Additional Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of November 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, after they have completed the first four weeks with practice exercises of CodingBat Python. Same way as in the original problem collection, these problems are listed roughly in their order of increasing difficulty. Complexity of the solutions for these additional problems ranges from straightforward loops up to rather convoluted branching recursive backtracking searches that allow all kinds of clever optimizations to prune the branches of the search.

The rules for solving and testing 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`, the same as you wrote 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):
```

Whenever 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 late chess grandmaster and overall colourful character Bobby Fischer generalized chess into a more exciting variation of “Chess 960” that plays otherwise the same except that the home rank pieces are randomly permuted before each match begins, which should 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. The function can assume that that `row` is a string that contains the eight correct chess pieces in their right multiplicities.

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

Discrete math enjoyers can later convince themselves that out of the $8! / (2! \times 2! \times 2!) = 5040$ 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	5
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 integers with a higher persistence than eleven are known, nor are believed to exist.

Top of the swops

```
def topswops(cards):
```

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 a superficial veil. This problem deals with topswops, a patience puzzle played with `cards` 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 this fixed state.

For every 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. (Discrete math enjoyers can try proving this guaranteed termination as a side quest.) Your function should count how many moves are needed to reach this fixed state from the given initial permutation of `cards`.

`cards`	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, so that finding even one x for the given y so that $y = f(x)$ is an exponentially arduous task.

Discrete rounding

```
def discrete_rounding(n):
```

Make the given positive integer n your current number. For each k counting down from $n - 1$ to 2, "round up" your current number by replacing it with the smallest positive integer that is exactly divisible by k and is greater or equal to your current number. Return the final number acquired after these replacements.

For example, starting with $n = 5$ and therefore counting down the values of k from 4 down to 2, the smallest positive integer that is divisible by 4 and is greater than equal to 5 would be 8. Next, the smallest positive integer that is divisible by 3 and is greater than equal to 8 would be 9. Last, the smallest positive integer that is divisible by 2 and is greater than equal to 9 would be 10, our final answer. The reader can similarly verify that starting from $n = 12$, the chain of values that this iteration goes through is 12 → 22 → 30 → 36 → 40 → 42 → 42 → 45 → 48 → 48 → 48.

n	Expected result
1	1
2	2
10	34
23	174
10**4	31833630
10**6	318310503562
10**7	31830995532658

Of course this problem makes for a simple basic exercise on Python loops and integer arithmetic for the first programming course, but the problem itself has a curious mathematical property worth looking at. If we denote the result of this process starting from n by $r(n)$, the expression $n^2 / r(n)$ approaches π in the limit as n approaches infinity! For example, when n equals one million, this expression already gets the first four decimal places of π correct, and for n equal to one billion, the first eight decimal places. The reader may enjoy pondering what connection integer arithmetic iterated in this manner could possibly have to circles so that it produces the famous transcendental number as its limit using only integer arithmetic.

Translate

```
def tr(text, ch_from, ch_to):
```

This simple text processing function implements the behaviour of the Unix command line tool `tr`, but of course operates on the given argument `text` as a function instead of reading or writing anything to the standard input and output. This function should return a string that is otherwise the same as the original `text`, except that every occurrence of any character inside the `ch_from` string has been replaced by the corresponding character in the same position in the `ch_to` string. Both strings `ch_from` and `ch_to` are guaranteed to be the same length, and `ch_from` will never contain the same character twice.

`text`	`ch_from`	`ch_to`	Expected result
`'X'`	`'Y'`	`'Z'`	`'X'`
`'jonny'`	`'jony'`	`'AAAA'`	`'AAAAA'`
`'abccba'`	`'bc'`	`'XY'`	`'aXYYXa'`
`'bee'`	`'fgabhed'`	`'jjidflb'`	`'dll'`
`'yeah!'`	`''`	`''`	`'yeah!'`
`'abcde'`	`'abcde'`	`'edcba'`	`'edcba'`

Count palindromic substrings

```
def count_palindromes(text):
```

Given a `text` string that consists of letters `a` and `b` only, count how many substrings of the `text` are **palindromes** whose length is at least three characters. A palindrome is a string that reads the same forward and backward, such as `'babab'` or `'abba'`. If the `text` contains several equal palindromic substrings, each of these substrings should be included separately in the count.

To count the palindromes efficiently without having to loop through a quadratic number of substrings and then test for each such substring whether it is a palindrome, we get a more efficient solution by realizing that every palindrome has a center that is either a single character for a palindrome of an odd length, or two equal characters for a palindrome of an even length. Loop through all positions `i` of the `text` as potential centers of palindromic substrings in the `text`. To count all the palindromes with the center `i`, use a nested while-loop to look at characters `k` positions left and right from the center `i`, with `k` initially set to equal 1. As long as the two characters that surround the center `k` steps away in both directions are equal to each other, increment both the total `count` of palindromes and the value of `k` for the next round of the inner loop. Once you have looked at all possible centers inside the `text` and added up all the palindromes emanating from them, return the total `count` accumulated throughout the function.

`text`	Expected result
`'aa'`	`0`
`'ababa'`	`4`
`'bbbbba'`	`6`
`'aababaabab'`	`10`
`'bbbabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaab'`	`747`
`'a' * 1000`	`498501`

Reasonable filename comparison

```
def tog_comparison(first, second):
```

In his list of Ten Most Wanted Design Bugs in the "Bughouse" collection of usability blunders that have plagued computer users longer than most students reading this have been walking on this Earth, the usability expert Bruce "Tog" Tognazzini includes "Stupid sort" when listing file names in alphabetical order to be helpful for the user. If the order comparison of filenames is performed in per character basis, the files `foo1.txt`, `foo10.txt` and `foo2.txt` end up being listed in this order, since the stupid string comparison used in the sorting algorithm does not realize that integers that consist of consecutive digit characters from `'0'` to `'9'` ought to be compared according to the arithmetic order between integers, instead of the **lexicographic** dictionary order.

Since in this fine summer day up here in the year 2023, this author saw a screenshot of the Eclipse Java IDE that listed the results of running the JUnit test methods in this stupid order, the time has come to eradicate this idiocy once and for all. This function should compare the two given text strings `first` and `second` in this reasonable manner, returning `-1` if the `first` string is lower than `second`, `+1` if the `first` string is higher than `second`, and `0` if both strings are equal.

To keep this simple, the comparison function does not have to consider any minus sign that precedes the digits to be part of the number. What sort of pervert would number their files with negative numbers anyway? (Withdrawn, your honour. Rhetorical.)

`first`	`second`	Expected result
`'foo10.txt'`	`'foo2.txt'`	1
`'foo1.txt'`	`'foo2.txt'`	-1
`'qPX938.doc'`	`'qPX7105.doc'`	-1
`'60359Z30571.dat'`	`'60359Z30571.dat'`	0

List the Langford violations

```
def langford_violations(items):
```

A list of $2n$ elements where each number 1, …, n occurs exactly twice is called a Langford pairing if for every k from 1 to n, there are exactly k elements between the two occurrences of k in the list. For example, `[2, 3, 1, 2, 1, 3]` is a Langford pairing for $n = 3$, as the reader can quickly verify. Langford pairings exist whenever n is a multiple of four, or one less than a multiple of four. All Langford pairings for the given n can in principle be generated with a recursive backtracking algorithm, or a special algorithm can be used to rapidly produce just one such pairing for any given admissible value of n. However, the exact count of Langford pairings for the given n is a famous open problem in combinatorics, these exponentially growing exact counts still unknown from case $n = 31$ onwards.

Instead of generating Langford pairings, this function simply receives a list of `items` guaranteed to contain each number 1, …, n exactly twice, and should return a sorted list of all numbers that violate the Langford requirement in that list of `items`. You should implement this function to operate in a single pass through the `items`, instead of using two nested loops in a sluggish "Shlemiel" fashion. Keep track of which numbers you have already seen through the pass, and whenever you see some number the second time, `append` that number to the result list if the distance between its two occurrences is incorrect. After processing all the `items` in a single pass, return the sorted result list.

`items`	Expected result
`[2, 3, 4, 2, 1, 3, 1, 4]`	`[]`
`[5, 8, 4, 1, 7, 1, 5, 4, 6, 3, 8, 2, 7, 3, 2, 6]`	`[]`
`[5, 8, 1, 1, 10, 2, 11, 6, 2, 9, 8, 4, 6, 7, 4, 10, 7, 3, 11, 9, 5, 3]`	`[1, 4, 5, 6, 7]`
`[14, 4, 1, 1, 2, 13, 15, 2, 11, 10, 12, 5, 10, 6, 9, 14, 8, 5, 7, 13, 6, 11, 4, 12, 9, 8, 7, 3, 15, 3]`	`[1, 3, 4, 10, 11, 15]`
`[3, 4, 1, 15, 3, 16, 4, 2, 16, 13, 12, 20, 5, 8, 14, 7, 10, 2, 5, 9, 18, 6, 1, 13, 19, 17, 11, 10, 6, 14, 12, 17, 19, 20, 8, 15, 7, 9, 11, 18]`	`[1, 2, 7, 8, 9, 12, 15, 16, 17, 19, 20]`

Interested readers can check out the page "Langford's Problem" at Dialectrix, or the page "Langford's Pairing" by Susam Pal, for more information about this classic combinatorial problem.

Ten pins, not six, Dolores

```
def bowling_score(frames):
```

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 list of `frames` encoded as strings. Strikes are encoded as `'X'`, misses as `'-'`, and spares as `'/'` in each frame. For simplicity, we assume that no fouls take place during the game.

Rules for scoring a bowling game 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 the special three-character form of the last frame that contains either a strike or a spare.

frames	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 it 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`. There is a non-trivial chance that you can use this solution to dazzle your future job interviewer.

`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`

It's a game, a reflection

```
def abacaba(n):
```

As illustrated on the Wikipedia page ABACABA, this one of the simplest fractal patterns appears in all sorts of more or less familiar mathematical sequences and patterns. Defined over an alphabet starting from A, the ABACABA string S_k of order k can be defined recursively so that the string S_0 equals just A, and the string S_k is the concatenation of two copies of the string S_{k-1} with the k:th character of the alphabet stuck between those copies to separate them. For example, the string S_1 equals `ABA`, the string S_2 equals `ABACABA`, the string S_3 equals `ABACABADABACABA`, and so on.

Since each S_{k+1} contains the previous S_k as its prefix, it is meaningful to define the infinite ABACABA string as the limit of this process as k approaches infinity. This function should return the character located in the position `n` of this infinite ABACABA string, the position counting starting from zero. It is easy to prove by induction that the string S_k contains exactly $2^k - 1$ characters, so your function should not try to construct this string explicitly, but compute the result with integer computations.

So that the result can be returned meaningfully for arbitrarily large alphabets instead of merely our 26 letters of English the way God intended, this function should return the result as an integer so that the result 0 stands for `A`, 1 stands for `B`, 2 stands for `C`, and so on.

`n`	Expected result
`0`	`0`
`1`	`1`
`2`	`0`
`3`	`2`
`2**42 - 1`	`42`
`2**42 - 1 + 2**41`	`41`
`2**41 - 1 - 2**41`	`41`
`10**100`	`0`

Mian–Chowla sequence

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

The Mian–Chowla sequence is another interesting sequence of increasing positive integers defined with a simple rule from which a chaotic complexity automagically 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 simply 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

Stern–Brocot path

```
def stern_brocot(x):
```

The Stern–Brocot tree is an infinite binary tree that contains every positive rational number exactly once somewhere in its nodes, and always in its simplest reduced form. Construction of this tree starts with the lower and upper limits that do not exist in the tree. The lower limit is initially set to 0/1 that represents the zero, and the upper limit is set to 1/0 that represents the positive infinity.

The root node of the Stern–Brocot tree contains the mediant of the two initial limits. Since computing this mediant gives the numerator 0 + 1 = 1 and the denominator 1 + 0 = 1, the root node contains the rational number 1/1, the unity from which this entire panoply emerges. To compute the value stored in the left child of the given node that holds the value *x*, replace the current upper limit with *x* and keep the lower limit as is. Symmetrically for the right child, replace the current lower limit with *x* and keep the upper limit as is. The infinite subtree constructed to hang from that node then contains all rational numbers between that lower and upper limit.

Instead of constructing the entire tree in level order up to some particular depth, this function should return the list of values that appear in nodes on the linear path from the root node to the node that contains the positive rational number `x`. The **binary search algorithm** to achieve this is explained in the section "Mediants and binary search" on the Stern–Brocot tree Wikipedia page.

`x`	Expected result
`Fraction(1, 1)`	`[Fraction(1, 1]`
`Fraction(4, 5)`	`[Fraction(1, 1), Fraction(1, 2), Fraction(2, 3), Fraction(3, 4), Fraction(4, 5)]`
`Fraction(4, 3)`	`[Fraction(1, 1), Fraction(2, 1), Fraction(3, 2), Fraction(4, 3)]`
`Fraction(35, 32)`	`[Fraction(1, 1), Fraction(2, 1), Fraction(3, 2), Fraction(4, 3), Fraction(5, 4), Fraction(6, 5), Fraction(7, 6), Fraction(8, 7), Fraction(9, 8), Fraction(10, 9), Fraction(11, 10), Fraction(12, 11), Fraction(23, 21), Fraction(35, 32)]`

Mū tōrere boom-de-ay

```
def mu_torere_moves(board, player):
```

Mū tōrere is a traditional Māori board game with a surprising complexity of play strategy veiled behind its deceptively simple rules. Same as with the Oware move problem in the original collection, the minimax algorithm to play this game will have to wait until the course CCPS 721 *Artificial Intelligence I*, but we can already implement the **move generator** to list the possible moves for the given `player` in the current `board`. Being computer scientists, we of course also immediately generalize this game from eight points to $2k$ points around the shared center for arbitrary $k > 0$.

The reader should consult the linked Wikipedia page for the setup and rules of this game, or the page "Mu Torere" at Mayhematics for a deeper analysis of game positions. Pieces never capture each other or get removed from the board, but on his turn, a player must move any one of his pieces to a neighbouring empty space. The stalemated player who can't make any legal moves loses the game. So that the game wouldn't end immediately after the first move, a piece can move into the empty center space only if there was at least one opponent's piece next to the piece being moved.

The `board` is given as a string of $2k + 1$ characters, each either `'B'` or `'W'` for the black and white pieces and the `'-'` character denoting the empty space. The first $2k$ characters of the `board` list the pieces around the center, and the last character gives the contents of the center. This function should return the list of possible boards that the player can get to with a single move from the current `board`. To make the result unique, this list should be returned in alphabetically sorted order.

board	player	Expected result
'-BW'	'W'	['WB-']
'-BW'	'B'	['B-W']
'BBWW-'	'B'	['-BWWB', 'B-WWB']
'WBWWBB-'	'W'	['-BWWBBW', 'WB-WBBW', 'WBW-BBW']
'BBBBWW-WW'	'B'	[]
'BBBBWW-WW'	'W'	['BBBBW-WWW', 'BBBBWWW-W', 'BBBBWWWW-']
'BWWBWBWB-'	'B'	['-WWBWBWBB', 'BWW-WBWBB', 'BWWBW-WBB', 'BWWBWBW-B']

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 can be proven to be 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.

Scoring a tournament bridge hand

```
def bridge_score(strain, level, vul, doubled, made):
```

Once the dust has settled after both bidding and play of a bridge hand are over, it's time to add up the score while muttering under your breath about the mistakes that your partner again did. To keep this problem simple, we will look at only successful contracts. Given the `strain`, `level`, `vul`nerability and the `doubled` status of the contract (as one of the possibilities `'-'`, `'X'` and `'XX'`) along with the level of tricks `made`, this function should compute the score for the declarer.

The rules for scoring the successful bridge contract can be found in many places on the Internet, for example in the Wikipedia page "Bridge Scoring". Being gentlemen instead of some kind of animals that wallow in their own filth, the scoring is done according to the rules of tournament bridge, instead of **rubber bridge** better suited for kitchen bridge players.

strain	level	vul	doubled	made	Expected result
'notrump'	1	True	'-'	2	120
'spades'	3	False	'X'	4	650
'diamonds'	3	True	'-'	5	150
'hearts'	4	True	'-'	4	620
'clubs'	6	False	'-'	7	940
'notrump'	7	True	'XX'	7	2980

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

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

Unlike insertion sort and other such non-Lysenkovian **comparison sorting** algorithms devised by the CIA to weaken the spirit of the hardy people of Brutopia, **Stalin sort** is a famous joke sort algorithm that wastes no time with bourgeois considerations of “fairness” or “equality”. Inspired by 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 predecessor, a one-way train ticket to gulag. After this loop, the elements that remain in the list will be in sorted order! For example, given the list `[3, 2, 4, 1]`, Stalin sort would remove the wreckers 1 and 2 (note that 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 the order they were sent away. For example, the previous list becomes `[2, 1, 3, 4]` after the first round of purges. The same algorithm is then applied to this new list as many times as required to achieve and verify the utopia, here reached once comrade 1 has returned from his second re-education journey.

Instead of returning the sorted list, which you could do simply by calling the list function `sort`, the automated tester expects this function not to be alienated from the toils of production but prove having done this work by returning the number of rounds needed to arrange the `items` to sorted order and verify that order. However, you should be aware of that wretched wrecker Shlemiel who wants to sabotage this glorious effort by making each purge operate in quadratic time by hiding the inner loop inside some seemingly innocent list operation such as `remove`. Proper praxis of the glorious revolution will steamroll through the `items` in linear time with a single for-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 item and ask why it be so, comrade!

Boxed away

```
def fewest_boxes(items, weight_limit):
```

Each item in the given list of `items` has a weight that is a unique positive integer, listed in ascending sorted order. You are given an unlimited supply of identical boxes, each of which can hold up to two items, provided that the weight of these items is at most equal to the given `weight_limit` of an individual box. No box is able to hold three or more items, even if those items together happened to fit within the weight limit. Your function should calculate the minimum number of boxes needed to pack away all the `items`.

The bin packing problem, the general version of this problem where each box can hold not just two but any number of items within the weight limit, is known to be NP-hard so that no efficient polynomial time algorithms are believed to even exist for it. Even the special case of partition problem that asks to determine whether two boxes, each box with the weight limit equal to one half of the total weight of the `items`, suffice to exactly pack away all the given `items` is already NP-hard, so any algorithm aiming to solve this problem in the general case is doomed to juggle these `items` between the two boxes through an exponential number of partitions. However, hardcoding the capacity of each box to at most two items should eliminate all this massive branching and allow the optimal solution to be extracted in a far easier time.

`items`	`weight_limit`	Expected result
[41]	42	1
[4, 5, 8]	11	2
[3, 6, 12, 14, 21]	24	3
[12, 23, 29, 42, 53, 54, 56]	58	5
[15, 18, 31, 37, 38, 50, 57]	74	4
[19, 23, 44, 69, 72, 77, 89, 118, 141]	162	5
[18, 20, 51, 73, 118, 159, 172, 213, 233, 235, 282]	308	6

’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. Any move that ends up outside the bounds of the board makes the knight fall 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 **normalize** these probabilities to add up to one, where x' and y' go through all the possible squares reachable by a knight's move from (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. Of course, very few companies would probably interview candidates using problems that involve probabilities and recursion. But if you are lucky enough to hired by such a company, aim to make yourself indispensable there.

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`, and the problem is still perfectly well-defined with a definite answer.)

`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 these rolls, regardless of `d`. It's always nice to have numerical confirmation for intuitively acceptable things that you have only proven 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 will solve this problem more efficiently by defining a recursive function $S(row, col)$ that gives the side length of the largest square whose **bottom right 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'`, the value of $S(row, col)$ is computed with the formula $1 + \min(S(row - 1, col), S(row - 1, col - 1), S(row, col - 1))$ that depends on the solutions of its west, northwest and north neighbours. DUCY?

`board`	Expected result

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

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

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

Digit string partition

```
def digit_partition(digits, n):
```

This function should determine whether it is possible to partition the given `digit` string into pieces so that the sum of these pieces added up as integers equals the goal value `n`. For example, the `digit` string `"123456789"` can be broken into pieces 123 + 4 + 56 + 789 to add up to the goal value 972, and into pieces 12 + 34567 + 89 to add up to the goal value 34668.

This problem can be solved recursively by looping through all possible ways to choose the first number from the beginning of `digits`, and for each such first number, determining recursively whether it is possible to partition the remaining digits to add up to the original goal minus that first chosen number. Alternatively, you can loop through all possible ways to choose the last number in the partition, and try to partition the preceding digits to add up to the remaining goal. However, unlike in most other recursions seen in this collection, subproblems do not tend to repeat themselves through the recursive search, so using the `@lru_cache` would only be a waste of both time and memory. You should also ensure that your function handles the zero digits correctly, regardless of whether these zeros are at the beginning, middle or the end of the digit string.

`digits`	`n`	Expected result
`'00010203000'`	`6`	`True`
`'1000055'`	`100010`	`True`
`'26389'`	`26389`	`True`
`'0453962'`	`1337`	`False`
`'482951'`	`4880`	`True`
`'0024644418'`	`24644417`	`False`
`'625950539'`	`1196`	`True`
`'6013721263902447919922'`	`5940825159181`	`False`
`'241333152384669659'`	`523846937993`	`True`

Hofstadter's figure-figure sequences

`def hofstadter_figure_figure(n):`

In his now classic tome "Gödel, Escher, Bach", Douglas Hofstadter also defined a bunch of wacky and chaotic self-referential sequences that make for interesting coding problems. This problem looks at Hofstadter's figure-figure sequences, two complementary sequences R and S of increasing positive integers. Starting the position numbering from zero like computer scientists made of sterner stuff, the sequences are initialized with $R(0) = 1$ and $S(0) = 2$. After that, $R(n) = R(n - 1) + S(n - 1)$, and $S(n)$ equals the smallest positive integer that has not appeared in either sequence so far. The reader can easily verify that these sequences start out as

R: 1, 3, 7, 12, 18, 26, 35, 45, 56, 69, 83, 98, 114, 131, 150, 170, 191, 213, 236, 260, ...
S: 2, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, ...

with the R-sequence growing approximately quadratically while the S-sequence patiently crawls behind to collect all the numbers that the R-sequence skipped on its merry way.

Given the zero-indexed position n, this function should return a tuple of two values $R(n)$ and $S(n)$. The automated tester will be giving your function values of n from zero up to a hundred thousand in order. Therefore to pass the test within the time limit, you should somehow cache the sequences generated so far, along with whatever other information you choose to store to quickly determine the smallest positive integer that has not yet made its debut appearance.

n	Expected result
0	(1, 2)
6	(35, 10)
30	(602, 38)
49	(1509, 59)
100000	(5028615179, 100432)

[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 the words. In this problem, you will write a function that performs a similar transformation for the given `word` made of lowercase English letters, boxing the chemical element `symbols` between square brackets. Your task is to break the word into chemical symbols to minimize the cost function where each unboxed letter costs two points, each single-letter chemical symbol costs one point, and each two-letter chemical symbol costs zero points. If two different breakdowns of `word` have the same cost, return the one that uses a two-letter symbol earlier than the other solution does.

This problem is best solved with **dynamic programming**. Given the n-letter `word`, start by creating two lists `cost` and `move` so that `cost[i]` gives the count of unbroken letters in the best solution for breaking the substring `word[i:]` into chemical elements, and `move[i]` says what symbol the optimal solution for `word[i:]` starts with. You then fill these two lists from right to left, making the decision for each `move[i]` and its ensuing optimal `cost[i]` based on the later parts of these lists filled in the previous rounds of this loop. After both lists `cost` and `move` have been filled, a simple iteration from left to right can construct the optimal solution.

`word`	Expected result (using the real chemical elements)
`'no'`	`'[No]'`
`'overgrows'`	`'[O][V][Er]gr[O][W][S]'`
`'multinode'`	`'m[U]l[Ti][No]de'`
`'reprised'`	`'[Re][Pr][I][Se]d'`
`'oxysalicylic'`	`'[O]x[Y][S][Al][I][C][Y][Li][C]'`
`'felina'`	`'[Fe][Li][Na]'`

Total covered area

```
def squares_total_area(points):
```

You are given a list of `points` located in the nonnegative quadrant of the infinite two-dimensional integer grid. Your task is to process these points in order so that using each point on its turn as the **top right corner**, you construct the **largest possible axis-aligned square** that does not reach across the *x*- or *y*-axis into the negative coordinate values and does not share a nonzero area with any the previously created squares. This function should add up and return the total area together covered by the squares constructed in this manner.

For example, the reader equipped with a pen and some grid paper can quickly verify that processing the `points` given as `[(6, 2), (5, 3), (9, 4)]` would first create a 2-by-2 square whose top right corner is at `(6, 2)`. Then, a 1-by-1 square is added with its top right corner at `(5, 3)`. Finally, a 3-by-3 square is added with its top right corner at `(9, 4)`. Adding up the areas of these squares gives the total area of 4 + 1 + 9 = 14. Note how the order in which the `points` are listed can sometimes make a big difference; had these points been given as `[(9, 4), (6, 2), (5, 3)]`, the resulting squares would have added up to the total area of 16 + 0 + 9 = 25 instead.

The automated tester will give your function enough test cases with the point *x*- and *y*-coordinates large enough so that any brute force solution that loops through the integer grid points inside each square to diligently add up these points one by one is guaranteed to run out of time before reaching the finish line of the test. The correct and efficient logic for this problem uses only integer arithmetic and comparisons to swiftly cut the squares under construction down to their final sizes, and would not much care even if these point coordinates were hovering in the octillions.

`points`	Expected result
`[(5, 3)]`	9
`[(3, 3), (5, 5), (7, 3)]`	17
`[(4, 5), (3, 4), (9, 6)]`	41
`[(2, 2), (5, 5), (6, 2)]`	17
`[(0, 8), (6, 0), (0, 10), (2, 9)]`	4
`[(10, 2), (9, 8), (5, 11), (2, 16)]`	53
`[(3, 20), (26, 6), (11, 24), (14, 33), (21, 2)]`	190
`[(11, 81), (80, 99), (76, 32), (97, 144), (159, 14), (145, 28), (124, 30), (24, 157)]`	8467

Balsam for the code

```
def measure_balsam(flasks, goal):
```

You are a medieval balsam merchant equipped with a tuple of `flasks` of their capacities in ounces, listed in descending sorted order. The largest flask is initially full of balsam, and the smaller flasks are empty. Your task is to measure exactly `goal` ounces of balsam into any one of these flasks in the smallest possible number of moves. Since these flasks are unmarked except for their capacities so that you can't measure the desired pour amount just on eyesight and feel, each move must keep pouring balsam from some nonempty flask into some other flask until either the flask that you pour from becomes empty, or the flask that you pour into becomes full. This function should return the smallest number of moves to measure exactly `goal` ounces into some flask, or `None` if this measurement is not possible to achieve in any number of moves with the given `flasks`.

This problem is perhaps easiest solved using **breadth-first search**. Initially, the list of reachable states contains only the initial starting state. As long as none of your reachable states contains a flask with exactly `goal` ounces of balsam in it, loop through these reachable states to produce the list of **successor states** that can be reached from the previous row of states in one pouring move. You should also maintain the set of all states that you have already seen at any time during the search, and ignore such previously seen states should they happen to be generated again, to prevent your search from running around in a circle like some mangy dog chasing its own tail. Eventually one of the two things must happen; either a desired goal state appears in the reachable states, or the list of current states becomes empty, which allows you to conclude that the desired `goal` was impossible to measure using the given `flasks` to begin with.

`flasks`	`goal`	Expected result
`(8, 5, 1)`	`7`	`1`
`(10, 7, 2)`	`9`	`7`
`(5, 3, 3)`	`1`	`None`
`(12, 8, 3, 1)`	`10`	`3`
`(15, 11, 8, 6, 5, 4)`	`9`	`1`
`(27, 22, 21, 20, 20, 10, 9, 9)`	`23`	`5`

The original version of this problem asks three medieval balsam thieves to divide the stolen flask full of balsam equally using three given empty flasks of different sizes so that each thief gets to scurry off in a different direction with one of these flasks with his equal share of the loot. The reader can ponder the difference between this version of the problem and our current version.

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, the 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	67681661646610430136498140330603313423346621090724909 0703330489389102173000617274221681128634706638779211

Shotgun sequence

```
def shotgun(n):
```

Usually programmers like to number sequence positions starting from zero, but starting the numbering from one works out better for the wacky Shotgun sequence problem. Consider the infinite sequence of positive integers 1, 2, 3, 4, 5, 6, 7, 8, 9, ... Pair up numbers in positions divisible by two with each other and swap each number with its pair for the new sequence 1, 4, 3, 2, 5, 8, 7, 6, 9, ... Next, look at the numbers in positions that are divisible by three, and similarly pair them up and swap each number with its pair. In general, in the k:th round, look at the numbers currently in positions divisible by $k - 1$, and pair up and swap those numbers. After $n - 1$ rounds, the number that ended up in the position n will never move again, and this function should return that number.

Denoting the number in position n after k rounds of this operation by $s(n, k)$, the base case of the recursive formula is simply $s(n, 0) = n$ for any n. If n is not divisible by k, then $s(n, k) = s(n, k - 1)$, again simple enough. If n is divisible by k, find the position m with which the position n is swapped in that round (this can be easily determined from parity of n / k), and solve $s(m, k - 1)$ to find out what number was previously in position m. However, the automated tester will give you values of n large enough to make the resulting deep recursion hit a **stack overflow**, so you need to convert your **tail recursion** into a for-loop that will be executed entirely in the same function activation.

n	Expected result
1	1
4	6
1912	2385
3348444	7610786

This "deranged" permutation of positive integers is a bizarre combination of the famous Hilbert Hotel and the Sieve of Eratosthenes in that these repeated swaps keep pushing all prime numbers infinitely far right along the number line, so that this function can never return a prime number for any finite n. Consider any prime number p. Since p is not divisible by any integer $1 < k < p$, it will move the first time in the round $p - 1$ when it will be swapped with the integer currently in the position $2p$. The integer in that position must be either the original $2p$ or some integer that was moved there in some previous round $k < p$, and thus necessarily composite. The prime p will then remain in its new position $2p$ until the round $2p - 1$, when it will be swapped with the integer currently in the position $4p$, another composite number, and so on.

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 entire game, what is the result for the given `cards` for this **principal variation** of perfect play?

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 evaluated 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, so the result for one player is the negation of the result for the other one.

`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 constructed 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.

As usual, the automated tester will feed your function lists long enough so that any "Shlemiel" solving this problem with two nested for-loops will run out of the 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

Tailfins and hamburgers

```
def trip_plan(motels, daily_drive):
```

Your vacation plan for the summer consists of driving through the nostalgic highway that crosses the entire country, staying in some motel each night to catch some Z's before the next day's drive across the fruited plain. You are given the list of `motels` stretched along the highway in order of ascending distance, and start your trip at the milestone zero of the highway. (Note that this zero milestone is not the same thing as the first motel in position zero of the `motels` list.) Staying at any other motel is optional during the trip, but your trip must end at "The Gobbler", always the last motel in the given list, a historical landmark from a bygone era of happy motoring optimism.

You are able to drive any distance during each day, and can make your vacation take as many days as you like. However, you would prefer to have your daily drive to be as close to `daily_drive` as possible. We define the **misery** of each day to be the square of the difference between the actual distance and the daily_drive. For example, should you prefer to drive 300 miles each day, the misery for a day where you drove only 250 miles equals $(300 - 250)^2 = 2500$. This function should generate the trip plan that minimizes your total misery over the entire trip.

This trip plan should be returned as a list of the locations of the motels where you choose to stay each night, in ascending order. If there exist multiple travel plans whose total misery is equal, this function must return the lexicographically lowest such plan.

`motels`	`daily_drive`	Expected result
`[7]`	5	`[7]`
`[3, 16, 32, 43]`	22	`[16, 43]`
`[16, 27, 49, 59, 72]`	32	`[27, 72]`
`[13, 23, 28, 55, 71, 82]`	38	`[28, 55, 82]`
`[13, 18, 32, 44, 81, 88, 114, 144]`	49	`[44, 88, 144]`
`[21, 41, 74, 98, 130, 171, 183, 198, 216, 267]`	93	`[98, 183, 267]`

Tower of cubes

```
def cube_tower(cubes):
```

The six faces of a cube are numbered from 0 to 5 so that the face opposite to the face number `i` is given by the expression `(i+3)%6`. Your function is given a list of `cubes` where the sides of each individual cube have been coloured with some colours encoded as natural numbers. These `cubes` are of all different sizes, and are listed in order of increasing size. Your task is to build a tower with the largest possible number of cubes under the rules that a larger cube may not be placed on top of a smaller one, and that the touching faces of adjacent cubes must always be of the same colour. Each cube can be freely rotated to an arbitrary orientation during the build. Return the largest number of cubes that the tower can contain under these constraints.

This problem is probably easiest to solve with the classic "take it or leave it" recursive search. Either you take the largest remaining cube as the base of the tower, or you don't. Either way, continue to build the best tower using the remaining smaller cubes, making sure that the colour of the face of your next chosen cube always matches the face of the previously chosen cube that it rests on. Since the same subproblems can and will repeat exponentially during this recursive search, you should again use the `@lru_cache` decorator to prune the repeated branches of this search tree.

`cubes`	Expected result
`[[0, 2, 2, 1, 2, 4], [3, 1, 2, 0, 3, 2], [4, 1, 4, 4, 1, 2]]`	3
`[[2, 9, 1, 4, 8, 6], [1, 7, 6, 2, 3, 2], [0, 5, 5, 2, 6, 2], [2, 9, 8, 3, 6, 8]]`	3
`[[7, 21, 14, 13, 19, 9], [14, 16, 22, 22, 20, 7], [2, 3, 0, 4, 2, 18], [2, 19, 5, 12, 21, 8], [0, 12, 13, 11, 4, 15], [15, 2, 5, 3, 16, 7], [15, 16, 1, 2, 3, 12], [10, 13, 22, 3, 10, 15], [1, 20, 11, 6, 2, 12], [15, 9, 1, 14, 10, 1], [4, 14, 9, 17, 7, 15], [7, 22, 6, 4, 16, 19], [7, 21, 19, 18, 10, 18], [4, 18, 4, 22, 2, 5], [13, 10, 0, 20, 17, 21], [1, 10, 15, 10, 1, 3], [15, 2, 4, 6, 0, 17], [21, 21, 11, 16, 0, 18], [5, 1, 2, 14, 17, 8], [16, 19, 3, 21, 15, 12]]`	10

This problem is adapted from the problem 10051 at Online Judge, "Tower of Cubes", included in the Programming Challenges compilation by Steven Skiena of "The Algorithm Design Manual" fame.

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 solved with the classic **two pointers** approach, using two indices `lo` and `hi` 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. Then update `s` to give the sum of squares of the new range, and continue 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.

Kimberling's expulsion sequence

```
def kimberling_expulsion(start, end):
```

Kimberling's expulsion sequence is one of the numerous wacky sequences of positive integers devised by Clark Kimberling. Assume that positions of positive integers 1, 2, 3, 4, 5, 6, 7, 8 ... are indexed starting from zero. To produce the *i*:th element of the result sequence, remove the element from position *i*, and rearrange the first 2*i* elements of the remaining sequence by **riffling** (same as in "Riffle shuffle kerfuffle", the problem 3 in the original *109 Python Problems* collection) the *i* elements before and the *i* elements after the position *i* together with an **in-shuffle**, that is, the first element of the riffle is taken from the block that follows the position *i*.

For example, the element in the position 0 of the result is 1, leaving the sequence 2, 3, 4, 5, 6, 7, 8, ... of remaining elements. The next element of the result is 3, the element in position 1 of the remaining elements, and riffling its surrounding elements produces the new sequence 4, 2, 5, 6, 7, 8, ... of remaining elements. The next element of the result is 5, the element in position 2 of the remaining elements, and riffling its surrounding elements produces the new sequence 6, 4, 7, 2, 8, ... of remaining elements. And so on.

This function should return the subsequence of the Kimberling's exclusion sequence from the position `start` (inclusive) up to position `end` (exclusive). The automated tester is guaranteed to give you test cases so that start is 0 in the first test case, and from every test case onwards, `start` will always equal the `end` of the previous test case. Your function should therefore store the remaining sequence information constructed so far in some global variables between the calls, so that it doesn't have to start producing each new subsequence all the way from the beginning.

`start`	`end`	Expected result
0	2	`[1, 3]`
2	5	`[5, 4, 10]`
12	14	`[31, 14]`
710	725	`[1525, 2086, 1849, 1471, 1958, 836, 1035, 351, 648, 1263, 299, 579, 1531, 1019, 1682]`

It is still unknown whether this sequence contains every positive integer at some point.

Kimberling's repetition-resistant sequence

```
def repetition_resistant(n):
```

Right on the heels of the previous problem, here is another self-referential sequence devised by Clark Kimberling whose chaotic behaviour emerges organically from the iteration of a simple deterministic rule. This binary sequence consists of zeros and ones, and starts with a zero.

To determine the next bit of the sequence σ constructed so far, consider its **longest repeated subsequence** that occurs in the sequence at least twice. For example, the longest repeated subsequence of `01100111` would be `011`, of length three. Partially overlapping repetitions count, so the longest repeated subsequence of `010101` would be `0101`. Let m be the length of the longest repeated subsequence of σ. To decide whether the sequence σ will continue with `0` or `1`, look at the sequences σ`0` and σ`1`, acquired by extending σ by either `0` or `1`, respectively. The sequence is extended by `1` if and only if the sequence σ`0` contains a repeated subsequence of length $m + 1$, but the sequence σ`1` does not contain such a subsequence. Reader can now verify that this infinite sequence begins with

`010001101011100100111101100000101000011101001011011111000100000000110`…

This function should return the character in the position `n` of this sequence, the position numbering starting from zero. To speed up your function, note that adding a new character to the end of the sequence constructed so far can create a new longest repeated subsequence of length $m + 1$ only if that subsequence lies at the very end of the new sequence. The automated tester will be giving your function values of n from 0 to 9999, so your function should save the sequence generated so far and the length of its longest repeated into global variables stored between the function calls.

`n`	Expected result
`0`	`0`
`1`	`1`
`1000`	`0`
`10000`	`0`

According to Kimberling, it was proven back in the year 2003 that this infinite sequence contains every possible finite bit string somewhere in it. (Therefore it actually contains every bit string not just once but infinitely many times; DUCY?) The reader can also compare this sequence to the similarly spirited Linus sequence, problem 52 in this author's collection of Java problems for CCPS 209.

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 Slavic 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 on the notion of **self-modifying code**. This language is not Turing-complete in its basic form, but is still powerful enough to give 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, position indexing again starting from zero. The function should return the number of execution steps needed to terminate the program. 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 execution states of tortoise and hare ever become equal, you can safely conclude that the `program` execution is 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`

Casinos hate this Toronto man!

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

One of James Swain's airport potboilers featuring 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 a 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 the 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 using the known `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 full tilt Frank Fontaine mode 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`

How's my coding? Call 1-800-3284778

```
def keypad_words(number, words):
```

The E.161 recommendation maps the letters from A to Z to touchtone keypad digits from 2 to 9 so that phone numbers that can be difficult for humans to memorize can be given catchy phonetic mnemonics such as 1-800-DOCTORB. Since each digit can denote three or four different letters, the same series of digits could represent many different words and combinations of words. We would like to simplify the task of finding a catchy mnemonic with a function that generates the list of all combinations of words that map to the `number` that was randomly given to us by Ma Bell, so that we humans could leisurely scan this list and choose the snappiest mnemonic for that `number`.

To keep the size of the returned lists manageable, the list of `words` will only contain words whose length is between three and seven letters, inclusive. There are still 97,106 such words in the file `words-sorted.txt`, so you need to somehow speed up the search for words that match the given sequence of digits. For improved readability, you should use minus signs to separate the individual words in the mnemonic. To make the results unique to allow automated testing, this function should return the list of mnemonics in sorted order.

`number`	Expected result (using the wordlist `words-sorted.txt`)
`'4444444'`	`['ghi-gigi', 'ghi-high', 'gig-gigi', 'gig-high', 'gigi-ghi', 'gigi-gig', 'gigi-ihi', 'gigi-iii', 'high-ghi', 'high-gig', 'high-ihi', 'high-iii', 'ihi-gigi', 'ihi-high', 'iii-gigi', 'iii-high']`
`'2244638'`	`['aah-gneu', 'ach-gneu', 'bag-gneu', 'bagh-met', 'bagh-meu', 'bagh-mev', 'bagh-net', 'bagh-oft', 'bah-gneu', 'bai-gneu', 'baignet', 'bch-gneu', 'cag-gneu', 'cai-gneu']`
`'8572539'`	`['tlr-akey', 'tlr-alew', 'tlr-alex', 'tlr-blew', 'tlr-clew']`
`'8634633'`	`['tod-gode', 'tod-goff', 'tod-hoed', 'tod-inde', 'tod-iode', 'toe-gode', 'toe-goff', 'toe-hoed', 'toe-inde', 'toe-iode', 'ume-gode', 'ume-goff', 'ume-hoed', 'ume-inde', 'ume-iode', 'undined', 'unfined', 'vod-gode', 'vod-goff', 'vod-hoed', 'vod-inde', 'vod-iode', 'voe-gode', 'voe-goff', 'voe-hoed', 'voe-inde', 'voe-iode']`

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 pawn nearest to the queen in 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)]`	`False`
`(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 must also 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 located 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 blissful separation.

As is 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 ought to 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. Each move must take place in one of the four compass directions, so diagonal moves are not allowed.

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 unvisited neighbour square.

`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'`)

The round number round

```
def des_chiffres(board, goal):
```

One segment of the popular French television game show "Des chiffres et des lettres" gives the contestant a `board` of n positive integers, not necessarily distinct, and a positive `goal` integer that the contestant must get to appear on the `board` with a suitably clever sequence of moves.

Each move consists of the contestant removing any two numbers a and b of his choice from the `board` and replacing them with any one of the values $a + b$, $a - b$, $a \times b$ or a / b, provided that the resulting value is also a positive integer. This function should compute how many moves are necessary to produce the given `goal` on the given `board`. If this is not possible within the limit of $n - 1$ moves after which there is only one number on the `board`, this function should return `None`.

This problem can be solved by looping through all possible pairs of elements a and b on the `board`, and for each pair, trying out the result of the four arithmetic operations one at the time to determine recursively whether the `goal` can be reached from the new `board` given by this move. The dizzying number of possible choices at each level of recursion sprouts into an exponentially large tree of possibilities, so perhaps the reader can devise a pair of conceptual shears to prune down this mess.

Another good idea is to use **iterative deepening** so that the nested recursive search function gets the recursion `limit` as an additional parameter. The `limit` is decremented in every recursive call, and the case of `limit` becoming zero is treated as a dead end base case of the search. In the function proper, keep calling your nested recursive search function with increasing values of `limit` until a sufficiently loose `limit` allows the search to get to the `goal`.

`board`	`goal`	Expected result
`[3, 6, 7]`	`32`	`None`
`[7, 9, 12, 12]`	`1`	`1`
`[2, 4, 4, 12]`	`88`	`3`
`[1, 3, 5, 14, 14]`	`77`	`4`
`[6, 7, 10, 13, 15]`	`282`	`4`
`[4, 4, 6, 8, 8, 9]`	`594`	`4`
`[3, 4, 4, 9, 12, 16, 17, 20]`	`208`	`2`

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 doesn't even know how 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 the pairs of rooks must be 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 should be pretty straightforward for anyone who has been reading this material this far. However, no algorithms to generate all possible Costas arrays of size *n* have been discovered, at least more efficiently than the trivial **brute force backtracking** through all *n*! permutations and cherry picking the permutations whose direction vectors between all pairs of rooks are unique.

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 simply 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 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 109 Python problems collection is a trivial special case of this function.