109 Additional Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of July 14, 2024

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

After the first couple of dozen problems suitable for the first or second course on Python programming, the remaining problems would be more suited for more advanced programming and algorithm courses for computer science majors during the sophomore, junior and senior years of their undergraduate degree. These problems would also make for excellent preparation material for a technical coding interview, the problems in the last fifth or so even for the FAANG level.

Same as in the original problem collection, these problems are listed roughly in their order of increasing difficulty. The complexity of the solutions for these additional problems ranges from straightforward loops up to convoluted branching recursive backtracking searches that require all sorts of clever optimizations to prune the branches of the search sufficiently to keep the total running time of the test suite within the twenty second time limit.

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 that must be named `labs109.py`, the exact same way as you wrote your solutions to the original 109 problems. The automated test suite to check that your solutions are correct, along with all associated files, is available in the GitHub repository `ikokkari/PythonProblems`.

Python coders of all levels will hopefully find something interesting in this collection. The author believes that there is room on Earth for all races to live, prosper and get strong at coding while having a good time by solving these problems. Same as with the original 109 problems, if some problem doesn't spark joy to you, please don't get stuck with that problem, but just skip it and move on to the next problem that interests you. Life is too short to be wasted on solving stupid problems.

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 generally 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. Furthermore, 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. Your 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` (king is not between rooks)
`'rkbQKbkr'`	`True`
`'rrkkKbQb'`	`False` (king is not between rooks, bishops same parity)
`'rbbkKQkr'`	`True`
`'brbQKkkr'`	`False` (bishops same parity)
`'rKrQkkbb'`	`True`
`'kbQbkrKr'`	`False` (bishops same parity)
`'bQrKkrkb'`	`True`

Discrete math enjoyers can convince themselves that exactly 960 out of the 8! / (2! × 2! × 2!) = 5040 possible permutations of these chess pieces satisfy the required 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 again try proving this guaranteed termination as a side quest.) Your function should compute and return the number of moves 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 generate a permutation for the given *n* that requires the largest possible number of moves to terminate. Exact solutions for this 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.

Lowest common dominator

```
def lowest_common_dominator(beta, gamma):
```

The **prefix sum sequence** of the given integer sequence is defined so that its each element equals the sum of the elements of the original sequence up to and including that position. For example, as the reader can easily verify the prefix sum sequence of the sequence `[3, -1, 0, 5, 2]` would be `[3, 2, 2, 7, 9]`.

An integer sequence α **dominates** another integer sequence β of the same length if every element of the prefix sequence of α is greater than equal than the corresponding element of the prefix sequence of β. It is easy to see that the concept of domination defined in this manner is a **partial order** that is **reflexive** (every sequence dominates itself) and **transitive** (if α dominates β and β dominates γ, then α dominates γ). However, note that it is not necessary for every element of α to be greater than equal to the corresponding element of β for the sequence α to dominate β; for a trivial example, the sequence `[10, -2, -3]` dominates the sequence `[-10, 10, 4]`.

One can further prove that for any two sequences β and γ, there exists the unique **lowest common dominator** sequence α that dominates both β and γ, so that any other sequence that dominates β and γ will also dominate α. (If you want to get all technical about it, the domination relation defines a **lattice** where any two elements have a unique **supremum** and **infimum**.) This function should return the lowest common dominator of the sequences `beta` and `gamma`.

`beta`	`gamma`	Expected result
`[5]`	`[7]`	`[7]`
`[1, -2]`	`[0, 0]`	`[1, 0]`
`[5, 5, -5]`	`[2, 5, 0]`	`[5, 10, 7]`
`[0, 3, -9, -12]`	`[-8, -4, 13, -13]`	`[0, 3, 1, -12]`
`[15, 12, -6, 5, 11]`	`[5, -1, -14, -17, 20]`	`[15, 27, 21, 26, 37]`

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 than 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 or equal to 5 would be 8. Next, the smallest positive integer that is divisible by 3 and is greater than or equal to 8 would be 9. Last, the smallest positive integer that is divisible by 2 and is greater than or 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 well worth taking a closer look 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. Readers 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 and divisibility.

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 Python 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'`

Ifs and butts

`count_cigarettes(n, k):`

As we can see from expressions such as “to treat someone like a dog”, the past was not only a different country, but generally a poor, filthy and unhygienic place to live in. This is illustrated by the classic puzzle from “the obvious immediate answer is actually wrong” genre, about a nicotine-addicted bum who can make himself a complete cigarette from the remains of five cigarette butts. Quickly now; if the bum has a pack of 25 cigarettes to begin with, how many cigarettes does he get to smoke? The answer is not the obvious 30 but actually 31, since the five cigarettes created from the butts of the original twenty-five will themselves produce five butts to create one more cigarette.

This function should compute the total number of cigarettes produced by *n* original cigarettes, assuming that *k* cigarette butts are needed to create a new cigarette.

n	k	Expected result
25	5	31
9	3	13
219	3	328
126291199661946087897236120463864135117068196536406426448 78	47	129036660524162307199349949169600311967439244287197870502 01

Word positions

```
def word_positions(sentence, word):
```

Here is a simple problem taken from the exercise set of the famous MIT Python course 6.0001. Your function is given a `sentence` made up from some words separated by single whitespace characters, and one particular `word` to look for. This function should return the list of all **word positions** in the `sentence` where that particular `word` appears. Note that this function does not ask for the character positions in the string, but operates at the granularity of individual words.

For simplicity, the sentence does not contain any punctuation marks, but only the words and the single whitespaces that separate these words. Capitalization counts, so that the lowercase and uppercase versions of the word count as separate words.

This problem is a pretty straightforward exercise on Python loops and text strings, but aspiring Pythonistas can take as a challenge to implement this function as a **one-liner** using the Python **list comprehensions** along with some tactically chosen Python string methods.

`sentence`	`word`	Expected result
`'Stix nix hix pix'`	`'hix'`	`[2]`
`'Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo'`	`'buffalo'`	`[1, 3, 4, 5, 7]`
`'James while John had had had had had had had had had had had a better effect on the teacher'`	`'had'`	`[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]`
`'That that is is that that is not is not is that it it is'`	`'that'`	`[1, 4, 5, 11]`
`'That that is is that that is not is not is that it it is'`	`'is'`	`[2, 3, 6, 8, 10, 14]`

Deterministic finite automata

```
def dfa(rules, text):
```

Deterministic finite automata (DFA) are an important text processing tool in computer science. A DFA consists of a set of **states**, for simplicity numbered with integers 0, 1, 2, ..., and **transitions** that determine how the automaton changes its state in response to the stimuli of characters that it reads. Initially, the DFA starts its operation in state 0. It then processes the characters of the given `text` one at the time from left to right. At each character `c`, the automaton changes its internal state to the successor state `rules[(s, c)]`, where `s` denotes the state the automaton is currently in.

This function should return the final state that the automaton with given `rules` ends up being in after processing the given `text`. You can assume that the `rules` dictionary is **total** in that it gives the unique successor state for every possible pair of the current state and `text` character.

`rules`	`text`	Expected result
`{(0, 'a'): 1, (0, 'b'): 0, (1, 'a'): 1, (1, 'b'): 1}`	`'bb'`	0
`{(0, 'a'): 0, (0, 'b'): 1, (0, 'c'): 0, (1, 'a'): 0, (1, 'b'): 2, (1, 'c'): 1, (2, 'a'): 0, (2, 'b'): 2, (2, 'c'): 2}`	`'bca'`	0
`{(0, 'a'): 3, (0, 'b'): 0, (0, 'c'): 3, (0, 'd'): 2, (1, 'a'): 1, (1, 'b'): 3, (1, 'c'): 3, (1, 'd'): 1, (2, 'a'): 4, (2, 'b'): 4, (2, 'c'): 2, (2, 'd'): 3, (3, 'a'): 2, (3, 'b'): 4, (3, 'c'): 1, (3, 'd'): 3, (4, 'a'): 2, (4, 'b'): 0, (4, 'c'): 1, (4, 'd'): 1}`	`'aadbcabba'`	2

Deterministic finite automata are important in theory of computer science because any **regular expression**, an important technique for advanced pattern matching inside text strings, can be converted to an equivalent DFA so that the original regular expression matches the given `text` if and only if the corresponding DFA ends up to one of its **accepting states** after processing through that `text`. Once the regular expression has been converted into an equivalent DFA, that automaton can be used to match that regular expression with a blazing speed in a single linear pass through the `text`, regardless of the complexity of the original regular expression.

The parking lot is full

```
def parking_lot_permutation(preferred_spot):
```

A parking lot contains n parking spots 0 to $n - 1$, arranged in a circle. A total of n cars arrive to the lot one at the time, also numbered from 0 to $n - 1$ in order of their arrival. When the car number i arrives to the lot, it first drives to the spot `preferred_spot[i]`. If that spot is still available, the car will park there. Otherwise, the car will try out the spots after that preferred spot in linear order until it finds some available spot, and parks in the first empty spot that is finds this way. This function should return the list of parking spots that each car ends up in under this discipline.

`preferred_spot`	Expected result
`[0, 0]`	`[0, 1]`
`[2, 0, 1]`	`[1, 2, 0]`
`[3, 2, 2, 2]`	`[2, 3, 1, 0]`
`[2, 4, 4, 0, 1, 4]`	`[3, 4, 0, 5, 1, 2]`
`[4, 7, 6, 7, 4, 6, 0, 6]`	`[3, 5, 6, 7, 0, 4, 2, 1]`
`[2, 5, 6, 0, 0, 3, 5, 6, 2, 5]`	`[3, 4, 0, 5, 8, 1, 2, 6, 7, 9]`

Under this discipline, the result will always be some permutation of numbers from 0 to $n - 1$. Of course, if each car has its own unique parking spot that no other car prefers, the result will simply be the **inverse** of the `preferred_spot` permutation. Other variations of this combinatorial problem give rise to interesting calculations and theorems, such as the fact that exactly $(n + 1)^{n-1}$ different `preferred_spot` lists of length n force no car to cycle back to the beginning of the lot. Interested readers looking for some Goofy time can check out these variations in the survey article "Parking Functions: Choose Your Own Adventure".

Lychrel numbers

```
def lychrel(n, giveup):
```

Starting from the given positive integer *n*, repeatedly add up the number to the number that you get from reversing its digits until the resulting number becomes a **palindrome**, that is, the same number when read forward and backward. This function should return the number of iterations needed to reach a palindrome number from the given initial *n*. If some palindrome number has not been reached until `giveup` iterations have been executed, return `None`.

As documented in the Wikipedia page "Lychrel Numbers", even relatively small starting numbers display great diversity on how quickly this process converges to a palindromic number. For example, it is still an open problem whether this process will ever even terminate starting from *n* = 196.

n	giveup	Expected result
101	100	0
168	100	3
196	10000	None
296	100	7
10911	100	55

Starting from *n* = 196, the process will eventually reach numbers so large that they exceed Python's internal limit for integer-to-string conversion, used by this author to reverse the given integer when implementing the solution. Reversing the digits of an integer using only integer arithmetic without any string conversions would be an interesting side exercise for aspiring Python mavens.

First fit bin packing

```
def first_fit_bin_packing(items, capacity):
```

Bin packing is a famous problem in combinatorial arithmetics. The function is given a list of `items`, each item a positive integer giving the size of that item, and the `capacity` of an individual bin giving the maximum total size of items placed in that bin. The task is to distribute these `items` efficiently into bins so that the total number of bins is minimized.

The general bin packing problem is quite difficult, and even the best algorithms will take an exponential time to find and verify the optimal distribution of `items` to bins. However, many **approximation algorithms** have been devised for this problem. These algorithms are guaranteed to find a near-optimal solution quickly, but can sometimes produce a suboptimal solution that uses more bins than necessary. These approximation algorithms are typically **online algorithms**, meaning that the `items` are considered to arrive one at the time (no peeking ahead allowed!) and the decision of which bin to place the current item must be done immediately and cannot be taken back later, even if some later item turns out to have fit that bin better than the current item.

In this problem you will implement the simple **first-fit algorithm** for online bin packing. This algorithm maintains the list of bins that have been created so far, initially empty. It then loops through the `items` as given, and places each item in the first available bin that still has enough capacity left. If no available bin exists, create a new empty bin to the end of the list, and place the item there. This function should return the list of total bin sizes after executing this algorithm.

`preferred_spot`	`capacity`	Expected result
`[3, 2, 3]`	`7`	`[5, 3]`
`[8, 6, 1, 9, 7]`	`9`	`[9, 6, 9, 7]`
`[9, 10, 6, 1, 4]`	`13`	`[10, 10, 10]`
`[11, 8, 3, 14, 10, 13, 14]`	`17`	`[14, 8, 14, 10, 13, 14]`
`[18, 12, 6, 10, 2, 3, 4, 17, 7]`	`53`	`[51, 28]`

The first-fit algorithm can use up to 1.7 times as many bins as the optimal solution, for a particularly pathological sequence of `items`. Even when looking ahead is allowed, no polynomial time approximation algorithm can guarantee better ratio than 3/2 unless P = NP, in which case the entire problem could be solved exactly in efficient polynomial time.

Count possible triangles

```
def count_triangles(sides):
```

The famous **triangle inequality** says that the sum of the two shorter sides of a triangle has to be at least equal to the longest side for that triangle to be geometrically possible. Given a list of possible lengths of triangle `sides`, this function should count the number of ways to choose three different sides to create a triangle. Since we care only about triangles that are prim and proper and ignore triangles that degenerate into a line segment, we require the sum of the two shorter sides to be **strictly larger** than the longest chosen side length for the integer triple to count as a triangle.

Of course you can solve this problem by iterating through all possible ways to choose three lengths from the `sides` list, either with three nested for-loops or `combinations(items, 3)`. However, a more clever design eliminates swathes of possible side length triples that can't possibly combine into a triangle, resulting for a good speedup of running time.

`sides`	Expected result
`[1, 2, 3]`	`0`
`[1, 2, 4]`	`1`
`[3, 4, 7, 13, 28]`	`0`
`[5, 10, 12, 13, 14, 15]`	`19`
`[7, 14, 20, 25, 29, 33, 36, 39, 41, 42, 43, 44]`	`171`

Arrow walk with me

```
def arrow_walk(board, position):
```

This cute little problem taken from a recent Stack Overflow Code Golf post has surprising depths to think about both in its theory and implementation. You are initially standing at the given `position` of a one-dimensional `board` encoded as a string of angle bracket characters. In each time step, you move one step to the left or to the right depending on the direction of the angle bracket that you are currently standing on. As you step away from a bracket, that bracket flips to point in the opposite direction. This function should return the number of steps needed to get off the board.

Since Python strings are **immutable**, you will definitely want to first convert the `board` to a list whose individual items can be mutated in constant time. Doing so will speed up your function by an order of magnitude compared to the naive implementation that insists on keeping the `board` as a string throughout the execution.

`board`	`position`	Expected result
`'><>>'`	`1`	`5`
`'<<><<<'`	`1`	`2`
`'<>><>><>>'`	`3`	`18`
`'<<<><>>><<>>>'`	`0`	`1`
`'><>>><<>><<<<<>><<<<><>><>>><><>>><<<><<'`	`22`	`400`

Discrete math enthusiasts might enjoy proving that this process can never get stuck in an infinite loop, but must terminate in finite time for any finite `board` and starting `position`.

Count Friday the Thirteenths

```
def count_friday_13s(start, end):
```

The problems in both this and the original 109 Python Problems collection are designed to not assume any knowledge about any special libraries and frameworks. However, dealing with dates and times is an important exception to this general principle. As you can see from the classic document "Falsehoods programmers believe about time", a part of the larger "Awesome Falsehood" curated collection of such lists from various walks of life, anyone trying to roll their own date and time calculation functions is virtually guaranteed to mishandle many unexpected edge case situations.

Instead of trying to reinvent this wheel for no good reason whatsoever, any and all calculations that involve real world dates and times should always be delegated to the appropriate standard library carefully designed for this purpose, such as the `datetime` module in Python. To practice using the `date` and `timedelta` datatypes of this library, this function should calculate how many days from the given `datetime.date` object `start` to the similar `datetime.date` object `end`, inclusive, happen to be Fridays that fall on the thirteenth day of that particular month.

`start`	`end`	Expected result
`datetime.date(2024, 3, 4)`	`datetime.date(2024, 5, 21)`	`0`
`datetime.date(2024, 3, 4)`	`datetime.date(2024, 12, 13)`	`2`
`datetime.date(2022, 1, 28)`	`datetime.date(2026, 12, 14)`	`9`
`datetime.date(1986, 8, 29)`	`datetime.date(2035, 9, 12)`	`84`
`datetime.date(1987, 12, 3)`	`datetime.date(2064, 11, 10)`	`130`

Median filter

```
def median_filter(items, k):
```

A time series of measurement values that contains noise that occasionally produces random outlier values is often run through some kind of **smoothing** process to create a series of values with the same general shape but is less bouncy with the random outlier values brought back closer to the actual trend. Of countless such filters devised, the ***k*-median filter** is best suited for the purposes of this course, since this filter can be implemented solely with element order comparisons without the need for any imprecise floating point arithmetic.

Given a list of `items` that are such noisy measurements, create a result of the same length where each item has been replaced by the **median** value of that item and the $k - 1$ immediately preceding items. The median of the given sequence of values is defined to be the value that would end up at the center position if the list of those values were sorted. So that this definition is meaningful also for the first $k - 1$ items in the list, we pretend that the `items` list has been implicitly prepended with $k - 1$ copies of its first item. To ensure that the median element is unambiguous at each location, the value of the parameter `k` is guaranteed to be odd.

`items`	`k`	Expected result
`[2, 2, 3]`	`3`	`[2, 2, 2]`
`[33, -44, -9, -30, -3]`	`3`	`[33, 33, -9, -30, -9]`
`[-25, -28, -41, -15, 14, 46, 31, 34]`	`3`	`[-25, -25, -28, -28, -15, 14, 31, 34]`
`[62, -20, 49, 1, -59, 65, 102, -98, 41, -45, -40, -94, -63, -48, -61]`	`5`	`[62, 62, 62, 49, 1, 1, 49, 1, 41, 41, -40, -45, -45, -48, -61]`

Man of La Cancha

```
def jai_alai(n, results):
```

Back in the days of Magic City, the hottest action in Miami used to take place at the jai alai frontons, but the last time that this trad Chad sport has shown up in mainstream culture was probably in the opening titles of *Miami Vice*. Even if we can't return to the golden days of tailfins, bowling shirts and rock hard *pelotas*, we can appreciate the clever manner that this game allows multiple teams to simultaneously face each other in an orderly fashion without the game devolving into a chaos.

The game of *jai alai* is played between *n* teams, one or two players per team. Before the game, these teams are randomly numbered from 0 to *n* – 1. Initially, team 0 holds the serve on the court as teams 1 to *n* – 1 wait for their turn in this order in a **queue**, probably best implemented with the `Queue` standard library class. Each round, the first team in the queue enters the court to play against the team currently holding the serve. The team that scores a point stays on the court, the other team goes back to the end of the queue, and the next team from the head of the queue enters the court to play the winner of the previous point.

Given the `results` of each point, this function should return the points that each team has scored so far. In these `results`, the letter `'W'` means that the team that was holding the court scored the point, whereas the letter `'L'` means that the team that came in scored that point.

n	`results`	Expected result
2	`'WLWL'`	`[2, 2]`
2	`'LLLLLL'`	`[3, 3]`
4	`'WWWWWWWWW'`	`[9, 0, 0, 0]`
4	`'LLLLWWWW'`	`[5, 1, 1, 1]`
4	`'LLLLLWLLL'`	`[2, 3, 2, 2]`
5	`'WWLLWWWWWWWLLW'`	`[2, 2, 0, 2, 8]`

Curiously, the above rules make it maximally difficult or a **colluding** cabal of subset of players to secretly choose the winner in style of tournament poker collusion teams or mob-controlled "point shaving" schemes in basketball. Combinatorics appreciators may enjoy the challenge of analyzing how many of the *n* teams need to secretly collude to throw (heh) the points to make one of this cabal the winner with high probability, over all possible random permutations of teams.

Nondeterministic finite automata

`nfa(rules, text):`

The **deterministic finite automata (DFA)** from the earlier problem are a powerful tool for regular expression pattern matching, but their huge downside is the potentially exponentially large number of states relative to the length of that regular expression. To keep the number of states reasonably small while still allowing efficient pattern matching algorithms, we can generalize these automata into **nondeterministic finite automata (NFA)** so that for each possible pair of current state `s` and character `c`, the dictionary item `rules[(s, c)]` is not merely a single state, but a list of possible successor states that the nondeterministic automaton can move to after reading the character `c` while in the current state `s`.

This function should return the list of all possible states that the given NFA can end up in after processing the characters in the given `text`, these possible final states listed in sorted ascending order. Instead of just keeping track of the current state with an integer scalar variable, your function now needs to maintain an entire `set` of possible states that the automaton could be after having processed the current prefix of the `text`, and for each character of the `text`, compute the set of possible successor states from the set of possible current states.

Note also that it is quite possible for some item of `rules[(s, c)]` to be the empty list, which makes it possible for some NFA and `text` inputs that the automaton ends up having an empty set of possible final states.

`rules`	`text`	Expected result
`{(0, 'a'): [1], (0, 'b'): [1], (1, 'a'): [0], (1, 'b'): [1]}`	`'aab'`	`[1]`
`{(0, 'a'): [], (0, 'b'): [], (1, 'a'): [0], (1, 'b'): [1]}`	`'ab'`	`[]`
`{(0, 'a'): [1, 0], (0, 'b'): [], (0, 'c'): [0, 3, 2], (0, 'd'): [2, 0], (1, 'a'): [0], (1, 'b'): [2], (1, 'c'): [0, 3], (1, 'd'): [0], (2, 'a'): [0], (2, 'b'): [], (2, 'c'): [3, 1], (2, 'd'): [3, 0], (3, 'a'): [3, 2], (3, 'b'): [3], (3, 'c'): [], (3, 'd'): [0]}`	`'cacabaa'`	`[0, 1, 2, 3]`

Count unicolour rectangles

```
def count_unicolour_rectangles(grid):
```

The body of a Python for-loop can consist of any statements, including for-loops nested for arbitrary depths to create wheels spinning within wheels. This problem lets you play with a whopping **four** levels of nested for-loops; two for the coordinates of the top left corner of the rectangle, with either one or two more levels of nested loops then inside those outer two.

You are given a two-dimensional `grid` whose rows are strings made up from the alphabet `"abcd"` to represent four different possible colours in each cell. This function should count how many axis-aligned rectangles the `grid` contains so that all four corners of that rectangle have the same colour. That is, count the number of coordinates (x, y) and width-height pairs (w, h) with $w > 0$ and $h > 0$ so that all four cells (x, y), $(x + w, y)$, $(x, y + h)$ and $(x + w, y + h)$ have the same colour.

`grid`	Expected result

`['aba',`
` 'ccc',`
` 'ada']`
	`1`

`['bbbbccba',`
` 'cdacaabc',`
` 'acdcdcdc',`
` 'cddacdab',`
` 'cbaaaabd']`
	`5` (how quickly can you find them all?)

This problem is connected to an interesting combinatorial question of filling an n-by-n rectangle with four colours without creating any unicolour rectangles. For small values n, the problem can be easily solved by filling the grid randomly, and as long as there is a unicolour rectangle, choose one of the corners randomly and flip it to a different random colour. This approach doesn't work for larger values of n, since the solutions get too sparse among the exponentially many possible colourings for this headless chicken to have a snowball's chance of accidentally hitting them, so more finesse is needed. As discussed in the post "The 17×17 Challenge" by Brian Hayes, when $n > 18$, colouring the grid with four colours without creating at least one unicoloured rectangle is known to be impossible. Colourings without unicolour rectangles have been constructed for all n up to 17, but the case of $n = 18$ remains tantalizingly open.

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`
`'bbbabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaab'`	`747`
`'a' * 1000`	`498501`

Longest mirrored substring

```
def str_rts(text):
```

Right on the heels of the previous problem, here is a similar problem about substrings, taken from Jeff Erickson's collection of old homework and exam questions. Your function should find the longest string that occurs both as it is and as its reversed version somewhere inside the `text` of lowercase English characters so that these two occurrences do not overlap with each other. Since this substring is not necessarily unique, this function should return the length of the longest possible substring with this property. Note that as the last example string shows, the first substring may well occur after the halfway point of the `text`.

For the purposes of this lab, a simple **brute force** solution that loops through all possible pairs of starting points of the two strings is not adequate to pass the automated tests, so a faster solution is needed. One possible way to do this starts by first building a dictionary `pos` so that for each character `c`, `pos[c]` is the list of the positions where that character occurs in the `text`. For example, constructed for the text `"helloworld"`, `pos['o'] == [4, 6]`. Once you have constructed this dictionary in a single loop over the `text`, iterating through all possible starting points of the longest substring and its mirror image should be a breeze. Perhaps the reader can think up even faster techniques to solve this problem.

text	Expected result
`''`	0
`'abcd'`	0
`'aaa'`	1
`'uftftuftft'`	3
`'hahahexoxehah'`	5
`'ggotjprjtogzgotjpr'`	4
`'xczxqwcduczxaaxzcudcwqx'`	10
`'djettezzezzettezexeexezczzettezexeevtveexezettez'`	11

Square lamplighter

```
def square_lamps(n, flips):
```

All lamps arranged in an *n*-by-*n* square grid are initially turned off. A series of `flips` is then performed so that each flip changes the state of every lamp of either a single row (that is, every lamp on that row that is on is turned off, and every lamp on that row that is off is turned on) or a single column. The rows and columns of the grid have been numbered from 1 to *n*. Each flip *f* with $f > 0$ applies to the row *f*, whereas a flip with $f < 0$ applies to the column $-f$. This function should compute how many lamps on the grid are on after completing the given sequence of `flips`.

This problem could be solved by separately keeping track of the state of each individual lamp, and then tallying the lit lamps after performing all the `flips`. However, this brute force solution would be horrendously inefficient for large *n* in both time and memory.

To solve this problem in linear time, we first need to realize that the order in which the `flips` are performed is irrelevant for the end state of each individual lamps, and that two flips performed in the same row or same column cancel each other out as if those two flips had never been performed at all. Finally, permuting the order of rows or the order of columns after the `flips` does not affect the final tally of lamps that are lit. With these insights, the invisible light bulb hovering over your head should suddenly turn on and illuminate your way to compute the final tally with integer arithmetic after merely keeping track of the **parity** of flips done for each individual row and column.

n	flips	Expected result
1	`[1, -1]`	0
2	`[1, 2]`	4
3	`[-3, 2, 1, 3, 1, 1]`	6
4	`[-2, 3, -2, 2, -1, 2, -3, 4, -1, 1, 1]`	8
7	`[2, 2, 3, -4, -3, 6, 1, 6, -4, 5, 7, -7, 4, -6, 4, 2, -4, 5, 4, -7, 4]`	25
10000	`list(range(-10000, 0)) + list(range(1, 10001))`	0

Pairwise lamps

```
def lamp_pairs(lamps):
```

A one-dimensional row of `lamps` is represented as a string where `0` denotes a lamp that is turned off, and `1` denotes a lamp that is turned on. In a single move, you can choose any two adjacent lamps that are in the same state (that is, either both lamps are on, or both lamps are off) and flip them together to the opposite state. What is the smallest number of moves after which the entire row of `lamps` is simultaneously turned on?

Since each flips maintains the **parity** of the number of lamps that are off, it is easy to see that for a solution to exist, it is necessary for initial number of lamps that are off to be even. However, as you can see from the test case `'00010'`, this necessary condition is not sufficient, since no sequence of flips can turn on all lamps simultaneously, and the function should return `None`.

To solve this problem, find the leftmost lamp that is off. If its immediate neighbour to the right is also off, flip both these lamps on, and continue with the rest of the row. Otherwise, keep going right until you find the first pair of lamps that are in the same state (again, either both are on or both are off), and use them to make the right neighbour of that leftmost lamp to also be off, so that you can flip both these lamps together on. Implemented properly, this entire function should be essentially one linear loop going through the `lamps` in a single pass from left to right.

`lamps`	Expected result
`'100'`	`1`
`'0110'`	`3`
`'00010'`	`None`
`'011000'`	`4`
`'00110111011111001110010010001011'`	`45`
`'00010000000000001001001101010 11'`	`None`

This problem is “Paired furnaces” in the excellent Kattis problem collection.

Accumulating merge

```
def accumulating_merge(items1, items2):
```

The `itertools` standard library module contains many useful functions for common transformations of sequences and pairs of sequences. However, it does not contain the function for the following artificial variation of **merging** the elements of two sequences `items1` and `items2` into a result that contains each element of both sequences exactly once.

This function should start by creating an initially empty result list, and then iterate through both sequences one element at the time, but not necessarily in lockstep. This function should keep track of the position that it is at each list (both initially at the beginning of the list), and the **accumulated sum** of elements for each list that it has processed so far (both sums initially zero). The function should also remember which list is currently being processed, initially always the first list `items1`.

Each round, the function appends the next element of the list currently being processed to the result, and also adds that same element to the accumulated sum of that list. If processing of the current list reaches the end of that list or the accumulated sum of the current list becomes strictly larger than the accumulated sum of the other list, the function switches to process the elements of the other list, provided that the function has not yet reached the end of that list. Processing the elements one at the time continues in this same manner until the result list contains all elements from both lists.

`items1`	`items2`	Expected result
`[1, 2, 3]`	`[4, 5]`	`[1, 4, 2, 3, 5]`
`[3, 6, 2, 14]`	`[5, 9, 11, 3]`	`[3, 5, 6, 9, 2, 14, 11, 3]`
`[15]`	`[1, 17, 14]`	`[15, 1, 17, 14]`
`[22, 17, 4, 16]`	`[1]`	`[22, 1, 17, 4, 16]`
`[11, 2, 8, 4, 16, 5, 19]`	`[9, 19, 18, 12]`	`[11, 9, 19, 2, 8, 4, 16, 18, 5, 19, 12]`
`[16, 8, 20, 17, 11, 16, 10]`	`[14, 11, 4, 5]`	`[16, 14, 11, 8, 20, 4, 5, 17, 11, 16, 10]`

And they walk in twos or threes or more

```
def twos_and_threes(n):
```

The fact that any positive integer can be represented as a sum of distinct powers of two in exactly one way allows representing integers as **binary numbers**. However, a more curious representation uses distinct terms of the form 2^a3^b where a and b are natural numbers, binary numbers being a trivial special case of this with $b = 0$ in every term. For example, the integer 21 can be expressed as the sum $21 = 12 + 9 = 2^2 3^1 + 2^0 3^2$. Unlike with powers of two, this representation allows the terms 2^a3^b to be chosen so that no chosen term is divisible by any other chosen term! The **constructive** proof discovered by Paul Erdős himself gives us a recursive algorithm to compute these terms efficiently for any given n.

If $n = 2m$ is an even number, simply construct this sum for m, and then just multiply its each term by two. Since the original terms were not divisible by each other, nothing changes when every such term is multiplied by two. On the other hand, if n is an odd number, find the largest b so that 3^b is at most equal to n. Then construct the sum for $(n - 3^b) / 2$, half of the remainder, and multiply the terms of this latter sum by two. These terms for the remainder plus the original 3^b is then the desired sum whose terms add up to n. Since the term 3^b clearly does not contain the prime factor two, whereas all the remaining terms contain that factor and their exponent for three must be lower than b, it is again easy to verify that none of these terms can divide each other.

This function should compute the twos-and-threes representation of the given positive integer n, returning the list of terms in descending order from largest to smallest. To make the answers more concise and useful, each individual term 2^a3^b should be given as a two-tuple `(a, b)` of its exponents of two and three, instead of the actual integer that these powers give multiplied together.

n	Expected result
3	`[(0, 1)]`
4	`[(2, 0)]`
19	`[(0, 2), (1, 1), (2, 0)]`
2901	`[(0, 7), (1, 5), (2, 3), (4, 1), (3, 2)]`
771084	`[(2, 11), (11, 2), (5, 6), (8, 4)]`
2578298957581455	`[(0, 32), (1, 30), (2, 29), (4, 25), (5, 24), (13, 19), (28, 2), (15, 17), (26, 3), (25, 5), (24, 6), (17, 15), (23, 9), (20, 12)]`

Ladies and gentlemen, Conway Bitty

```
def conway_coin_race(a, b):
```

Two players each choose a bit sequence, expressed here as two strings `a` and `b`, and the player whose chosen bit sequence appears first in the series of repeated flips of a fair random coin (0 means heads and 1 means tails) wins the race. If both players choose a sequence of equal length, one might expect that both players have the same 1/2 probability of winning, but surprisingly this isn't so. For example, the sequence `'100'` beats the sequence `'000'` with probability 7/8, since the second sequence can win only if it appears at the very beginning of the coin flip series, after which it has no chance of winning, regardless of how the rest of the flips come up. (DUCY?)

An ingenious algorithm to determine the probability of the sequence `a` beating the sequence `b` was devised by John Horton Conway, because of course it was. This algorithm uses a **leading function** *lead*(*x*, *y*) defined between two bit strings *x* and *y*, not necessarily of the same length, so that the result of *lead*(*x*, *y*) is a binary string the same length as *x*. For each position of the result, the bit in that position is on if and only if the **suffix** of the string *x* starting at that position is also a **prefix** of the string *y*. For example, the reader can quickly verify that *lead*(01101, 11010) = 01001. (To help you work with binary numbers, the Python `int` function to convert a string into an integer takes an optional base parameter, so that for examples, `int('10010', 2)` equals 18.)

Odds of the sequence `a` winning over sequence `b` can now be expressed in the **odds form**

$$lead(b, b) - lead(b, a) : lead(a, a) - lead(a, b)$$

which is then converted to a `Fraction` of the probability that the sequence `a` wins the race. For example, the odds of 67:29 would become the probability 67 / (67 + 29) = 67 / 96.

a	b	Expected result (as `Fraction`)
`'1'`	`'0'`	`1/2`
`'00'`	`'10'`	`1/4`
`'100'`	`'110'`	`1/3`
`'1101111111'`	`'10001101'`	`60/317`
`'01010010111'`	`'1110'`	`7/1024`

As explained in “The Colossal Book of Mathematics” by Martin Gardner, the ordering between sequences of same length is **intransitive**. For any sequence of three or more bits, a better sequence to beat it first places the complement of its second-to-last bit in front of it, then discards the last bit.

Lowest fraction between two fractions

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

Given two `Fraction` objects `first < second`, this function should compute a third `Fraction` object `result` strictly between the other two so that `first < result < second`, and that among all possible `result` fractions with this property, the returned `result` has the lowest possible **denominator**. (If multiple fractions with the same denominator happen to fit in the wide gap between `first` and `second`, return the one with lowest numerator.)

To make the brute force search for the result fractions more efficient, let us denote these three fractions by $a/b < e/f < c/d$. Multiplying each term by the so far unknown denominator f of the `result` (note that f is already known to be nonzero), this is equivalent to $(a\ f)/b < e < (c\ f)/d$. It therefore suffices to try out increasing values of f starting from one until you get to the value of f that is large enough to make the whole number parts of the two fractions $(a\ f)/b$ and $(c\ f)/d$ different. The numerator of the `result` then equals the whole number part of the first fraction $(a\ f)/b$ plus one, while the denominator equals the f that you ended up at.

`first`	`second`	Expected result (as `Fraction`)
`Fraction(7, 3)`	`Fraction(10, 3)`	`3`
`Fraction(49, 100)`	`Fraction(51, 100)`	`1/2`
`Fraction(11, 8)`	`Fraction(13, 8)`	`3/2`
`Fraction(11, 9)`	`Fraction(196, 153)`	`5/4`
`Fraction(127, 48)`	`Fraction(53345, 20112)`	`45/17`
`Fraction(5, 3)`	`Fraction(97, 57)`	`17/18`

The idea for this problem comes from the collection "Mathematical Diamonds" by the late great Ross Honsberger.

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" of listing file names in alphabetical order in a mechanistic manner. Since in this fine summer day up here in the year 2023 writing this, this author saw a screenshot of the Eclipse Java IDE that listed the results of running the JUnit test methods in the stupid sort order, the time has come to put a stop to this idiocy once and for all.

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 of their characters. Comparisons between non-digit characters are done according to the **Unicode code points** of these characters, the way that the Python string comparison operators < et al. automatically already do for individual characters.

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. Also to keep this problem 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 perverts would number their files with negative numbers anyway?

first	second	Expected result
`'foo10.txt'`	`'foo2.txt'`	`1`
`'foo1.txt'`	`'foo2.txt'`	`-1`
`'bar42qux99joe.txt'`	`'bar42qux100moe.txt'`	`-1`
`'qPX938.doc'`	`'qPX7105.doc'`	`-1`
`'60359Z30571.dat'`	`'60359Z30571.dat'`	`0`

List the Langford violations

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

A list of 2*n* 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 also 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`

Add like an Egyptian

```
def greedy_egyptian(f):
```

As explained on the Wikipedia page "Egyptian Fractions", ancient Egyptians expressed integer fractions as sums of **unit fractions** (that is, fractions of the form $1/n$ with $n > 0$) so that no denominator was ever repeated. For example, the ordinary fractions $2/3$ and $43/48$ can be expressed as the Egyptian fraction sums $1/2 + 1/6$ and $1/2 + 1/3 + 1/16$, respectively. (The simpler representation of $2/3$ as $1/3 + 1/3$ is not allowed here, as it repeats a denominator.) Every positive integer fraction can be expressed this way as a sum of unit fractions. In fact, any fraction could be expressed as a sum of distinct unit fractions in infinitely many ways, since any particular unit fraction $1/n$ can always be further broken down into two unit fractions $1/(n + 1) + 1/(n(n + 1))$.

Several algorithms to construct a unit fraction representation for the given fraction are known. This function has you implement the greedy algorithm first described and proven to always terminate by Fibonacci himself. Even though this algorithm works for any positive integer fraction, we will restrict our attention to fractions of the form a/b where $a < b$. If $a = 1$, the fraction is already its own unit fraction representation, and the function can add n to the result and terminate. Otherwise compute the smallest positive integer n for which $a/b > 1/n$ (you can do this with integer arithmetic without any loops), add n to the result list, and continue converting the remaining $a/b - 1/n$.

This function should return the list of integer denominators of the greedy unit fraction representation of the given `Fraction` object `f`, these integer denominators listed in ascending order.

`f`	Expected result
`Fraction(1, 2)`	`[2]`
`Fraction(2, 3)`	`[2, 6]`
`Fraction(43, 48)`	`[2, 3, 16]`
`Fraction(9, 19)`	`[3, 8, 66, 5016]`
`Fraction(4, 25)`	`[7, 59, 5163, 53307975]`
`Fraction(53, 71)`	`[2, 5, 22, 977, 1271729, 2425940702433, 11770376583439808963536545]`

As you can see from the last example, the greedy breakdown into unit fractions can sometimes end up quite unbalanced even for relatively simple fractions. Other algorithms can produce more balanced breakdowns, and possibly produce sums with fewer terms.

Sorting by pairwise swaps

```
def pair_swaps(perm):
```

The parameter `perm` is a **permutation** of order *n*, that is, a list that contains each integer from 0 to *n* – 1 exactly once somewhere in it. (Being computer scientists, we start counting from zero, not one.) A single move consists of swapping any two elements of this permutation with each other. What is the minimum number of **pairwise element swaps** necessary to sort this permutation to ascending order so that `perm[i] == i` for every position `i` in the list?

Any permutation can be trivially sorted in at most *n* – 1 pairwise swaps by first swapping the element 0 to its correct location if not already there, followed by swapping the element 1 to its correct location if not already there, and so on. However, a smaller total number of swaps can be realized for some permutations by preferring swaps that move two elements in their final resting positions in a single move, or if no such fortunate double swaps are not available, choose a swap that creates such a double swap for the next move. How can you organize and implement this computation the simplest and most efficient way?

`perm`	Expected result
`[0, 2, 1]`	1
`[2, 0, 1]`	2
`[0, 1, 2, 3, 4]`	0
`[2, 4, 1, 0, 5, 3]`	5
`[2, 5, 0, 3, 1, 4]`	3
`[26, 13, 7, 4, 5, 14, 0, 2, 19, 16, 10, 17, 15, 27, 20, 8, 22, 21, 24, 9, 1, 11, 12, 25, 3, 23, 6, 18]`	21

This problem is from the classic collection "The Moscow Puzzles: 359 Mathematical Recreations".

Van der Corput sequence

```
def van_der_corput(base, n):
```

Van der Corput sequence is an infinite low-discrepancy sequence of integer fractions that fills in the open unit interval (0, 1). Any finite prefix of this sequence can be used as positions of sample points over the unit interval, possibly aided with some small random **jittering**, so that all regions of the interval get sampled fairly so that the sampling does not concentrate on some parts of the interval and leave other parts without attention. The sequence is usually defined using the binary base of two, but can in principle be defined in any `base`. The position numbering of the sequence starts from one, not zero. This function should compute the value in the position `n` in the van der Corput sequence of the given `base`.

To compute the value in position `n`, start by expressing `n` in the given `base`. For example, in the binary van der Corput sequence, the position $n = 6$ would be 110_2 in binary. Reading the individual digits in this representation from lowest to highest, use each digit in position i inside the number as the coefficient of the corresponding negative power $(-1 - i)$ of the `base`. For base two, these negative powers would be 1/2, 1/4, 1/8, 1/16, and so on. The value in position `n` is the sum of these negative powers multiplied by their respective coefficients. For example, the value in position 6 of the binary van der Corput sequence would be 1/4 + 1/8 = 3/8.

`base`	`n`	Expected result
2	1	`Fraction(1, 2)`
3	7	`Fraction(5, 9)`
4	9	`Fraction(3, 8)`
5	17	`Fraction(13, 25)`
42	`10**10`	`Fraction(23346056329, 230539333248)`

Interested readers can check out the blog post "When Random Numbers Are Too Random: Low Discrepancy Sequences" at "The blog at the bottom of the sea" to learn more about the need and application of this and more advanced low discrepancy sequences in fair sampling over one- and higher-dimensional intervals.

Condorcet election

```
def condorcet_election(ballots):
```

In the usual **first-past-the-post election** used in Anglosphere countries, each voter in the district casts a ballot for exactly one candidate, and the candidate who gets the plurality of votes in that district is elected to represent that district. This election method gives rise to tactical voting and other downsides. In a Condorcet election, the ballot of each voter is a **permutation** of all candidates, listing these candidates is the descending **preference ranking** of that voter. For example, if the four candidates are denoted by numbers 0 to 3, a ballot `[2, 3, 0, 1]` means that that voter would prefer the candidate 2 to win, but if that is not to be, he would prefer the candidate 3, and so on.

To compute the winner of such an election, look at each pair of candidates `c1` and `c2` separately, and count the ballots that rank `c1` above `c2`, versus the ballots that rank `c2` above `c1`. The candidate who got more pairwise wins earns one **matchpoint** in the election. The margin of victory is irrelevant here; winning the pairwise race by one ballot gains the same one matchpoint as winning by a million ballots. The candidate with the most matchpoints wins the entire election.

This function should return the winning candidate of the Condorcet election for the given `ballots`. To make these results deterministic to allow for automated testing, every tie must be resolved to favour the lower-numbered candidate, instead of using a fair coin flip.

`ballots`	Expected result
`[[1, 0], [1, 0], [0, 1], [1, 0], [1, 0]]`	1
`[[0, 2, 1], [2, 1, 0], [1, 2, 0]]`	2
`[[1, 2, 0], [1, 2, 0], [0, 2, 1], [0, 1, 2], [0, 1, 2], [2, 0, 1], [2, 0, 1]]`	0
`[[0, 2, 3, 1], [3, 2, 1, 0], [2, 0, 3, 1], [0, 2, 3, 1], [1, 2, 3, 0], [2, 1, 0, 3], [1, 2, 0, 3], [1, 0, 2, 3], [3, 1, 2, 0], [1, 0, 2, 3]]`	1
`[[2, 4, 0, 3, 1], [0, 2, 1, 4, 3], [3, 2, 1, 0, 4], [2, 4, 3, 0, 1], [3, 1, 2, 0, 4], [1, 2, 0, 4, 3], [3, 0, 2, 4, 1], [2, 4, 0, 1, 3], [1, 0, 2, 4, 3], [0, 4, 2, 1, 3], [3, 0, 4, 1, 2], [2, 3, 1, 4, 0], [1, 2, 0, 3, 4]]`	2

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`

Reverse a Fibonacci-like sequence

```
def reversenacci(i, n):
```

A sequence of integers is said to be **Fibonacci-like** if after the given first two elements, each element equals the sum of the previous two elements. Of course, the original Fibonacci sequence starting with (1, 1) is the most famous and important of such a series, but other choices of the first two elements give us an infinity of sequences, such as the Lucas sequence starting with (2, 1).

At this stage, it would be anticlimactic to ask the reader to generate the Fibonacci-like sequence from the given two elements. A more interesting question is to reverse this problem and ask whether there exists some Fibonacci-like sequence that has the value `n` in the position `i`, so that all previous elements are positive and non-descending. (Position counting starts from zero.)

The reader may enjoy thinking up ways to speed up this search compared to brute forcing all possible ways to start the sequence and generating the sequence from each start value up to position `i`.

`i`	`n`	Expected result
`5`	`8`	`True`
`4`	`12`	`False`
`4`	`13`	`True`
`4`	`14`	`True`
`17`	`28048`	`False`
`17`	`28047`	`True`

Recamán sequence

```
def recaman(n):
```

The Recamán sequence is a famous integer sequence where the iteration of a simple rule creates chaotic and unpredictable behaviour. The first element of this sequence at the position $n = 0$ equals zero. After that, the element a_n at the position n equals $a_{n-1} - n$, provided that this difference is positive and the value $a_{n-1} - n$ has not already appeared anywhere in the sequence so far. If this is not the case, the element a_n equals $a_{n-1} + n$, even if this value has already appeared in the sequence.

This function should compute and return the element a_n located at the position n of this sequence. The automated tester will call this function with values from 1 up to one million, by far the largest number of test cases given to any function in this problem collection! Since these test cases are given to your function in strictly increasing order, your function needs to remember only the most recent element a_{n-1} (initially zero), and the set of values that have appeared in the sequence so far.

Since storing a Python set of up to one million elements seems a bit excessive, there is a good programming technique worth knowing that you can apply in this situation. When maintaining a **monotonic** set of positive integers where no element ever gets removed from the set (this is a common scenario in problems where the set is used to remember integers that the function has seen and done), in addition to the set itself you should also maintain a scalar integer variable x that remembers the smallest positive integer that is not a member of the set. The set itself then only needs to remember the elements in the set that are larger than x, since all values lower than x are automatically members of the set. When the threshold element x is itself added to the set, you can keep incrementing x as long as it is a member of the set, and remove the elements that you skip over from the set, to use those memory bytes to store larger elements in the future.

n	Expected result
1	1
10	11
100	164
1000	3686
10000	199508
100000	2057164

The Numberphile video "The Slightly Spooky Recamán Sequence" discusses the curious properties of this seemingly simple sequence.

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. This sequence starts with the element 1 in the first position $n = 0$. 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 forth 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. Same as in the previous Recamán sequence, 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)]`

Carryless multiplication

```
def carryless_multiplication(a, b):
```

The blog post "I Could Carry Less" by Brian Hayes links to the article "Carryless Arithmetic Mod 10" that defines interesting addition and multiplication operations for integers. These operations work otherwise the same as the familiar addition and multiplication, except that addition and multiplication of individual digits is performed **reductively** modulo 10, meaning that any carries over to the next column are simply ignored.

This weird but mathematically perfectly consistent operation gives birth to interesting integer sequences that the curious reader might want to check out after completing this problem. (Especially the section that explains how to properly define the concept of **primality** in this algebra should be highly educational.) However, in this problem you are only asked to implement the carryless multiplication operation between the given two nonnegative integers `a` and `b`.

It is probably useful to first implement a similar function `carryless_addition` to add two given integers in the carryless fashion, if only to get the gist of the general behaviour of this wacky arithmetic. This function will also simplify the logic of the `carryless_multiplication` proper.

a	b	Expected result
12	0	0
2	5	0
7	666	222
643	59	417
123	456	43878
59999888866666	66666228887466	4826468481515148026666862 6

The reader may also compare and contrast this operation to Problem 87 in the original 109 Python Problems collection, "Lunatic multiplication", where instead of ignoring the carry digits, there is never any carry since addition and multiplication were defined as taking the maximum and minimum of the digits being added and multiplied, respectively.

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 2*k* 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 2*k* + 1 characters, each either `'B'` or `'W'` for the black and white pieces and the `'-'` character denoting the empty space. The first 2*k* 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, probably just to annoy you. 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`

Manimix

```
def manimix(expression):
```

The given `expression` string consists of digit characters placed inside properly nested and balanced pairs of parentheses and square brackets. Let us define that a string made of digit characters inside **square brackets** evaluates to the **maximum** of these digits, and a string made of digit characters inside **parentheses** evaluates to the **minimum** of these digits. For example, the string `'[283]'` evaluates to 8, and the string `'(283)'` evaluates to 2. Nested expressions are evaluated inside out, so that `'([28](56))'` evaluates as if it were `'(85)'` that then evaluates to 5. This function should evaluate the given `expression` under these rules. There is no limit how long an `expression` can be and how deep its nesting can reach.

Probably the simplest way to solve this problem is to use a list as an initially empty **stack**, and loop through the characters of the `expression` one at the time from left to right. Any character other than a closing parenthesis or a closing square bracket is simply `append`ed to the stack. Whenever the loop encounters a closing parenthesis or square bracket, it should use an inner loop to `pop` characters from the stack up to and including the corresponding open parenthesis or square bracket, and push the minimum (if parentheses) or the maximum (if square brackets) of the popped digits into the stack. Once the entire expression has been processed in this manner, the stack will contain the final answer as its only element.

`expression`	Expected result
`'(123456)'`	1
`'[(45)(27)]'`	4
`'(88(0(41)))'`	0
`'[5[0[5743]](7(816)8942)8]'`	8
`'[[(872)(824)30(6565)][5379][[11][89]][(92([9170](0295)44)(1[00742])(35391))[6[2507]746][[[1701]9(1557)]5]90]4]'`	9

Count distinct substrings

```
def count_distinct_substrings(text):
```

A nonempty substring can be sliced from the given `text` of length n in $n(n-1)/2$ different ways, for all possible ways to choose the start and end of that slice. There is exactly one substring of length n, namely the `text` itself, two substrings of length $n-1$, three substrings of length $n-2$, and so on, down to the n substrings of length one. If all characters in `text` are distinct, all these substrings will also end up being distinct, but if the `text` contains repeated characters, some of these substrings will correspondingly end up being equal to each other.

This function should compute how many different nonempty substrings the given `text` contains. This problem could be easily solved by looping through all $n(n-1)/2$ possible slices of `text` to add these slices to the result set one by one, and returning the size of that result set as the final answer. However, this approach would be horrendously inefficient whenever the `text` contains plenty of repeated characters and substrings, making most of these substrings equal to each other.

A better solution springs forth from the realization that any substring of length $k-1$ in `text` must necessarily be either a prefix or suffix of some substring of length k. (DUCY?) The function should therefore maintain the set of all substrings of length k, initially containing only the `text` itself for the starting state $k = n$. We can easily compute the set of all substrings of length $k-1$ by looping through each substring `s` in the set of known substrings of length k, and adding both strings `s[1:]` and `s[:-1]` to the set of substrings of length $k-1$ currently being constructed. After completing the set of all substrings of length $k-1$, add the size of this set to the current tally, and go to the next round to compute the set of all substrings of length $k-2$.

`text`	Expected result
`'ab'`	`3`
`'bbbb'`	`4`
`'fegg'`	`9`
`'lfvffffv'`	`32`
`'eokooooooo'`	`33`
`'tbobbooobbbb'`	`61`
`'ab' * 10000`	`39999`

Replacement with perfect hindsight

```
def mmu_lru(n, pages):
```

The topic of page replacement algorithms in computer virtual memory management, the intricacies of which are to be explored further in the later course CCPS 530 *Operating Systems*, can be illustrated with an analogy to give us an interesting coding problem here in the first programming course. You are carrying a binder that has `n` **slots** available, each slot initially empty and able to carry one **page**. You are given a list of `pages` that you need to process in this exact order. Each individual page is identified with a nonnegative integer, and the same pages can appear in `pages` multiple times. On its turn to be processed, the page must first be in a binder. If that page is not already somewhere in the binder, you need to place that page to some slot there, costing you one **page fault**.

If the binder contains unused slots to hold the current page, there is no problem. However, if the binder is full and there are no unused slots available, one of the pages in the binder has to be tossed out to create an empty slot. An interesting problem in memory management is to choose the page to ditch to minimize the expected number of future page faults. Following the general idea of how the best predictor of the future behaviour of a system is its observed past behaviour, a straightforward discipline is to remove the **least recently used** page as the best guess of the page that won't be used in the future, in the style and spirit of Marie Kondo. This function should compute the total number of page faults for the given sequence of `pages` when using this least recently used page discipline, and return that count.

n	pages	Expected result
2	`[3, 1, 4, 4, 3, 3]`	4
2	`[4, 1, 2, 2, 4, 1, 2]`	6
1	`[3, 1, 1, 3, 1, 3, 3, 3]`	5
2	`[4, 0, 8, 4, 0, 8, 8, 4, 8]`	7
5	`[8, 6, 3, 8, 3, 6, 8, 3, 3, 6]`	3
4	`[5, 8, 10, 5, 10, 8, 1, 10, 5, 10]`	4
3	`[1, 12, 4, 4, 4, 11, 4, 1, 1, 8, 1, 4]`	6

Replacement with perfect foresight

```
def mmu_lru(n, pages):
```

After solving the previous problem in the theme of page replacement algorithms in virtual memory management, we can see that making an educated guess of which page to ditch to make room for the current page can be difficult for anybody who is not a descendant of Madame de Thèbes and blessed with the gift of perfect foresight of future events. However, let us pretend for the purposes of the present problem that we possess that perfect foresight, so that we already know all the future `pages` that will be coming in to be processed.

Instead of throwing out the least recently used page to make room for the newcomer, we instead throw out the page whose next use is the **longest time away in the future**. Of course, if some page in the binder will never be needed in the future, that page is automatically the one that gets tossed. This function should compute and return the total number of page faults over the given `pages` under this discipline.

n	pages	Expected result
2	`[3, 1, 4, 4, 3, 3]`	3
2	`[1, 5, 2, 1, 5, 5, 1, 1]`	4
2	`[4, 0, 8, 4, 0, 8, 8, 4, 8]`	5
3	`[1, 12, 4, 4, 4, 11, 4, 1, 1, 8, 1, 4]`	5
4	`[12, 12, 5, 15, 16, 6, 6, 12, 12, 16, 6, 12, 6, 16, 12]`	5

Discrete math enjoyers can later prove that this prescient approach is guaranteed to minimize the total number page faults over the processing of the given list of `pages` in all possible cases. (Take any optimal solution, and show that replacing some other choice with the choice done here can never make that solution worse; in other words, this problem has the **greedy choice property**.)

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 two base cases of the recursion are when the coordinates (*x*, *y*) are outside the board so that the survival probability is zero regardless of the value of *k*, and when the coordinates are inside the board with $k = 0$ so that 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 single knight's move from the square (*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.

Arithmetic skip

```
def arithmetic_skip(n):
```

A person is standing at the origin point zero of the sequence of integers, reaching infinitely far in both positive and negative directions. As his first move, he can take either one step to the left, or one step to the right. As his second move, he can take either two steps to the left, or two steps to the right. In general, as his *k*:th move, he can take either *k* steps to the left, or *k* steps to the right. It can be proven that it is possible to reach any goal number `n` from the origin in such moves. This function should compute the smallest number of moves necessary to reach the number `n`.

This problem can be solved with a **breadth-first** search, but this approach would be far too slow to pass the test case torrent from the automated tester. Instead, you should first define a recursive helper function `can_reach(n, k)` that returns `True` if it is possible to reach the goal *n* from the origin in precisely *k* moves, and returns `False` otherwise. The base case of this recursion is when *k* = 0, in which the answer is the same as testing whether *n* = 0. It is also useful for efficiency to have another base case for when `n` is too large to be reached from the origin in *k* moves, using the well known formula for the **arithmetic sum** of the first *k* positive integers. Otherwise, *n* can be reached from the origin in exactly *k* moves if and only if either one of the possible predecessor states *n* – *k* or *n* + *k* can be reached from the origin in *k* – 1 moves.

After writing this utility function, the actual function should keep calling `can_reach(n, k)` for ever increasing values of `k` until it receives the answer `True`, and return the `k` that achieves this. Since the same recursive subproblems will be reached many times during the entire run of automated test cases, you should use the `@lru_cache` decorator on your `can_reach` function to store these subproblem solutions for future use. To cut the size of this cache in half, note that –*n* can be reached from the origin in *k* moves if and only if +*n* can be so reached.

`n`	Expected result
`0`	`0`
`2`	`3`
`-15`	`5`
`38`	`11`
`215`	`21`
`-999`	`45`

Carving Egyptian fractions

```
def carving_egyptian(f):
```

The earlier problem "Add like an Egyptian" asked you to express a positive integer fraction as a sum of distinct unit fractions using the greedy algorithm to achieve this breakdown. This problem continues with the same task, but with a different method based on the observation that if any fraction a/b that is not a unit fraction can always be trivially expressed as $(a - 1)/b + 1/b$, immediately giving us one unit fraction $1/b$ and leaving a fraction with a smaller numerator to be converted.

The difficulty is keeping the generated unit fractions distinct. This can be achieved with the following technique that maintains the set of unit fractions accepted into the `result` set so far, along with the `buffer` list of unit fractions waiting to come in. Each time that you carve out the new unit fraction $1/b$ in the outer loop, append b to the `buffer`. Then, as long as the `buffer` is nonempty, pop out any one element n from the `buffer`. (The result will not be affected in the choice of element to pop, so you might as well `pop` out the last element of the list.)

If n is not already in the `result` set, add it there. If n is already in the `result` set, remove it from there, and continue depending on the parity of n. If n is even, append $n/2$ to the `buffer`. If n is odd, append two values $(n + 1)/2$ and $n(n + 1)/2$ to the `buffer`. Either way, process the `buffer` in this manner until it becomes empty, and continue carving the remaining fraction $(a - 1)/b$.

`f`	Expected result
`Fraction(1, 2)`	`[2]`
`Fraction(2, 3)`	`[2, 6]`
`Fraction(43, 48)`	`[2, 4, 8, 48]`
`Fraction(9, 19)`	`[3, 15, 19, 48, 4560]`
`Fraction(4, 25)`	`[7, 91, 163, 52975]`
`Fraction(53, 71)`	`[3, 5, 15, 18, 23, 45, 71, 160, 320, 1035, 1278, 102240, 204480]`

As the reader may note by comparing the above example results to those received from the same arguments in the greedy Egyptian fractions problem, the carving method tends to produce more terms, but these terms also tend to be more balanced so that the subdivision doesn't contain ugly thin slivers.

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

Infection affection

```
def infected_cells(infected):
```

On the infinite two-dimensional grid of integers, some cells (*x*, *y*) have become infected with a virus. Once infected, there is no cure, but an infected cell will remain infected forever. The rule to determine the spread of the infection is that as soon as at least two of the four neighbours (in the main compass axis directions) of some cell have become infected, that cell itself also becomes infected. This function should compute the total number of infected cells at the end once the infection has reached all the cells that it can and will infect.

To efficiently determine all the cells that the infection will reach, you should maintain a `set` of all the cells infected so far, and a `list` representing the queue of cells waiting to be examined. Initialize both of these contain the `infected` cells. While the queue is not empty, pop any one cell from there, and look at its four neighbours. For each neighbour that is not yet infected, count the number of its infected neighbours. If this count is at least two, that neighbour becomes infected, so you add that neighbour in both the queue and the infected set. Once the queue has become empty, the infection can spread no further, and you can return the size of the infected set as your final answer.

`infected`	Expected result
`[(0, 0), (1, 1), (1, 3), (3, 2)]`	`16`
`[(-1, 1), (1, -1)]`	`2`
`[(0, 1), (-1, 0), (0, -2)]`	`8`
`[(0, 1), (2, -1), (5, -1), (4, -2), (0, -1)]`	`24`
`[(8, 1), (5, 4), (7, 0), (6, -1), (5, 6), (5, 3), (3, 2), (4, 1)]`	`48`
`[(x, x) for x in range(1000)]`	`1000000`

This particular problem was inspired by a famous discrete math exercise of proving claims with **invariants**. Can you prove rigorously that for all the cells of an *n*-by-*n* box to become infected in this manner, at least *n* cells inside the box must be initially infected? This problem is rather tricky, since any kind of handwaving "But surely at least one cell in each row must initially be infected, *maaan*" is not correct. However, once you get the "Aha!" moment of the lightbulb turning on above your head to see that the total **perimeter** of all infected cells is an invariant of the infection process that cannot increase in any infection step, it immediately follows that to completely infect the *n*-by-*n* box that has a perimeter 4*n*, at least *n* cells of perimeter 4 per each cell must be initially infected.

Prize strings

```
def prize_strings(n, late_limit, absent_limit):
```

The problems offered in Project Euler generally tend to be beyond the reach of undergraduate students. Even if the problem itself would be simple, adding up the results for gazillion possible inputs tends to require mathematical knack beyond what can be expected at this level of your studies. However, some problems are simple enough to allow them to not only be included here, but generalized beyond the original problem setting. Case in point is the problem 191, "Prize Strings".

The attendance record of a student over n days can be expressed as an n-character string consisting of characters `'O'` for "on time", `'A'` for "absent" and `'L'` for "late". For example, the attendance record `'OOALAO'` means that the student was on time the first two days, absent the third, late the fourth, absent again on fifth, and on time the sixth day. The student gets a plaster reward if they are late fewer than `late_limit` days in total, and are never absent for any `absent_limit` consecutive days. This function should calculate how many different attendance record strings allow the student the reward.

This problem can be solved with a nested recursive function, aided by `@lru_magic` to keep the running time from blowing up exponentially. This recursive function needs three parameters to describe the state reached in the present day; the number of days remaining, the late days accrued so far in total, and the days of absenteeism that immediately precede the current day. The function should then compute the number of ways to extend the sequence for each of the three possible ways to spend the current day (on time, late, or absent), and add up the results for these three ways for the answer returned to the previous level of recursion.

n	late_limit	absent_limit	Expected result
4	2	3	43
30	2	3	1918080160
5	4	3	211
20	2	6	10444797
31	1	4	13752712373
59	1	5	221333601789363969

String shuffle

```
def is_string_shuffle(first, second, result):
```

Analogous to shuffling together two halves of a deck of cards, characters of two strings can be shuffled together to create a `result` string that contains all characters of both original strings so that the order of characters coming from each original string remains unchanged. This problem is not quite as simple as it seems when the same characters can appear in both strings. For example, two strings `"hello"` and `"world"` can be shuffled together in exponentially many ways to create many result strings such as `"worldhello"`, `"hewolrldlo"` or `"wohrellldo"`.

This function should determine whether the `first` and `second` string can be shuffled together to produce the given `result` string, and return that answer.

`first`	`second`	`result`	Expected result
`'knickerbocker'`	`'kamala'`	`'kkanacmilareebockkr'`	`False`
`'chandler'`	`'mxuzptlk'`	`'chmandlxuzerptlk'`	`True`
`'laurie'`	`'grayson'`	`'grlaysaurione'`	`True`
`'joy'`	`'travis'`	`'jtoaryvis'`	`False`
`'ouagadougou'`	`'ellie'`	`'ouagadoeulglouie'`	`True`
`'turk'`	`'urban'`	`'urbanturk'`	`True`

Weak Goodstein sequence

```
def goodstein(n, k):
```

Another nice problem from Project Euler suitable for this collection is problem 396, "Weak Goodstein sequence". As you may know, other than the fact that we have ten fingers, there is nothing magical about the base ten that we use to represent integers; analogous to how every species in the universe calls itself "human" in its own language, every base is "base 10" from its own point of view! The Python built-in function `int` allows the caller to supply the base as the second argument that defaults to 10, but this argument has a maximum value that is way too small for the purposes of this problem. You will therefore have to write the base conversion routines back and forth all by yourself using repeated remainder and integer division operations.

The **weak Goodstein sequence** starts from the given value $g_1 = n$ (for simplicity, we index its position starting from one), and continues there with the rule that involves base conversions. To compute the next element g_{i+1} from the current element g_i, first compute the list of digits of g_i in base i. Then interpret these very same digits as a number in the next higher base $i + 1$, and subtract one. For example, as shown in the above Project Euler problem page, the sequence for $n = 6$ starts out as 6, 11, 17, 25, 35, 39, 43, 47, 51, 55, 59, 62, 65…

One might expect these values to keep increasing forever, since of course the exact same digits interpreted in base $i + 1$ represent a larger number than they do in base i. However, the mere act of subtracting one at each step is enough to rein in this growth so that for any starting value n, the weak Goodstein sequence must eventually fall all the way down to zero and remain there! This function should return the value in the one-based position `k` for the Goodstein sequence starting from `n`.

`n`	`k`	Expected result
`1`	`1`	`1`
`3`	`3`	`3`
`5`	`3`	`15`
`16`	`187`	`13328639`
`33`	`642`	`513618635782`
`41`	`504`	`32844064606486`

The rule that defines this wacky sequence is still pretty weak sauce compared to the real deal Goodstein sequence whose termination time can and will reach some pretty dizzying heights.

Markov distance

```
def markov_distance(triple1, triple2):
```

As explained on the Wikipedia page "Markov Numbers", the positive integer solutions to the polynomial equation $x^3 + y^3 + z^3 = 3xyz$ have a curious property that if (x, y, z) is one solution triple that satisfies that equation, then $(x, y, 3xy - z)$ is another solution for that equation. Since this equation is symmetric, any one of the three values of x, y and z can be permuted to the role of the last term, thus giving us up to three more solution triples from any known solution triple (x, y, z). For example, the solution triple (1, 5, 13) gives us three other solutions (1, 2, 5), (1, 13, 34) and (5, 13, 194). Furthermore, these solutions form an infinite tree depicted on the Wikipedia page so that any solution triple can be reached from any other solution triple by traversing up and down this tree.

Your function should compute how many steps away the two given solution triples `triple1` and `triple2` are from each other in this tree. Knowing that the space of solutions forms a tree, we don't need the full power of **breadth-first search** that can find the shortest path in arbitrary state spaces, but can solve this problem merely by climbing up the tree. Start by initializing a step counter to zero. While the two solution triples are not equal to each other, replace the solution triple with the largest maximum value with its parent triple in this tree, and increment the step counter by one. Once the two solution triples become equal, stop and return the final value of the step counter.

`triple1`	`triple2`	Expected result
`(1, 2, 5)`	`(1, 2, 5)`	0
`(13, 34, 1325)`	`(2, 29, 169)`	5
`(13, 1325, 51641)`	`(2, 29, 169)`	6
`(14701, 56399710225, 2487396418774357)`	`(5, 13, 194)`	8
`(985, 4360711162037, 12885900008113189)`	`(4400489, 33809973988413085, 446341255878891162918658)`	14

A very graphy caterpillar

```
def is_caterpillar(edges):
```

An **undirected graph** consists of n nodes numbered from 0 to $n - 1$. Each node is connected to some other nodes with **edges** that are **undirected** so that if the node u is a neighbour of node v, then the node v is also a neighbour of node u. These `edges` are given as a list to your function so that each element `edges[u]` lists the neighbours of node u.

This function should determine if the given graph is a caterpillar tree, a very special type of tree graph that consists of some linear path of nodes from which all the other nodes are one step away. To determine whether the given graph is a caterpillar, you need to first check whether that graph is **connected**, which is easiest to determine by checking that a breadth first search from any node reaches all the nodes of the graph. After checking for connectedness, you can choose any one of the equivalent definitions from the Wikipedia page and implement that check as Python code. The easiest one is probably the one that removes leaf nodes (that is, nodes connected to only one other node) one at the time, and then checks whether the remaining graph is a path graph, the latter check being trivial to do by looking at the number of neighbours of each node.

`edges`	Expected result
`[[1, 2, 3, 4], [0], [0], [0, 5], [0], [3]]`	`True`
`[[2, 5], [2], [0, 1], [5], [], [0, 3, 6], [5]]`	`False`
`[[3], [2], [1, 3, 4, 5], [0, 2, 6, 7], [2], [2], [3], [3]]`	`True`
`[[2, 6, 7], [7], [0, 4, 5], [5], [2], [2, 3, 7], [0], [0, 1, 5]]`	`False`

Most problems on graphs become utterly trivial when restricted to caterpillar trees. One interesting exception is the bandwidth minimization problem that stubbornly remains NP-complete even when restricted to caterpillars.

St. Bitus' Dance

```
def super_tiny_rng(seed, n, bits):
```

Allowing us to handle individual integers stored in computer memory as their smallest elementary particles above the hardware level, the low-level **bitwise arithmetic** and its operators are not often seen in introductory Python courses. However, students who are familiar with the concept and bitwise operators in general might enjoy implementing a function explained in the post "A Super-Tiny Random Number Generator" from The blog at the bottom of the sea.

Every **pseudorandom number generator** maintains an internal **state**, initialized to the given `seed` value. Whenever a new random integer is requested, the generator performs some operation to its current state to produce the new state, and returns some part of the new state as that random quantity. This hardcoded operation applied to the state has been carefully chosen so that even though it is perfectly deterministic so that the same `seed` will always produce the same random numbers, it is highly **chaotic** so that its results behave as if they were produced by a genuine random process shaking and twisting like some poor soul caught in a *danse macabre*.

The super-tiny generator given in the above blog post uses an internal state of 32-bit integer. It updates its internal state `x` each time using the assignment `x += (x*x)|5`, where the `|` operator is the **bitwise inclusive or**, to make the bits scramble like in some mad chaotic dance. The bitwise operation to chop the resulting `x` to 32 bits in a Procrustean manner is left as an exercise for the reader. Since the lowest bits of the state are highly non-random, each update uses only the highest order bit as the random bit, this update needs to be performed `bits` times to produce a result integer consisting of the given number of `bits`. Use the first random bit produced this way as the highest order bit of the generated random number, and the later bits to fill in the lower order bits.

Starting from the given `seed` value, this function should produce a list of `n` random unsigned integers, each one consisting of the given number of `bits`.

`seed`	`n`	`bits`	Expected result
`831769172`	`3`	`8`	`[39, 170, 1]`
`2376066489`	`3`	`12`	`[2138, 3320, 2731]`
`2528054762`	`3`	`31`	`[1425110951, 1066116851, 1565670465, 1613085173, 1921709156]`
`1419340007`	`7`	`64`	`[2372072775304310267, 14757952967543449990, 2275344065262535408, 668817182494655816, 14695084848187285985, 6287545658816032331, 13919832216633589446]`

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`
`'601372126390244791992'`	`5940825159181`	`False`
`'2413331523846696659'`	`523846937993`	`True`

Hofstadter's figure-figure sequences

```
def hofstadter_figure_figure(n):
```

In his now classic tome "Gödel, Escher, Bach", Douglas Hofstadter 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 us 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 skips over 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 the given `word` have the same cost, return the one that uses a two-letter symbol earlier than the other solution.

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]'`

Spiral matrix

```
def spiral_matrix(n, row, column):
```

A **spiral matrix** of order *n* is an *n*-by-*n* grid that contains each integer from 0 to $n^2 - 1$ exactly once, these values arranged in a clockwise spiral starting from the top left corner. For example, the spiral matrix for *n* = 5 would look like this:

```
0     1     2     3     4
15    16    17    18    5
14    23    24    19    6
13    22    21    20    7
12    11    10    9     8
```

This function should return the element located in the given `row` and `column` of the spiral matrix of order *n*, the numbering of both rows and columns starting from 0 at the top left corner. As always, your function can assume that `row` and `column` numbers fall properly inside the matrix.

The technique of writing out the successive values in order to the spiral matrix until you reach the goal position would be very inefficient for large *n*, and cause the automated tester to reject your function for being too slow to solve the torrent of test cases. You need to therefore think up some shortcut to speed up this evaluation. You can start by noticing that snug as a bug in a rug, inside a spiral matrix of order *n* there is a smaller spiral matrix of order *n* – 2 whose starting number is offset by 4(*n* – 1), the number of border elements surrounding this smaller spiral matrix.

n	row	column	Expected result
1	0	0	0
2	1	1	2
4	2	2	14
5	2	2	24
25	18	13	481
10001	5000	5000	100020000
10000001	5000000	5000000	100000020000000

Baker–Norine dollar game

```
def baker_norine_dollar_game(edges, balance):
```

An **undirected graph** consists of n nodes numbered from 0 to $n - 1$. Each node is connected to some other nodes with **edges** that are **undirected** so that if the node u is a neighbour of node v, then the node v is also a neighbour of node u. These `edges` are given as a list to your function so that each element `edges[u]` lists the neighbours of node u.

In the **Baker–Norine variant** of the classic chip-firing game, each node has a coin `balance` that can initially be positive, negative or zero. A node is **ready to fire** if its balance is at least equal to its **degree**, that is, its number of neighbours. A single move consists of choosing any node that is ready to fire, and moving one coin from that node to each one of its neighbours, maintaining the overall total of coins through the entire graph. This function should determine whether there exists some sequence of moves starting from the initial `balance` to make the balance of every node positive.

This problem can be solved with simple graph search in a **state space graph** whose nodes are the coin balances of the original nodes. Your function should maintain the list of states that constitute the current **search frontier**, initially containing only the initial `balance`. Repeatedly `pop` one state out of this frontier (doesn't really matter which one), and loop through all possible nodes that can fire in that state. For each such node ready to fire, compute the resulting successor state. If the balance of every node is positive in this successor state, your function can return `True`. Otherwise, if that successor state has not been previously encountered during the search (use a separate `set` to keep track of this, storing each state in that set as an immutable **tuple**), append that successor state in the search frontier. Should the search frontier become empty once all reachable state configurations have been exhausted without reaching a solution, your function can return `False`.

`edges`	`balance`	Expected result
`[[1, 2], [2, 0, 3], [1, 0], [1]]`	`[7, -1, -1, -1]`	`False`
`[[1, 2], [4, 0], [0, 3], [4, 2], [3, 1]]`	`[6, -1, -1, 3, -2]`	`True`
`[[5, 1, 7, 3, 6], [0, 3, 2, 5], [5, 4, 1, 7, 6], [4, 1, 0, 6, 5], [3, 2, 7], [1, 0, 3, 2], [2, 0, 3], [4, 0, 2]]`	`[7, 9, -1, -1, 10, -1, -1, -1]`	`True`

There is some pretty deep graph-theoretic math in this problem, for anyone interested in such stuff.

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 the top right corner at `(5, 3)`. Finally, a 3-by-3 square is added with the 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 given can sometimes make a big difference; had these points been `[(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 coordinate values 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 for 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 exactly one third of the balsam.

Flip those trips

```
def trip_flip(switches):
```

In front of you lies a finite row of `switches`, each switch an integer with four possible values from zero to three. In a single move, you can choose any three consecutive switches whose values are all different, and flip all three switches together to the fourth possible value. For example, one possible move done to the row `[0, 2, 1, 3, 2, 3]` turns that row into `[0, 2, 0, 0, 0, 3]`.

These flips can be performed in any order, not necessarily strictly left to right. However, no matter how the initial `switches` are positioned and how you choose your moves, you are inevitably destined to reach a dead end after a finite number of flips so that no further moves are possible.

Your function should compute the maximum number of moves that can be performed for the given `switches`. Note that two flips done on disjoint triples of switches are **commutative** in that these flips can be performed in either order without affecting the outcome. Therefore, if your previous flip in the prefix of the flip sequence constructed so far took place at position *i*, the search for the next flip can start at the position *i* – 2 instead of all the way from the beginning of the list.

`switches`	Expected result
`[3, 0, 2, 2, 1]`	1
`[3, 3, 0, 2, 0, 3, 2, 0]`	4
`[3, 0, 1, 0, 1, 2, 0, 1, 0, 2]`	6
`[3, 1, 2, 2, 3, 1, 3, 1, 0, 2, 0, 3, 2, 3, 3, 2]`	13
`[2, 3, 0, 1, 3, 3, 0, 3, 2, 1, 2, 3, 3, 2, 1, 1, 2]`	7
`[1, 2, 0, 2, 2, 3, 0, 3, 0, 0, 2, 0, 3, 0, 2, 1, 0]`	12

This sparkling gem of a problem is adapted from a recent tweet by Joel David Hamkins, whose writings on logic would be highly recommended not just for any computer science undergraduate aiming at graduate studies in theoretical computer science, but anyone looking for serious but accessible material on these topics. The tomes "Lectures on Philosophy of Mathematics" and "Proof and the Art of Mathematics" are basically top notch undergraduate courses on their own. The tweet asked for a proof that this process is guaranteed to eventually terminate. This is easiest by noticing that the leftmost position that ever gets flipped in the sequence will be flipped only once and never again after that, and the result follows by indiction for the rest of the positions and the flip sequence.

Maximal disk placement

```
def place_disks(points, r):
```

This function should place as many non-overlapping disks of radius `r` on the two-dimensional plane as possible, when the list of `points` gives you the possible centers that these disks can be placed on, and return the number of disks placed. We count two **kissing** disks as overlapping, so two disks centered on `(x1, y1)` and `(x2, y2)` overlap if `(x2-x1)**2 + (y2-y1)**2 <= 4*r*r`.

To speed up the recursive search for the optimal placement, you should first preprocess the `points` by creating a dictionary that associates each center point to the list of other points where a disk would overlap the disk placed on that center point. Your recursive backtracking search should maintain the set of points on which placing a disk is still possible without overlapping any disk that has already been placed. Each time you place a disk on some point, eliminate these overlapping disks from future consideration. Be careful not to bring back a point that has been eliminated at multiple levels of recursion until the recursive search has backtracked all the way up to the level where that point was first eliminated.

`points`	`r`	Expected result
`[(4, 10), (0, 5), (6, 11)]`	`2`	2
`[(10, 4), (-4, 0), (4, 13), (0, 16)]`	`3`	3
`[(10, 4), (-4, 0), (4, 13), (0, 16)]`	`3`	3
`[(41, 17), (35, 11), (27, 17), (46, 12), (57, 3), (69, 15), (61, 24), (10, 18)]`	`6`	6
`[(x, y) for x in range(10) for y in range(10)]`	`3`	4

Nice sequence

```
def nice_sequence(items, start):
```

🎶 *Many n, came here as numbers, many n, will pass this way; Many n, will count the factors, as they fill the longest chain; Many n, are smooth and clique-y, many n, are here to stay; Many n, won't see the base case, when it ends the longest chain…* 🎶 – Paul Anka, *The Longest Chain*

This problem idea is taken from "Problems with a Point" by William Gasarch and Clyde Kruskal. Using the positive integer `items` given in sorted order so that each item can be used at most once, create the longest possible sequence so that after the given `start` value, each element is always some integer multiple or a factor of the immediate previous element. Since this sequence might not be unique, the function needs to return the length of that sequence.

This problem can be solved with a recursive function that receives the sequence constructed so far, and tries out all possible ways to extend the sequence from the current last element. It might be useful to preprocess the `items` into a dictionary that maps each individual item to the list of other items that are its multiples or factors, to speed up looping through the local possibilities during the recursion. The reader may also think up other ways to speed up the algorithm to make the tests complete within the time limit.

`items`	`start`	Expected result
`[2, 8, 12, 26, 108, 132, 264]`	`132`	`6`
`[3, 9, 55, 95, 106, 190]`	`106`	`1`
`[4, 5, 11, 17, 22, 24, 52, 88, 93, 120, 264]`	`5`	`9`
`[3, 6, 9, 27, 28, 45, 60, 84, 90, 168, 360]`	`28`	`11`

Posed alternatively, your task is to find the longest non-repeating path through a graph whose nodes are the given integer `items`, and two nodes are connected by an edge if and only if one of the numbers is an integer multiple of the other. Under this definition, these edges are unidirectional so that each edge can be traversed to either direction. The general problem of finding the longest non-repeating path in an undirected graph is **NP-complete** so that no algorithms exist guaranteed to solve it efficiently in every case. Perhaps this restricted form of nodes may (or may not) allow helpful optimizations that are not available for general graphs. Who knows?

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`	`67681661646610430136498140330603313423346621090724909070333048938910217300061727422168112863470663877921 1`

Independent dominating set

```
def independent_dominating_set(edges):
```

An **undirected graph** consists of n nodes numbered from 0 to $n - 1$. Each node is connected to some other nodes with **edges** that are **undirected** so that if the node u is a neighbour of node v, then the node v is also a neighbour of node u. These `edges` are given as a list to your function so that each element `edges[u]` lists the neighbours of node u.

An independent set is any subset of nodes in the graph so that no two nodes of this subset are neighbours of each other. On the other hand, a dominating set of a graph is a subset of nodes so that every node in the graph either is in this subset, or is a neighbour of at least one node in this subset. On their own, both problems are known to be NP-complete so that any algorithm to find the largest possible independent set or the smallest possible dominating set in the given graph will take an exponential time to find and verify the solution for some particularly tricky graph.

However, combining these two problems restricts the possibilities enough that at least for the graphs generated by our trusty automated tester, this function should be able to find the smallest **independent dominating set** fast enough with a recursive backtracking search. For each node u, you have a simple two-way choice. You can choose to take the node u in the set, in which case you are not allowed to take any of the neighbours of u in the set. Or, you can choose to leave the node u out, in which case you must eventually take at least one of the neighbours of u in the set.

It is probably a good idea to first sort the nodes in descending order of their number of neighbours before having the recursion search through the nodes. Since the smallest independent dominating set of the given graph is not necessarily unique, this function should only return the size of that set.

`edges`	Expected result
`[[4, 3, 2], [2, 3], [1, 0], [1, 0], [0]]`	2
`[[5], [2, 5, 3], [1, 3, 5], [1, 2, 5], [], [1, 0, 3, 2]]`	2
`[[2, 5, 6], [6, 2, 4], [5, 4, 1, 0, 3, 6], [4, 2], [3, 2, 5, 1], [4, 0, 2], [2, 1, 0]]`	1
`[[2, 1, 7, 3, 9], [0, 8, 4], [4, 0, 9], [4, 0, 6], [3, 2, 1, 7], [7, 6], [5, 7, 3, 9], [4, 0, 9, 6, 5, 8], [1, 7], [2, 7, 0, 6]]`	3

Vertex cover

```
def vertex cover(edges):
```

An **undirected graph** again consists of n nodes numbered from 0 to n – 1, given the same way as list of `edges` as in the previous problem of finding the minimal independent dominating set. However, this time your function needs to find the minimal vertex cover; that is, a subset of nodes so that for every edge (u, v) of the graph, at least one of the nodes u and v belongs to the chosen subset.

The backtracking recursion to solve this problem is similar to the recursion to find the minimal independent dominating set. It is straightforward to see that for every node u, the backtracking recursion gives you a plain two-way choice: either you take the current node u in the vertex cover, or if you choose not to take the node u in, then you have to take **every** neighbour of u in the vertex cover. (In the dominating set problem, you were forced to take in at least one neighbour of u. It follows that the minimal vertex cover must necessarily contain at least as many nodes as the minimal dominating set, independent or not.)

It is again useful to sort the nodes in descending order of their number of neighbours before having the backtracking recursion go through these nodes. Another good thing to notice is that if u is a node with only one neighbour v, we might as well nail in this neighbour v in the vertex cover before starting the recursive search, since v covers not only the edge between u and v, but also all the edges emanating from v to the rest of the graph. This heuristic does not apply to construction of the optimal independent dominating set, although it does apply to unrestricted dominating set (DUCY?). Also, be careful not to accidentally take both u and v in the vertex cover in the situation where these nodes form a **duopole** of two nodes that are each other's only neighbours.

`edges`	Expected result
`[[4, 3, 2], [2, 3], [1, 0], [1, 0], [0]]`	2
`[[5], [2, 5, 3], [1, 3, 5], [1, 2, 5], [], [1, 0, 3, 2]]`	3
`[[2, 5, 6], [6, 2, 4], [5, 4, 1, 0, 3, 6], [4, 2], [3, 2, 5, 1], [4, 0, 2], [2, 1, 0]]`	4
`[[2, 1, 7, 3, 9], [0, 8, 4], [4, 0, 9], [4, 0, 6], [3, 2, 1, 7], [7, 6], [5, 7, 3, 9], [4, 0, 9, 6, 5, 8], [1, 7], [2, 7, 0, 6]]`	6

Since the nodes that were not chosen in the vertex cover form an independent set, finding a minimal vertex cover is equivalent to finding a maximal independent set. To-*may*-to, to-*mah*-to. All NP-complete search problems are fundamentally the same problem anyway under different disguises.

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 $k < p$, 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 $4p$, by the same logic 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.

The remains

```
def repeating_decimal(a, b):
```

Unless the prime factors of the denominator *b* of an integer fraction *a* / *b* are powers 2 and 5, the decimal representation of that integer fraction ends up being infinitely long. However, for all rational numbers, this decimal representation is guaranteed to eventually become periodic so that after some initial prefix of decimal digits, the same pattern of decimal digits keeps repeating infinitely. Given an integer fraction *a* / *b*, this function should compute the initial part and the repeating part, returning the answer of the tuple of two strings containing the digits of these two parts. You may assume that 0 < *a* < *b,* so that your function only needs to handle numbers between 0 and 1.

As explained in the section "Decimal expansion and recurrence sequence" on the Wikipedia page "Repeating decimal", the initial and repeating parts of the decimal representation can be computed by performing the long division of *a* / *b*, just like you were taught back when you were a wee kid in the math class, to generate the digits of the decimal representation. However, to avoid guessing when the decimal digits look like they might be repeating (both the initial part and the repeating part of the decimal representation can be arbitrarily long, even for seemingly simple fractions that don't seem to warrant this), your function needs to somehow keep track of the position where it has previously encountered each possible remainder. Once the long division encounters a remainder that it has already encountered during the same division, the rest of the division will be periodic and you can stop and extract the repeating part as the decimal digits between the two occurrences of the same remainder. Whatever digits precede this repeating part will then be the initial part.

a	b	Expected result
1	7	`('0.', '142857')`
10	11	`('0.', '90')`
4	29	`('0.', '13793103448275862068965517 24')`
3	44	`('0.06', '81')`
1	48	`('0.0208', '3')`
67	432	`('0.1550', '925')`
66	433	`('0.', '152424942263279445727482678983833718244803695150115473441108545034642032332563510392609699769053117782909930715935334872979214780600461893764434180138568129330254041570438799076212471131639722863741339491916859122401847575057736720554272517321016166281755196304849884526558891454965357967667436489607390300230946882217090069284064665127020785219399538106235565819861431870669745958429561200923787528868360277136258660508083140877598')`

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]`

Split the digits, maximize the product

```
def max_product(digits, n, m):
```

A string contains exactly *n* + *m* decimal `digits` listed in descending order. Your task is to use these `digits` to build two positive integers of exactly *n* and *m* digits, respectively, to maximize the product of your two integers. Obviously, digits chosen to both these integers will appear in descending order, but in which integer will you place each digit to maximize the resulting product?

This problem is generalized from the adorable classroom story given in the post "Mrs. Nguyen's prestidigitation" by Brian Hayes. Once reasoned out, the algorithm is quite short and sweet… provided that there are no duplicate digits or zero digits! However, your function must work correctly even without these restrictions.

This problem can be split into recursive two-way decisions by realizing that the first digit must go either to the first or to the second integer being constructed. After this two-way choice, continue by placing the second digit to either one or the other, and so on. The base case of this recursion is when one of the numbers reaches its required length, at which point you can stuff the remaining digits into the other number and compare the resulting product to the current maximum. Make sure to also handle each block of *k* consecutive identical digits as a branch of *k* + 1 possibilities of how many of these digits you place into the first number, as opposed to a branch of 2^k redundant possibilities to divvy up these identical digits between the two numbers.

`digits`	`n`	`m`	Expected result
`'6540'`	`3`	`1`	`3240`
`'87452'`	`3`	`2`	`63168`
`'942200'`	`5`	`1`	`379800`
`'9877100'`	`4`	`3`	`8448700`
`'8866666555554222211110'`	`12`	`10`	`7510047809441458785210`
`'9998888876544332000000'`	`11`	`11`	`9877523839516576000000`

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.

Gijswijt sequence

```
def gijswijt(n):
```

Gijswijt sequence is an interesting sequence of positive integers that starts from value 1. After that, each element of this sequence equals the largest integer *k* for which there exists some suffix of the preceding sequence, no matter of what length, that repeats *k* times consecutively. For example, the element following the initial sequence 1, 1, 2, 1, 1, 2, 2, 2 would be 3, since the preceding sequence contains the suffix [2] that is repeated three times consecutively.

This function should return the element in position `n` of the Gijswijt sequence, when position counting again starts from 0 for us budding computer scientists. The automated tester is guaranteed to give your function values of n in increasing order, so your function should construct this sequence to a global list initialized to contain some prefix of the Gijswijt sequence, and each time extend this sequence one element at the time until the stored sequence list contains the requested position `n`.

As explained in the article "Seven Staggering Sequences" by Neil Sloane, this sequence contains all positive integers somewhere in it. However, this sequence grows at such an incredibly glacial pace that element 4 makes its first appearance in the position 219 (using zero-based position counting), and element 5 makes its first appearance in the position that equals about 10 raised to the power of 10^{23}. (Read that sentence again to make sure that you understood it correctly.) Your function may therefore safely assume that each element in this sequence for the values of `n` requested by the tester will be either 1, 2, 3 or 4. To solve the element in position n, your function should try out all possible repeated suffix lengths *k* from 1, 2, 3 ... and quickly determine how many times the suffix of length k is repeated. Don't be a Shlemiel who extracts entire sublists to compare for equality; rather use a loop to compare pairwise elements until a mismatch is found, which will usually happen after looking at only a couple of elements.

`n`	Expected result
`0`	`1`
`6`	`2`
`91`	`3`
`219`	`4`
`564`	`2`

Minimal Egyptian fractions

```
def minimal_egyptian(f, k):
```

The two earlier problem of Egyptian fractions had you implement the greedy algorithm and the pairing algorithm to express an integer fraction a/b as a sum of distinct unit fractions $1/n$, and you should solve at least one of those problems before attempting this problem to get an idea of how unit fractions work. Neither of the previous algorithms is guaranteed to result in the smallest possible number of distinct unit fractions. The function required here should determine whether the given `Fraction` object `f` can be expressed as a sum of at most `k` distinct unit fractions.

This problem can be solved with recursive search that receives an additional third parameter `n`, the smallest positive integer that you are allowed to use as a unit fraction denominator. This parameter is used to ensure that the result will use only distinct unit fractions. The base cases of this recursion are when $k = 0$ so that the answer is trivially `False`, and when the fraction is of the form $1/m$ with m no smaller than n, so that the answer is trivially `True`. Otherwise, try out the possible values for n for the first fraction $1/n$ in the breakdown (the arithmetic used in the greedy algorithm works here just as well), and for each such n, determine recursively whether the remaining $f - 1/n$ can be broken down into at most $k - 1$ distinct unit fractions whose denominators are at least $n + 1$.

f	k	Expected result
`Fraction(25, 51)`	2	`False`
`Fraction(4, 51)`	3	`True`
`Fraction(52, 53)`	4	`False`
`Fraction(45, 62)`	4	`True`
`Fraction(97, 145)`	4	`False`

The still open Erdős–Straus conjecture asks whether there exists some integer fraction of the form $4/n$ whose breakdown into distinct unit fractions requires more than four terms.

Unity partition

```
def unity_partition(n):
```

John D. Cook recently tweeted out the following interesting algebra fact, originally proven by none other than Ronald L. Graham himself: Every positive integer $n > 77$ can be expressed as a sum of distinct positive integers whose reciprocals add up to exactly one. For example, the integer 87 can be partitioned into a sum 2 + 4 + 6 + 15 + 60, and we leave it for the reader to convince himself that the sum 1/2 + 1/4 + 1/6 + 1/15 + 1/60 does indeed add up to unity.

This function should return the list of these integers that add up to the given *n* and satisfy the required property of the sum of reciprocals adding up to one, these integers listed in ascending order. Since the asked integer partition is not necessarily unique, to satisfy the automated testing we force the result to be unique by requiring this function to return the **lexicographically smallest** solution list that starts with the smallest integer that can be extended to a working solution, followed by the lexicographically smallest way to complete this solution list.

For anybody who has been solving these problems up to this far, Python should make this a relatively easy task by using recursion to enumerate the possibilities, aided with the standard library `Fraction` class to perform the exact arithmetic of fractions. However, the difficulty in this problem is pruning the search space and recognizing the inevitable dead ends as soon as possible, so that all required results are computed within the time limit imposed in the tester.

n	Expected result
78	`[2, 6, 8, 10, 12, 40]`
88	`[2, 4, 8, 20, 24, 30]`
121	`[2, 3, 8, 36, 72]`
150	`[2, 4, 9, 18, 26, 39, 52]`
200	`[2, 3, 9, 36, 60, 90]`
250	`[2, 3, 12, 28, 44, 77, 84]`
300	`[2, 3, 10, 40, 50, 75, 120]`

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.

Domino poppers

```
def domino_pop(dominos):
```

A row of `dominos` has been laid down in a chain, the pips at the end of each tile not necessarily matching the pips at the start of the neighbouring tile. Each domino is a two-tuple of its pips. In a single move, you can choose any neighbouring pair of dominos whose touching pips match each other. (As usual, zero pips serve as a **joker** that matches any other number of pips in the other domino.) Your chosen two matching dominos disappear, and the rest of the chain moves to the left to fill in the gap to keep the entire chain contiguous. This function should compute the smallest possible number of dominos left in the chain when these removals are done the best possible way.

This problem is adapted from the problem "Bad to the bone" from 1992 ACM East Central Regional Programming Contest, as collected by Ed Karrels. In this problem, the leftmost pair of matching dominos was always eliminated first. This simpler version of our problem allows for an efficient **stack-based** solution that iterates through the `dominos` in one pass from left to right, the current domino either eliminating the matching domino at the top of the stack, or being itself pushed on top of the stack, with no branching choices along the way.

However, our version of the problem cannot be solved greedily by always removing the leftmost matching domino pair, as illustrated by the chain `[(1, 2), (2, 3), (3, 5), (2, 4)]` that allows the entire chain to be eliminated by starting with the two dominos in the center. You must therefore adapt the stack-based solution to branch recursively to examine both possibilities of the current domino either eliminating or not eliminating the domino at the top of the stack.

`dominos`	Expected result
`[(2, 5)]`	1
`[(3, 5), (5, 6), (6, 6)]`	1
`[(6, 4), (4, 2), (2, 5), (5, 0)]`	0
`[(5, 0), (3, 6), (6, 2), (0, 3)]`	0
`[(3, 1), (6, 6), (5, 1), (4, 3)]`	4
`[(2, 3), (3, 3), (2, 2), (3, 3), (3, 2), (0, 1)]`	0

As an additional note, this Ed Karrels fellow is also somewhat of a pi enthusiast.

Balance of power

```
def shapley_shubik(weights, quota):
```

In a democracy, we are normally accustomed to the idea of one vote per person, but in corporate meetings, individual members can have a different number of votes. In such situations, the power of each voter is not necessarily linear with respect to the number of votes. For example, assuming **majority voting** and four individual voters with vote weights `[4, 1, 1, 1]`, the first voter is effectively a **dictator** who gets to decide how every vote will go. On the other hand, for three voters with vote weights `[10000, 10000, 1]`, the Lilliputian third voter wields a massively disproportionate power to unilaterally decide the outcome of every issue where the big boys disagree!

The Shapley–Shubik power index can be used to gauge the relative power of each voter in such situations. Given the `weights` of each individual voter followed by the `quota`, the total number of votes required to win the vote (this is normally half the sum of `weights` plus one, but can also be something else, such as the entire sum in a jury trial, or just one in a **vetocracy**), the algorithm should iterate through all *n*! possible permutations of the *n* voters. For each permutation of voters, the algorithm should add up the votes from the beginning to the point of the **pivot** voter of that permutation, that is, the voter at which point the sum of votes so far becomes greater or equal to `quota`. The Shapley–Shubik power index of each voter is defined simply as the number of times that that voter ends up the pivot voter over the *n*! possible permutations.

Your function should recursively generate the voter permutations one at the time. However, to save time, you can stop the generation of the current permutation when you reach the pivot voter of that permutation. Before returning, add *m*! to the power index of that pivot voter, where *m* is the number of remaining voters including that pivot voter. (It might be handy to precompute the factorials into a list for quick lookup during the recursion.)

`weights`	`quota`	Expected result
`[1, 1, 1]`	`3`	`[2, 2, 2]`
`[4, 1, 1, 1]`	`4`	`[24, 0, 0, 0]`
`[4, 1, 1, 1]`	`1`	`[6, 6, 6, 6]`
`[9, 8, 6, 1, 1]`	`18`	`[54, 54, 4, 4, 4]`
`[23, 22, 17, 16, 5, 1, 1]`	`48`	`[1428, 1428, 1008, 1008, 168, 0, 0]`
`[38, 38, 35, 34, 34, 21, 10, 1]`	`153`	`[7872, 7872, 6528, 6528, 6528, 3840, 1152, 0]`

Longest zigzag subsequence

```
def longest_zigzag(items):
```

As we may recall, a **subsequence** of the given list of `items` consists of some of these items, not necessarily consecutive, listed in the order in which they appear in the original `items`. For example, the list `[2, -1, 3, 2]` is one possible subsequence of the list `[8, 2, -1, 2, 3, 2]`. We next define a subsequence to be **zigzag** if the signs of the differences between consecutive elements strictly alternate between +1 and –1, where the first such difference can be either positive or negative. This function should find the longest zigzag subsequence of the given list of `items`. Since this sequence is not necessarily unique, you should return its length.

This classic exercise on **dynamic programming** can be solved by tabulating the values of the function $s(i, d)$, where i is a position to the `items` list and d is the direction of the next allowed step, either +1 or –1. When $d = +1$, $s(i, +1)$ equals one plus the maximum of the values $s(j, -1)$ for the positions $j > i$ for which `items[j] > items[i]`. The formula for $s(i, -1)$ is defined symmetrically. For the position m at the end of the `items` list, $s(i, +1) = s(i, -1) = 1$. You can compute these values either by tabulating them from the end of the list to the beginning, or by using a recursive formula once again aided with the `@lru_cache` function decorator to eliminate the exponentially repeated subproblem computations. The final answer to the problem is then simply the larger of the two values $s(0, +1)$ and $s(0, -1)$.

`items`	Expected result
`[7, 3]`	2
`[3, 4, 1]`	3
`[10, 6, 4, 1, 5]`	3
`[5, 1, 3, 0, 1]`	5
`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`	2
`[5, 6, 3, 2, 2, 9, 9, 0, 3]`	6

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

`0100011010111001001111011000001010000111010010110111111000100000000110`...

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.

Game with multiset

```
def game_with_multiset(queries):
```

This nifty little problem is the Problem 1913C from CodeForces, "Game with Multiset". Your function is given a list of `queries` so that each individual query is of the form `('A', x)` or `('G', x)`, where `x` is some non-negative integer. Your function needs to process these queries in the exact order given. To do this, your function needs to somehow maintain a **monotonic multiset** of nonnegative integers, initially empty. Unlike in an ordinary set, the same value can occur in the multiset any number of times.

Each query `('A', x)` adds the integer `2**x` to your multiset. (Since the numbers that we are going to store are powers of two, of course you will only store the raw exponent `x`, instead of `2**x`.) Each query `('G', v)` asks you to determine whether the integer `v` can be constructed as a sum of some values that have been added to the multiset so far. Each value can be used in the sum up to as many times as it currently occurs in the multiset at that point. Your function should return the list of truth values for the results of the `('G', v)` queries.

This problem would be easy if each `x` could occur in the set only once, since you could just express `v` in binary and see if the corresponding powers of two occur in the set. (This would be literally just one line of code using clever **bitwise arithmetic**.) Multisets give us more freedom and possibilities in constructing the sum so that if you don't have some particular power that you need stored in the set, using two instances of the one notch lower power (or four instances of the two notches lower power, or any suitable combination of lower powers for that matter) does the job as well. The question is how you arrange all this efficiently. One linear loop from the highest bit of `v` down to lowest should be enough, keeping track of how many lower powers it needs to fulfill the bit deficit that it has accumulated down the line. (You can also note the handy Python built-in `bin`.)

`queries`	Expected result
`[('A', 0), ('A', 0), ('A', 2), ('G', 6), ('G', 3), ('G', 5)]`	`[True, False, True]`
`[('A', 3), ('A', 1), ('A', 4), ('G', 26), ('G', 23), ('G', 18), ('G', 1), ('A', 2), ('A', 1), ('G', 22)]`	`[True, False, True, False, True]`
`[('A', 4), ('A', 2), ('A', 2), ('G', 24), ('A', 1), ('A', 1), ('A', 5), ('G', 1), ('A', 4), ('A', 3), ('G', 10)]`	`[True, False, True]`

Tower of Babel

```
def tower_of_babel(blocks):
```

The side lengths of a solid three-dimensional block are given as a three-tuple `(a, b, c)`. A block can be freely rotated and placed on a flat surface in three different ways, depending on which face you decide to make its bottom face. Your task is to reach for the heavens by building the tallest possible tower under the constraint that a block can be placed on another block only if its bottom face is **strictly smaller** in both dimensions than the top face of the block directly below it. You are allowed to place only one block directly on the block (or on the table underneath), and are not allowed to combine multiple thin pillar blocks together on top of the same block to form a wide block.

This function should return the height of the tallest possible tower that can be built from the given list of `blocks`. This can once again be done with recursion that first chooses the bottom block and then tries to recursively build the tallest possible tower on top of that using the remaining blocks. It will probably speed up the search to preprocess these `blocks` in some useful manner, at least initially rotating each block so that its three dimensions are listed in descending sorted order.

`blocks`	Expected result
`[(5, 6, 4), (5, 7, 3)]`	`12`
`[(8, 10, 1), (1, 8, 6), (10, 7, 9), (4, 6, 2)]`	`20`
`[(1, 4, 6), (3, 1, 11), (11, 2, 1), (2, 10, 4), (7, 8, 3)]`	`29`
`[(5, 4, 9), (2, 8, 8), (8, 1, 7), (4, 4, 6), (12, 13, 3), (7, 10, 9)]`	`33`
`[(1, 2, 9), (4, 10, 1), (16, 8, 11), (3, 1, 1), (14, 4, 10), (15, 3, 3), (13, 5, 15), (11, 16, 5)]`	`65`

(This problem is problem 437, "Tower of Babylon", in the Online Judge collection.)

Make a list self-describing

```
def self_describe(items):
```

A list of positive integer `items` is **self-describing** if every *x* that appears somewhere in that list appears exactly *x* times. For example, `[1]`, `[2, 4, 4, 2, 4, 4]` and `[3, 2, 1, 2, 3, 3]` would be self-describing in this sense. Your function receives a list of *n* integer `items`, each item taken from the range 1, ..., *n*. Assuming that you are allowed to change any of these `items` to any values of your choice, your function should compute and return the smallest number of such changes required to turn the `items` list self-describing. Since you could always trivially make each element equal *n*, this problem has a well-defined solution for every possible list of `items`.

This problem can be solved in various different ways. This author suggests you first define a **recursive generator** function to `yield` all possible integer partitions of the given integer *n* that don't have any repeating elements. For example, when *n* = 10, this generator would `yield` the lists `[10]`, `[9, 1]`, `[8, 2]`, `[7, 3]`, `[7, 2, 1]`, `[6, 4]`, `[6, 3, 1]`, `[5, 4, 1]`, `[5, 3, 2]` and `[4, 3, 2, 1]`, one such list at the time.

Armed with this generator, your function should then preprocess the `items` by computing the counter dictionary `c` so that `c[x]` equals the number of times that x appears in `items`. Then, loop through all the possible non-repeating partitions of *n* and compute the cost of converting the `items` into a list that contains precisely the numbers that appear in that partition in their correct multiplicities, and finally return the smallest cost found this way.

`items`	Expected result
`[1]`	`0`
`[1, 2, 3]`	`1`
`[2, 2, 5, 5, 5, 5]`	`2`
`[1, 2, 4, 3, 2]`	`2`
`[3, 2, 3, 4, 3, 1, 2, 2, 2]`	`3`
`[3, 6, 2, 7, 3, 5, 7, 2, 5, 2]`	`4`

Two pins, not three, Dolores

```
def kayles(pins):
```

Kayles is a combinatorial game made famous by Conway, Guy and Berlekamp, and is often used as an example in teaching basic concepts of combinatorial game theory. The game is a hypothetical variation of bowling where some pins have been arranged in a linear row. Both players are perfectly accurate so that either player can knock down any one pin or any two adjacent pins of his choice, but is never able to knock down three or more pins, no matter how he would like to throw. Two players take alternate turns knocking down pins, and the player who knocks down the last pin and leaves his opponent unable to move wins the game.

The game has a trivial winning strategy for the first player if the pins are together in a single group, so to make this more interesting, let's assume that some pins have been randomly removed from the row before the actual game begins. This function receives the current situation as a list of groups of pins. For example, the list `[7, 3, 4, 2]` means that the first group contains seven consecutive pins, the second group contains three consecutive pins, and so on. The function should determine whether the first player has a winning strategy starting from the given list of `pins`. This can be done with the search that tries all possible moves available for the current player, and recursively evaluates the resulting situations from the opponent's point of view. The current situation is **hot** (winning) if there exists any one move that leads to a situation that is **cold** (losing) for the opponent so that all his possible moves from that situation lead to hot states.

The base case is when only one non-empty group of pins remains, as above. A similar argument also shows that any two groups with the same number of pins cancel each other out, since neither player can gain more than a temporary respite by making any play in either one of those groups. For example, `[7, 4, 4, 4, 2, 2, 1]` plays essentially the same as `[7, 4, 1]`.

`pins`	Expected result
`[3, 1]`	`True`
`[4, 1]`	`False`
`[2, 7, 1, 3]`	`True`
`[2, 3, 1]`	`False`
`[2, 2, 3, 1, 1, 1, 2]`	`False`
`[2, 6, 1, 5, 3, 2, 5, 5, 1, 4, 3]`	`True`

Knight jam

```
def knight_jam(knights, player):
```

The vast majority of **combinatorial games of complete information** such as tic-tac-toe, chess and go are **partisan** so that the moves available for each player on his turn are different from the other player's possible moves. For example, in chess, one player can move only the white pieces whereas the other one can move only the black pieces. This problem, however, concerns a curious **impartial** game inspired by chess where the exact same moves are available for both players on their turn.

A set of chess `knights`, all same colour and thus on the same side, stand on some squares of the infinite two-dimensional grid of pairs of natural numbers. On his turn, player zero can move any knight from its current square (x, y) into either $(x - 2, y + 1)$ or $(x - 2, y - 1)$. Player one, on the other hand, can move the knight from its current square (x, y) into either $(x + 1, y - 2)$ or $(x - 1, y - 2)$. A knight can jump over other knights, but cannot jump into a square already occupied by another knight, nor jump outside the game board onto coordinates whose x- or y-coordinate is negative.

The players alternate making their moves in this manner, and the player stuck and unable to make a move loses. (This must eventually happen, since each move strictly decreases the **Manhattan distance** of that knight from the origin.) This function should determine whether the given `player` has a winning move with the given `knights`, that is, any one move for which the opponent does not have winning response. This can be determined by looping through the possible moves for the given `knights` for that `player`. For each possible move, determine recursively whether the opponent has a winning move from the new situation produced from that move.

`knights`	`player`	Expected result
`{(2, 2), (3, 0)}`	`0`	`True`
`{(0, 1), (2, 2)}`	`0`	`False`
`{(0, 1), (1, 2), (2, 1)}`	`1`	`True`
`{(4, 4), (2, 4), (4, 1), (2, 1)}`	`0`	`True`
`{(0, 4), (3, 4), (3, 3), (3, 2), (4, 1)}`	`0`	`False`
`{(0, 1), (1, 2), (1, 5), (0, 3), (2, 0), (5, 1), (5, 2)}`	`1`	`True`

The recursive definition of the winning move may seem a bit queer in that it does not appear to have a base case. But the base case is there, even if you might need to squint a little bit to see it.

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 from one bus to another in the same city 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 within the same day before the next midnight, 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`

The sharpest axes

```
def vector_add_reach(start, goal, vectors, giveup):
```

This problem was inspired by yet another excellent Quanta Magazine article "An Easy-Sounding Problem Yields Numbers Too Big for Our Universe" that the reader should first read for a good explanation for what this deceptively simple problem is all about. Given the list of direction `vectors` that determine the possible moves from the current point, this function should compute how many steps are needed to get from the `start` coordinates to the `goal` coordinates. If it is not possible to reach the `goal` state within `giveup` moves, this function should return `None`.

This problem can be solved with the simple **breadth-first search**, the same as many previous problems. Initially the first layer consists only of the `start` point. Repeatedly compute the next layer of reachable points by looping through the points in the current layer and generating the list of all points reachable from these points with the given `vectors`, terminating your search either when you find the `goal` point or when more than `giveup` layers have been generated. However, the **reversible** nature of the moves allows the use of bidirectional search to cut the effective search depth through this exponentially large search tree in half, for massive savings in both time and space!

Operate two separate instances of breadth first search, one emanating from the `start` point using the `vectors` to the forward direction, the other emanating from the `goal` point using the negated `vectors` to the backward direction. Alternate generating the successive layers of reachable points between these two searches, terminating when either search "meets the other one in the middle" by finding some point already discovered by the opposite search.

The following table contains examples only from two-dimensional space to keep it compact, but your function should work correctly for points of any number of dimensions.

start	goal	vectors	giveup	Expected result
(3, 0)	(2, 2)	[(-2, 1), (0, 2)]	6	None
(3, 2)	(4, 1)	[(2, -3), (1, 2), (-1, 2)]	6	2
(5, 7)	(0, 4)	[(-3, -5), (2, 1), (-4, 3), (2, 5), (3, 5)]	15	11
(9, 1)	(8, 3)	[(-1, 2)]	10	1

Vidrach Itky Leda

```
def itky_leda(board):
```

This cute little problem of state space searching is taken from Jeff Erickson's collection of old homework and exam questions. The `board` is an *n*-by-*n* square grid with a positive integer in every cell. A red token initially sits in the corner cell (0, 0), and a blue token initially sits in the opposite corner cell (*n* – 1, *n* – 1). The problem is to exchange the locations of these two tokens in the smallest possible number of moves.

Each move consists of the player choosing any one of these two tokens and one of the four compass directions, and moving the token in that direction the number of steps given by the number in the cell currently occupied by the other token. A token may not move outside the `board`, or end up in the cell occupied by the other token. This function should return the smallest possible number of moves necessary to exchange the positions of the red and blue token, or return `-1` if this task is impossible on the given `board`.

This problem can be solved with breadth first search, but same as in the previous problem, the reversible nature of the moves allows the search to be tightened up exponentially with the aid of bidirectional search. Same as before, you should operate two separate instances of breadth first search, one starting from the initial state and the other starting from the goal state. Alternate between these two searches to generate the successive layers of reachable states, terminating the search when either search "meets the other one in the middle" by finding some state already discovered by the opposite search. Chopping the depth of the search tree and therefore the exponent of the algorithm's worst case running time in half is surely nothing to sneeze at!

`board`	Expected result
`[[1, 0],` ` [0, 1]]`	`-1`

`[[1, 2, 4, 3],`
` [3, 4, 1, 2],`
` [3, 1, 2, 3],`
` [2, 3, 1, 2]]`
	`5`

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, immediately removing it from the board. 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.

Sneaking

```
def sneaking(n, start, goal, knights):
```

Separated from his beloved queen who is wreaking havoc on a different board in the previous problem, a lone chess king finds himself at the `start` position on the *n*-by-*n* chessboard populated by sleeping enemy `knights`. The king would very much like to escape by reaching the `goal` position containing the exit before the enemy knights wake up. Unless waken up, these enemy knights will remain in their slumber and do not move during this demented version of the game of the kings. However, these sleeping knights are essentially spiders who have set up sensitive tripwires on all the squares that they guard in a single knight's jump. Should the king step on a guarded square, all knights guarding that square will instantly wake up and behead the king.

This function should determine whether it is possible for the king to reach the `goal` without losing his head. So that this problem isn't merely a simple breadth-first search through the unguarded squares on the board, the king can also sneak into a square of any sleeping knight that is not guarded by some other knight, and quietly slice his throat to continue his journey from that square. Solving some of these instances requires the king to do precisely that to unguard some square that is necessary for the rest of the journey.

To speed up this function, it will be handy to precompute an *n*-by-*n* table that keeps track of how many knights are currently guarding each square. During the recursion, update this table whenever some knight gets his throat cut open, and downdate the table back to the previous values whenever the recursive search backtracks to restore that knight back to life.

n	start	goal	knights	Expected result
4	(3, 2)	(1, 1)	[(1, 2), (2, 1), (0, 3), (2, 3), (0, 2)]	False
4	(0, 3)	(0, 1)	[(1, 2), (1, 3)]	True
5	(0, 0)	(3, 0)	[(2, 3), (2, 0), (4, 2)]	False
6	(5, 1)	(1, 4)	[(4, 4), (1, 5), (3, 1), (1, 1), (2, 5), (1, 3)]	True
8	(0, 7)	(5, 3)	[(4, 4), (5, 1), (4, 2), (5, 6), (6, 0), (3, 2), (5, 2)]	False

Inverse pair sums

```
def pair_sums(n, sums):
```

Given a list of *n* distinct positive integers, constructing the list of all possible `sums` of adding two integers from that list would be a good exercise for an introductory course. However, since that one is just a one-liner `sorted(x+y for (x, y) in combinations(items, 2))` for anyone who has been solving these problems this far, we rather turn this problem upside down and ask for a function to reconstruct the original list of `n` distinct integers from these pairwise `sums`. For example, given the pairwise sums `[3, 4, 5, 5, 6, 7]`, this function should return `[1, 2, 3, 4]` as that original list of numbers.

Your search algorithm should try out all possible values for the last number of the original list of *n* elements, and try to fill in the smaller elements recursively from there. Note that once you have chosen this last number, the last two items of the `sums` list immediately give you the second and third largest numbers of the original list without any branching. Make sure to use an efficient dictionary data structure to keep track of which pairwise sums have been used up by your current choices and which ones are still available.

The automated tester will give your function only `sums` lists computed from some list of distinct positive integers, so that there always exists at least one solution. Since this solution is not necessarily unique, your returned solution must be the highest in **colexicographic** order; that is, your solution must use the largest possible value for its last element. Given that last value, the solution must then use the largest possible value for its second last element, and so on.

`sums`	n	Expected result
`[5, 8, 9]`	3	`[2, 3, 6]`
`[8, 13, 15, 18, 20, 25]`	4	`[3, 5, 10, 15]`
`[7, 8, 11, 13, 16, 17, 17, 20, 21, 26]`	5	`[2, 5, 6, 11, 15]`
`[4, 8, 10, 12, 14, 15, 17, 18, 21, 23, 25, 25, 28, 29, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 39, 41, 42, 43, 44, 46, 47, 49, 50, 50, 50, 50, 52, 54, 55, 56, 57, 58, 60, 62, 63, 63, 64, 64, 65, 68, 70, 71, 71, 72, 75, 76, 77, 79, 83, 85, 88, 89, 92, 97, 104, 110]`	12	`[1, 3, 7, 11, 14, 22, 27, 28, 36, 43, 49, 61]`

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 co-ordinates 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* × *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`

Oppenhoppenheimereimer

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

Lindenmayer systems, or simply **L-systems** for all of us too busy to have time to spit out 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.

Bonus problem 110: String stretching

```
# For Charmaine: Delayed response is sometimes too early
def string_stretching(text):
```

This wonderful little problem is titled "String stretching" in the Kattis problem collection. The parameter string `text` has been constructed by making the `text` initially equal the empty string, and then repeatedly inserting that original pattern into a randomly chosen position of the `text` constructed so far. For example, if the initial pattern is `'bax'`, the `text` could develop from the empty string to `'bax'` to `'babaxx'` to `'babbaxaxx'` to `'babbaxaxbaxx'`, the newly inserted copy of the pattern shown with the green background in each step, to produce the final `text` equal to the string `'babbaxaxbaxx'`. Plenty of other possible result `text`s are equally possible, of course.

Your function must compute and return the shortest possible pattern that could have produced the given `text` in this manner after some number of insertions. If there are multiple solutions, your function should return the first solution in alphabetical order. Obviously the `text` length must be some multiple of the pattern length (if nothing shorter works, the `text` and the pattern could always simply be equal), and the first and the last characters of the `text` are necessarily also the first and the last characters of the pattern.

The problem is made tricky by the fact that the pattern may contain the same character multiple times. This makes your matching search potentially branch into several directions whenever you encounter the first character of the pattern in the `text`. Was that position the start of yet another insertion of the pattern, or is that character also part of the pattern after the first character?

`text`	Expected result
`'aaacccccc'`	`'aac'`
`'mimmiimi'`	`'mi'`
`'cccc'`	`'c'`
`'aadbabadbabdbab'`	`'adbab'`
`'aagaaagaagaaaaa'`	`'aagaa'` (but not equally possible `'agaaa'`)
`'aiaiiiaiiiiaiiiiaiaiiiii'`	`'aiii'`
`'mmmmammmmamammmammammammammmm'`	`'mmam'`

Bonus problem 111: Casinos hate this Toronto man!

```
# For Alain: Nimis malum quod potest etiam numerare
def optimal_blackjack(deck):
```

Legendary casino moguls such as Benny Binion, Steve Wynn and Veijo Bellagio built their opulent palaces of sin on the foundation of truth that the average punter has no chance against the house in the long run, unless he gets some crooked help under the table. 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 remaining cards, for an ultimate edge in card counting. In this problem you will implement this 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`

Bonus problem 112: Word bin packing

```
# for Debra: Add a point if necessary
def word_bin_packing(words):
```

Define that two words are **compatible** if their **Hamming distance** equals zero, that is, these words don't have the same letter common in any position. For example, `'oscar'` and `'floyd'` are compatible this way (the letter `o` is in a different position in both words), whereas `'oscar'` and `'shram'` are not compatible. Your task is to place given `words` in smallest possible number of separate **bins** so that all words placed in the same bin are compatible with each other.

This problem can be solved with recursive backtracking search. The recursive search function receives the list of bins built so far as parameter, initially an empty list at the top level call. Each bin itself is a list of words that have been placed in that bin in that branch of the recursive search. The recursive function then taken the next word, and branches through the options of placing that word into one of the existing bins where it is compatible with the existing words, or starting a brand new bin that now contains only that word. For each of these options, continue to place the remaining words optimally into the new list of bins.

As always in this sort of combinatorial searches, it is a good idea to accept the hard constraints that you are not allowed to violate, and use these constraints as supports to prune the exponentially branching tree of search possibilities. Since every word must eventually be placed somewhere, you might as well take the bull by the horns and place the most difficult words first in the recursive search, to find out how many bins minimum you are going to need. Before the recursive search, you should therefore sort the words in order of how many conflicts they have with other words.

words	Expected result
`['yarb', 'inez', 'doke', 'fray']`	1
`['klop', 'surv', 'aery']`	2
`['wagh', 'wash', 'vady', 'glob']`	3
`['cyan', 'cute', 'geic', 'cote', 'nori', 'boke', 'name', 'size', 'abye']`	6
`['fana', 'vamp', 'cars', 'scap', 'guff', 'sagy', 'juju', 'orbs', 'uang', 'taha', 'piky', 'turp', 'flow', 'eaux', 'irpe']`	7

Bonus problem 113: Probabilistic tic-tac-toe

```
# For Bill: As expected and excepted
def tic_tac(board, player, probs):
```

The trivial game of three-by-three **tic-tac-toe** can be made more interesting by assigning each cell a probability that the player trying to draw his mark in that cell is successful in this task, a failure causing the other player getting his mark placed in that very cell! Whichever mark appears in that cell, next comes the opponent's turn to choose his response under the same conditions. The two-dimensional three-by-three list `probs` gives this probability of the player successfully marking each particular cell on his turn. Unlike in the version of the game linked above, there is no possibility of a move leaving the cell empty, so exactly one of the two players' marks will appear on that cell. This ensures that the **state space** of this game is a **tree** that can be analyzed with the **expectimax** algorithm where each player always chooses the move that maximizes the probability of his win.

The player who first creates a three-in-a-row with his own marks wins the game outright. If the `board` fills up with marks but so that neither player has three in a row, the player whose turn it is to move loses, as is customary in **combinatorial games**. Finally, in case that the randomly generated test board simultaneously contains a three-in-a-row for both players, the player whose turn it is currently to play wins that game.

This function should calculate the probability the given `player` wins the entire game from his turn to make the next move on the given `board`, assuming that both players make their optimal moves throughout the remainder of the game. The function must return its answer as an exact `Fraction` object, and not use any floating point computations at any point.

This function takes too many parameters to allow tabular representation of test cases, so we shall show just one example. Given the `board` as `[['.', 'O', 'O'], ['.', '.', 'X'], ['O', 'X', 'X']]`, and the `probs` as `[[Fraction(17, 19), Fraction(58, 67), Fraction(40, 41)], [Fraction(43, 74), Fraction(11, 13), Fraction(1, 1)], [Fraction(4, 5), Fraction(17, 24), Fraction(5, 9)]]`, the `player` using the marker `'O'` has the exact probability `Fraction(225, 247)` of winning the entire game.

Bonus problem 114: Bandwidth minimization

```
# For Ron: It possibly could be that too
def bandwidth(edges):
```

An **undirected graph** consists of n nodes numbered from 0 to $n - 1$. Each node is connected to some other nodes with **edges** that are **undirected** so that if the node u is a neighbour of node v, then the node v is also a neighbour of node u. These `edges` are given as a list to your function so that each element `edges[u]` lists the neighbours of node u.

This function should renumber the nodes of the graph so that each node is given a unique number from 0 to $n - 1$, so that the maximum difference between to neighbouring nodes is minimized over the entire graph. This minimum of maximum differences between neighbouring node numbers in the given graph is called the bandwidth of that graph, which is what this function should return.

Of course this problem can be solved with a backtracking recursion that tries out all $n!$ possible ways to number the given nodes, and returning the bandwidth of the best such numbering, but this approach would be horrendously inefficient. A better technique would be to use **iterative deepening** in that you first try to construct a numbering whose bandwidth is equal to the largest degree of node in the graph. If no such numbering is possible, try to construct a numbering with bandwidth one larger than that, and so on. As soon as your recursion finds it impossible to number some node, it can backtrack to try out the next possible way to number the previous node. It might also be useful to preprocess the graph to determine the best order to fill in the numbers in these nodes.

`edges`	Expected result
`[[4, 3, 2], [2, 3], [1, 0], [1, 0], [0]]`	2
`[[5], [2, 5, 3], [1, 3, 5], [1, 2, 5], [], [1, 0, 3, 2]]`	3
`[[5], [5, 6], [4], [6], [6, 2, 7], [1, 7, 0], [1, 4, 3], [4, 5, 8], [7]]`	2
`[[4, 13, 6], [11, 4], [3], [4, 10, 2, 11], [0, 3, 9, 12, 8, 11, 1], [6], [12, 5, 8, 0], [13, 9, 11], [9, 6, 4, 13], [8, 11, 7, 4], [3, 11], [4, 1, 7, 10, 3, 9], [4, 6], [8, 7, 0]]`	6