Additional Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of April 19, 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.

Table of Contents

The Fischer King

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

If you think that you have found a good solution, you should still look around a bit more, since a great solution might be waiting just around the corner. 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 convince themselves that out of the 8! / (2! × 2! × 2!) = 5040 possible permutations of these chess pieces, exactly 960 permutations satisfy the above constraints.

Multiplicative persistence

```
def multiplicative_persistence(n, ignore_zeros=False):
```

Digits of the given positive integer `n` are multiplied together to produce the next value of `n`, and this operation is repeated until `n` becomes a single digit. For example, `n = 717867` becomes 16464, which becomes 576, which becomes 210, which finally becomes 0 where it will then stay. This function should return the multiplicative persistence of `n`, that is, the number of iteration rounds required to turn `n` into a single digit in this manner.

Since the appearance of even one zero digit in the number makes the entire product to be zero, most values of `n` become zeros in short order. To make this process that much more interesting, if the keyword argument `ignore_zeros` is set to `True`, zero digits are ignored during multiplication.

`n`	`ignore_zeros`	Expected result
`717867`	`False`	`4`
`717867`	`True`	`4`
`36982498883598862928`	`False`	`2`
`36982498883598862928`	`True`	`6`
`7899978679899687`	`False`	`3`
`7899978679899687`	`True`	`6`

When zeros are not ignored, the number *n* = 277777788888899 holds the highest known multiplicative persistence of 11 rounds. Since multiplication doesn't depend on the order of digits, any permutation of those same digits has the same multiplicity, and listing the digits in ascending order gives the smallest number with that persistence. Persistences seem to go up to eleven but no higher, since no numbers with a higher persistence than this are known, nor are believed to exist.

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 up to $n = 17$. This problem is a good illustration of the principle that even if some function f is straightforward to compute and understand, properties of its inverse f^{-1} might not be equally simple, and finding an x for the given y so that $y = f(x)$ becomes an exponentially arduous task.

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 used different variations of spelling to name themselves. 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

Strict majority element

```
def has_majority(items):
```

This function should determine whether the given list of `items` contains a **strict majority** element, that is, some element that occurs in the list more times than all other elements put together. Unlike the sloppy definition "at least half", this definition makes the majority element unique if one exists.

This problem can be solved efficiently in two different ways. The first way uses a **counting dictionary** to tally up the occurrence counts of each element in `items`, and then iterates through the keys of this counter dictionary to check at how many times that key occurred inside `items`, keeping track of the element with the highest occurrence count seen so far. Nothing wrong with this technique, and Python makes it easy to implement for a good enough solution for this problem.

However, a more clever way to crack this dusty old algorithmic chestnut is to notice that if you ignore any even-length prefix of `items` that contains k copies of some element along with k any other elements, the majority element of the original list will necessarily also be the majority elements of the rest of the list after ignoring these $2k$ elements. This second way allows this problem to be solved with two sequential for-loops iterating over the `items` that operate **in place**, that is, use only a constant amount of bookkeeping memory in addition to the `items`.

`items`	Expected result
`[]`	`False`
`[42]`	`True`
`[1, 2, 1, 2]`	`False`
`[7, 5, 5, 5, 1, 5, 5, 1, 5]`	`True`
`[16, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4]`	`True`
`[3, 9, 20, 33, 33, 4, 20, 33, 3, 20, 20, 33, 4, 33, 3, 33, 33, 33, 33, 3, 33, 33, 5, 3, 6, 3, 3, 3, 3, 3, 3, 3, 3, 6, 3, 3]`	`False`

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 with 1, 2, 4, 8, giving the first impression of being the 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`

A place for everything and everything in its place

```
def insertion_sort_swaps(items):
```

Of all the simple sorting algorithms, **insertion sort** is not merely the shortest but also provably the most efficient. Starting from a singleton prefix that contains only one element and is therefore sorted by default, insertion sort maintains and grows a prefix of sorted elements. The outer loop of insertion sort iterates through the `items` from left to right. The inner loop grabs the element just past the current sorted prefix, and swaps that element with the preceding larger elements until it either ends up at the beginning, or is preceded by an element at most equal to it.

For example, the list `[5, 2, 7, 4, 1]` first becomes `[2, 5, 7, 4, 1]`, which then becomes `[2, 5, 7, 4, 1]`, which becomes `[2, 4, 5, 7, 1]`, which becomes `[1, 2, 4, 5, 7]`. Since your function could just pass the buck of sorting the `items` to its native `sort` method, and our automated tester doesn't have X-ray glasses to see how exactly your function does the job, your function must return the total number of adjacent element swaps during the insertion sort as **proof of work** of actually having gone through the rigmarole of the insertion sort algorithm.

`items`	Expected result
`[1, 2, 3, 4]`	`0`
`[2, 1, 0, -1]`	`6`
`[4, 3, 1, 1, 0, -1, 2, -3]`	`23`
`[3, 2, 21, 7, 23, 5, -25, 7, -18, -21, 8, 11, 23, -19]`	`43`
`list(range(10000, 0, -1))`	`49995000`

For the given list of `items`, this number of required element swaps is an important combinatorial quantity known as its **inversion count**, that is, the number of element pairs in unsorted order relative to each other. It is easy to see that a sorted list has zero inversions, and that swapping two adjacent elements can decrease the inversion count by at most one. Therefore any sorting algorithm limited to swap adjacent elements only must necessarily consume time at least in linear proportion to the inversion count of that list. Insertion sort requires exactly that much, plus the linear time needed to take a butcher's at each element even before any comparisons and swaps, and is thus optimal among sorting algorithms limited to compare and swap adjacent elements only.

When there's no item, there's no problem

```
def stalin_sort(items):
```

Unlike insertion sort and other bourgeois comparison sorting algorithms, **Stalin sort** is a famous joke sort algorithm that wastes no time with decadent notions of fairness or similar considerations. Inspired and named after the original anti-capitalist social justice warrior, Stalin sort operates in one brutally effective loop through the list of `items` by giving every subversive element, that is, those smaller than their predecessors, a one-way ticket to gulags. After this loop, the elements that remain in the list will be in sorted order! For example, applied to the list `[3, 1, 2, 4]`, Stalin sort would remove the elements 1 and 2 (after removing the element 1, the next element 2 is compared to its surviving predecessor 3), leaving in the sorted items `[3, 4]`.

Stalin sort can be turned into an inefficient but working sorting algorithm by allowing the subversive elements to return after having learned their lesson in the gulag. After each purge, the subversive elements are returned in front of the list in order they were sent away. For example, the previous list would become `[1, 2, 3, 4]` after the first round of purges. The modified Stalin sort algorithm is then applied to this new list as many times as required to achieve the sorted harmony.

Instead of returning the sorted list, which you could do simply by calling the list function `sort`, the automated tester expects this function to prove actually having done this work by returning the number of rounds required to arrange the `items` to sorted order. However, you should be aware of that wretched wrecker Shlemiel who will try to make each purge to operate in quadratic time and hide the inner loop inside some seeming innocent list `remove` operation. Proper praxis will steamroll through the `items` in linear time with a single loop.

`items`	Expected result
`[1, 2, 3, 4]`	`1`
`[2, 1, 0, -1]`	`4`
`[4, 3, 1, 1, 0, -1, 2, -3]`	`6`
`list(range(10000, 0, -1))`	`10000`

It should be clear and rational to everyone who doesn't belong to the oppressor class that this algorithm is guaranteed to terminate after a finite number of rounds. Don't be afraid to place your finger on every statement and ask why it be so, comrade.

'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 an exact `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 so the survival probability is zero, and when the coordinates are inside the board and $k = 0$ so the survival probability is one. Otherwise, the survival probability is the sum of the eight survival probabilities $S(n, x', y', k - 1)$, multiplied by the transition probability 1/8 to make these probabilities add up to one, where x' and y' go through all the possible squares reachable by 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. However, very few companies would probably interview candidates using problems that involve probabilities and recursion.

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 at least 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 Honsberger, 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.

Largest square of ones

```
def largest_ones_square(board):
```

Here is another classic job interview chestnut that has a cute solution either with recursion aided by the `lru_cache` memoization magic, or with **dynamic programming** to directly fill in the memoization table. Your function receives a `board` of *n*-by-*n* elements, each row of the board given as a string consisting of characters `'0'` and `'1'`. This function should find the largest square made ones on the `board`, and return the side length of this square.

Once again, some "Shlemiel" would solve this with four levels of nested loops to iterate through all possible top left corners and areas of that square. We can solve this problem more efficiently by defining a recursive function *S*(*row*, *col*) that gives the side length of the largest square whose bottom left corner is at the given *row* and *col*umn. The base case is *S*(*row*, *col*) = 0 whenever that position contains a `'0'` or is out of bounds of the `board`. For positions that contain a `'1'`, *S*(*row*, *col*) is computed with the formula 1 + min(S(*row* – 1, *col*), *S*(*row* – 1, *col* – 1), *S*(*row*, *col* – 1)) that depends on the subproblems solutions of its three neighbours to the west, northwest and north. DUCY?

`board`	Expected result

`['0000',`
` '1111',`
` '0111',`
` '1111']`
	3

`['00111',`
` '11010',`
` '11001',`
` '00110',`
` '10110']`
	2

`['0010000',`
` '1101111',`
` '1111111',`
` '0111111',`
` '0111111',`
` '0111111',`
` '1011110']`
	4

[Be]t[Te]r [C][Al]l [Sm][Al]l

```
def break_bad(word, symbols):
```

The title card of the television series *Breaking Bad* famously stylized the show title with chemical element symbols boxed inside words. In this problem, you will write a function that performs similar transformation for the given `word` made of lowercase English letters, boxing the chemical element symbols between square brackets. You don't need to hardcode the chemical `symbols` in your function code, since they are given to your function as the second argument.

Since the vast majority of English words cannot be broken down perfectly into a sequence of chemical element symbols, your function should produce a breakdown that minimizes the number of letters that are not part of a chemical symbol. If two or more optimal solutions exist for the given `word`, your function must return the solution that, after the initial common prefix of those solutions, uses a chemical symbol first, preferring a two-character symbol over a one-character symbol.

This problem can be solved recursively in style of the script `morse.py` given among the example programs of this course. For the given `word`, you can always leave its first character as it is, or possibly box its one- or two-character prefix to a matching chemical symbol. For each of these possibilities, calculate recursively the best way to break down the rest of the word into symbols, and combine the parts into the optimal solution, favouring the two-character prefix in case of a tie. Since the same subproblems are repeated along many paths during recursion, use the `@lru_cache` function decorator to ensure that each of the `len(word)` subproblems will be evaluated only once.

word	Expected result
`'no'`	`'[No]'`
`'overgrows'`	`'[O][V][Er]gr[O][W][S]'`
`'multinode'`	`'m[U]l[Ti][No]de'`
`'reprised'`	`'[Re][Pr][I][Se]d'`
`'diadokokinesis'`	`'d[I]ad[O][K][O][K][In][Es][I][S]'`
`'oxysalicylic'`	`'[O]x[Y][S][Al][I][C][Y][Li][C]'`
`'felina'`	`'[Fe][Li][Na]'`

Forbidden digit

```
def forbidden_digit(n, d):
```

Suppose that all nonnegative integers that do not contain the digit `d` are listed in increasing order. For example, if `d` equals 3, this list would start as 0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, ...

This function should return the integer located in the position `n` of this list, counting of positions starting from zero. Of course, the automated tester will again give your function large enough values of `n` so that constructing this list explicitly is a futile effort from the start. Instead, you need to use a bit of combinatorial thinking to find out how to get to the result sooner. After the necessary flash of insight to how to achieve this, the solution does not actually require any higher mathematics.

`n`	`d`	Expected result
`3`	`3`	`4`
`5`	`0`	`6`
`21`	`3`	`24`
`7085`	`4`	`10752`
`29030`	`7`	`43835`
`10**100`	`5`	`67681661646610430136498140330603313423346621090724909070333048938910217300061727422168112863470663877921 1`

Card row game

```
def card_row_game(cards):
```

In this classic recursive programming chestnut, two players face each other in a game played with a row of `cards` turned face up. Alternating turns, each player can pick up either the card currently at the beginning of the row, or the card currently at the end of the row. Each card contains an integer, and both players try to maximize the sum of integers in the cards that they picked up. The result of the game is the difference between the sums of the cards of the first and the second player, a negative result thus indicating that the second player won. Assuming that both players make their decisions optimally throughout the game, what is the result for the given `cards`?

The greedy strategy of always picking up the larger of the two cards does not generally produce the optimal play in this game. This problem should be modelled with a recursive **minimax** equation C(*b*, *e*) where *b* and *e* are the positions in `cards` where the current row begins and ends. The base case is *b* = *e* when only one card remains, so the result of the game is the value of the card in that position. Otherwise, C(*b*, *e*) equals the maximum of the two quantities `cards[b]` − C(*b* + 1, *e*) and `cards[e]` − C(*b*, *e* − 1), respectively giving the value of picking up the card from the beginning or the end of the current row. Note how in both cases, the result of the recursive call is subtracted from the value of the card picked up, since that call evaluates the new situation from the opponent's point of view. In a zero-sum game, your opponent's gain exactly equals your loss and vice versa.

`cards`	Expected result
`[5, 5]`	`0`
`[3, 6, 2]`	`-1`
`[1, 6, 6, 10]`	`9`
`[2, 2, 23, 15, 3]`	`-5`
`[65, 17, 19, 68, 26, 90, 35, 28, 33, 46]`	`119`

After solving this problem, you can analyze its **first-in vigorish**, that is, how much of an advantage there is in getting to make the first move for a randomly chosen board.

Count sublists with odd sums

```
def count_odd_sum_sublists(items):
```

Given a list of nonnegative integer `items`, this function should count the number of contiguous sublists whose sum of elements is an odd number. For example, the list `[0, 3, 2, 4]` contains six such sublists `[0, 3]`, `[0, 3, 2]`, `[0, 3, 2, 4]`, `[3]`, `[3, 2]`, and `[3, 2, 4]` read in order of starting position and breaking ties by length.

Of course, this problem could be solved with two nested for-loops. The outer loop iterates through all the possible starting points of the sublist. The inner loop then iterates through the positions from that starting point all the way to the end of the list, keeping track of the sum of the elements of that sublist so far. Whenever the current sum is an odd number, increment the tally by one.

Short and simple, but we can actually do way better and solve this problem with just one for-loop! Have the inner loop of the previous algorithm loop through the positions of the entire list, keeping track of the sum of the elements up to that position. Maintain two integer variables `odd_count` and `even_count` to remember how many prefixes of the list have had odd and even sums of elements. Depending on the parity of the current element, increment the corresponding counter, and update the total tally of sublists with odd sums by the number of sublists of suitable parity that you have encountered so far.

Of course, the automated tester will feed your function lists long enough so that any "Shlemiel" solving this problem using two nested for-loops will run out of time limit well before terminating.

`items`	Expected result
`[2, 0, 4, 3]`	`4`
`[5, 0, 1, 2, 2]`	`8`
`[2, 1, 8, 0, 5, 6, 5, 0]`	`20`
`[9, 0, 5, 2, 8, 3, 3, 1, 9, 8, 3]`	`35`
`[22, 1, 12, 8, 5, 3, 5, 17, 8, 17, 19, 3, 1, 15, 14, 18, 8, 18, 0, 11, 18, 10]`	`132`

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.

Out where the buses don't run

```
def bus_travel(schedule, goal):
```

Cities in Poldavia have been numbered starting from zero. As a happy-go-lucky backpacker, you want to make your way to the `goal` city from the zero city by travelling on the bus network of that pastoral country. You have a `schedule` that gives the list of buses leaving from each city that day. Each leg of the bus travel is given as a triple `(destination, leave, arrive)` giving the times that the bus takes off from the current city and arrives at its `destination`. For simplicity, we assume that transferring to another bus takes zero time. Being a vaguely Eastern European fictional country, Poldavia uses a 24-hour clock `(hour, minute)` to measure the day.

This function should determine the earliest time that you can arrive at the `goal` according to the bus schedule. To achieve this with a simplified version of **Dijkstra's algorithm**, your function should keep track of the earliest possible arrival time to each city. For the starting city 0, this time is the starting midnight. For the rest of the cities, you will update their earliest arrival times whenever you find faster routes to get to these cities. If it's not possible to reach the `goal` city at all, return the midnight hour `(24, 0)` to indicate this.

The function should maintain a **frontier** of cities that need processing. Initially this frontier contains only the starting city 0. Then, repeatedly pop out one city from the frontier, doesn't even matter which one. Loop through the buses taking off from your chosen city after the earliest possible arrival time to that city. If the arrival time of some route to its `destination` is earlier than your current earliest known arrival time to that destination, update the earliest known arrival time of that destination, and insert the destination city to the frontier. Repeat this until the frontier becomes empty. (Also, since buses can't travel backward in time, you can ignore all buses whose arrival time at their destination comes after the currently known earliest arrival time at the `goal`.)

`schedule`	`goal`	Expected result

`{0: [(1, (15, 54), (17, 27)), (2, (17, 46), (19, 7)), (4, (18, 1), (19, 17))],`
`1: [(0, (17, 27), (17, 46)), (2, (20, 29), (21, 6)), (3, (15, 14), (16, 47)), (4, (15, 27), (16, 3)), (5, (18, 3), (19, 39))],`
`2: [(1, (19, 7), (20, 29)), (3, (21, 6), (21, 44))],`
`3: [(0, (16, 47), (18, 1)), (1, (13, 53), (15, 14))],`
`4: [(5, (13, 37), (14, 17)), (5, (16, 3), (17, 26)), (5, (19, 17), (20, 36))],`
`5: [(1, (14, 17), (15, 27)), (1, (17, 26), (18, 3)), (3, (12, 46), (13, 53))]}`
	`2`	`(19, 7)`

SMETANA interpreter

```
def smetana_interpreter(program):
```

SMETANA is a wacky esoteric programming language based the notion of **self-modifying code**. This language is not Turing-complete in its basic form, yet gives us an interesting coding problem. Your function should simulate the execution of the `program` given as a list of statements, each statement a string of the form `'GOTO x'` or `'SWAP x y'`, where `x` and `y` are statement numbers in the list, positions indexing from zero. The function should return the number of execution steps needed to terminate. If the `program` execution enters an infinite loop, your function should return `None`.

Execution starts at the statement 0, and ends when the execution goes past the last statement in the list. The effect of `'GOTO x'` is to make the execution jump to the statement currently in position `x`. The effect of `'SWAP x y'` is to swap the statements currently in positions `x` and `y`, and continue the program execution from the next statement after the current position.

To detect the program execution being stuck in an infinite loop, you should simulate the execution of the given `program` in two separate instances of execution that operate in lockstep. The first instance, **the hare**, executes the `program` two steps forward for every step that the second instance, the **tortoise**, executes that same `program`. Should the tortoise and hare ever become equal, you can safely conclude that the `program` execution is now stuck in an infinite loop.

`program`	Expected result
`['GOTO 2', 'SWAP 1 0', 'SWAP 2 1']`	`2`
`['SWAP 1 2', 'GOTO 0', 'GOTO 3', 'GOTO 1']`	`None`
`['SWAP 2 0', 'SWAP 0 3', 'SWAP 1 2', 'GOTO 3']`	`4`
`['SWAP 2 3', 'SWAP 1 3', 'GOTO 4', 'SWAP 4 0', 'GOTO 0', 'SWAP 4 1', 'SWAP 0 2']`	`None`
`['GOTO 5', 'GOTO 2', 'GOTO 8', 'SWAP 3 6', 'SWAP 6 5', 'GOTO 3', 'SWAP 0 7', 'SWAP 1 7', 'SWAP 1 4']`	`18`
`['SWAP 10 8', 'SWAP 1 4', 'GOTO 1', 'GOTO 7', 'GOTO 10', 'GOTO 7', 'GOTO 8', 'SWAP 6 2', 'SWAP 9 4', 'SWAP 8 1', 'SWAP 5 6']`	`5`

Optimal blackjack play

```
def optimal_blackjack(deck):
```

One of James Swain's thrillers about the Las Vegas casino security expert Tony Valentine mentioned a device to calculate the optimal blackjack strategy over all possible distributions of the remaining cards, for an ultimate edge in card counting. In this problem you will implement such device for a simplified version of blackjack over the known `deck` of cards. Since suits don't matter in blackjack, cards are given simply as their blackjack numerical values, with tens and faces all given as 10, and aces given as 11. Aces can be used either as 11 ("soft") or 1 ("hard") in either hand.

Each deal gives the first two cards to player and the next two cards to the dealer. There are no immediate blackjack bonuses, but an ace and a face just make an ordinary 21. Knowing the contents of the `deck`, the player should take as many cards as needed to optimize not just the immediate outcome of the current deal, but the total outcome of all deals from the given `deck`. (Our simplified blackjack does not allow splitting, doubling down or surrendering.) Once the player has chosen to stand and has not gone bust, the dealer must keep taking cards until he has 17 or higher, soft or hard, regardless of the player's current total. The winner pays one dollar to the loser, with nothing paid in a push. Deals continue as long as the `deck` has at least four cards. If the `deck` runs out of cards in the middle of a deal, cards are taken from the beginning of the `deck` in a cyclic fashion.

The function should return the best possible score for the player assuming optimal overall play in style of Frank Fontaine over the entire `deck`. Since the eye in the sky is not here to stop the action after suspicious plays, it may sometimes even be optimal to go intentionally bust in the current hand to get to play the remaining deals from a more favourable position of the `deck`.

`deck`	Expected result
`[8, 10, 2, 11, 6, 7, 5, 6, 10, 8, 4, 7, 3, 8, 6, 10, 10, 4, 10, 10]`	`0`
`[10, 6, 6, 10, 9, 9, 2, 6, 9, 2, 5, 7, 4, 2, 10, 10, 11, 3, 2, 3, 10, 5, 8]`	`3`
`[9, 9, 9, 10, 4, 5, 10, 11, 6, 10, 10, 8, 10, 10, 9, 7, 3, 10, 10, 8, 2, 2, 9, 7, 8, 6, 5, 6, 3, 9, 10, 10, 11]`	`-2`

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.

Blocking pawns

```
def blocking_pawns(n, queens):
```

Some chess `queens` have been randomly placed on the generalized *n*-by-*n* chessboard. To maintain peace and harmony among these queen bees, we need to place some neutral pawns on the board so that any two queens on the same row, column or diagonal are separated by at least one pawn placed between them. This function should compute and return the minimum number of pawns required to achieve this separation.

As usual in combinatorial problems of this nature, we need to have the serenity to accept the things we cannot change, courage to change the things we can, and the wisdom to accept the difference. Your function should start by finding all attacking pairs of queens. Since you have no choice but to place one pawn between them, you can try out each such pawn position to see what happens when you try to recursively solve the remaining problem with the same approach. Ordering the attacking pairs and the squares between each pair appropriately will speed up this search.

n	queens	Expected result
8	`[(1, 0), (5, 6), (0, 3), (1, 7), (5, 1)]`	2
9	`[(5, 0), (3, 8), (0, 5), (7, 5), (3, 5), (0, 0), (8, 2)]`	7
10	`[(7, 7), (9, 1), (3, 1), (1, 5), (4, 4), (8, 4), (2, 8)]`	3
13	`[(9, 8), (5, 6), (7, 11), (3, 10), (11, 0), (9, 4), (0, 8), (6, 0), (5, 3), (11, 11)]`	8

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 *n* queens.

To make this problem with rooks worth our 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 should be pretty straightforward at this stage. 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 merely 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 too busy to have time for long words, are a famous formalism typically used to generate fractal strings converted into various 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 zero 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'`

As can be seen in the last row of the above table, the Grand Slam problem “Infinite Fibonacci Word” in the original Python problems collection is a trivial special case of this function.