas array, apparently useful in certain radar and sonar applications. Recognizing a legal Costas array is straightforward. However, no algorithms to generate all possible Costas arrays of size *n* more efficiently than the trivial brute force iteration through all *n*! permutations and cherry picking those permutations that satisfy the criterion of unique direction vectors between all pairs of rooks have been discovered.

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 somehow 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`

A Lindenmayer system, bigly

```
def lindenmayer(rules, n, start='A'):
```

Lindenmayer systems, or just L-systems for all of us who don't have time for nonsense, are a famous grammar formalism typically used to generate fractal strings to be converted into visually appealing graphical forms. A Lindenmayer system consists of an alphabet of characters (conventionally, uppercase letters) and a dictionary of `rules` to rewrite each character into a string of characters.

The process starts from a string that consists of the given `start` character. Each round, each character `c` in the current string is replaced by the string `rules[c]`, all these replacements done logically simultaneously. For example, consider an L-system with the alphabet `A`, `B` and rules `{'A':'AB', 'B':'A'}`. Starting from the string `'A'`, the first round of application of rules turns this into `'AB'`. The second round turns this into `'ABA'`, which then turns into `'ABAAB'`, and so on.

This function should return the character in position `n` in the string produced by the given `rules`, once these `rules` have been applied as many times as needed to make the string long enough to include the position `n`. The automated tester will give your function such large values of `n` that you can't possibly generate the entire string and then look up the answer in there, but you need to be more clever about how you execute this computation. To begin, you should define a nested utility function `length(rules, start, k)` that computes the length of the string that the `rules` produce from the given `start` symbol after exactly `k` rounds. (Just add up the lengths of the strings these `rules` produce from the characters for the replacement of the `start` symbol in `k-1` rounds. Good recursion, with `@lru_cache`.) Armed with this function, you can zoom in only to the relevant substring at each round, instead of having to build up the entire universe-sized result string.

n	rules	start	Expected result
9	`{'A': 'CAB', 'B': 'ABC', 'C': 'ABA'}`	`'B'`	`'C'`
678	`{'A': 'DABD', 'B': 'CA', 'C': 'AD', 'D': 'BC'}`	`'A'`	`'B'`
10**100	`{'A': 'AB', 'B': 'A'}`	`'A'`	`'A'`

Note from the last row of the table how the Grand Slam problem number 109 in the original Python problems collection, “Infinite Fibonacci Word”, is a trivial special case of this function.