109 Python Problems for CCPS 109

This document (version of **December 24, 2020**) contains specifications for the **lab problems** for **CCPS 109 Computer Science I**, as taught by Ilkka Kokkarinen. You **must implement all these functions in a single Python source code file** named `labs109.py`. This allows you to run the `tester109.py` script at any time to validate the functions you have completed so far, so you always know exactly where you are standing on this course. These tests are executed in the same order that your functions appear inside the `labs109.py` file.

Your functions may assume that the arguments given to them them are as promised in the problem specification, and never have to recover from invalid arguments. Note that zero integers and empty lists are often legitimate arguments, and your functions must handle them correctly. **The test for each individual function should take at most a couple of seconds to complete.** If some test takes a minute or more to complete, your code is too slow and its logic desperately needs to be streamlined. This is usually done by shaving off one level of nesting from your structured loops, sometimes with a `set` or a `dict` to remember what your function has seen and done.

SILENCE IS GOLDEN. When called, none of your functions should ever print anything on the console, but silently return the expected result. You can do some debugging outputs during the development, but remember to comment them out before submission. Also, unless explicitly requested to do so by the problem specification, **these functions should never mutate the lists and other sequences that they receive as arguments**.

This specification document and the automated tester script are released under GNU Public License version 3 to be distributed and adapted freely. The author has compiled these problems over time from a gallimaufry of sources ranging from the original lab and exam problems of his old Java version of this course to a multitude of programming puzzle and code challenge sites such as LeetCode, CodeAbbey, Stack Exchange Code Golf, and Wolfram Programming Challenges. Recreational mathematics columns by Martin Gardner and his spiritual disciples have also inspired many problems both old and new.

The author wishes to thank **Shu Zhu Su**, **Mohammed Waqas** and **Zhouqin He** for going above and beyond the call of duty in completing solutions that either revealed errors in the original problem specifications or their private model solutions, or independently agreed with the private model solution and this way raised the confidence to the correctness of both. All remaining errors, ambiguities and inconsistencies are the sole responsibility of Ilkka Kokkarinen. Report them to ilkka.kokkarinen@gmail.com to help nudge (or budge) this document one step closer to the perfection that it still is *n* steps away from, the exact value or even the scale of *n* still unknown.

Ryerson letter grade

```
def ryerson_letter_grade(pct):
```

Given the percentage grade, calculate and return the letter grade that would appear in the Ryerson grades transcript, as defined on the page Ryerson Grade Scales. This letter grade should be returned as a string that consists of the uppercase letter followed by the modifier '+' or '-', if there is one. This function should work correctly for all values of `pct` from 0 to 150.

Same as all other programming problems that follow this problem, this can be solved in various different ways. At this point of your studies, the simplest way to solve this problem would probably be to use an **if-else ladder**. The file `labs109.py` given in the repository `ikokkari/ PythonProblems` already contains an implementation of this function for you to run the tester script `tester109.py` to verify that everything is hunky dory.

pct	Expected result
45	'F'
62	'C-'
89	'A'
107	'A+'

As you learn more Python and techniques to perform computations within it, you may think up other ways to solve this problem. Some of these would be appropriate for actual productive tasks done in a professional coding environment, whereas others are intended to be taken in jest as a kind of conceptual performance art. A popular genre in all programming languages is to solve some straightforward problem with an algorithmic equivalent of a needlessly complicated Rube Goldberg machine, to demonstrate the universality and unity of all computation.

Ascending list

```
def is_ascending(items):
```

Determine whether the sequence of `items` is **strictly ascending** so that each element is **strictly larger** (not just merely equal to) than the element that precedes it. Return `True` if the list of `items` is strictly ascending, and return `False` otherwise. Note that the empty sequence is ascending, as is also every one-element sequence, so be careful that your function returns the correct answers in these seemingly insignificant **edge cases** of this problem. (If these sequences were not ascending, pray tell, what would be the two elements that violate the requirement and make that particular sequence not be ascending?)

items	Expected result
`[]`	`True`
`[-5, 10, 99, 123456]`	`True`
`[2, 3, 3, 4, 5]`	`False`
`[-99]`	`True`
`[4, 5, 6, 7, 3, 7, 9]`	`False`
`[1, 1, 1, 1]`	`False`

In the same spirit, note how every possible universal claim made about the elements of an empty sequence is trivially true. For example, if `seq` is the empty sequence, the two claims "All elements of `seq` are odd" and "All elements of `seq` are even" are both equally true, as is also the claim "All elements of `seq` are [colourless green ideas that sleep furiously](#)". For some reason, many people find this highly counterintuitive or even paradoxical, but it is a consequence of the fact that any logical formula of the form "*A* implies *B*" is automatically true whenever *A* is false.

Riffle

```
def riffle(items, out=True):
```

Given a list of `items` whose length is guaranteed to be even (notice here that "oddly" enough, zero is an even number), create and return a list produced by performing a perfect **riffle** to the `items` by interleaving the items of the two halves of the list in an alternating fashion.

When performing a perfect riffle shuffle, also known as the [Faro shuffle](#), the list of `items` is split in two equal sized halves, either conceptually or in actuality. The first two elements of the result are then the first elements of those halves. The next two elements of the result are the second elements of those halves, followed by the third elements of those halves, and so on up to the last elements of those halves. The parameter `out` determines whether this function performs an [out shuffle](#) or an [in shuffle](#) that determines which half of the deck the alternating card is first taken from.

items	out	Expected result
[1, 2, 3, 4, 5, 6, 7, 8]	True	[1, 5, 2, 6, 3, 7, 4, 8]
[1, 2, 3, 4, 5, 6, 7, 8]	False	[5, 1, 6, 2, 7, 3, 8, 4]
[]	True	[]
['bob', 'jack']	True	['bob', 'jack']
['bob', 'jack']	False	['jack', 'bob']

Only odd digits

```
def only_odd_digits(n):
```

Check that the given positive integer n contains only odd digits (1, 3, 5, 7 and 9) when it is written out. Return `True` if this is the case, and `False` otherwise. Note that this question is not asking whether the number n itself is odd or even. You therefore will have to look at every digit of the given number before you can proclaim that the number contains no odd digits.

Hint: to extract the lowest digit of a positive integer n, use the expression `n%10`. To chop off the lowest digit and keep the rest of the digits, use the expression `n//10`. Or, if you don't want to be this fancy, convert the number into a string first and work from there with string operations.

n	Expected result
8	False
1357975313579	True
42	False
71358	False
0	False

Cyclops numbers

```
def is_cyclops(n):
```

A nonnegative integer is said to be a **cyclops number** if it consists of an **odd number of digits** so that the middle (more poetically, the "eye") digit is a zero, and all other digits of that number are nonzero. This function should determine whether its parameter integer `n` is a cyclops number, and return either `True` or `False` accordingly.

n	Expected result
0	True
101	True
98053	True
777888999	False
1056	False
675409820	False

As an extra challenge, you can try to solve this problem using only loops, conditions and integer arithmetic operations, without first converting the integer into a string and working from there. Dividing an integer by 10 with the integer division `//` effectively chops off its last digit, whereas the remainder operator `%` can be used to extract that last digit. These operations allow us to iterate through the digits of an integer one at the time from lowest to highest, almost as if that integer were some kind of lazy sequence.

(Even better, this operation doesn't work merely for the familiar base ten, but it works for any base by using that base as the divisor. Especially using two as the divisor instead of ten allows you to iterate through the **bits** of the **binary representation** of any integer, which will come handy in problems in your later courses that expect you to be able to manipulate these individual bits.)

Domino cycle

```
def domino_cycle(tiles):
```

A single **domino tile** is represented as a two-tuple of its **pip values**, such as `(2,5)` or `(6,6)`. This function should determine whether the given list of `tiles` forms a **cycle** so that each tile in the list ends with the exact same pip value that its successor tile starts with, the successor of the last tile being the first tile of the list since this is supposed to be a cycle instead of a chain. Return `True` if the given list of domino tiles form such a cycle, and `False` otherwise.

tiles	Expected result
`[(3, 5), (5, 2), (2, 3)]`	True
`[(4, 4)]`	True
`[]`	True
`[(2, 6)]`	False
`[(5, 2), (2, 3), (4, 5)]`	False
`[(4, 3), (3, 1)]`	False

Count dominators

```
def count_dominators(items):
```

An element of `items` is said to be a dominator if **every** element to its right (and not just the one that is immediate to its right) is strictly smaller than it. This function should count how many elements in `items` are dominators, and return that count. By this definition, the last item of the list is automatically a dominator. For example, dominators of `[42, 7, 12, 9, 13, 5]` would be the elements 42, 13 and 5.

Before writing any code for this function, think about the tale of "Shlemiel the painter" and how this seemingly silly little tale from a far simpler time might relate to today's computational problems for lists, strings and other sequences. This problem will be the first of many that you will encounter during and after this course to illustrate the important principle of using only one loop to achieve in a tiny fraction of time the same end result that Shlemiel needs two nested full loops to achieve, your workload therefore increasing only **linearly** with respect to the number of `items` instead of **quadratically** (that is, as a function of the **square** of the number of items), same as how Shlemiel's task of running back and forth will increase as the function of the square of the length of the fence, so that doubling the length of the fence will quadruple the amount of running...

items	Expected result
`[42, 7, 12, 9, 2, 5]`	4
`[]`	0
`[99]`	1
`[42, 42, 42, 42]`	1
`range(10**7)`	1
`range(10**7, 0, -1)`	10000000

Trying to hide the inner loop of some Shlemiel algorithm inside a function call (including Python built-ins such as `max` and list slicing) will only enrage the Gods of Compubook Headings and summon them to return with tumult to claim their lion's share of execution time.

Extract increasing integers from digit string

```
def extract_increasing(digits):
```

Given a string of digits guaranteed to only contain ordinary integer digit characters 0 to 9, create and return the list of increasing integers acquired from reading these digits in order from left to right. The first integer in the result list is made up from the first digit of the string. After that, each element is an integer that consists of as many following consecutive digits as are needed to make that integer **strictly larger** than the previous integer. The leftover digits at the end of the digit string that do not form a sufficiently large integer are ignored.

This problem can be solved with one for-loop through the `digits` that looks at each digit exactly once regardless of the position of that digit in the beginning, end or middle of the string. Keep track of the `current` number (initially zero) and the `previous` number to beat (initially minus one). Each digit `d` then updates `current = 10*current+d` to tack that digit to the end of `current`.

digits	Expected result
`'045349'`	`[0, 4, 5, 34]`
`'77777777777777777777 77'`	`[7, 77, 777, 7777, 77777, 777777]`
`'12233344445555566666 6'`	`[1, 2, 23, 33, 44, 445, 555, 566, 666]`
`'1234567890987654321'`	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 98, 765, 4321]`
`'31415926535897932384626433832795028884'`	`[3, 14, 15, 92, 653, 5897, 9323, 84626, 433832, 795028]`
`'27182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952 60'`	`[2, 7, 18, 28, 182, 845, 904, 5235, 36028, 74713, 526624, 977572, 4709369, 9959574, 96696762, 772407663, 3535475945, 7138217852, 51664274274, 66391932003, 599218174135, 966290435729]`
`'123456789' * 100`	A list that contains 75 elements, the last one of which equals 345678912345678912345 67891

Words that contain given letter sequence

```
def words_with_letters(words, letters):
```

This might be a good place to bring in some general discrete math terminology that makes our problem specifications less ambiguous. A **substring** of a string consists of characters taken **in order** from consecutive positions. Contrast this with the similar concept of **subsequence** of characters still taken in order, but not necessarily at consecutive positions. For example, the strings `""`, `"e"`, `"put"`, `"ompu"` and `"computer"` are both substrings and subsequences of `"computer"`, whereas `"cper"` and `"out"` are subsequences of that word, but not substrings.

Note that the empty string is automatically a substring of every possible string. Every string is its own substring, although not a **proper substring** the same way how all other substrings are proper. Concepts of **sublist** and **subsequence** are defined for lists in an analogous manner. Since **sets** have no internal order on top of the element membership in that set, sets can meaningfully have both proper and improper subsets. However, the concept of subsequence would be nonsensical for sets and dictionaries.

Now that you know all that, given a list of `words` sorted in alphabetical order, and a string of required `letters`, find and return the list of words that contain `letters` as a sub*sequence*.

`letters`	Expected result (using the wordlist `words_sorted.txt`)
`'klore'`	`['booklore', 'booklores', 'folklore', 'folklores', 'kaliborite', 'kenlore', 'kiloampere', 'kilocalorie', 'kilocurie', 'kilogramme', 'kilogrammetre', 'kilolitre', 'kilometrage', 'kilometre', 'kilooersted', 'kiloparsec', 'kilostere', 'kiloware']`
`'brohiic'`	`['bronchiectatic', 'bronchiogenic', 'bronchitic', 'ombrophilic', 'timbrophilic']`
`'azaz'`	`['azazel', 'azotetrazole', 'azoxazole', 'diazoaminobenzene', 'hazardize', 'razzmatazz']`

Taxi zum zum

```
def taxi_zum_zum(moves):
```

A Manhattan taxicab starts at the origin point of $\mathbb{Z}^2$, the infinite two-dimensional lattice grid of all possible pairs of integers. The taxicab also has a direction that is always one of the four main compass directions, initially heading north. This taxicab then faithfully executes the given sequence of `moves`, guaranteed to be a string made up of the characters `'L'` for turning 90 degrees left (while standing in place), `'R'` for turning 90 degrees right (ditto), and `'F'` for moving one step forward towards the current heading. This function should return the final position of the taxicab in the integer grid coordinates of Manhattan.

moves	Expected result
`'RFRL'`	`(1, 0)`
`'LFFLF'`	`(-2, -1)`
`'LLFLFLRLFR'`	`(1, 0)`
`'FR' * 1000`	`(0, 0)`
`'FFLLLFRLFLRFRLRRL'`	`(3, 2)`

As an aside, why do these problems always seem to take place in Manhattan and evoke nostalgic visuals of Jackie Mason or that Woodsy Allen fellow talking with the grumpy immigrant cabbie and various colourful bystander characters, instead of being set in, say, the mile high city of Denver whose street grid is rotated 45 degrees from the main compass axes to equalize the amount of daily sunlight on streets laid out in both orientations? That ought to make for an interesting variation to many problems of this spirit. Unfortunately, diagonal moves always maintain the total **parity** of the coordinates, which makes it impossible to reach any coordinates of the opposite parity in this manner, as in that old joke with the punchline "Gee... I don't think that you can get there from here."

Giving back change

```
def give_change(amount, coins):
```

Given the `amount` of money (expressed as an integer as the total number of cents, one dollar being equal to 100 cents) and the list of available denominations of `coins` (similarly expressed as cents), create and return a list of coins that add up to `amount` using the **greedy approach** where you use as many of the highest denomination coins when possible before moving on to the next lower denomination. The list of coin denominations is guaranteed to be given in descending sorted order, as should your returned result also be.

amount	coins	Expected result
64	[50, 25, 10, 5, 1]	[50, 10, 1, 1, 1, 1]
123	[100, 25, 10, 5, 1]	[100, 10, 10, 1, 1, 1]
100	[42, 17, 11, 6, 1]	[42, 42, 11, 1, 1, 1, 1, 1]

This problem, along with its countless variations, is a computer science classic when modified to minimize the total number of returned coins. The above greedy approach then no longer produces the optimal result for all possible coin denominations. For example, using simple coin denominations of `[50, 30, 1]` and the amount of sixty kopecks to be exchanged, the greedy solution `[50, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]` ends up using eleven coins by its unfortunate first choice that prevents it from using any of the 30-kopeck coins that would be handy here, seeing that the optimal solution `[30, 30]` needs only two such coins! A more advanced **recursive** algorithm examines both sides of the "take it or leave it" decision for each coin and chooses the choice that ultimately leads to a superior outcome. The intermediate results of this recursion should then be **memoized** to avoid blowing up the running time exponentially.

Rooks on a rampage

```
def safe_squares_rooks(n, rooks):
```

A generalized *n*-by-*n* chessboard has been invaded by a parliament of `rooks`, each rook represented as a two-tuple `(row, column)` of the row and the column of the square that the rook is in. Since we are again computer programmers instead of chess players and other normal folks, these rows and columns are numbered from 0 to *n* - 1. A chess rook covers all squares that are in the same row or in the same column. Given the board size `n` and the list of `rooks` on that board, count the number of empty squares that are safe, that is, are not covered by any rook.

Hint: count separately how many rows and columns on the board are safe from any rook. Because permuting the rows and columns does not change the answer to this question, we can imagine all these safe rows and columns to have been permuted to form a rectangle at the top left corner of the board. The area of that safe rectangle is the product of its known width and height, of course, so the result is simply the product of these two one-dimensional counts.

n	rooks	Expected result
10	[]	100
4	[(2, 3), (0, 1)]	4
8	[(1, 1), (3, 5), (7, 0), (7, 6)]	20
2	[(1, 1)]	1
6	[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]	0
100	[(r, (r*(r-1))%100) for r in range(0, 100, 2)]	3900
10**6	[(r, r) for r in range(10**6)]	0

Try a spatula

```
def pancake_scramble(text):
```

Analogous to flipping a stack of pancakes by sticking a spatula inside the stack and flipping over the stack of pancakes resting on top of that spatula, a **pancake flip** of order k performed for the `text` string reverses the prefix of first k characters and keeps the rest of the string as it were. For example, the pancake flip of order 2 performed on the string `'ilkka'` would produce the string `'likka'`. The pancake flip of order 3 performed on the same string would produce `'klika'`.

A **pancake scramble**, as [defined in the excellent Wolfram Challenges programming problems site](#), consists of the sequence of pancake flips of order 2, 3, ... , n performed in this exact sequence for the given n-character `text` string. For example, the pancake scramble done to the string `'ilkka'` would step through the intermediate results `'likka'`, `'kilka'`, `'klika'` and `'akilk'`. This function should compute and return the pancake scramble of its parameter `text` string.

text	Expected result
`'hello world'`	`'drwolhel ol'`
`'ilkka'`	`'akilk'`
`'pancake'`	`'eanpack'`
`'abcdefghijklmnopqrstuvwxyz'`	`'zxvtrpnljhfdbacegikmoqsuwy'`

For those of you who are interested in this sort of stuff, the follow-up question "[How many times you need to pancake scramble the given string to get back the original string?](#)" is also educational, especially once the strings get so long that the answer needs to be computed analytically (note that the answer depends only on the length of the string but not the content, as long as all characters are distinct) instead of actually performing these scrambles until the original string appears. A more famous problem of [pancake sorting](#) asks for the shortest series of pancake flips whose application rearranges the elements of the given list in sorted order.

Words with given shape

```
def words_with_given_shape(words, shape):
```

The shape of the given word of length *n* is a list of *n* - 1 integers, each one either -1, 0 or +1 to indicate whether the next letter following the letter in that position comes later (+1), is the same (0) or comes earlier (-1) in the alphabetical order of English letters. For example, the shape of the word `'hello'` is `[-1, +1, 0, +1]`, whereas the shape of `'world'` is `[-1, +1, -1, -1]`. Find and return a list of all `words` that have that particular `shape`, listed in alphabetical order.

Note that your function, same as all the other functions specified in this document that operate on lists of words, should not itself try to read the wordlist file `words_sorted.txt`, even when Python makes this possible with just a couple of lines of code. The tester script already reads in the entire wordlist and builds the list of words from there. Your function should use this given list of `words` without even caring which particular file it came from.

shape	Expected result (using wordlist words_sorted.txt)
`[1, -1, -1, -1, 0, -1]`	`['congeed', 'nutseed', 'outfeed', 'strolld']`
`[1, -1, -1, 0, -1, 1]`	`['axseeds', 'brogger', 'cheddar', 'coiffes', 'crommel', 'djibbah', 'droller', 'fligger', 'frigger', 'frogger', 'griffes', 'grogger', 'grommet', 'prigger', 'proffer', 'progger', 'proller', 'quokkas', 'stiffen', 'stiffer', 'stollen', 'swigger', 'swollen', 'twiggen', 'twigger']`
`[0, 1, -1, 1]`	`['aargh', 'eeler', 'eemis', 'eeten', 'oopak', 'oozes', 'sstor']`
`[1, 1, 1, 1, 1, 1]`	`['aegilops']`

Motivated students can take on as an extra challenge to find the shape of length *n* - 1 that matches the largest number of words, for the possible values of *n* from 3 to 20. Alternatively, try to count how many possible shapes of length *n* - 1 do not match any words of length *n* at all. What is the shortest possible shape that does not match any words?

Chirality

```
def is_left_handed(pips):
```

Even though this has no effect on fairness, pips from one to six are not painted on dice just any which way, but so that the [pips on the opposite faces always add up to seven](). In each of the $2^3 = 8$ corners of the cube, one value from each opposite pair 1-6, 2-5 and 3-4 meets two other values from each of the other two opposite pairs.

This still leaves two possibilities for painting these pips. Look at the corner shared by the faces 1, 2, and 3, and read these numbers clockwise around that corner starting from 1. If they read out as 123, that die is **left-handed**, and if they read out as 132, that die is **right-handed**. The three numbers read around any other corner of the die determine the three numbers in the unseen opposite side, and therefore also the handedness of the entire die just as firmly. Analogous to a pair of shoes separately made for the left and right foot, left- and right-handed dice are in one sense identical, yet no matter how you twist and turn, you can't change either one to match the other, at least not without taking that three-dimensional pancake "Through the Looking-Glass" to flip it around in the fourth dimension!

Given the three-tuple of `pips` around a corner read in clockwise order, determine whether that die is left-handed. There are only $2^3*3! = 8*6 = 48$ possible pip combinations to test for, but you should find and cheerily exploit as many of the ample symmetries in this problem as you can.

pips	Expected result
(1, 2, 3)	True
(1, 3, 5)	True
(5, 3, 1)	False
(6, 3, 2)	True
(6, 5, 4)	False

After solving that, imagine that our physical space had k dimensions, instead of merely just the familiar three. How would dice even be cast (in both senses of this word) in k dimensions? How would you generalize your function to find the chirality of an arbitrary k-dimensional die?

The card that wins the trick

```
def winning_card(cards, trump=None):
```

Playing cards are again represented as tuples of (rank, suit) as in the cardproblems.py lecture example program. In **trick taking** games such as **whist** or **bridge**, four players each play one card from their hand to the trick, committing to their play in clockwise order starting from the player who plays first into the trick. The winner of the trick is determined by the following rules:

1. If one or more cards of the trump suit have been played to the trick, the trick is won by the highest ranking trump card, regardless of the other cards played.
2. If no trump cards have been played to the trick, the trick is won by the highest card of the suit of the first card played to the trick. Cards of any other suits, regardless of their rank, are powerless to win that trick.
3. Ace is the highest card in each suit.

Note that the order in which the cards are played to the trick greatly affects the outcome of that trick, since the first card played in the trick determines which suits have the potential to win the trick in the first place. Return the winning card for the list of cards played to the trick.

cards	trump	Expected result
[('three', 'spades'), ('ace', 'diamonds'), ('jack', 'spades'), ('eight', 'spades')]	None	('jack', 'spades')
[('ace', 'diamonds'), ('ace', 'hearts'), ('ace', 'spades'), ('two', 'clubs')]	'clubs'	('two', 'clubs')
[('two', 'clubs'), ('ace', 'diamonds'), ('ace', 'hearts'), ('ace', 'spades')]	None	('two', 'clubs')

Boustrophedon

```
def create_zigzag(rows, cols, start=1):
```

This function creates **a list of lists** that represents a two-dimensional **grid** with the given number of `rows` and `cols`. This grid should contain the integers counting the `rows*cols` numbers from `start` in ascending order. However, similar to the way an ox slowly plows a field somewhere back in ancient Greece, the elements of every odd-numbered row must be listed in descending order. However you choose to enforce that discipline is up to you.

In all of the examples of the following table, the keyword parameter `start` is not given and therefore equals one. Your function must still work correctly for arbitrary values of `start`.

rows	cols	Expected result
3	5	`[[1, 2, 3, 4, 5], [10, 9, 8, 7, 6], [11, 12, 13, 14, 15]]`
10	1	`[[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]]`
4	2	`[[1, 2], [4, 3], [5, 6], [8, 7]]`

Multidimensional knight moves

```
def knight_jump(knight, start, end):
```

An ordinary [chess knight](#) on a two-dimensional board of squares can make an "L-move" into up to eight possible neighbours. However, we can generalize the entire chessboard into k dimensions from just the puny two. A natural extension of the knight's move to keep moves symmetric with respect to these dimensions is to define the possible moves as some k-tuple of **strictly decreasing** nonnegative integer offsets. Each one of these k offsets must be used for exactly one dimension of your choice during the move, either as a positive or a negative version.

For example, the three-dimensional `(4, 3, 1)`-knight makes its way by first moving four steps along any one of the three dimensions, then three steps along any other dimension, and then one step along the remaining dimension, whichever dimensions that was. Once the knight has stepped along some dimension, it can no longer step along that same dimension within the same move. These steps are considered to be performed together as a single jump that does not visit or is blocked by any of the intermediate squares. Given the `start` and `end` positions as k-tuples of integer coordinates, determine whether the `knight` can get from `start` to `end` in one jump.

knight	start	end	Expected result
(2, 1)	(12, 10)	(11, 12)	True
(7, 5, 1)	(15, 11, 16)	(8, 12, 11)	True
(9, 7, 6, 5, 1)	(19, 12, 14, 11, 20)	(24, 3, 20, 11, 13)	False

A quick combinatorial calculation reveals that exactly $k! * 2^k$ possible neighbours are reachable in a single move, assuming that none of these moves takes the knight outside the board. In this notation, the ordinary chess knight is a `(2, 1)`-knight that can reach $2! * 2^2 = 2 * 4 = 8$ possible neighbouring squares in one jump. A 6-dimensional knight reaches a whopping $6! * 2^6 = 46080$ different neighbours in one jump. Since the number of moves emanating from each position to its neighbours grows exponentially with respect to k, pretty much everything ends up being close to pretty much everything else in high-dimensional spaces. (Density in general is known to have both advantages and disadvantages in all walks of life.)

Sevens rule, zeros drool

```
def seven_zero(n):
```

Seven is considered a lucky number in Western cultures, whereas [zero is what nobody wants to be](). We bring these two opposites briefly together with positive integers that consist of some solid sequence of sevens, followed by some (possibly empty) solid sequence of zeros. Examples of such numbers are 7, 77777, 7700000, 77777700, or 700000000000000. A surprising theorem proves that for any positive integer n, there exist infinitely many integers of such seven-zero form that are divisible by n. This function should return the **smallest** such seven-zero integer.

This exercise is about efficiently generating all numbers of the constrained form of sevens and zeros in strictly ascending order to guarantee finding the smallest working such number. This logic might be best written as a **generator** to `yield` such numbers. The body of this generator consists of two nested loops. The outer loop iterates through the number of digits d in the current number. For each d, the inner loop iterates through all possible k from one to d to create a number that begins with a block of k sevens, followed by a block of d-k zeros. Most of the work done inside that helper generator, the `seven_zero` function itself is short and sweet.

n	Expected result
17	7777777777777777
42	7770
103	7777777777777777777777777777777
77700	77700
2**50	700000000000000000000000000000000000000000000000000
12345	(a behemoth that consists of a total of 822 sevens, followed by a single zero)

This problem is adapted from the excellent [MIT Open Courseware]() online textbook "*Mathematics for Computer Science*" ([PDF link]() to the 2018 version for anybody interested) that, like so many other **non-constructive** combinatorial proofs, uses the [pigeonhole principle]() to prove that *some* solution must exist for any integer n, but provides no clue about where to find that solution. Also as proven in that same textbook, whenever n is not divisible by either 2 or 5, the smallest such number will always consist of some solid sequence of sevens with no zero digits after them. This can speed up the search by an order of magnitude for such friendly values of n.

Fulcrum

```
def can_balance(items):
```

Each element in `items` is considered to be a [physical weight](#), and guaranteed to be a positive integer. Your task is to find and return a **fulcrum** position in this list of weights so that when balanced on that position, the total [torque](#) of the items to the left of that position equals the total torque of the items to the right of that position. The item on the fulcrum is assumed to be centered symmetrically on the fulcrum, and does not participate in the torque calculation.

In physics, the torque of an item with respect to the fulcrum equals its weight times distance from the fulcrum. If a fulcrum position exists, return that position, otherwise return -1 to indicate that the given `items` cannot be balanced, at least not without rearranging them.

items	Expected result
`[6, 1, 10, 5, 4]`	2
`[10, 3, 3, 2, 1]`	1
`[7, 3, 4, 2, 9, 7, 4]`	-1
`[42]`	0

The problem of finding the fulcrum position when rearranging elements is allowed would be an interesting but a far more advanced problem normally suitable for a third year computer science course. However, this algorithm could be built in an effective (albeit not as efficient) **brute force** fashion around this function by using the generator `permutations` in the Python standard library module [`itertools`](#) to try out all possible permutations in an outer loop until you find one permutation that works. In fact, quite a few problems of this style can be solved with this "**generate and test**" approach without the **backtracking** algorithms from third year courses.

(Yes, I pretty much wrote this problem only to get to say "fulcrum". What a cool word. And you know what is another really cool word? "[Phalanx](#)". That one even seems like something that could be turned into an interesting computational problem about lists of lists... and that is the *crux* of that entire matter.)

At least fail while daring greatly

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

Marred by dust and sweat and blood, the ancient world back "♪♪♪ when men were made of iron and their ships were made of wood ♪♪♪" was an entertainingly violent place, at least if we believe "300", "Spartacus", "Game of Thrones" and similar [historical docudramas](#) of swords and sandals. During [one particularly memorable incident](#), a group of extremist [zealots](#) (yes, Lana, *literally*) found itself surrounded by overwhelming Roman forces. To avoid capture and death by crucifixion that would have been as humiliating as it was slow, these men chose to commit mass suicide in a scheme that prevented any one of them from changing his mind. The zealots arranged themselves in a circle and used lots to choose a step size k. Starting from the first man, they repeatedly count k men ahead and quickly kill that man, removing his corpse from this grim circle. Being [normal people instead of computer scientists](#), they always start counting from one, not zero. (The concept of zero was unknown back then anyway.)

This deadly game of eeny-meeny-miney-moe continues until only one man remains to fall on his own sword to join his brothers. [Josephus](#) would very much prefer to be this last man since he has other ideas of surviving. Help him and his secret confederate survive with a function that, given n and k, returns the list of the execution order so that these men know which places let them be the survivors who get to walk away from this grim circle. There is a [cute mathematical solution](#) to determine the survivor position for *k* = 2. Unfortunately *k* can now be arbitrarily large, if only to make this game more exciting for us cold and timid souls [who point out how strong men stumble](#).

n	k	Expected result
4	1	[1, 2, 3, 4]
4	2	[2, 4, 3, 1]
4	999999	[3, 2, 4, 1]
10	3	[3, 6, 9, 2, 7, 1, 8, 5, 10, 4]
8	7	[7, 6, 8, 2, 5, 1, 3, 4]
30	4	[4, 8, 12, 16, 20, 24, 28, 2, 7, 13, 18, 23, 29, 5, 11, 19, 26, 3, 14, 22, 1, 15, 27, 10, 30, 21, 17, 25, 9, 6]
1000	99	(a sequence of 1000 elements whose first five elements are [99, 198, 297, 396, 495] and last five elements are [183, 762, 380, 966, 219])

Longest palindrome substring

```
def longest_palindrome(text):
```

A string is said to be a **palindrome** if it reads the same both forward and backward, for example `'racecar'`. Given `text`, find and return the longest consecutive substring inside `text` that is a palindrome. If multiple palindromes exist with the same largest length, return the leftmost one.

Note that since you are looking for the longest palindrome, of course you should loop through these substrings in descending order of length, and the substrings of same length should be looped through left to right. This way you can simply return the first palindrome substring that you find, safe in the knowledge that the `text` does not contain any longer palindromic substrings.

text	Expected result
`'saippuakauppias'`	`'saippuakauppias'`
`'abaababaaabbabaababababaa'`	`'aababababaa'`
`'xxzxxracecar'`	`'racecar'`
`'xyxracecaryxy'`	`'racecar'`

The straightforward implementation of the trivial algorithm "loop through all possible substrings and check which ones are palindromes, remembering the longest palindrome that you have seen" should pass the test in a reasonably short time. However, as a curiosity, the real challenge in this problem is making this function at least an order of magnitude faster than this, which is why you might meet this one in a later course on algorithmic techniques. The students who are already piqued in this kind of algorithmic tweaking might want to check out the Wikipedia page "[Longest Palindromic Substring](#)", as the **dynamic programming** technique used to solve this problem is a key to solving a thousand other problems of similar combinatorial nature of **overlapping recursive subproblems**.

All your fractions are belong to base

```
def group_and_skip(n, out, ins):
```

A pile of `n` identical coins lies on the table. In each move, the coins in the currently remaining pile are arranged into groups of exactly `out` coins each, where `out` is a positive integer greater than one. The `n%out` leftover coins that did not make a complete group of `out` elements are taken aside and recorded. From each complete group of `out` coins taken out, exactly `ins` coins are put back into the pile, and the rest of the coins of that group are discarded aside. Repeat this until the pile becomes empty, which must eventually happen whenever `out>ins`. Return a list that tells how many coins were taken aside in each step.

n	out	ins	Expected result
123456789	10	1	[9, 8, 7, 6, 5, 4, 3, 2, 1]
987654321	1000	1	[321, 654, 987]
255	2	1	[1, 1, 1, 1, 1, 1, 1, 1]
81	5	3	[1, 3, 2, 0, 4, 3]
10**9	13	3	[12, 1, 2, 0, 7, 9, 8, 11, 6, 8, 10, 5, 8, 3]

As you can see in the first three rows, this method produces the digits of n in base `out` in reverse order. So this entire setup turned out to be the cleverly disguised algorithm to construct the representation of integer n in base `out`. However, an improvement over the standard algorithm for such base conversion is that this version works not only for integer bases, but allows any fraction `out/ins` that satisfies `out>ins` and `gcd(out,ins)==1` to be used as a base! For example, the familiar integer 42 would be written as 323122 in base 4/3.

Yes, fractional bases are an actual thing. Take a deep breath to think about the implications, and then imagine trying to do your real world basic arithmetic in such a system. That certainly would have been some "[New Math](#)" for the frustrated parents in the sixties for whom balancing their chequebooks in the familiar base ten was already an exasperating ordeal!

Recamán's sequence

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

Compute and return the first n terms of the [Recamán's sequence](#), starting from the term $a_1 = 1$. See the linked definition for this sequence as it is defined on [Wolfram Mathworld](#).

Note how the definition for the current next element depends on whether a particular number is already part of the previously generated part of the sequence. To let your function execute in a speedy fashion even when generating a sequence that contains millions of elements, you should use a `set` to keep track of which values are already part of the previously generated sequence. This way you can generate each element in constant time, instead of having to iterate through the entire previously generated list like some "[Shlemiel](#)" would have done. The better technique that saves time by using more memory can create this list arbitrarily far in linear time, and should therefore be fast even for millions of elements, at least until it runs out of memory.

n	Expected result
10	`[1, 3, 6, 2, 7, 13, 20, 12, 21, 11]`
1000000	(a list of million elements whose last five elements are `[2057162, 1057165, 2057163, 1057164, 2057164]`)

Count the balls off the brass monkey

```
def pyramid_blocks(n, m, h):
```

Mysteries of the pyramids have fascinated humanity through the ages. Instead of packing your machete and pith helmet to trek through deserts and jungles to raid the hidden treasures of the ancients like Indiana Croft, or by gaining visions of enlightenment by intensely meditating under the apex set over a square base like Deepak Veidt, this problem deals with truncated [brass monkeys](#) of layers of discrete uniform spheres, in spirit of that [spherical cow running in a vacuum](#). Given that the *top* layer of the truncated brass monkey consists of n rows and m columns of spheres, and each solid layer immediately below the one above it always contains one more row and one more column, how many spheres in total make up this truncated brass monkey that has h layers?

This problem could be solved in a straightforward **brute force** fashion by mechanistically tallying up the spheres through these layers. However, the fair and just reward for such a blunt approach is that you get to watch the automated tester take roughly a minute to terminate! Some creative use of discrete math and summation formulas gives an **analytical closed form formula** that makes the answers come out pretty much immediately by plugging the values of n, m and h into that formula. (Seeing that the brass monkey is a discrete pyramidal shape, surely the [pyramidal numbers](#) might be of some assistance here in getting started...)

n	m	h	Expected result
2	3	1	6
2	3	10	570
10	11	12	3212
100	100	100	2318350
10**6	10**6	10**6	2333331833333500000

Count growlers

```
def count_growlers(animals):
```

Let the strings `'cat'` and `'dog'` denote that kind of animal facing left, and `'tac'` and `'god'` denote that same kind of animal facing right. Each individual animal, regardless of its species, growls if it sees more dogs than cats to the direction that the animal is facing. Given a list of such `animals`, return the count of how many of these animals are growling.

animals	Expected result
`['cat', 'dog']`	0
`['god', 'cat', 'cat', 'tac', 'tac', 'dog', 'cat', 'god']`	2
`['dog', 'cat', 'dog', 'god', 'dog', 'god', 'dog', 'god', 'dog', 'dog', 'god', 'god', 'cat', 'dog', 'god', 'cat', 'tac']`	11
`['god', 'tac', 'tac', 'tac', 'tac', 'dog', 'dog', 'tac', 'cat', 'dog', 'god', 'cat', 'dog', 'cat', 'cat', 'tac']`	0

I admit that I was high as a kite back when I originally thought up this problem, at least high enough to perceive the letter `'t'` as a tail of a happy cat held up high, and `'d'` as the snout and stand-up ears of a curious dog, perhaps some kind of spitz or a similar breed. (Yeah, good luck trying to un-see that now.) And yet for some reason, this problem is somehow more tricky than it would initially seem in a fair, predictable and just world, and does not seem to conveniently allow itself to be twisted into a clean and Pythonic solution. Readers are welcome to suggest their best solutions to the author, should they come up with nifty ways to solve this problem.

Bulgarian solitaire

```
def bulgarian_solitaire(piles, k):
```

You are given a row of `piles` of pebbles, all identical, and a positive integer `k` so that the total number of pebbles in these piles equals `k*(k+1)//2`, the **arithmetic sum** formula that equals the sum of positive integers from `1` to `k`. Almost as if intended as a metaphor for the bleak daily life behind the Iron Curtain, all pebbles are identical and you don't have any choice in your moves. Each move picks up one pebble from each pile (even if doing so makes that pile vanish), and creates a new pile from the resulting handful. For example, starting with `[7, 4, 2, 1, 1]`, the first move turns this into `[6, 3, 1, 5]`. The next move turns that into `[5, 2, 4, 4]`, which then turns into the five-pile configuration `[4, 1, 3, 3, 4]`, and so on.

This function should count how many moves are needed to convert the given initial `piles` into the **goal state** where each number from `1` to `k` appears as the size of **exactly one pile**, and return that count. These numbers from `1` to `k` may appear in any order inside the list, not necessarily in sorted order. Applying the basic move to this goal state simply leads right back in that same goal state. (In mathematical lingo, the goal state is a **fixed point** of the transformation function between these configurations of pebbles, so this process gets stuck in that state forever once it falls into it.)

piles	k	Expected result
`[1, 4, 3, 2]`	4	0
`[8, 3, 3, 1]`	5	9
`[10, 10, 10, 10, 10, 5]`	10	74
`[3000, 2050]`	100	7325
`[2*i-1 for i in range(171, 0, -2)]`	171	28418

This problem hails from the Martin Gardner column "Bulgarian Solitaire and Other Seemingly Endless Tasks". It was used there as an example of a while-loop where it is not immediately obvious that this loop will always eventually reach its goal and terminate, analogous to the behaviour of the Collatz 3x+1 problem back in `mathproblems.py`. However, unlike in that still annoyingly open problem, a combinatorial proof shows that Bulgarian solitaire can never get stuck, but must terminate from any starting configuration of pyramidal numbers after at most $k(k$-1$)/2$ steps.

Scylla or Charybdis?

```
def scylla_or_charybdis(moves, n):
```

This problem was inspired by the article "[A Magical Answer to the 80-Year-Old Puzzle](#)" in [Quanta Magazine](#). Thanks to your recent cunning stunts, your [nemesis](#) in life is trapped inside a devious game where, for a refreshing change from the usual way of things, you get to be the Final Boss. (Everyone is the hero in their own story until they become a villain in somebody else's, after all.) This final level is a one-dimensional video game platform that reaches n-1 steps from its center to both directions. At each end of this platform exactly n steps away from the center, your two lethal friends [Scylla and Charybdis](#) are licking their lips in anticipation of a tasty morsel.

Your nemesis starts at the center of this platform. To start this deadly combat of wits, she must immediately commit to her entire sequence of future `moves`, given to this function as a string of characters '+' ("♪♪ just a step to the ri-i-i-i-ight ♪♪♪") and '-' (move one step to the left). Your task is to find and return a positive integer k so that executing every k:th step of `moves` (so this subsequence starts from position k-1 and includes every k:th element from then on) makes your nemesis fall into one of the two possible dooms n steps away from the center to each direction.

If several values of k would work, return the smallest value of the possible values of k that minimizes the number of steps before the downfall. Otherwise, those freaking Tolkien eagles will once again swoop down to rescue your nemesis. She will then return in a third year analysis of algorithms course as an **adversary** who puts your functions through a similar wringer of wits!

moves	n	Expected result
`'-++--++-++++'`	2	3
`'--++++---+-++-+++++++++++'`	5	5
`'+-++---+-+-++-+++---+++-++-+---+--+++++++++'`	5	7
`'+--++---+-++-+++------++++---++-+--+++-+--++-++-++-+-++++++++++++++++++'`	9	1

Arithmetic progression

```
def arithmetic_progression(items):
```

An **arithmetic progression** is a numerical sequence so that the **stride** between each two consecutive elements is constant throughout the sequence. For example, `[4, 8, 12, 16, 20]` is an arithmetic progression of length 5, starting from the value 4 with a stride of 4.

Given a non-empty list `items` of positive integers in strictly ascending order, find and return the longest arithmetic progression whose all values exist somewhere in that sequence. Return the answer as a tuple `(start, stride, n)` of the values that define the progression. To ensure unique results to facilitate automated testing, if there exist several progressions of the same length, this function should return the one with the lowest `start`. If several progressions of equal length emanate from the lowest `start`, return the progression with the smallest `stride`.

`items`	Expected result
`[42]`	`(42, 0, 1)`
`[2, 4, 6, 7, 8, 12, 17]`	`(2, 2, 4)`
`[1, 2, 36, 49, 50, 70, 75, 98, 104, 138, 146, 148, 172, 206, 221, 240, 274, 294, 367, 440]`	`(2, 34, 9)`
`[2, 3, 7, 20, 25, 26, 28, 30, 32, 34, 36, 41, 53, 57, 73, 89, 94, 103, 105, 121, 137, 181, 186, 268, 278, 355, 370, 442, 462, 529, 554, 616, 646, 703]`	`(7, 87, 9)`
`range(1000000)`	`(0, 1, 1000000)`

(Bridge players like to distinguish between "best result possible" and "best possible result", which are not at all the same thing. In the same spirit, can you see the difference between two deceptively similar concepts of "leftmost of longest" and "longest of leftmost" in this problem?)

Tukey's ninther

```
def tukeys_ninthers(items):
```

Back in the day when computers were far slower and had a lot less RAM for our programs to burrow into, special techniques were necessary to achieve many [things that are trivial today with a couple of lines of code](). In this spirit, "[Tukey's ninther]()" is an **approximation algorithm** from the seventies to quickly find some value that should be "reasonably close" to the **median element** of the given unsorted list. For the purposes of this problem, the median element of the list is defined to be the element that would end up in the middle position if that list were sorted. This makes the median unambiguous, regardless of the elements and their multiplicities. This function is not tasked to find the true median, which would be a trivial one-liner by sorting `items`, but find and return the same element that Tukey's ninther algorithm would return for those `items`.

Tukey's algorithm splits the list into triplets of three elements, and finds the median element for each triplet. These medians-of-three are then collected into a new list that this same operation is applied to, until only one element remains. For simplicity, your function can assume that the length of `items` is always some power of three. In the following table, each row contains the result produced by applying one round of Tukey's algorithm to the list in the row immediately below.

`items`	Expected result
`[15]`	15
`[42, 7, 15]`	15
`[99, 42, 17, 7, 1, 9, 12, 77, 15]`	15
`[55, 99, 131, 42, 88, 11, 17, 16, 104, 2, 8, 7, 0, 1, 69, 8, 93, 9, 12, 11, 16, 1, 77, 90, 15, 4, 123]`	15

Tukey's algorithm is extremely **robust**, as can be appreciated by giving it a bunch of randomly shuffled lists of distinct numbers to operate on, and plotting the histogram of results to admire how heavily centered around the actual median this distribution ends up being. (These distinct numbers could come from distributions over arbitrary scales, since this **comparison-based** algorithm never performs any arithmetic between elements.) For example, the median of the last example list in the above table is really 15, pinky swear for grownup realsies. Even better, if all `items` are distinct and the length of the list is some power of three, the returned result can never be from the true top or bottom third of the sorted elements, thus eliminating all risk of using some funky outlier as the approximation for the true median.

Whenever they zig, you gotta zag

```
def is_zigzag(n):
```

A positive integer n is called a **zigzag number** (also called "alternating number" in some combinatorics material) if the series of differences between its consecutive digits strictly alternates between positive and negative steps. The step from the first digit to the second may be either positive or negative. The function should determine whether n is a zigzag number.

In the negative examples in the table below, the part of the number that violates the zigzag property is highlighted in red.

n	Expected result
7	True
25391	True
90817263545463728185	True
16329	False
10417510101096715	False
49573912009	False

Crack the crag

```
def crag_score(dice):
```

Crag (see [the Wikipedia page](#) for the scoring table needed in this problem) is a dice game similar to the more popular games of [Yahtzee](#) and [Poker dice](#) in style and spirit, but with much simpler combinatorics of roll value calculation due to this game using only three dice. Players repeatedly roll three dice and assign the resulting patterns to **scoring categories** so that once some roll has been assigned to a category, that category is considered to have been spent and cannot be used again for any future roll. These tactical choices between safety and risk-taking give this game a little bit more tactical flair on top of merely relying on the favours of Lady Luck for rolling the bones.

Given the list of pips of the three dice of the first roll, this function should compute and return the highest possible score available when **all categories of the scoring table are still available for you to choose from**, so that all that matters is maximizing this first roll. Note that the examples on the Wikipedia page show the score that some dice would score **in that particular category**, which is not necessarily even close to the maximum score in principle attainable with that roll. For example, the roll [1, 1, 1] inefficiently used in the category "Ones" would indeed score only three points, whereas that same roll would score a whopping 25 points in the more fitting category "Three of a kind". (The problem "Optimal crag score" later in this collection then has you distribute your rolls into distinct categories to maximize the total score.)

This problem ought to be a straightforward exercise on if-else ladders combined with simple sequence management. Your function should be swift and sure to return the correct answer for every one of the $6^3 = 216$ possible pip combinations. However, you will surely design your if-else structures to handle entire equivalence classes of pip combinations in a single step, so that your entire ladder consists of *far* fewer than 216 separate steps...

dice	Expected result
[1, 2, 3]	20
[4, 5, 1]	5
[3, 3, 3]	25
[4, 5, 4]	50
[1, 1, 1]	25
[1, 1, 2]	2

Three summers ago

```
def three_summers(items, goal):
```

Given a list of positive integer `items` guaranteed to contain at least three elements with all of its **elements in sorted ascending order**, determine whether there **exist precisely three** separate `items` that together add up to the given positive integer `goal`, no more and no less.

You could, of course, solve this problem with three nested loops to go through all possible ways to choose three elements from `items`, checking for each triple whether it adds up to the `goal`. However, this approach would get rather slow as the number of elements in the list increases, and of course the automated tester used to grade this function will make those lists larger just to make such solutions reveal themselves with their excessive consumption of running time.

Since `items` are known to be sorted, better technique will find the answer significantly faster. See the new example function `two_summers` in listproblems.py to quickly find two elements from the given sorted list that together add up to the given `goal`. You can simply use this function as a subroutine to speed up your search for three summing elements, once you realize that the list contains three elements that add up to `goal` if and only if it contains some element `x` so that the remaining list contains two elements that add up to `goal-x`.

items	goal	Expected result
[10, 11, 16, 18, 19]	40	True
[10, 11, 16, 18, 19]	41	False
[1, 2, 3]	6	True

For the general **subset sum problem** used as an example of inherently exponential branching recursion in that lecture, the question of whether the given list of integers contains some subset of k elements that together add up to given `goal` can be determined by trying each element `x` in turn as the first element of this subset, and then recursively determining whether the remaining elements after `x` contain some subset of k - 1 elements that adds up to `goal-x`.

Sum of two squares

```
def sum_of_two_squares(n):
```

Some positive integers can be expressed as a sum of two squares of some positive integers greater than zero. For example, $74 = 49 + 25 = 7^2 + 5^2$. This function should find and return a tuple of two positive integers whose squares together add up to n, or return `None` if the parameter n cannot be broken into a sum of two squares.

To facilitate the automated testing, the returned tuple must present the larger of its two numbers first. Furthermore, if some integer can be broken down to a sum of squares in several ways, return the breakdown that maximizes the larger number. For example, the number 85 allows two such representations $7^2 + 6^2$ and $9^2 + 2^2$, of which this function must return `(9, 2)`.

The technique of **two approaching indices** that start from the beginning and end of a sequence, respectively, and approach each other until they meet somewhere, used in the function `two_summers` in one of our class examples, is directly applicable to this problem. In this problem, these indices crawl towards each other only one small step at the time, whereas in **binary search**, one index would jump halfway towards the other one in every round.

n	Expected result
1	None
2	(1, 1)
50	(7, 1)
8	(2, 2)
11	None
123**2 + 456**2	(456, 123)
55555**2 + 66666**2	(77235, 39566)

Carry on Pythonista

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

Two positive integers `a` and `b` can be added together to produce their sum with the usual integer column-wise addition algorithm that we all learned back when we were but wee little children. Instead of the sum `a+b` that the language could compute for you anyway, this problem asks you to count how many times there is a **carry** of one into the next column caused by adding the two digits in the current column (possibly including the carry from the previous column), and return that total count. Your function should be efficient even when the parameter integers `a` and `b` are enormous enough to require thousands of digits to write down.

To extract the lowest digit of a positive integer n, use the expression `n%10`. To extract all other digits except the lowest one, use the expression `n//10`. You can use these simple integer arithmetic operations to execute the steps of the **column-wise integer addition** so that you don't care about the actual result of the addition, but only the carry produced in each column.

a	b	Expected result
99999	1	5
11111111111	2222222222	0
123456789	987654321	9
2**100	2**100 - 1	13
3**1000	3**1000 + 1	243

Expand positive integer intervals

```
def expand_intervals(intervals):
```

An **interval** of consecutive positive integers can be succinctly described as a string that contains its first and last value, inclusive, separated by a minus sign. (This problem is restricted to positive integers so that there can be no ambiguity between the minus sign character used as a separator and an actual unary minus sign in front of an integer.) For example, the interval that contains the numbers 5, 6, 7, 8, 9 could be more concisely described as `'5-9'`. Multiple intervals can be described together by separating their descriptions with commas. An interval that contains only one value is given as only that value.

Given a string that contains one or more such comma-separated interval descriptions, guaranteed to be given in sorted ascending order and never overlap with each other, create and return the list that contains all the integers contained inside these intervals. In solving this problem the same as any other problems, it is always better to not have to reinvent the wheel, but always check out first whether the string objects have useful methods to make your job easier...

intervals	Expected result
`''`	`[]`
`'42'`	`[42]`
`'4-6,10-12,16'`	`[4, 5, 6, 10, 11, 12, 16]`
`'1,3-9,12-14,9999'`	`[1, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 9999]`

Collapse positive integer intervals

```
def collapse_intervals(items):
```

This function is the inverse of the previous question of expanding positive integer intervals. Given a nonempty list of positive integers that is guaranteed to be in sorted ascending order, create and return the unique description string where every maximal sublist of consecutive integers has been condensed to the notation `first-last`. If some maximal sublist consists of a single integer, it must be included in the result string just by itself without the minus sign separating it from the now redundant `last` number. Make sure that the string returned by your function does not contain any whitespace characters, and does not have a redundant comma at the end.

items	Expected result
`[1, 2, 4, 6, 7, 8, 9, 10, 12, 13]`	`'1-2,4,6-10,12-13'`
`[42]`	`'42'`
`[3, 5, 6, 7, 9, 11, 12, 13]`	`'3,5-7,9,11-13'`
`[]`	`''`
`range(1, 1000001)`	`'1-1000000'`

Reversing the deserver

```
def reverse_reversed(items):
```

Create and return a new list that contains the `items` in reverse, but so that whenever each item is itself a list, its elements are also reversed. This reversal of sublists must keep going on all the way down, no matter how deep the nesting of these lists, so you ought to use recursion to solve this problem. The base case handles any argument that is not a list. When the `items` parameter is a list (use the Python function `type` or the `isinstance` operator to check this), recursively reverse the elements that this nested list contains. (List comprehensions might come handy in doing this part of the problem.)

Note that this function must create and return a new list to represent the result, and should not rearrange or otherwise mutate the contents of the original list. This same principle should apply to all functions that receive a list argument. In your own programs that compute some important list that you pass to some function from the standard library, isn't life easier when you never have to worry of that function silently mutating the contents of your list?

items	Expected result
`[1, [2, 3, 4, 'yeah'], 5]`	`[5, ['yeah', 4, 3, 2], 1]`
`[[[[[[1, 2]]]]]]`	`[[[[[[2, 1]]]]]]`
`[42, [99, [17, [33, ['boo!']]]]]`	`[[[[['boo!'], 33], 17], 99], 42]`

Count word dominators

```
def count_word_dominators(words):
```

If you already solved the earlier `count_dominators` problem, you might notice that even though the problem was originally stated for lists of integers, the logic of domination did not depend on this fact in any way. As long as the individual elements can be compared with each other for order, the Pythonic spirit of **duck typing** allows the very same `count_dominators` function to handle a list of strings just as smoothly as it would handle a list of integers! For example, the function call `count_dominators(['dog','emu','cat','bee'])` would return 3, since `'emu'`, `'cat'` and `'bee'` dominate all words coming after them in the list when using the **lexicographic order** comparison. If your `count_dominators` does not pass this hurdle, try to rewrite it to contain no baked-in assumptions about elements being specifically integers.

However, things become more interesting if we define domination between words of equal length with a rule that says that for a word to dominate another word, for more than half of the positions the character in the first word is **strictly greater** than the corresponding character in the other word. Note also the intentional wording "*more* than half" to break ties between words of even length such as `'aero'` and `'tram'`, so that no two words can ever both dominate each other.

words	Expected result
['sky', 'yat']	2
['pun', 'ean', 'fiz', 'coe']	3
['toph', 'prow', 'koku', 'okey']	2
['ufo', 'hny', 'hun', 'ess', 'kab']	3
['cagit', 'libri', 'sured', 'birls', 'golgi', 'shank', 'bailo', 'senex', 'cavin', 'ajiva', 'babby']	5

Duplicate digit bonus

```
def duplicate_digit_bonus(n):
```

Some people ascribe deep significance to numerical coincidences, so that consecutive repeated digits or other low description length patterns, such as a digital clock blinking 11:11, seem special and personal to them. Such people then find numbers with repeated digits to be more valuable and important than all the ordinary and pedestrian numbers without any obvious pattern. For example, getting inside a taxicab flashing an exciting number such as 1234 or 6969 would be far more instagrammable than a more pedestrian taxicab adorned with some dull number such as 1729.

Assume that some such person assign a meaningfulness score to every positive integer so that every maximal block of `k` consecutive digits with `k>1` scores `10**(k-2)` points for that block. A block of two digits scores one point, three digits score ten points, four digits score a hundred points, and so on. However, just to make this more interesting, there is also a special rule in effect that if this block of digits is at the lowest end of the number, that block scores twice as many points as it would in any other position. Given a positive integer `n>0`, this function should compute and return its meaningfulness score as the sum of its individual block scores.

n	Expected result
43333	200
2223	10
777777777	20000000
3888882277777731	11001
211111174711111177777700	12002
999999777774444488872222	21210
1234**5678	15418

Nearest smaller element

```
def nearest_smaller(items):
```

Given a list of integer `items`, create and return a new list of the same length but where each element has been replaced with the nearest element in the original list whose value is smaller. If no smaller elements exist because that element is the minimum of the original list, the element in the result list should remain as that same minimum element.

If there exist smaller elements equidistant in both directions, you must resolve this by using the smaller of these two elements. This again makes the expected results unique for every possible value of `items`, which is necessary for the automated testing framework to work at all. [Being permissive in what you accept while being restrictive in what you emit](#) is a pretty good principle to follow in all walks of life, not just in programming.

items	Expected result
[42, 42, 42]	[42, 42, 42]
[42, 1, 17]	[1, 1, 1]
[42, 17, 1]	[17, 1, 1]
[6, 9, 3, 2]	[3, 3, 2, 2]
[5, 2, 10, 1, 13, 15, 14, 5, 11, 19, 22]	[2, 1, 1, 1, 1, 13, 5, 1, 5, 11, 19]
[1, 3, 5, 7, 9, 11, 10, 8, 6, 4, 2]	[1, 1, 3, 5, 7, 9, 8, 6, 4, 2, 1]

Interesting, intersecting

```
def squares_intersect(s1, s2):
```

A square on the two-dimensional plane can be defined as a tuple `(x, y, r)` where `(x, y)` are the coordinates of its **bottom left corner** and `r` is the length of the side of the square. Given two squares as tuples `(x1, y1, r1)` and `(x2, y2, r2)`, this function should determine whether these two squares **intersect**, that is, their **areas** have at least one point in common, even if that one point is merely the shared corner point when these two squares are placed kitty corner. This function **should not contain any loops or list comprehensions of any kind**, but should compute the result using only integer comparisons and conditional statements.

This problem showcases an idea that comes up with some problems of this nature; that it is actually easier to determine that the two squares do **not** intersect, and then just negate that answer. Two squares do not intersect if one of them ends in the horizontal direction before the other one begins, or if the same thing happens in the vertical direction. (This technique generalizes from rectangles lying on the flat two-dimensional plane to not only three-dimensional cuboids, but to hyper-boxes of arbitrary high dimensions.)

s1	s2	Expected result
(2, 2, 3)	(5, 5, 2)	True
(3, 6, 1)	(8, 3, 5)	False
(8, 3, 3)	(9, 6, 8)	True
(5, 4, 8)	(3, 5, 5)	True
(10, 6, 2)	(3, 10, 7)	False
(3000, 6000, 1000)	(8000, 3000, 5000)	False
(5*10**6, 4*10**6, 8*10**6)	(3*10**6, 5*10**6, 5*10**6)	True

Reap what you sowed

```
def oware_move(board, house):
```

The African board game of [Oware](#) is one of the most popular [Mancala variations](#). After appreciating the simple but flexible structure of sowing games that allows such games to be played with minimal equipment that is amply available everywhere on this planet, we might as well generalize the game mechanics of Oware for arbitrary $k > 1$ houses per player and any number of seed. As interesting as the discrete mathematics and game theoretical aspects of these games would be for a later course that deals with such topics, we shall here content ourselves with capturing (heh) the mechanics of this game by simulating the execution of a single move. The **minimax** function that chooses the best move among the legal moves will have to wait until you take CCPS 721, *Artificial Intelligence*.

The `board` is a list with $2k$ elements, the first k representing the houses of the current player and the last k representing those of the opponent. (The stores that keep track of the captured stones do not appear anywhere in this list.) This function should return the outcome of picking up the seed from the given `house` on your side (numbering of houses starts from zero) and sowing them counterclockwise around the `board`. The original `house` is skipped during sowing, should that `house` originally hold enough seed to see this sowing process to come around a full lap.

After sowing, capturing starts from the house that the last seed was sown into, and continues backwards as long as the current house is on the opponent's side and contains two or three seed. To keep this problem simple, we ignore the edge situations that stem from the **grand slam rule** and the gentlemanly requirement of a legal move to leave at least one seed for the opponent.

board	house	Expected result
[0, 2, 1, 2]	1	[0, 0, 0, 0]
[2, 0, 4, 1, 5, 3]	0	[0, 1, 5, 1, 5, 3]
[4, 4, 4, 4, 4, 4, 4, 4]	2	[4, 4, 0, 5, 5, 5, 5, 4]
[10, 10, 10, 10]	0	[0, 14, 13, 13]
[4, 10, 4, 1, 1, 1, 4, 4]	1	[5, 0, 6, 3, 0, 2, 5, 5]
[4, 5, 1000, 1, 2, 3]	2	[204, 205, 0, 201, 202, 203]
[0, 0, 0, 6, 1, 1, 2, 1, 2, 1]	3	[0, 0, 0, 0, 2, 0, 0, 0, 0, 0]

That's enough of you!

```
def remove_after_kth(items, k=1):
```

Given a list of `items`, some of which may be duplicated, create and return a new list that is otherwise the same as `items`, but only up to `k` occurrences of each element are kept, and all occurrences of that element after those first `k` are discarded.

Hint: loop through the `items`, maintaining a dictionary that remembers how many times you have already seen each element. Update this count as you go, and append each element to the `result` list only if its count is still at most equal to `k`.

items	k	Expected result
`[42, 42, 42, 42, 42, 42, 42]`	3	`[42, 42, 42]`
`['tom', 42, 'bob', 'bob', 99, 'bob', 'tom', 'tom', 99]`	2	`['tom', 42, 'bob', 'bob', 99, 'tom', 99]`
`[1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 4, 3, 2, 1]`	1	`[1, 2, 3, 4, 5]`
`[1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5]`	3	`[1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 1, 5]`
`[42, 42, 42, 99, 99, 17]`	0	`[]`

Note the counterintuitive and yet completely legitimate edge case of `k==0` that has the well defined and unambiguously correct answer of an empty list! Once again, an often missed and yet very important part of becoming a programmer is learning to see zero as a number...

Brussel's choice

```
def brussels_choice_step(n, mink, maxk):
```

This problem is adapted from another jovial video "[The Brussel's Choice](#)" of [Numberphile](#) whose first five minutes you might want to watch to get an idea of what is going on. This function should compute and return the list of all numbers that the positive integer `n` can be converted to by treating it as a string and replacing some substring `m` of its digits with the new substring of either `2*m` or `m/2`, the latter substitution allowed only whenever `m` is even so that dividing it by two produces an integer result.

This function should return the list of numbers that can be produced from `n` in a single step. To keep the results more manageable, we also impose an additional constraint that the number of digits in the chosen substring `m` must be between `mink` and `maxk`, inclusive. The returned list must contain the numbers in ascending sorted order.

n	mink	maxk	Expected result
42	1	2	`[21, 22, 41, 44, 82, 84]`
1234	1	1	`[1134, 1232, 1238, 1264, 1434, 2234]`
1234	2	3	`[634, 1117, 1217, 1268, 1464, 1468, 2434, 2464]`
88224	2	3	`[44124, 44224, 84114, 84124, 88112, 88114, 88212, 88248, 88444, 88448, 176224, 176424, 816424,816444]`
123456789	7	9	`[61728399, 111728399, 126913578, 146913569, 146913578, 246913489, 246913569, 246913578]`
123456789	1	9	(a list with 65 elements, the last three of which are `[1234567169, 1234567178, 1234567818]`)

Count consecutive summers

```
def count_consecutive_summers(n):
```

Like a majestic wild horse waiting for someone to come and tame it, positive integers can be broken down as sums of **consecutive** positive integers in various ways. For example, the integer 42 often used as placeholder in this kind of discussions can be broken down into such a sum in four different ways: (a) 3 + 4 + 5 + 6 + 7 + 8 + 9, (b) 9 + 10 + 11 + 12, (c) 13 + 14 + 15 and (d) 42. As the last solution (d) shows, any positive integer can always be trivially expressed as a **singleton sum** that consists of that integer alone. Given a positive integer n, determine how many different ways it can be expressed as a sum of consecutive positive integers, and return that count.

The count of how many different ways a positive integer n can be represented as a sum of consecutive integers is also called its [politeness](#), and can be alternatively computed by counting how many odd divisors that number has. However, note that the linked Wikipedia definition includes only sums that consist of at least two components, so according to their definition, the politeness of 42 equals 3 due to its odd divisors being 3, 7 and 21.

n	Expected result
42	4
99	6
92	2

Powers of two are therefore the least polite of all numbers. As an idle exercise in combinatorics, how would you concisely characterize the "most polite" numbers that have more ways to be represented as sums of consecutive integers than any number lower than them?

McCulloch's second machine

```
def mcculloch(digits):
```

[Esoteric programming languages](#) should be appreciated as works of conceptual performance art. These tar pits are not designed to serve as any sort of practical programming tools, but demonstrate tongue-in-cheek how some deceptively simple mechanism (or merely disguised as simple, with complexity emerging organically from its simple rules) unexpectedly grants its wielder the powers of **universal computation**. Introduced by [Raymond Smullyan](#) in one of his clandestine brain teasers on logic and computability, [McCulloch's second machine](#) is a **string rewriting** system that receives a string of `digits` between one and nine. The rewrite rule applied to the rest of the `digits` (denoted below by X) depends on the first digit. Notice how rule for the leading digit 2 is applied to X itself, whereas rules for leading digits 3 to 5 are applied to Y=mcculloch(X). This function should return `None` whenever X or Y is `None`.

Form of digits	Formula for the expected result	(computational operation)
2X	X	quoting
3X	Y + '2' + Y	concatenation with separator mark
4X	Y[::-1]	reversal
5X	Y + Y	concatenation
anything else	None	N/A

digits	Expected result
'329'	'929'
'53231'	'3123131231'
'4524938'	'83948394'
'343424859355'	'4859355248593552485935524859355'
'433342717866'	'71786627178662717866271786627178662717866271 78662717866'

That's enough for you!

```
def first_preceded_by_smaller(items, k=1):
```

Given a list of items, find and return the value of the first element for which at least k smaller elements precede that element in items. If no such element exists inside the list, this function should return None. The required k smaller items do not need to be consecutive immediate predecessors of the current item, but they can be any k items positioned before the current element.

items	k	Expected result
[4, 4, 5, 6]	2	5
[42, 99, 16, 55, 7, 32, 17, 18, 73]	3	18
[42, 99, 16, 55, 7, 32, 17, 18, 73]	8	None
['bob', 'carol', 'tina', 'alex', 'jack', 'emmy', 'tammy', 'sam', 'ted']	4	'tammy'
[9, 8, 7, 6, 5, 4, 3, 2, 1, 10]	1	10
[42, 99, 17, 3, 12]	2	None

Pull down your neighbours

`def eliminate_neighbours(items):`

Given the sequence of integer `items` that are guaranteed to be some **permutation** of positive integers from 1 to n where n is the length of the list, find the smallest number among those that still remain in the list, and remove from the list both that number and whichever of its current immediate neighbours is larger. The function should repeat this basic operation until the largest number in the original list gets eliminated. Return the number of removal operations that were needed to achieve this goal.

For example, given the list `[5, 2, 1, 4, 6, 3]`, the operation would remove element 1 and its current larger neighbour 4, resulting in the list `[5, 2, 6, 3]`. Applied again, that operation would remove 2 and its current larger neighbour 6, thus reaching the goal in two steps.

items	Expected result
`[1, 6, 4, 2, 5, 3]`	1
`[8, 3, 4, 1, 7, 2, 6, 5]`	3
`[8, 5, 3, 1, 7, 2, 6, 4]`	4
`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`	5
`range(1, 10001)`	5000
`[1000] + list(range(1, 1000))`	1

The bottleneck of the running time is computing the new list that results from removing two elements from it. Try to think up some way to solve this problem that alleviates this expensive operation. Note again that the problem specification guarantees that the list elements cannot be just whatever, but are precisely the integers from 1 to *n*, which allows your function to loop through these elements in ascending order...

What do you hear, what do you say?

```
def count_and_say(digits):
```

Given a string of digits that is guaranteed to contain only digit characters from `'0123456789'`, read that string "out loud" by saying how many times each digit occurs consecutively in the current bunch of digits, and then return the string of digits that you just said out loud. For example, given the digits `'222274444499966'`, we would read it out loud as "four twos, one seven, five fours, three nines, two sixes" for the result string `'4217543926'`.

digits	Expected result
`'333388822211177'`	`'4338323127'`
`'11221122'`	`'21222122'`
`'123456789'`	`'111213141516171819'`
`'777777777777777'`	`'157'`
`''`	`''`
`'1'`	`'11'`

As silly and straightforward as this "[count-and-say sequence](#)" problem might initially seem, it required the genius of no lesser mathematician than John Conway to notice and capture into a symbolic equation the tremendous complexity ready to burst out from just below its surface, as the man himself explains [in this Numberphile video](#). Interested students can also check out the related construct of the infinitely long and yet self-describing [Kolakoski sequence](#) where only the lengths of each consecutive block of digits is stored, not the actual digits.

Bishops on a binge

```
def safe_squares_bishops(n, bishops):
```

A generalized *n*-by-*n* chessboard has been taken over by some [bishops](#), each represented as a tuple `(row, column)` of the row and the column of the square the bishop stands on. Same as in the earlier version of this problem with rampaging rooks, the rows and columns are numbered from 0 to *n* - 1. Unlike a chess rook whose moves are **axis-aligned**, a chess bishop covers all squares that are on the same **diagonal** with that bishop arbitrarily far into any of the four diagonal compass directions. Given the board size `n` and the list of `bishops` on that board, count the number of safe squares that are not covered by any bishop.

To determine whether two squares `(r1, c1)` and `(r2, c2)` are reachable from each other in one diagonal move, use `abs(r1-r2) == abs(c1-c2)` to check whether the horizontal distance between those squares equals their vertical distance, which is both necessary and sufficient for those squares to lie on the same diagonal. This way you don't need to write the essentially identical block of logic separately for the four diagonal directions, but a single test handles all four diagonals in one swoop.

n	bishops	Expected result
10	`[]`	100
4	`[(2, 3), (0, 1)]`	11
8	`[(1, 1), (3, 5), (7, 0), (7, 6)]`	29
2	`[(1, 1)]`	2
6	`[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]`	18
100	`[(row, (row*row) % 100) for row in range(100)]`	6666

Dem's some mighty tall words, pardner

```
def word_height(words, word):
```

The **length** of a word is easy enough to define and compute by counting its characters. Taking the road less traveled, we define the **height** of the given word with the following **recursive** rule. As the base case, any character string that is not a member of the list of all words automatically has the height of zero. Otherwise, the height of the given word equals one plus the whichever is larger of the two heights of its nonempty left and right pieces that it can be split into. To make the heights unambiguous, splitting into the left and right pieces must be done to produce the tallest height. For example, the height of 'blamability' is two, since it can be split into 'blam' and 'ability' that split no further, and therefore have a height of one by the easy virtue of being words.

Since the list of all words is sorted, you should use **binary search** (as implemented as the function bisect_left in the bisect module) to quickly determine whether a piece is a legitimate word. Be mindful of pieces that start with two or more copies of the letter z. They do occur.

word	Expected result (using the wordlist words_sorted.txt)
'chukker'	1
'asteroid'	3
'pedicured'	1
'quadrants'	4
'napthalin'	5
'undepressed'	7

Curiously enough, the tallest word inside words_sorted.txt with its height of ten is a well-known joke word that is also a solution of a classic recreational linguistics puzzle.

Up for the count

```
def counting_series(n):
```

The [Champernowne word](#) 1234567891011121314151617181920212223... is an infinitely long string of digits made up of all positive integers written in ascending order without any separators between the individual numbers. This function should return the integer digit that lies in the position n of the Champernowne word, position count again starting from zero as usual.

Of course, the automated tester will give your function values of n large enough that anybody trying to solve this problem by explicitly constructing the series as a string would run out of time and space long before receiving the answer. Instead, you should observe that the structure of this infinite sequence is quite straightforward, as it starts with 9 single-digit numbers, followed by 90 two-digit numbers, followed by 900 three-digit numbers, and so on. Such a predictably self-similar structure allows you to skip over prefixes of this series in exponentially widening leaps and bounds, until you reach the position n and find out the digit that is waiting for you there.

n	Expected result
0	1
100	5
10000	7
10**100	6

Out of curiosity, has the digit that was waiting for you in position n been waiting for you there since the dawn of time? Or did the answer come to existence the moment that somebody first posed this problem, or only when somebody actually solved this problem up to the position n? What is your stance on this thorny and ancient philosophical issue that is well above this author's pay grade? Do all mathematical structures and the answers to all possible questions about these structures already exist in the static and timeless Platonic plane of perfect forms for us to discover by means of reason? Or are at least some mathematical truths lazily created by social agreement as the need for them emerges, so that their existence is similar to that of fictional characters such as Donald Duck and James Bond, woven into our reality by the arbitrary decisions of their creators?

Revorse the vewels

```
def reverse_vowels(text):
```

Given a `text` string, create and return a new string constructed by finding all its **vowels** (for simplicity, in this problem vowels are the letters found in the string `'aeiouAEIOU'`) and reversing their order, while keeping all other characters exactly as they were in their original positions. However, to make the result look prettier, the capitalization of each moved vowel must be the same as that of the vowel that was originally in the target position. For example, reversing the vowels of `'Ilkka'` should produce `'Alkki'` instead of `'alkkI'`. Applying this operation to random English sentences often seems to give them a curious pseudo-Mediterranean vibe.

Along with many possible other ways to perform this square dance, one straightforward way to reverse the vowels starts with collecting all vowels of `text` into a separate list, and initializing the `result` to an empty string. After that, iterate through all positions of the original `text`. Whenever the current position contains a vowel, take one from the end of the list of the vowels. Convert that vowel to either upper- or lowercase depending on the case of the vowel that was originally in that position, and append it to `result`. Otherwise, append the character from the same position of the original `text` to the `result` as it were.

text	Expected result
`'Revorse the vewels'`	`'Reverse the vowels'`
`'Ilkka Markus Kokkarinen'`	`'Elkki Markos Kukkaranin'`
`'Hello world'`	`'Hollo werld'`
`'abcdefghijklmnopqrstuvwxyz'`	`'ubcdofghijklmnepqrstavwxyz'`
`'This is Computer Science I!'`	`'This es Cempiter Scuonci I!'`

Everybody come do the Scrooge Shuffle

```
def spread_the_coins(piles, left, right):
```

Some coins have been placed on the integer line $\mathbb{Z}$ so that for every `i in range(len(piles))`, the position `i` initially contains `piles[i]` coins. All other positions on $\mathbb{Z}$, both positive and negative, are initially empty. A position that contains at least `left+right` coins is **unstable**. As long as some position `i` is unstable, exactly `left` coins from that position `i` spill into its predecessor position `i−1`, and exactly `right` coins spill into its successor position `i+1`. Continue until everything is stable.

The unique terminal state of this back-and-forth dance will necessarily be reached after a finite number of steps, independent of the order of processing the unstable positions in the interim. This function should return this terminal state as tuple `(start, coins)` where `start` is the leftmost position that contains at least one coin. The second component lists the `coins` that ended in each position, this list spanning the positions from `start` up to the last non-empty position.

To speed up this process, note how whenever some pile contains `k*(left+right)` coins, you can scoop `k*left` coins into the predecessor and `k*right` coins into the successor pile as a single move. The new difficulty of this problem is handling the negative positions during computation, and also quickly finding some unstable position to process next.

piles	left	right	Expected result
[0, 0, 2]	1	1	(1, [1, 0, 1])
[20]	3	2	(−4, [3, 1, 4, 2, 4, 2, 4])
[8, 4]	3	2	(−2, [3, 1, 3, 3, 2])
[111, 12, 12]	19	6	(−6, [19, 13, 13, 13, 13, 13, 10, 23, 18])
[101]	1	1	(a tuple whose first element equals −50 and the second element is a list of 101 copies of number 1)

Calkin-Wilf sequence

```
def calkin_wilf(n)
```

The nodes of the [Calkin-Wilf tree](#), when read in **level order** so that the elements in each level are read from left to right, produce the linear sequence of all possible **positive rational numbers**. Almost as if by magic, this construction guarantees every positive integer fraction to appear exactly once in this sequence. Even more wonderfully, this construction makes every rational number to appear in its lowest reduced form! To perform the following calculations, you should `import` the data types `Fraction` and [`deque`](#) from the [`fractions`](#) and [`collections`](#) modules.

Your function should return the n:th element of this sequence. First, create a new instance of `deque` and `append` the first fraction `1/1` to "prime the pump", so to speak, to initiate the production of the values of this sequence. Then repeat the following procedure `n` times. Pop the fraction currently in front of the queue using the `deque` method `popleft`, extract its numerator and denominator `p` and `q`, and push the two new fractions `p/(p+q)` and `(p+q)/q` to the back of the queue, in this order. Return the fraction object that was popped in the final round.

n	Expected result
10	3/5
1000	11/39
100000	127/713

(Actually, once you reach the position `n//2+1`, the queue already contains the result you need, so you can stop pushing in any new values and that way save some significant memory.)

Hippity hoppity, abolish loopity

```
def frog_collision_time(frog1, frog2):
```

A frog hopping along on the infinite two-dimensional lattice grid of integers is represented as a 4-tuple of the form `(sx, sy, dx, dy)` where `(sx, sy)` is its **starting position** at time zero, and `(dx, dy)` is its constant **direction vector** for each hop. Time advances in discrete integer steps 0, 1, 2, 3, ... so that each frog makes one hop at every tick of the clock. At time `t`, the position of that frog is given by the formula `(sx+t*dx, sy+t*dy)` that can be nimbly evaluated for any `t`.

Given two frogs `frog1` and `frog2` that are guaranteed to initially stand on different squares, return the time when both frogs hop into the same square. If these two frogs never simultaneously arrive at the same square, return `None`.

This function should not contain any loops whatsoever. The result should be calculated using conditional statements and integer arithmetic. Perhaps the best way to get cracking is to first solve a simpler version of this problem with one-dimensional frogs restricted to hop along the one-dimensional line of integers. Once you get that function working correctly, including all its possible edge cases such as either one or both frogs hopping in place, use it to solve for `t` separately for the *x*- and *y*-dimensions in the original problem. Combine these two one-dimensional answers into your final answer.

frog1	frog2	Expected result
(0, 0, 0, 2)	(0, 10, 0, 1)	10
(10, 10, -1, 0)	(0, 1, 0, 1)	None
(0, -7, 1, -1)	(-9, -16, 4, 2)	3
(-28, 9, 9, -4)	(-26, -5, 8, -2)	None
(-28, -6, 5, 1)	(-56, -55, 9, 8)	7
(620775675217287, -1862327025651882, -3, 9)	(413850450144856, 2069252250724307, -2, -10)	206925225072431

Double trouble

```
def double_trouble(items, n):
```

Suppose, if just for the sake of argument, that the following operation is repeated `n` times for the given list of `items`: remove the first element, and append that same element twice to the end of `items`. Which one of the `items` would be removed and copied in the last operation performed?

Sure, this problem could be finger-quotes "solved" by actually performing that operation `n` times, but the point of this exercise is to come up with an analytical solution to compute the result much faster than actually going through that whole rigmarole. To gently nudge you towards thinking in symbolic and analytical solutions, the automated tester is designed so that anybody trying to **brute force** their way through this problem by performing all `n` operations one by one for real will run out of time and memory long before receiving the answer, as will the entire universe.

To come up with this analytical solution, tabulate some small cases (you can implement the brute force function to compute these) and try to spot the pattern that generalizes to arbitrarily large values of n. You can again also check out Wolfram Alpha or similar systems capable of manipulation of symbolic formulas to simplify combinatorial and summing formulas.

items	n	Expected result
['joe', 'bob', 42]	10	'joe'
[17, 42, 99]	1000	17
[17, 42, 99]	10**20	99
['only', 'the', 'number', 'of', 'items', 'matters']	10**1000	'the'

The reason to pay attention in the discrete math and combinatorics courses is learning to derive symbolic solutions to problems of this nature so that you don't have to brute force their answers in a time that would be prohibitively long to ever be feasible. We met this very same theme back in the question of counting the number of cannonballs in the given brass monkey pyramid...

Nearest polygonal number

```
def nearest_polygonal_number(n, s):
```

Any positive integer $s > 2$ defines an infinite sequence of **s-gonal numbers** whose i:th element is given by the formula $((s - 2)\ i^2 - (s - 4)\ i)\ /\ 2$, as explained on the Wikipedia page "[Polygonal Number](#)". In this formula, positions start from 1, not 0, and we use the letter i for the position since we will be using the letter n for something else. For example, the sequence of "[octagonal numbers](#)" that springs forth from $s = 8$ starts with 1, 8, 21, 40, 65, 96, 133, 176...

Given the number of sides `s` and an arbitrary integer `n`, this function should find and return the s-gonal integer that is closest to `n`. If `n` falls exactly halfway between two s-gonal numbers, return the smaller of those numbers. As you can see from the last row of the following table, this function must be efficient even for gargantuan values of `n`.

n	s	Expected result
5	3	6
27	4	25
450	9	474
10**10	42	9999861561
10**100	91	1000000000000000000000000000000000000000000000000 0000004163327535183294788977557947043340030 03544212420356

The simplest way to make this function efficient is to harness the power of **repeated halving** to pull your wagon with a clever application of **binary search**. Start with two integers `a` and `b` wide enough that they satisfy `a<=i<=b` for the currently unknown position `i` that the nearest polygonal number is stored in. (Just initialize these as `a=1` and `b=2`, and keep squaring `b` until the number in that position gets too big. It's not like these initial bounds need to be accurate.) From there, compute the **midpoint** position `(a+b)//2`, and look at the element in that position. Depending on how that midpoint element compares to `n`, bring either `b` or `a` to the midpoint position. Continue this until the gap has become small enough so that `b−a<2`, at which point one last comparison tells you the correct answer.

Postfix interpreter

```
def postfix_evaluate(items):
```

When arithmetic expressions are given in the familiar infix notation 2 + 3 * 4, we need to use parentheses to force a different evaluation order than the usual **PEMDAS** order determined by *precedence* and *associativity*. Writing arithmetic expressions in *postfix* notation (also known as [Reverse Polish Notation](#)) may look strange to us humans accustomed to the conventional *infix* notation, but is computationally far easier to handle, since postfix notation allows any evaluation order to be expressed unambiguously without using any parentheses at all! A postfix expression is given as a list of `items` that can be either individual integers or one of the strings `'+'`, `'-'`, `'*'` and `'/'` for the four possible arithmetic operators.

To evaluate a postfix expression using a simple linear loop, use a list as a **stack** that is initially empty. Loop through the `items` one by one, in order from left to right. Whenever the current item is an integer, just `append` it to the end of the list. Whenever the current item is one of the four arithmetic operations, `pop` two items from the end of the list, perform that operation on those items, and `append` the result to the list. Assuming that `items` is a legal postfix expression, which is guaranteed in this problem so that you don't need to do any error handling, once all items have been processed this way, the one number that remains in the stack is returned as the final answer.

To avoid the intricacies of floating point arithmetic, you should perform the division operation using the Python integer division operator `//` that truncates the result to the integer part. Furthermore, to avoid the crash caused by dividing by zero, in this problem we shall artificially make up a rule that dividing anything by zero will simply evaluate to zero instead of crashing.

`items`	(Equivalent infix)	Expected result
`[2, 3, '+', 4, '*']`	(2+3) * 4	20
`[2, 3, 4, '*', '+']`	2 + (3*4)	14
`[3, 3, 3, '-', '/']`	3 / (3 - 3)	0
`[7, 3, '/']`	7 / 3	2
`[1, 2, 3, 4, 5, 6, '*', '*', '*', '*', '*']`	1 * 2 * 3 * 4 * 5 * 6	720

(By adding more operators and another auxiliary stack, an entire Turing-complete programming language can be built around postfix evaluation. See the Wikipedia page "[Forth](#)", if interested.)

Fractran interpreter

```
def fractran(n, prog, giveup=1000):
```

[John Conway,](#) who was quite a character among mathematicians, is best known for [The Game of Life,](#) not to be confused with the family board game sharing the same name. That achievement eclipsed basically all other [wacky and original creations](#) of Conway, so in the name of fairness, we ought to give his less appreciated creations an occasional turn in the flashing lights of the red carpet fame.

This lab has you write an interpreter for the simple [esoteric programming language](#) called [FRACTRAN](#), named as a pun of the word "fraction" and the [FORTRAN programming language.](#) (Things used to be named in uppercase back in the day, with occasional dollar signs interspersed as separators.) A program written in such mysterious and hard-to-decipher form consists of nothing but a list of positive integer fractions, in this problem given as tuples of the numerator and the denominator. Of course, you are not merely allowed but encouraged to use the `Fraction` data type of the Python `fractions` module to simplify the computations inside your function.

Given a positive integer `n` as its start state, the next state is the product `n*f` for the first fraction listed in `prog` for which `n*f` is an exact integer. That integer then becomes the new state for the next round. Once `n*f` is not an integer for any of the fractions `f` listed in `prog`, the execution terminates. Your function should compute and return the sequence of integers produced by the given FRACTRAN program, with a forced termina... sorry, forced **halting** (after all, Conway was British) taking place after `giveup` steps, if the execution has not halted by then.

n	prog	giveup	Expected result
2	`[(17, 91), (78, 85), (19, 51), (23, 38), (29, 33), (77, 29), (95, 23), (77, 19), (1, 17), (11, 13), (13, 11), (15, 2), (1, 7), (55, 1)]`	20	`[2, 15, 825, 725, 1925, 2275, 425, 390, 330, 290, 770, 910, 170, 156, 132, 116, 308, 364, 68, 4, 30]`
9	(same as above)	20	`[9, 495, 435, 1155, 1015, 2695, 3185, 595, 546, 102, 38, 23, 95, 385, 455, 85, 78, 66, 58, 154, 182]`

Permutation cycles

```
def permutation_cycles(perm):
```

In this problem, a **permutation** is a list of *n* elements that contains every integer from 0 to *n* - 1 exactly once. For example, [5, 2, 0, 1, 4, 3] is one of the 6! permutations for *n* = 6. Each permutation can be considered a **bijective function** that maps each position 0 to *n* - 1 into some unique position so that no two positions map to the same position. For example, the previous permutation maps the position 0 into 5, position 1 into 2, and so on.

A **cycle** of a permutation is a list of positions so that each position is mapped into the next position in the cycle, with the last element looping back to the first position of that cycle. Positions inside a cycle are treated in a "necklace" fashion so that [1, 2, 3], [2, 3, 1] and [3, 1, 2] are simply different ways to represent the same cycle, distinct from another possible cycle of these three elements given as either [1, 3, 2], [3, 2, 1] or [2, 1, 3]. To construct the cycle that the position i belongs to, start at that position i and keep moving from the current position j to its successor position perm[j] until you return to the original position i. The cycles of the above permutation are [0, 5, 3, 1, 2] and [4], the second cycle a **singleton** as this permutation maps the position 4 back into itself.

This function should list each individual cycle starting from its highest element, and the cycles should be combined to the increasing order of their starting positions. For example, the previous example cycles [4] and [5, 3, 1, 2, 0] must be listed in this order, since 4 < 5. Furthermore, instead of returning a list of cycles, these cycles must be encoded as a **flat list** that simply lists its cycles without any separators such as square brackets. Amazingly, this run-on **standard representation** of the permutation still allows unique reconstruction of its original cycles!

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

Squaring off

```
def subtract_square(queries):
```

Two players play a game of "[Subtract a square](#)", starting with a positive integer. On their turn to play, each player must subtract some **square number** (1, 4, 9, 16, 25, 36, 49, ...) from the current number so that the result does not become negative. Under the **normal play convention** of these games, the player who moves to zero wins, leaving his opponent stuck with no possible moves. (In the **misère** version of the game that has otherwise identical rules but where you win by losing the original game, these definitions would be adjusted accordingly.)

This and many similar [combinatorial games](#) can be modelled with recursive equations. This game is an example of an [impartial game](#) since the moves available to both players are exactly the same, as opposed to "partisan games" such as chess where each player can only move pieces of one colour. A game state is therefore determined by the remaining number alone, but not by which player has to make the next move. A state is called "**hot**" ("winning") if it allows at least one move that is part of the winning line regardless of the opponent's response, under the assumption that the current player will then also continue making the best moves until the opponent says uncle. The game state is "**cold**" ("losing") if no winning move exists in that state.

Since the possible states of this game are the nonnegative integers, the base case for the recursion is that zero is cold. The state `n` is hot if there exists at least one move `m` so that `n-m*m` is cold, otherwise `n` is cold. Since the heat of each state `n` depends on heats of states lower than `n`, we might as well combine the computations of states to be as a "batch job". Given a list of `queries` so that each element is a state this function should return a list of results for those queries so that `True` means hot and `False` means cold. You should compute the heat values of states as a single list that you build up from left to right, so that the computation of the heat value of each state can simply read from this list the already computed heat values of lower states.

queries	Expected result
`[7, 12, 17, 24]`	`[False, False, False, True]`
`[2, 3, 6, 10, 14, 20, 29, 39, 57, 83, 111, 149, 220, 304, 455, 681]`	`[False, True, True, False, True, False, True, False, False, True, True, True, True, True, True, True]`
`[7, 12, 17, 23, 32, 45, 61, 84, 120, 172, 251, 345, 510, 722, 966, 1301, 1766, 2582, 3523, 5095, 7812, 10784, 14426, 20741]`	`[False, False, False, True, True, True, True, True, True, True, True, True, False, False, True, True, True, True, True, True, True, True, True, True]`

ztalloc ecneuqes

`def ztalloc(shape):`

The famous [Collatz sequence](#) was used in the lectures as an example of a situation that requires the use of a `while`-loop, since we cannot know beforehand how many steps are needed to get to the goal from the given starting value. The answer was given as the list of integers that the sequence visits before terminating at its goal. However, we can also look at this sequence in a binary fashion depending on whether each value steps **up** ($3x + 1$) or **down** ($x // 2$) from the previous value, denoting these steps with letter `'u'` and `'d'`, respectively. For example, starting from n $=$ 12, the sequence `[12, 6, 3, 10, 5, 16, 8, 4, 2, 1]` would have the step shape `'ddududddd'`.

This function should, given the step `shape` as a string that is guaranteed to consist of only letters u and d, determine which starting value for the Collatz sequence produces that `shape`. However, this function must also recognize that some shape strings are impossible as entailed by the transition rules of Collatz problem, and correctly return `None` for all such shapes. You should start from the goal state 1, and perform the given transitions in reverse. Make sure that your function does not accept moves that would be illegal in the original forward Collatz sequence.

`shape`	Expected result
`'ududududdddudddd'`	15
`'dudududduddduddd'`	14
`'uduuudddd'`	None
`'d'`	2
`'uuududdddduuuuuudddddd'`	None
`'duuudddddddd'`	None

Balanced centrifuge

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

This problem was inspired by yet another [Numberphile](#) video, this time "[The Centrifuge Problem](#)", that you should first watch to get an idea what this problem is about. (Despite the chemical inspiration of this problem, the solution is not an overly alkaline pH balance.) Alternatively, those who read faster than they listen can consult "[The Balanced Centrifuge Problem](#)" by Matt Baker.

A **centrifuge** has n identical slots arranged in a rotating circle. For safe operation of this fast spinning machine, the identical test tubes must be placed on these slots so that their mutual center of gravity lies exactly on the center of the centrifuge. This turns out to be possible for k test tubes precisely when **both** quantities k and n−k can somehow be expressed as sums of **prime factors** of n. For example, when n equals 6 so that its prime factors are 2 and 3, it is possible to load the centrifuge safely with 2, 3, 4 or 6 test tubes. (When k equals n, n−k equals zero that is always expressible as the **empty sum**, regardless of the available factors of n.) However, there is no way to safely balance one or five test tubes into six slots; even though 5 = 2+3, there is no way to counterbalance the empty slot without making the machine wobble dangerously. (Nope, one is still not a prime number, and therefore not a prime factor either.)

This function does not actually need to construct the balanced configuration of test tubes, but simply determine whether some balanced configuration exists at all.

n	k	Expected result
6	3	True
6	1	False
7	0	True
222	107	True
1234	43	False

Reverse ascending sublists

```
def reverse_ascending_sublists(items):
```

Create and return a new list that contains the same elements as the argument `items` list, but reversing the order of the elements inside every **maximal strictly ascending** sublist. Note the modifier "strictly" used there to require each element to be greater than the previous element, not merely equal to it. As is the case with all functions unless otherwise specified, this function should not modify the contents of the original list, but create and return a brand new list object that contains the result, no matter how tempting it might be to perform this operation right there in the original list to "save some memory".

In the table below, different colours highlight the maximal strictly ascending sublists, and are not part of the actual argument object given to the Python function. (It's not like this is *Mathematica* or some other high level **symbolic computation** system that deals directly with symbolic expressions in their unevaluated forms, allowing you to do things of such nature...)

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

Brangelina

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

Combining the first names of celebrity couples into a catchy shorthand for media consumption turns out to be simple to automate. Start by counting how many maximal groups of consecutive vowels (*aeiou*, since to keep this problem simple, the letter *y* is always a consonant) exist inside the first name. For example, `'brad'` and `'jean'` have one vowel group, `'jeanie'` and `'britain'` have two, and `'angelina'` and `'alexander'` have four. Note that each vowel group can contain more than one vowel, as in the word `'queueing'` that has an entire fiver.

If the first name has only one vowel group, keep only the consonants before that group and throw away everything else. For example, `'ben'` becomes `'b'`, and `'brad'` becomes `'br'`. Otherwise, if the first word has *n* > 1 vowel groups, keep everything before the **second last** vowel group *n* - 1. For example, `'angelina'` becomes `'angel'` and `'alexander'` becomes `'alex'`. Concatenate that with the string you get by removing all consonants at the beginning of the second name. All names given to this function are guaranteed to consist of the 26 lowercase English letters only, and each name will have at least one vowel and one consonant somewhere in it.

`first`	`second`	Expected result
`'brad'`	`'angelina'`	`'brangelina'`
`'angelina'`	`'brad'`	`'angelad'`
`'sheldon'`	`'amy'`	`'shamy'`
`'amy'`	`'sheldon'`	`'eldon'`
`'frank'`	`'ava'`	`'frava'`
`'britain'`	`'exit'`	`'brexit'`

These rules do not always produce the best possible result. For example, `'ross'` and `'rachel'` meld into `'rachel'` instead of `'rochel'`, and `'joey'` and `'phoebe'` meld into `'joebe'` instead of `'joeybe'`. The reader is invited to think up more advanced rules that cover a wider variety of name combinations and special cases. (Some truly advanced set of rules, perhaps trained with **deep learning** techniques to also implicitly recognize the **semantic** content of word combinations, might even combine `'donald'` and `'hillary'` into a metaphorically fitting name `'dollary'` instead of `'dillary'`…)

Line with most points

```
def line_with_most_points(points):
```

A point on the two-dimensional grid of integers is given as a two-tuple of *x*- and *y*-coordinates, for example (2, 5) or (10, 1). As originally postulated by Euclid, given any two distinct points on the plane, there exists **exactly one** straight line that goes through both points. (In differently shaped spaces, different rules and their consequences apply.) Of course, this same line, being shamelessly infinite to both directions, will also pass through an infinity of other points on that plane.

Given a list of `points` on the integer grid, find the line that contains the largest number of points from this list. To guarantee the uniqueness of the expected result for every test case, this function should not return the line itself, but merely the count of how many of these `points` lie on that line. The list of `points` is guaranteed to contain at least two points and that all points in the list are distinct, but otherwise these points are not given in any specific order.

To get started, consult the example program `geometry.py` for the **cross product** function that can be used to quickly determine whether three points are collinear

points	Expected result
[(42, 1), (7, 5)]	2
[(1, 4), (2, 6), (3, 2), (4, 10)]	3
[(x, y) for x in range(10) for y in range(10)]	10
[(3, 5), (1, 4), (2, 6), (7, 7), (3, 8)]	3
[(5, 6), (7, 3), (7, 1), (2, 1), (7, 4), (2, 6), (7, 7)]	4

This problem can be **brute forced** with three nested loops, but the point (heh) of this problem is not to do too much more work than you really need to. (Being lazy is a virtue in programming, and should more fittingly be called being elegant or optimal.)

Om nom nom

```
def cookie(piles):
```

The beloved *Sesame Street* character [Cookie Monster](.) has stumbled upon a table with `piles` of cookies, each pile a positive integer. However, the monomaniacal obsessiveness of [The Count](.) who set up this fiesta has recently escalated to a new level of severity. To allow the Cookie Monster to enjoy this crumbly fiesta, The Count insists that these cookies must be eaten in the smallest possible number of moves. Each move consists of the Cookie Monster choosing one of the remaining pile sizes `p`. This move removes `p` cookies from every pile that contains at least `p` cookies (thereby eradicating the piles whose size is exactly `p`), and leaves all smaller piles untouched as they were. This function should compute and return **the smallest number of moves** that allows Cookie Monster to scarf down these cookies.

piles	Expected result	(optimal moves)
[1, 2, 3, 4, 5, 6]	3	[4, 2, 1]
[2, 3, 5, 8, 13, 21, 34, 55, 89]	5	[55, 21, 8, 3, 2]
[1, 10, 17, 34, 43, 46]	5	[34, 9, 8, 3, 1]
[11, 26, 37, 44, 49, 52, 68, 75, 87, 102]	6	[37, 31, 15, 12, 7, 4]
[2**n for n in range(10)]	10	[512, 256, 128, 64, 32, 16, 8, 4, 2, 1]

This problem has a curious property in that the moves of the optimal solution can be performed in any order, which is rare for combinatorial problems of this general spirit. Without loss of generality, it suffices to examine only strictly descending move sequences. Even then, [this problem is far deeper than it would initially seem](.). **Greedy strategies** that might seem natural for the addiction-riddled brain of the Cookie Monster, such as "choose `p` to maximize the number of cookies eaten at the current move" or "choose `p` to eliminate the largest possible number of piles" (yep, think about that one), do not always produce the shortest sequence of moves.

Autocorrect for sausage fingers

```
def autocorrect_word(word, words, df):
```

In this day and age all you whippersnappers are surely familiar with **autocorrect** that replaces a non-word with the "closest" real word in the dictionary. Many techniques exist to guess more accurately what the user intended to write. For many old people such as your instructor, baffled by technological doodads and thingamabobs, typos occur when they inadvertently press a virtual key somewhere around the intended key. For example, when the non-existent word is `"cst"`, the intended word seems more likely to have been `"cat"` than `"cut"`, assuming that the text was entered using the ordinary QWERTY keyboard where the characters `a` and `s` are adjacent.

Given a `word`, a list of `words`, and a **distance** function `df` that tells how "far" the first argument is from the second (for example, `df('a','s')==1` and `df('a','a')==0`), find and return the word in the list of `words` that have the same number of letters and whose distance, measured as the sum of the distances of the characters in the same position, is the smallest. If several words are equidistant from `word`, return the first such word in the dictionary order.

word	Expected result
`'qrc'`	`'arc'`
`'jqmbo'`	`'jambo'`
`'hello'`	`'hello'`
`'interokay'`	`'interplay'`

More advanced autocorrect techniques use the surrounding context to suggest the best replacement, seeing that not all words are equally likely to be the intended word, given the rest of the sentence. For example, the misspelling `'urc'` should probably be corrected very differently in the sentence "The gateway consists of an urc of curved blocks" than in "The brave little hobbit swung his sword at the smelly urc." Similarly, even though `'lobe'` is a perfectly valid word by itself, the sentence `'I lobe you!'` is most likely a typo...

Uambcsrlne the wrod

```
def unscramble(words, word):
```

Smdboeoy nteoicd a few yreas ago taht the lretets isndie Eisgnlh wdors can be ronmaldy slmecbrad wouhtit antifecfg tiehr rlaibdiatey too mcuh, piovredd that you keep the frsit and the last lteters as tehy were. Gevin a lsit of `words` gtuaraened to be soterd, and one serlcmbad `wrod`, tihs fctounin shulod rterun the list of wrdos taht cloud hvae been the orgiianl word taht got seambclrd, and of csorue retrun taht lsit wtih its wdros in apcabihaetll oerdr to enrsue the uaigntbmuiy of atematoud testing. In the vast maitjory of ceass, this list wlil catnoin only one wrod.

wrod	Etecxepd rssuelt (unsig the wrosldit `words_sorted.txt`)
`'tartnaas'`	`['tantaras', 'tarantas', 'tartanas']`
`'aierotsd'`	`['asteroid']`
`'ksmrseah'`	`['kersmash']`
`'bttilele'`	`['belittle', 'billette']`

Writing the filter to transform the given plaintext to the above scrambled form is also a good little programming exercise in Python. This leads us to a useful estimation exercise: how long would the plaintext have to be for you to write such a filter yourself instead of scrambling the plaintext by hand as you would if you needed to scramble just one word? In general, how many times do you think you need to solve some particular problem until it becomes more efficient to design, write and debug a Python script to do it? Would your answer change if it turned out that millions of people around the world also have that same problem, silently screaming for a saviour to come and automate this repetitive and error-prone task for them?

Substitution words

```
def substitution_words(pattern, words):
```

Given a list of `words` (once again, guaranteed to be in sorted order and each consist of the 26 lowercase English letters only) and a `pattern` that consists of uppercase English characters, this function should find and return a list of precisely those words that match the `pattern` in the sense that there exists a substitution from uppercase letters to lowercase letters that turns the `pattern` into the word. Furthermore, this substitution must be **injective**, meaning that no two different uppercase letters in the pattern are mapped to the same lowercase letter in that word.

For example, the word `'akin'` matches the pattern `'ABCD'` with the substitutions A→a, B→k, C→i and D→n. However, the word `'area'` would not match that same pattern, since the pattern characters A and D would both have to be mapped to the same letter a, against the rules.

pattern	Expected result (using the wordlist `words_sorted.txt`)
`'ABBA'`	`['abba', 'acca', 'adda', 'affa', 'akka', 'amma', 'anna', 'atta', 'boob', 'deed', 'ecce', 'elle', 'esse', 'goog', 'immi', 'keek', 'kook', 'maam', 'noon', 'otto', 'peep', 'poop', 'sees', 'teet', 'toot']`
`'ABABA'`	`['ajaja', 'alala', 'anana', 'arara', 'ululu']`
`'CEECEE'`	`['beebee', 'booboo', 'muumuu', 'seesee', 'soosoo', 'teetee', 'weewee', 'zoozoo']`
`'ABCDCBA'`	`['adinida', 'deified', 'hagigah', 'murdrum', 'reifier', 'repaper', 'reviver', 'rotator', 'senones']`
`'DEFDEF'`	76 words, first three of which are `['akeake', 'atlatl', 'barbar']`
`'ABCDEFGIJKLM'`	243 words, first three of which are `['absorptively', 'adjunctively', 'adsorptively']`

(Extra math problem for the readers who are into combinatorics and discrete math: can you construct an encoding function from strings to natural numbers so that two strings map into the same number if and only if these two strings both match the same `pattern`?)

Manhattan skyline

```
def manhattan_skyline(towers):
```

This classic problem in [computational geometry](#) (essentially, geometry that can be done using only integer arithmetic; yes, that is an actual thing) is best illustrated by pictures and animations such as those on the page "[The Skyline problem](#)", so you can first check that it out to get an idea of what is going on. Given a list of rectangular `towers` as tuples `(s, e, h)` where `s` and `e` are the start and end *x*-coordinates (satisfying `e>s`) and `h` is the height of that tower, compute and return the **total visible area of the towers**, being careful not to double count two or more towers that are partially overlapping. All towers share the same flat ground baseline at the height.

The classic solution illustrates the important [sweep line technique](#) that starts by creating a list of precisely those *x*-coordinate values where something relevant to the problem takes place. In this problem, the relevant *x*-coordinates are those where some tower either starts or ends. Next, loop through this list in ascending order, updating your computation for the interval between the current relevant *x*-coordinate and the previous one. In this particular problem, you need to maintain a **list of active towers** so that tower `(s, e, h)` becomes active when `x==s`, and becomes inactive again when `x==e`. Inside each interval, only the tallest active tower affects the area summation.

The automated tester script will produce start and end coordinates from an increasing scale to prevent this problem being solved with the inferior method that builds up the list of all positions from zero up to the maximum end coordinate in `towers`, loops through the `towers` to update the tallest tower height at each position, and sums up these heights for the final area.

towers	Expected result
`[(2, 3, 39)]`	39
`[(6, 8, 56), (5, 14, 81), (3, 13, 71)]`	871
`[(6, 18, 95), (3, 20, 95), (14, 31, 22), (5, 12, 93)]`	1857
`[(16, 88, 20), (11, 75, 22), (43, 73, 27), (21, 42, 37), (20, 89, 12), (67, 68, 19), (1, 65, 24), (78, 91, 34), (65, 117, 9)]`	2871

Count overlapping disks

```
def count_overlapping_disks(disks):
```

Right on the heels of the previous Manhattan skyline problem, here is another classic of similar spirit for us to solve efficiently with a **sweep line algorithm**. Given a list of `disks` on the two-dimensional plane as tuples `(x,y,r)` so that `(x,y)` is the center point and `r` is the radius of that disk, count how many pairs of disks **intersect** each other in that their areas have at least one point in common. To test whether disks `(x1,y1,r1)` and `(x2,y2,r2)` intersect, use the Pythagorean formula `(x2-x1)**2+(y2-y1)**2<=(r1+r2)**2`. Note again how this precise formula needs only integer arithmetic whenever the individual components are integers, with no square roots or any other irrational numbers gumming up the works with their infinite chaos of decimals that we can only approximate by never complete.

For this problem, crudely looping through all possible pairs of disks would work but be horrendously inefficient as the list grows larger. However, a sweep line algorithm can solve this problem by looking at a far fewer pairs of disks. Again, sweep through the space from left to right for all relevant *x*-coordinate values and maintain **the set of active disks** at the moment. Each individual disk `(x,y,r)` enters the active set when the vertical sweep line reaches the *x*-coordinate `x-r`, and leaves the active set when the sweep line reaches `x+r`. When a disk enters the active set, check for an intersection only between that disk and the disks in the active set.

disks	Expected result
`[(0, 0, 3), (6, 0, 3), (6, 6, 3), (0, 6, 3)]`	4
`[(4, -1, 3), (-3, 3, 2), (-3, 4, 2), (3, 1, 4)]`	2
`[(-10, 6, 2), (6, -4, 5), (6, 3, 5), (-9, -8, 1), (1, -5, 3)]`	2
`[(x, x, x // 2) for x in range(2, 101)]`	2563

An unordinary ordering of ordinary cardinals

```
def sort_by_digit_count(items):
```

Sorting can be done according to arbitrarily chosen comparison criteria, as long as they satisfy the mathematical requirements of a **total ordering** relation. To play around with this concept, let us define a wacky ordering comparison of positive integers so that for any two integers, the one that contains the digit 9 more times than the other is considered to be larger, regardless of the magnitude and other digits of these numbers. For example, $99 > 12345678987654321 > 10^{1000}$ in this ordering. If both integers contain the digit 9 the same number of times, the comparison proceeds to the next lower digit 8, and so on, until the first distinguishing digit has been discovered. If both integers contain every digit from 9 to 0 pairwise the same number of times, the ordinary integer order comparison will determine their mutual ordering.

items	Expected result
`[9876, 19, 4321, 99, 73, 241, 111111, 563, 33]`	`[111111, 33, 241, 4321, 563, 73, 19, 9876, 99]`
`[111, 19, 919, 1199, 911, 999]`	`[111, 19, 911, 919, 1199, 999]`
`[1234, 4321, 3214, 2413]`	`[1234, 2413, 3214, 4321]`
`list(range(100000))`	(a list of 100,000 elements whose first five elements are `[0, 1, 10, 100, 1000]` and the last five are `[98999, 99899, 99989, 99998, 99999]`)

As explained in discrete math courses, the above relation is transitive and antisymmetric, and therefore a legal ordering relation for integers. Outside its entertainment value it is very nearly useless, though, since it is hard to imagine any useful integer arithmetic operation that would preserve equal values as equals under our wacky equivalence. The ordinary ordering for ordinals (how is that for a name for a hipster indie band?) that we unthinkingly take as some sort of law of nature is useful because it is compatible with useful arithmetic operations in the previous sense, and makes these operations therefore immensely more powerful.

Count divisibles in range

```
def count_divisibles_in_range(start, end, n):
```

Let us take a breather by tackling a problem simple enough that its solution needs only a couple of conditional statements and some arithmetic, but not even one loop or anything even more fancy. The difficulty is coming up with the conditions that cover all possible cases of this problem exactly right, including all of the potentially tricksy **edge and corner cases**, without being **off-by-one**. Given three integers `start`, `end` and `n` so that `start <= end`, count how many integers between `start` and `end`, inclusive, are divisible by `n`. Sure, you *could* solve this problem with the list comprehension one-liner

```
return len([x for x in range(start, end+1) if x % n == 0])
```

but of course the automated tester is designed so that anybody trying to solve this problem in such a blunt fashion will only find themselves running out of both time and space! Your code should have **no loops at all**, but use only integer arithmetic and conditional statements to get to the truth. Also, be careful with various edge cases and off-by-one pitfalls lurking in the bushes. Note that either `start` or `end` can be negative or zero, but `n` is guaranteed to be greater than zero.

start	end	n	Expected result
7	28	4	6
−77	19	10	9
−19	−13	10	0
1	10**12 − 1	5	199999999999
0	10**12 − 1	5	200000000000
0	10**12	5	200000000001
−10**12	10**12	12345	162008911

Bridge hand shape

```
def bridge_hand_shape(hand):
```

In the card game of [bridge](#), each player receives a hand of exactly thirteen cards. The *shape* of the hand is the distribution of these cards into the four suits in the exact order of **spades, hearts, diamonds, and clubs.** Given a bridge hand encoded as in the example script `cardproblems.py`, return the list of these four numbers. For example, given a `hand` that contains five spades, no hearts, five diamonds and three clubs, this function should return `[5, 0, 5, 3]`. Note that the cards in `hand` can be given to your function in any order, since in this question the player has not yet manually sorted his hand. Your answer still must list all four suits in their canonical order, so that other players will also know what you are talking about.

hand	Expected result
`[('eight', 'spades'), ('king', 'diamonds'), ('ten', 'diamonds'), ('three', 'diamonds'), ('seven', 'spades'), ('five', 'diamonds'), ('two', 'hearts'), ('king', 'spades'), ('jack', 'spades'), ('ten', 'clubs'), ('ace', 'clubs'), ('six', 'diamonds'), ('three', 'hearts')]`	`[4, 2, 5, 2]`
`[('ace', 'spades'), ('six', 'hearts'), ('nine', 'spades'), ('nine', 'diamonds'), ('ace', 'diamonds'), ('three', 'diamonds'), ('five', 'spades'), ('four', 'hearts'), ('three', 'spades'), ('seven', 'diamonds'), ('jack', 'diamonds'), ('queen', 'spades'), ('king', 'diamonds')]`	`[5, 2, 6, 0]`

Milton Work point count

```
def milton_work_point_count(hand, trump='notrump'):
```

Playing cards are represented as tuples of form `(rank,suit)` as in `cardproblems.py` example program. The trick taking power of a **bridge** hand is estimated with Milton Work point count, of which we shall implement a version that is simple enough for beginners of either Python or the game of bridge. Looking at a bridge hand that consists of thirteen cards, first give it 4 points for each ace, 3 points for each king, 2 points for each queen, and 1 point for each jack. That should be simple enough. This **raw point count** is then adjusted with the following rules:

- If the hand contains one four-card suit and three three-card suits, subtract one point for being **flat**. (Flat hands rarely play as well as non-flat hands with the same point count.)
- Add 1 point for every suit that has five cards, 2 points for every suit that has six cards, and 3 points for every suit with seven cards or longer. (Shape is power in offence.)
- If the `trump` suit is anything other than `'notrump'`, add 5 points for every **void** (that is, suit without any cards in it) and 3 points for every **singleton** (that is, a suit with exactly one card) for any other suit than the `trump` suit. (Voids and singletons are great when you are playing a suit contract, but very bad when you are playing a notrump contract. Being void in the trump suit is, of course, extremely bad in that suit contract!)

hand (each hand below has been sorted by suits for readability, but your function can receive these 13 cards from the tester in any order)	trump	Expected result
[('four', 'spades'), ('five', 'spades'), ('ten', 'hearts'), ('six', 'hearts'), ('queen', 'hearts'), ('jack', 'hearts'), ('four', 'hearts'), ('two', 'hearts'), ('three', 'diamonds'), ('seven', 'diamonds'), ('four', 'diamonds'), ('two', 'diamonds'), ('four', 'clubs')]	'diamonds'	8
[('three', 'spades'), ('queen', 'hearts'), ('jack', 'hearts'), ('eight', 'hearts'), ('six', 'diamonds'), ('nine', 'diamonds'), ('jack', 'diamonds'), ('ace', 'diamonds'), ('nine', 'clubs'), ('king', 'clubs'), ('jack', 'clubs'), ('five', 'clubs'), ('ace', 'clubs')]	'clubs'	20
[('three', 'spades'), ('seven', 'spades'), ('two', 'spades'), ('three', 'hearts'), ('queen', 'hearts'), ('nine', 'hearts'), ('ten', 'diamonds'), ('six', 'diamonds'), ('queen', 'diamonds'), ('ace', 'diamonds'), ('nine', 'clubs'), ('four', 'clubs'), ('five', 'clubs')]	'notrump'	7

Never the twain shall meet

```
def hitting_integer_powers(a, b, tolerance=100):
```

Integer powers of the given base such as 7 in the series 7, 49, 393, 2401, 16807, 117649, ... soon blow up too large to be useful in our daily lives. Outside time and space, all alone with no human minds to watch over their journey that is paradoxically both infinite and instantaneous, these exponentially growing sequences of powers probe through the infinite space of positive integers with exponentially increasing gaps that eventually contain entire universes inside them. Fortunately, Python allows us to play around with integer powers with millions of digits, and yet its mechanical binary decisions will reliably and unerringly dissect even these behemoths that our mortal minds could not begin to visualize.

Except when the prime factors of `a` and `b` fit nicely,, the iron hand of the [Fundamental Theorem of Arithmetic](#) dictates that the integer powers `a**pa` and `b**pb` can never be exactly equal for any positive integer exponents `pa` and `pb`. However, in jovial spirit of "[close enough for government work](#)", we consider such powers to "hit" if their absolute difference `abs(a**pa-b**bp)` multiplied by the desired `tolerance` is at most equal to the smaller of those two powers. (This definition intentionally avoids division to keep it fast and accurate for arbitrarily large integers.) For example, `tolerance=100` requires their difference to be at most 1%. Given the positive integers `a` and `b` and your desired `tolerance`, find and return the tuple of the smallest integer exponents `(p1, p2)` that satisfy this requirement.

a	b	tolerance	Expected result
2	4	100	(2, 1)
2	7	100	(73, 26)
3	6	100	(137, 84)
4	5	1000	(916, 789)
10	11	1000	(1107, 1063)
42	99	100000	(33896, 27571)

(This problem was inspired by the Riddler Express problem in the column "[Can you get another haircut already?](#)" of [The Riddler](#), an excellent source of problems of this kind of general spirit.)

Bridge hand shorthand form

```
def bridge_hand_shorthand(hand):
```

In literature on the game of **contract bridge**, hands are often given in abbreviated form that makes their relevant aspects easier to visualize at a glance. In this abbreviated shorthand form, suits are always listed **in the exact order of spades, hearts, diamonds and clubs**, so no special symbols are needed to show which suit is which. The ranks in each suit are listed as letters from `'AKQJ'` for **aces and faces**, and all **spot cards** lower than jack are written out as the same letter `'x'` to indicate that its exact spot value is irrelevant for the play mechanics of that hand. These letters must be listed in descending order of ranks `AKQJx`. If some suit is **void**, that is, the hand contains no cards of that suit, that suit is abbreviated with a single minus sign character `'-'`. The shorthand forms for the individual suits are separated using single spaces in the result string, without any trailing whitespace.

hand (each hand below is sorted by suits for readability, but your function can receive these 13 cards from the tester in any order)	Expected result
[('four', 'spades'), ('five', 'spades'), ('ten', 'hearts'), ('six', 'hearts'), ('queen', 'hearts'), ('jack', 'hearts'), ('four', 'hearts'), ('two', 'hearts'), ('three', 'diamonds'), ('seven', 'diamonds'), ('four', 'diamonds'), ('two', 'diamonds'), ('four', 'clubs')]	`'xx QJxxxx xxxx x'`
[('three', 'spades'), ('queen', 'hearts'), ('jack', 'hearts'), ('eight', 'hearts'), ('six', 'diamonds'), ('nine', 'diamonds'), ('jack', 'diamonds'), ('ace', 'diamonds'), ('nine', 'clubs'), ('king', 'clubs'), ('jack', 'clubs'), ('five', 'clubs'), ('ace', 'clubs')]	`'x QJx AJxx AKJxx'`
[('three', 'spades'), ('seven', 'spades'), ('two', 'spades'), ('three', 'hearts'), ('queen', 'hearts'), ('nine', 'hearts'), ('ten', 'diamonds'), ('six', 'diamonds'), ('queen', 'diamonds'), ('ace', 'diamonds'), ('nine', 'clubs'), ('four', 'clubs'), ('five', 'clubs')]	`'xxx Qxx AQxx xxx'`
[('ace', 'spades'), ('king', 'spades'), ('queen', 'spades'), ('jack', 'spades'), ('ten', 'spades'), ('nine', 'spades'), ('eight', 'spades'), (seven, 'spades'), ('six', 'spades'), ('five', 'spades'), ('four', 'spades'), ('three', 'spades'), ('two', 'diamonds')]	`'AKQJxxxxxxxx - x -'`

(Then again, the author once saw a play problem where a freaking *eight* was a relevant rank...)

Losing trick count of a bridge hand

```
def losing_trick_count(hand):
```

The [Milton Work point count](#) that we saw in an earlier problem is the first baby step in estimating the playing power of a bridge hand. Once the partnership discovers they have a good trump fit, hand evaluation continues more accurately using some form of [losing trick count](#). For example, a small slam in spades with `'AKxxxxx – Kxxx xx'` facing `'xxxx xxxxx AQx –'` is a lock despite possessing only 16 of the 40 high card points in the deck, assuming that the opponents looking at their own massive shapes let you play that slam instead of making a great sacrifice by bidding seven of their suit. (Advanced partnerships are able to judge these things accurately from what has, and especially what has *not*, been bid.) On the other hand (heh), any slam is dead in the water with the `'QJxxx xx AKx QJx'` facing `'AKxxx QJ QJx AKx'`, thanks to the horrendous duplication of useful cards. ("If it quacks like a duck...")

In this problem, we compute the basic losing trick count as given in step 1 of "[Methodology](#)" section of the Wikipedia page "[Losing Trick Count](#)" without any finer refinements. Keep in mind that no suit can have more losers than it has cards, and never more than three losers even if the hand has ten cards of that suit! The following dictionary (composed by student Shu Zhu Su during the Fall 2018 term) might also come handy for the combinations whose losing trick count differs from the string length, once you convert each `J` of the shorthand form into an `x`:

```
{'-':0,'A':0,'x':1,'Q':1,'K':1,'AK':0,'AQ':1,'Ax':1,'KQ':1,'Kx':1,'Qx':2,'xx'
:2, 'AKQ':0, 'AKx':1, 'AQx':1,'Axx':2, 'Kxx':2, 'KQx':1, 'Qxx':2, 'xxx':3}
```

hand	Expected result
[('ten', 'clubs'), ('two', 'clubs'), ('five', 'clubs'), ('queen', 'hearts'), ('four', 'spades'), ('three', 'spades'), ('ten', 'diamonds'), ('king', 'spades'), ('five', 'diamonds'), ('nine', 'hearts'), ('ace', 'spades'), ('queen', 'spades'), ('six', 'spades')]	7
[('eight', 'hearts'), ('queen', 'spades'), ('jack', 'hearts'), ('queen', 'hearts'), ('six', 'spades'), ('ten', 'hearts'), ('five', 'clubs'), ('jack', 'spades'), ('five', 'diamonds'), ('queen', 'diamonds'), ('six', 'diamonds'), ('three', 'spades'), ('nine', 'clubs')]	8

Bulls and cows

```
def bulls_and_cows(guesses):
```

In the old two-player game of "Bulls and Cows" (reincarnated in the seventies with pretty colours under the name "Mastermind") the first player thinks up a four-digit secret number whose every digit must be different, for example 8723 or 9425. (For simplicity, we will not use the digit zero in this problem.) The second player tries to find this secret number by repeatedly guessing some four-digit number. After each guess, the first player reveals how many "bulls", the right digit in the right position, and "cows", the right digit but in the wrong position, that guess contains. For example, if the secret number is 1729, the guess 5791 contains one bull (the digit 7) and two cows (the digits 9 and 1). The guess 4385, on the other hand, contains no bulls or cows.

This function should determine which numbers could still be the secret number. Given a list of `guesses` that have been completed, each individual guess a three-tuple `(guess, bulls, cows)`, this function should return the list of all four-digit numbers that are consistent with all these guesses, sorted in ascending order. Note that it is very much possible for the result list to be empty, if no four-digit integer is consistent with all of the `guesses`.

Start by creating a list of all four-digit numbers that do not contain any repeated digits. Loop through the individual guesses given, and for each guess, use a list comprehension to create a list of numbers that were in the previous list and are still consistent with the current guess. After you have done all that, oi jolly well then, old chap, "*When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth.*" —Sherlock Holmes

guesses	Expected result
`[(1234, 2, 2)]`	`[1243, 1324, 1432, 2134, 3214, 4231]`
`[(8765, 1, 0), (1234, 2, 1)]`	`[1245, 1263, 1364, 1435, 1724, 1732, 2734, 3264, 4235, 8134, 8214, 8231]`
`[(1234, 2, 2), (4321, 1, 3)]`	`[]`
`[(3127, 0, 1), (5723, 1, 0), (7361, 0, 2), (1236, 1, 0)]`	`[4786, 4796, 8746, 8796, 9746, 9786]`

This problem and its myriad generalizations to an exponentially larger number of possibilities (for example, otherwise the same game but played with English words) can be solved in more clever and efficient ways than the above **brute force** enumeration. However, we shall let that one remain a topic for a later course on advanced algorithms.

Sort list by element frequency

```
def frequency_sort(items):
```

Sort the given list of integer `items` so that its elements end up in the order of **decreasing frequency**, that is, the number of times that they appear in `items`. If two elements occur with the same frequency, they should end up in the **ascending** order of their element values with respect to each other, as is the standard practice in sorting things.

The best way to solve this problem is to "Let George do it", or whichever way you wish to put this concept of passing the buck, by having the Python `sort` function perform the sorting against custom sorting criterion, as done in the example script countries.py. Start by creating a dictionary to keep track of how many times each element occurs inside the array. Then, use these counts stored in that dictionary as the **sorting key** of the actual array elements, breaking ties on the frequency by using the actual element value.

(If you also happen to remember that the order comparison of Python tuples is **lexicographic**, you don't even have to exhaust yourself with the work needed to break these ties between two equally frequent values...)

items	Expected result
`[4, 6, 2, 2, 6, 4, 4, 4]`	`[4, 4, 4, 4, 2, 2, 6, 6]`
`[4, 6, 1, 2, 2, 1, 1, 6, 1, 1, 6, 4, 4, 1]`	`[1, 1, 1, 1, 1, 1, 4, 4, 4, 6, 6, 6, 2, 2]`
`[17, 99, 42]`	`[17, 42, 99]`
`['bob','bob','carl','alex','bob']`	`['bob','bob','bob','alex','carl']`

Calling all units, B-and-E in progress

```
def is_perfect_power(n):
```

A positive integer `n` is a [perfect power](#) if it can be expressed as the power `b**e` for some two integers `b` and `e` that are both **greater than one**. (Any positive integer `n` can always be expressed as the trivial power `n**1`, so we don't care about those.) For example, the integers 32, 125 and 441 are perfect powers since they equal `2**5`, `5**3` and `21**2`, respectively.

This function should determine whether the given positive integer `n` is a perfect power. Your function needs to somehow iterate through a sufficient number of possible combinations of `b` and `e` that could work, returning `True` right away when you find some `b` and `e` that satisfy `b**e == n`, and returning `False` when all possibilities for `b` and `e` have been tried and found wanting. Since `n` can get pretty large, your function should not examine too many combinations of `b` and `e` above and beyond those that are both necessary and sufficient to reliably determine the answer. Achieving this efficiency is the central educational point of this problem.

n	Expected result
42	False
441	True
469097433	True
12**34	True
12**34 – 1	False

A tester for this problem can be easily built on [Catalan's conjecture](#), a mathematical theorem about integer powers that says that after the special case of two consecutive perfect powers 8 and 9, whenever the positive integer `n` is a perfect power, `n–1` cannot be a perfect power. This theorem makes it easy to generate pseudorandom test cases with known answers, both positive and negative. For example, we don't have to slog through all possible ways to potentially express the number as an integer power to know for a fact from the get-go that the unimaginable behemoth `12345**67890–1` is not a perfect power. This illustrates nicely the common asymmetry between performing a computation to one direction, versus trying to reverse that computation and perform it to the other direction. Given an integer such as `49222352429520267040371132431220080664` but not the formula that originally produced it, it is not easy to tell whether it is a perfect power, or a perfect power plus minus one.

Lunar multiplication

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

This problem was inspired by another jovial [Numberphile](#) video "[Primes on the Moon](#)'' that you should watch first to get an idea of how the "lunar arithmetic" works. Formerly known as "dismal arithmetic", addition and multiplication operators of natural numbers are redefined so that **adding** two digits is done by taking their **maximum**, whereas **multiplying** two digits is done by taking their **minimum**. For example, 2 + 7 = 7 and 2 * 7 = 2. Unlike ordinary addition, there will never be a carry to the next column of digits, no matter how many individual digits are being added together. For numbers with multiple digits, addition and multiplication algorithms work exactly as you learned back when you were a little kid, except that the shifted digit columns that stem from lunar multiplication of the individual digits are added together in the same lunatic fashion.

These operators define a perfectly legitimate **algebraic structure** where important identities such as $ab = ba$, $(a + b) + c = a + (b + c)$ and $a(b + c) = ab + ac$ hold, along with everything that is entailed by those identities. The unit for addition is zero, same as in ordinary arithmetic, but the unit for multiplication is not one but [number nine](#), since $9n = n$ for any natural number n, as you can easily verify yourself. This function should compute and return the lunar product of two integers `a` and `b`.

a	b	Expected result
2	3	2
8	9	8
11	11	111
17	24	124
123	321	12321
357	64	3564
123456789	987654321	12345678987654321

Distribution of abstract bridge hand shapes

```
def hand_shape_distribution(hands):
```

This is a continuation of the earlier "Bridge hand shape" problem that asked you to compute the shape of one given bridge hand. In that problem, the shapes `[6, 3, 2, 2]` and `[2, 3, 6, 2]` were considered different, as they very much would be in the actual bidding and play of the hand. However, in this combinatorial generalized version of this problem, we shall consider two hand shapes like these two to be the same *abstract shape* if they are equal when we care only about the sorted counts of the suits, but don't care which particular suits they happen to be.

Given a list of bridge `hands`, each hand given as a list of 13 cards encoded the same way as in all of the previous card problems, create and return a Python dictionary that contains all abstract shapes that occur within `hands`, each shape mapped to its count of occurrences in `hands`. Note that Python dictionary keys cannot be lists (Python lists are mutable, and changing the contents of a dictionary key would break the internal ordering of the dictionary) so you need to represent the abstract hand shapes as immutable **tuples** that can be used as keys inside your dictionary.

As tabulated on "Suit distributions" in "Durango Bill's Bridge Probabilities and Combinatorics", there exist precisely 39 possible abstract shapes of thirteen cards, the most common of which is 4-4-3-2, followed by the shape 5-3-3-2. Contrary to intuition, the most balanced possible hand shape 4-3-3-3 turns out to be surprisingly unlikely, trailing behind even the less balanced shapes 5-4-3-1 and 5-4-2-2 that one might have intuitively assumed to be far less frequent. (Understanding why randomness tends to produce variance rather than converging to complete uniformity is a great aid in understanding many other counterintuitive truths about the behaviour of random processes in computer science and mathematics.)

For example, if it were somehow possible to give to your function the list of all 635,013,559,600 possible bridge hands and not run out of the heap memory in the Python virtual machine, the returned dictionary would contain 39 entries for the 39 possible hand shapes, two examples of these entries being `(4,3,3,3):66905856160` and `(6,5,1,1):4478821776`. (Our automated tester will try out your function with a much smaller list of pseudo-randomly generated bridge hands, but at least for the common hand types that you might expect to see every day at the daily duplicate of the local bridge club, the percentage proportions really ought not be that different from the exact answers if measured over a sufficiently large number of random hands.)

Flip of time

```
def hourglass_flips(glasses, t):
```

An hourglass is given as a tuple `(u,l)` (the second character is the lowercase letter `L`, not the digit one) for the number of minutes in the upper and lower chamber. After `m` minutes have elapsed, the state of that hourglass will be `(u-min(u,m),l+min(u,m))` so that the total amount of sand remains constant inside the same hourglass, and neither chamber ever contains any negative anti-sand. Flipping the hourglass `(u,l)` produces the hourglass `(l,u)`, of course.

Given a list of `glasses`, each of form `(u,0)` so that all sand is initially in the upper chamber, and the amount of time `t` to be measured, you can only wait for the first hourglass to run out of time after its `u` minutes, since any eyeball estimation of hourglasses during these measurements is not allowed. Once the time in the chosen hourglass runs out, you may instantaneously flip **any subset** of these glasses (note that this subset can be empty as you are not required to flip any glasses, not even the one that just ran out of sand) to continue to measure the remaining time `t-u`.

This function should return **the smallest possible number of individual hourglass flips** that can exactly measure `t` minutes, or `None` if this task is impossible. The base cases of recursion are when `t` equals zero or exactly `t` minutes remain in some hourglass (no flips are needed), or when `t` is smaller than the time remaining in any hourglass (no solution is possible). Otherwise, wait for the first hourglass to run out after its `u` minutes, and loop through all possible subsets of your hourglasses, recursively trying each flipped subset to best measure the remaining `t-u` minutes.

glasses	t	Expected result
`[(7, 0), (11, 0)]`	15	2 ([see here](#))
`[(4, 0), (6, 0)]`	11	None
`[(7, 0), (4, 0), (11, 0)]`	20	3
`[(16, 0), (21, 0)]`	36	6
`[(5, 0), (8, 0), (13, 0)]`	89	7

(As an implementation hint, the handy generator `itertools.product([0,1], repeat=n)` produces all 2^n possible n-element tuples made of `[0, 1]` to represent the possible subsets of n individual `glasses` that will be flipped for the recursive call.)

Fibonacci sum

```
def fibonacci_sum(n):
```

[Fibonacci numbers](#) are a cliché in introductory computer science, especially in teaching recursion where this famous combinatorial series is mainly used to reinforce the belief that recursion is silly... but all clichés became clichés in the first place because they were so good that everyone and their brother just kept using them! Let us therefore traipse around the [Zeckendorf's theorem](#), a more amazing property of these famous numbers: **every positive integer can be expressed exactly one way as a sum of distinct non-consecutive Fibonacci numbers**.

The simple **greedy algorithm** can be proven to always produce the desired breakdown of `n` into a sum of distinct non-consecutive Fibonacci numbers: always add into the list the largest possible Fibonacci number `f` that is at most equal to `n`. Then convert the rest given by `n-f` in this same manner, until nothing remains to be converted. To allow for automated testing, the list of Fibonacci numbers must be returned sorted in **descending** order.

n	Expected result
10	`[8, 2]`
100	`[89, 8, 3]`
1234567	`[832040, 317811, 75025, 6765, 2584, 233, 89, 13, 5, 2]`
10**10	`[7778742049, 1836311903, 267914296, 102334155, 9227465, 3524578, 1346269, 514229, 75025, 6765, 2584, 610, 55, 13, 3, 1]`
10**100	(a list of 137 terms, the largest of which starts with digits `921684571`)

Note how this greedy construction guarantees that the chosen Fibonacci numbers are distinct and cannot contain two consecutive Fibonacci numbers F_i and F_{i+1}, since otherwise also their sum F_{i+2} would have fit inside n and been used instead of F_{i+1} and thus would have prevented the use of F_{i+1} with the same argument. A curious kind of **infinite ascent** proof towards the unknown but finite upper bound n in a "well-topped" sequence...

Wythoff array

```
def wythoff_array(n):
```

[Wythoff array](#) (see [the Wikipedia article](#) for illustration) is an infinite two-dimensional grid of integers that is seeded with one and two to start the first row. In each row, each element equals the sum of the previous two elements, so the first row contains precisely the Fibonacci numbers.

The first element of each later row is **the smallest integer c that does not appear anywhere in the previous rows**. Since every row is strictly ascending and grows exponentially fast, you can find this out by looking at relatively short finite prefixes of these rows. To determine the second element of that row, let `a` and `b` be the first two elements of the previous row. If the difference `c-a` equals two, the second element of that row equals `b+3`, and otherwise that element equals `b+5`. This construction guarantees the Wythoff array to be an **interspersion** of positive integers; every positive integer will appear **exactly once** in the entire infinite grid, with no gaps or duplicates! (This result nicely highlights the deeper combinatorial importance of the deceptively simple Fibonacci numbers as potential building blocks of other integers and integer sequences.)

The difficulty in this problem is determining the first two elements of each row, since the rest of the row is then utterly trivial to generate as far as needed. This function should return the position of `n` inside the Wythoff array as a two-tuple of (`row,    column`), both row and column numbers starting from zero.

n	Expected result
21	(0, 6)
47	(1, 5)
1042	(8, 8)
424242	(9030, 6)
39088169	(0, 36)
39088170	(14930352, 0)

Rooks with friends

```
def rooks_with_friends(n, friends, enemies):
```

Those dastardly rooks have once again gone on a rampage on a generalized *n*-by-*n* chessboard, just like in the earlier problem of counting how many squares were safe from their occupants being pulverized into dust under the wheels of these lumbering juggernauts. Each individual rook is again represented as a two-tuple `(row,column)` of the coordinates of the square it stands on. However, in this version of the problem we introduce a pinch of ethnic nepotism to the mix so that some of these rooks are your **friends** (same colour as you) while the others are your **enemies** (the opposite colour from you).

Friendly rooks protect the chess squares by standing between them and any enemy rooks that might want to threaten those squares. An enemy rook can attack only those squares in the same row or column that do not enjoy the protection of any friendly rook standing between them. Given the board size n and the lists of `friends` and `enemies`, count how many empty squares on the board are safe from the enemy rooks.

n	friends	enemies	Expected result
4	[(2,2), (0,1), (3,1)]	[(3,0), (1,2), (2,3)]	2
4	[(3,0), (1,2), (2,3)]	[(2,2), (0,1), (3,1)]	2
8	[(3,3), (4,4)]	[(3,4), (4,3)]	48
100	[(r, (3*r+5) % 100) for r in range(1, 100, 2)]	[(r, (4*r+32) % 100) for r in range(0, 100, 2)]	3811

Possible words in Hangman

```
def possible_words(words, pattern):
```

Given a list of possible `words`, and a `pattern` string that is guaranteed to contain only lowercase English letters `a` to `z` and asterisk characters `*`, create and return a sorted list of words that match the `pattern` in the sense of the famous pen-and-paper guessing game of [Hangman](#).

Any `pattern` can only match words of the exact same length. In all positions where the `pattern` contains some letter, the word must contain that same letter. In positions where the `pattern` contains an asterisk, the word character in that position may not be any of the letters that already occur inside the pattern. In the game of Hangman, all occurrences of such a letter would have been revealed in the earlier round when that letter was the current guess.

For example, the words `'bridge'` and `'smudge'` both match the pattern `'***dg*'`. However, the words `'grudge'` and `'dredge'` would **not** match that same pattern, since the first asterisk may not be matched with either of the letters `'g'` or `'d'` that appear inside the `pattern`.

pattern	Expected result (using wordlist `words_sorted.txt`)
`'***dg*'`	`['abedge', 'aridge', 'bludge', 'bridge', 'cledge', 'cledgy', 'cradge', 'fledge', 'fledgy', 'flidge', 'flodge', 'fridge', 'kludge', 'pledge', 'plodge', 'scodgy', 'skedge', 'sledge', 'slodge', 'sludge', 'sludgy', 'smidge', 'smudge', 'smudgy', 'snudge', 'soudge', 'soudgy', 'squdge', 'squdgy', 'stodge', 'stodgy', 'swedge', 'swidge', 'trudge', 'unedge']`
`'*t*t*t*'`	`['statute']`
`'a**s**a'`	`['acystia', 'acushla', 'anosmia']`
`'*ikk**'`	`['dikkop', 'likker', 'nikkud', 'tikker', 'tikkun']`

Factoring factorials

```
def factoring_factorial(n):
```

The [fundamental theorem of arithmetic](#) tells us that every positive integer can be broken down to the product of its **prime factors** in exactly one way. Given a positive integer n>1, create and return a list that contains all prime factors of n!, the product of all positive integers up to n, along with the exponents in the prime factorization. These prime factors should be listed in ascending order, each prime factor given as tuple (p, e) where p is the prime factor and e its exponent.

Since n will get into the thousands in the automated tester, you should accumulate the prime factors of n! separately for each individual term of the factorial as you go, rather than first computing the entire n! and only then chopping down that gargantuan beast back into its prime factors.

n	Expected result
5	[(2, 3), (3, 1), (5, 1)]
10	[(2, 8), (3, 4), (5, 2), (7, 1)]
20	[(2, 18), (3, 8), (5, 4), (7, 2), (11, 1), (13, 1), (17, 1), (19, 1)]
100	[(2, 97), (3, 48), (5, 24), (7, 16), (11, 9), (13, 7), (17, 5), (19, 5), (23, 4), (29, 3), (31, 3), (37, 2), (41, 2), (43, 2), (47, 2), (53, 1), (59, 1), (61, 1), (67, 1), (71, 1), (73, 1), (79, 1), (83, 1), (89, 1), (97, 1)]
1000	(a list with 168 terms, the first five of which are [(2, 994), (3, 498), (5, 249), (7, 164), (11, 98)])
10000	(a list with 1229 terms, the first five of which are [(2, 9995), (3, 4996), (5, 2499), (7, 1665), (11, 998)])

All paths lead to Rome

```
def lattice_paths(x, y, tabu):
```

You are standing on the point `(x, y)` in the **lattice grid** of pairs of nonnegative integers, and wish to make your way to the **origin** point `(0, 0)`. At any point, you are only allowed to move either one step left or one step down. Furthermore, you are never allowed to step into any of the points in the `tabu` list. This function should add up the number of different paths that lead from the point `(x, y)` to the origin `(0, 0)` under these constraints.

This constrained variation of the classic combinatorial problem turns out to have a reasonably straightforward recursive solution. As the base case, the number of paths from the origin `(0, 0)` to itself `(0, 0)` equals one, for the empty path. If the point `(x, y)` is in the `tabu` list, the number of paths from that point `(x, y)` to the origin equals zero. The same holds for all points whose either coordinate `x` or `y` is negative. Otherwise, the number of paths from the point `(x, y)` to origin `(0, 0)` equals the sum of paths from the two neighbours `(x-1, y)` and `(x, y-1)`.

However, this simple recursion branches into an exponential number of possibilities and would therefore be far too slow for us to execute. Therefore, you should either **memoize** the recursion, or even better, build up a two-dimensional list whose entries are the individual subproblem solutions, and fill this list with two `for`-loops instead of recursion, these loops filling the list in order that when these loops arrive at position `[x][y]`, the results for positions `[x-1][y]` and `[x][y-1]` have already been computed. (This idea to use loops to fill out the memoization tables with for-oops instead of recursion is called **dynamic programming**.)

x	y	tabu	Expected result
3	3	[]	20
3	4	[(2,2)]	17
10	5	[(6, 1), (2, 3)]	2063
6	8	[(4,3), (7,3), (7,7), (1,5)]	1932
10	10	[(0,1)]	92378
100	100	[(0,1), (1,0)]	0

Be there or be square

```
def count_squares(points):
```

This problem is adapted from "Count the Number of Squares" at Wolfram Challenges, so you might first want to check that page out for some illustrative visualizations of this problem.

Your function is given a `set` of `points`, each point a two-tuple `(x, y)` where `x` and `y` are positive integers. This function should count how many **squares** these points define so that all four corners of the square belong to the given set of `points`, and return this count. Note that these squares are not required to be **axis-aligned** so that their sides would have to be either horizontal and vertical. As long as all four sides have the exact same length, be that length an exact integer or some ir*ratio*nal number, well, that makes us "square" at least in my book, pardner!

To identify four points that constitute a square, note how every square has **bottom left corner** point `(x, y)` and **direction vector** `(dx, dy)` towards its upper left corner point that satisfies the constraints `dx>=0` and `dy>0`, so that the three points `(x+dx, y+dy)`, `(x+dy, y-dx)` and `(x+dx+dy, y-dx+dy)` are the top left, bottom right and top right corners of that square, respectively, all three also included in `points`. You can therefore loop through all possibilities for the bottom left point `(x, y)` and the direction vector `(dx, dy)` to be guaranteed to find all squares in the grid. But still try to be economical in these loops so this function sweeps up these squares swiftly.

points	Expected result
`[(0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2)]`	6
`[(4,3), (1,1), (5,3), (2,3), (3,2), (3,1), (4,2), (2,1), (3,3), (1,2), (5,2)]`	3
`[(x, y) for x in range(1, 10) for y in range(1, 10)]`	540
`[(3,4), (1,2), (3,2), (4,5), (4,2), (5,3), (4,1), (5,4), (3, 5), (2,4), (2,2), (1,1), (4,4), (2,5), (1,5), (2,1), (2,3), (4, 3)]`	15

Split within perimeter bound

```
def perimeter_limit_split(a, b, p):
```

Rectangles are represented as tuples `(a, b)` where `a` and `b` are positive integers. A rectangle can be cut into two smaller rectangles with the same total area using either a straight horizontal or a straight vertical cut along the integer axis of your choice. For example, one way among all possible ways to cut the rectangle `(5, 8)` is to cut it into `(2, 8)` and `(3, 8)` in the first dimension. Another way would be to cut it into `(5, 4)` and `(5, 4)` in the second dimension. These smaller pieces can then be further cut into smaller pieces, as long as the length of the side being cut is at least two. Also, at the risk of disappointing any lateral thinkers reading this, you are not allowed to cut multiple pieces in one cutting motion by stacking those pieces on top of each other.

Your task is to cut the given initial rectangle `(a, b)` into smaller pieces until the **perimeter** of each individual piece `2*(a+b)` is at most equal to `p`, the maximum allowed perimeter length. Since the optimal cuts are not generally unique, this function should compute and return the minimum number of cuts necessary to achieve some optimum result.

This problem is best solved with recursion. If the perimeter of the piece is within the limit `p`, return zero. Otherwise, loop through all possible ways to cut this piece into two smaller pieces, and recursively compute the best possible ways to cut up the resulting two pieces with `m1` and `m2` moves, respectively. Return the smallest value for `1+m1+m2` that you find. Since this branching recursion will visit its subproblems an exponential number of times, you might want to sprinkle some `@lru_cache` memoization on top of it, just to rein in that exponential tangle and downgrade it into a quadratic one.

a	b	p	Expected result
11	1	12	2
9	13	5	116
8	31	14	20
91	21	54	11
201	217	307	7

Sum of distinct cubes

```
def sum_of_distinct_cubes(n):
```

Positive integers can sometimes be broken down into sums of **distinct** cubes of **positive** integers, many of them even in multiple ways. This function should find and return the list of distinct cubes whose sum equals the given positive integer n. Should n allow several breakdowns into sums of distinct cubes, this function must return the **lexicographically highest** solution that starts with the largest possible first number a and is from there followed by the lexicographically largest representation of the rest of the number n–a*a*a. For example, when called with n=1072, this function should return [10, 4, 2] instead of [9, 7]. Whenever it is impossible to break the parameter n into a sum of distinct cubes, return None.

Unlike in the earlier, much simpler question of expressing n as a sum of exactly two squares, the result can now contain any number of distinct terms. This problem, like almost all such problems that cause an exponential blowup of two-way "take it or leave it" decisions, is usually best solved with recursion that examines both branches of each such decision and returns the one that gives a better result. The base case is when n itself is a cube. Otherwise, loop down through all possible values of the first element a in the result list, and for each such a, break down the remaining number n–a*a*a into a sum of distinct cubes using only integers that are smaller than your current a. Again, to let your function efficiently [gleam these cubes](#) even for large n, it might be a good idea to prepare something that allows you to quickly determine whether the given integer is a cube, and whenever it is not, to find the largest integer whose cube is smaller.

n	Expected result
8	[2]
11	None
855	[9, 5, 1]
sum([x*x*x for x in range(11)]	[14, 6, 4, 1]
sum([x*x*x for x in range(1001)])	[6303, 457, 75, 14, 9, 7, 5, 4]

This problem is intentionally restricted to positive integers to keep the number of possibilities to iterate through finite. Since the cube of a negative number is also negative, no finite upper bound for search would exist if negative numbers were allowed. Breaking some small and simple number into a sum of exactly three cubes can suddenly blow up to be a [highly nontrivial problem...](#)

Fractional fit

```
def fractional_fit(fs):
```

You are given a list of integer two-tuples `(a, b)` for which `0<=a<b`. When interpreted as an integer fraction from the expression `f=Fraction(a,b)`, these values fall within the unit interval. Given the list `fs` of available fractions, find the length of the longest possible list that can be constructed from those fractions under the following series of constraints:

- The first *two* fractions must lie in different *halves* of the unit interval. That is, one of them must satisfy `0<=f<1/2`, and the other one must satisfy `1/2<=f<1`.
- The first *three* fractions must all lie in different *thirds* of the unit interval.
- The first *four* fractions must all lie in different *quarters* of the unit interval.
- The first *five* fractions must all lie in different *fifths*... and so on!

Your function should return the length of the longest possible such list. Note how the order of your chosen fractions in this list is important. For example, `[(1, 4), (8, 9), (3, 7)]` works, but the order `[(3, 7), (1, 4), (8, 9)]` does not work, since both fractions 3/7 and 1/4 lie on the first half of the unit interval.

fs	Expected result
`[(6, 12), (0, 5), (5, 7), (4, 11)]`	3
`[(5, 15), (0, 5), (7, 19), (17, 23), (5, 18), (10, 11)]`	4
`[(34, 52), (61, 82), (71, 80), (36, 76), (15, 84), (36, 53), (79, 80), (5, 67), (31, 62), (15, 57)]`	6
`[(171, 202), (41, 42), (43, 85), (164, 221), (97, 130), (12, 23), (15, 62), (41, 128), (11, 25), (31, 49), (6, 35), (85, 137), (16, 241), (82, 225), (11, 26), (74, 149), (127, 203)]`	10

The same Martin Gardner column as in the Bulgarian solitaire problem also discussed this puzzle as an example of a seemingly unlimited process where it turns out that even if you were allowed to freely choose your fractions, every such sequence will inevitably have painted itself into a corner after at most 17 steps. No matter how you twist and turn, two of these fractions will inevitably fall on the same 1/18th of the unit interval, thus making your next chosen fraction, if you pardon a dad joke better suited for a "[Jumble](#)" newspaper puzzle, a "moot point".

Followed by its square

```
def square_follows(it):
```

Unlike our previous functions that receive entire lists as parameters, this function receives a lazy **iterator** guaranteed to produce some finite sequence of **strictly increasing positive integers**, but gives no guarantees of how long that sequence will be, and how large the numbers inside it or the gaps between them could grow to be. This iterator **is not a list** that would grant you **random access** to any of its element based on its position and this way let you jump back and forth as your heart desires. Therefore, processing the elements given by this iterator **must be done in a strictly sequential one-pass fashion** so that each element is examined only once, when it is its turn to come out of the sequence. However, you can and should use any available Python data structures to keep track of whatever intermediate results you need to store for your future decisions.

This function should create and return an ordinary Python list that contains, in ascending order, precisely those elements inside the sequence produced by the parameter iterator `it` whose square also appears somewhere later in that sequence.

it	Expected result
`iter([3, 4, 6, 8, 11, 13, 18])`	`[]`
`iter([2, 3, 4, 5, 8, 9, 11, 16])`	`[3, 4]`
`iter(range(1, 10**6))`	(a list that consists of exactly 998 elements, the first five of which are `[2, 3, 4, 5, 6]` and the last five are `[995, 996, 997, 998, 999]`)
`iter([x*x for x in range(1, 1000)])`	`[4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961]`
`iter([x**10 for x in range(1, 100)])`	`[1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824, 3486784401]`

Count maximal layers

```
def count_maximal_layers(points):
```

A point `(x1, y1)` on the two-dimensional plane **dominates** another point `(x2, y2)` if both inequalities `x1>x2` and `y1>y2` hold. A point inside the given list of `points` is **maximal** if it is not dominated by any other point in the list. Note that unlike in boring old one dimension, a maximal point does not need to dominate every other point. (Some lists may even contain maximal points that do not dominate *any* other point in that list!) Unlike in one dimension, a list of points on the two-dimensional plane can contain any number of maximal points, which then form the **maximal layer** for that particular list of points. For example, the points `(3, 10)` and `(9, 2)` are the maximal layer for `[(1, 5), (8, 1), (3, 10), (2, 1), (9, 2)]`.

Given a list of `points` from the upper right quadrant of the infinite two-dimensional plane so that all coordinates are guaranteed to be nonnegative, this function should compute how many times one would have to remove every point in the current maximal layer from the list for that list to become empty, and return that count.

points	Expected result
`[(1, 3), (2, 2), (3, 1)]`	1
`[(1, 5), (3, 10), (2, 1), (9, 2)]`	2
`[(x, y) for x in range(10) for y in range(10)]`	10
`[(x, x**2) for x in range(100)]`	100
`[(x, x**2 % 91) for x in range(1000)]`	28
`[((x**3) % 891, (x**2) % 913) for x in range(10000)]`	124

This is again one of those problems whose point (heh) and motivation is making this function to be fast enough to finish reasonably quickly even for a large list of points. Try not to do much more work than is necessary to identify and remove the current maximal points. Start by noting how each point can potentially be dominated only by those points whose distance from the origin `(0, 0)` is strictly larger. And it doesn't even matter which particular distance metric you use...

Maximum checkers capture

```
def max_checkers_capture(n, x, y, pieces):
```

Even though we again find ourselves on a generalized *n*-by-*n* chessboard, this time we are playing a variation of [checkers](#) so that your lone **king** currently stands at the coordinates (x, y), and the parameter `pieces` is a `set` that contains the positions of all of the opponent's pawns. This function should compute and return the maximum number of pieces that your king could potentially capture in a single move.

Some variants of checkers allow **leaping kings** that can capture pieces across arbitrary distances. For simplicity, the kings of this problem are more subdued so that they capture a piece only one step away to the four diagonal directions (the list `[(-1, 1), (1, 1), (1, -1), (-1, -1)]` might come handy) when the square behind the opponent piece in that diagonal direction exists and is vacant. Your king can then capture that piece by jumping over it into that vacant square, immediately removing that captured piece from the board. However, unlike in chess where each move can capture at most one enemy piece, the chain of captures continues from the square that your king jumps into, as long as more opponent pieces are available for capture.

The maximum number of pieces available for capture in one move is best computed with a method that locally loops through the four diagonal directions from the current position of the king. For each such direction that contains an opponent piece with a vacant space behind it, remove that opponent piece from the board, and recursively compute the number of pieces that can be captured from the vacant square that your king jumps into. Once that recursive call has returned, restore the first captured piece on the board, and continue looping to the next diagonal direction. Return the largest number that can be achieved from the four possible directions for the first capture, plus one for the piece captured first.

n	x	y	pieces	Expected result
5	0	2	set([(1,1), (3,1), (1,3)])	2
7	0	0	set([(1,1), (1,3), (3,3), (2,4), (1,5)])	3
8	7	0	set([(x, y) for x in range(2, 8, 2) for y in range(1, 7, 2)])	9

Collatzy distance

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

Let us make up a rule that says that from a positive integer n, you are allowed to move in a single step into either integer 3*n+1 or n//2. Even though these formulas were obviously inspired by the wonderfully chaotic [Collatz conjecture](#) that we encountered during the lecture of unlimited iteration, in this problem the parity of n does not constrain you to deterministically use just one of these formulas, but you may use either formula regardless of whether your current n is odd or even. This function should determine how many steps are needed to get from start to end, assuming that you wisely choose each step to minimize the total number of steps in the path.

This problem can be solved with **breadth-first search** the following way, given here as a sneaky preview for the later course that explains that and other **graph traversal** algorithms. For the given start value, the zeroth **layer** is the singleton list [start]. Once you have computed layer k, the next layer k+1 consists of all integers 3*n+1 and n//2 where n goes through all numbers in the previous layer k. For example, if start = 4, the first four layers numbered from zero to three would be [4], [13, 2], [40, 6, 7, 1] and [121, 20, 19, 3, 22, 0], the elements inside each layer possibly listed in some other order as your code found them.

Then, keep generating layers from the previous layer inside a while-loop until the goal number end appears, at which point you can immediately return the current layer number as the answer.

start	end	Expected result
10	20	7
42	42	0
42	43	13
76	93	23
1000	10	9

Once you get this function to work correctly, you can think up clever ways to speed up its execution. For starters, notice that any integers that have already made an appearance in an earlier layer before the current layer can be ignored while constructing the current layer.

Van Eck sequence

```
def van_eck(n):
```

Compute and return the *n*:th term of the [Van Eck sequence](#). The first term in the first position $i = 0$ equals zero. The term in later position i is determined by the term x in the previous position i - 1:

- If the position i - 1 was the first appearance of that term x, the term in the current position i equals zero.
- Otherwise, if the previous term x has now appeared at least twice in the sequence, the term in the current position i is given by the position difference i - 1 - j, where j is the position of the second most recent appearance of the term x.

The sequence begins with 0, 0, 1, 0, 2, 0, 2, 2, 1, 6, 0, 5, 0, 2, 6, 5, 4, 0, 5, 3, 0, 3, 2, 9, ... so you see how terms start making repeated appearances right away, especially the zero that automatically follows the first appearance of each term.

In the same spirit as in the earlier problem that asked you to generate terms of the [Recamán's sequence](#), unless you want to spend the rest of your day waiting for the automated tester to finish, this function should not repeatedly loop backwards through the already generated sequence to look for the most recent occurrence of each term that it generates. You should rather use a Python dictionary to remember the terms already seen in the sequence, mapped to the positions of their most recent appearance. With this dictionary, you don't need to explicitly store the entire sequence, but only the previous terms and positions that are relevant for the future.

n	Expected result
0	0
10	0
1000	61
10**6	8199
10**8	5522779

Next higher zigzag

```
def next_zigzag(n):
```

The definition of **zigzag numbers** is exactly the same as in the earlier problem that asked you to identify whether the given number is a zigzag number. In this problem, the function is given a positive integer `n` that **is guaranteed to be a zigzag number**. This function should return the positive integer `m>n` that is **the next higher zigzag number** up the line, so that none of the numbers strictly between `n` and `m` is also a zigzag number.

This problem could in principle be solved by counting up from `n+1` and using your earlier `is_zigzag` function to determine whether you have reached the next higher zigzag number. Unfortunately, as you can see in the table below, the gaps between consecutive zigzag numbers grow up to become arbitrarily large, so you need some more analytical approach based on the digit pattern found at the end of the number. This digit pattern might also be arbitrarily long before it allows you to decide where the next zigzag number lies, but that is why loops exist...

n	Expected result
801	802
1082845	1082846
92398	92401
27398	27450
89292523198989898	89292523230101010

Tips and trips

```
def trips_fill(words3, pattern, tabu):
```

A string of lowercase letters is **trip-filled** if every 3-substring (a "trip") inside it is a three-letter word from `words3`, the list of such words in sorted order. Some examples are `'pacepad'`, `'baganami'` and `'udianaahunat'`, provided that we use some of the more obscure three-letter words such as `'cep'` and `'gan'` from `words_sorted.txt`. Given a `pattern` made of lowercase letters and asterisks, this function should find and return **the lexicographically lowest trip-filled string** that can be constructed by replacing each asterisk of `pattern` with any letter of your choice. Furthermore, **no individual trip may appear in the solution more than once.** If no solution exists for the given `pattern`, return `None`.

This problem is best solved with recursion similar to `decode_morse` in the morse.py example. This function should loop through all trips that "fit into" the first three characters of `pattern`. (Let `bisect_left` be your friend in this time of need.) For each fitting trip, recursively fill in the rest of the `pattern` from position 1 onwards, with your currently chosen trip "stamped" into the first three characters of that `pattern`. As this function needs to return only the first complete solution instead of generating all of them, ordinary recursion suffices instead of an entire recursive generator. To enforce the uniqueness of the chosen trips, this recursion should `append` each trip into the `tabu` list on the way down, and then `pop` that trip out of the `tabu` list immediately after returning. (In the top level calls from the tester, `tabu` is always an empty list.)

pattern	Expected result (using the wordlist `words_sorted.txt`)
`'**q'`	`'aeq'`
`'*yo***'`	`'iyobaa'`
`'*m*gv**o'`	None
`'*ef***da**s'`	`'befloadabas'`
`'**l**a*l*ai*'`	`'ablobaalcaid'`
`'*e'*8`	`'lekereseyewemeve'`
`'*'*100`	`'aaahabaalabbloadabcdfthadcabdeaddtdryadebbsfmtgnubcfibabeamablschaeraboafbibobaccmlxxiseanaambdspace'`

Balanced ternary

```
def balanced_ternary(n):
```

The integer two and its powers abound all over computing whereas the number three seems to be mostly absent, at least until we get to the **theory of computation** and **NP-complete problems** where the number three turns out to be a very different thing from two not only quantitatively, but qualitatively. Stepping from two to three is where computational complexity usually emerges.

Normally we represent integers in base 10, each digit giving the coefficient of some power of 10. Since every positive integer can be uniquely represented as a sum of powers of any base, computers internally encode those same integers in the simpler (for machines, that is) **binary** of base two. Both of these schemes need [special tricks to represent negative numbers](#) in the absence of an explicit negation sign. The [balanced ternary](#) representation of integers uses the base three instead of two, with **signed** coefficients from three possibilities -1, 0 and +1. Every integer (positive or negative) can be broken down into a sum of signed powers of three exactly one way. Furthermore, the balanced ternary representation is perfectly symmetric around zero; the representation for *-n* can be constructed simply by flipping the signs of the terms of the representation of *+n*.

Given an integer *n*, return the list of signed powers of three that add up to *n*, listing these powers in the descending order of their absolute values. This problem could be solved in a few different ways. The page "[Balanced ternary](#)" shows one way to do this by first converting *n* to **unbalanced ternary** (this is just base three with coefficients 0, 1 and 2) and taking it from there. Another way to achieve the same end is to first find *p*, the highest power of 3 that is less than equal to *n*. Then, depending on the value of *n - p*, you either take *p* into the result and convert *n - p* for the rest, or take *3p* into the result and convert *n - 3p* for the rest. All roads lead to Rome.

n	Expected result
5	[9, -3, -1]
-5	[-9, 3, 1]
42	[81, -27, -9, -3]
-42	[-81, 27, 9, 3]
100	[81, 27, -9, 1]
10**6	[1594323, -531441, -59049, -6561, 2187, 729, -243, 81, -27, 1]

Lords of Midnight

`def midnight(dice):`

Midnight (see [the Wikipedia page](#) for the rules) is a dice game where the player has to choose which of the six dice to keep and which to re-roll to maximize his final score. However, all your hard work with the previous problems has now mysteriously rewarded you with the gift of perfect foresight (as the French might say, you might be a descendant of [Madame de Thèbes](#)) that allows you to predict what pip values each individual die will produce in its entire sequence of future rolls, expressed as a sequence such as `[2, 5, 5, 1, 6, 3]`. Aided with this foresight, your task is to return the maximum total score that could be achieved with the given dice rolls.

The argument `dice` is a list whose each element is a sequence of the pip values that particular die will produce when rolled. (Since this game will necessarily end after at most six rolls, this given future sequence needs to be only six elements long.) Note that the rules require you to keep at least one die in each roll, which is why the trivial algorithm "First choose the two dice that you will use to get the 1 and 4, and add up the maximum pip values for the four remaining dice" does not work, as demonstrated by the first row of the following table.

dice	Expected result
`[[3, 4, 6, 6, 6, 2], [3, 2, 6, 2, 3, 3], [2, 2, 2, 2, 2, 3], [6, 1, 4, 2, 2, 2], [2, 2, 2, 3, 2, 3], [2, 3, 3, 3, 3, 2]]`	14
`[[2, 6, 2, 5, 2, 5], [5, 3, 3, 2, 5, 3], [2, 2, 2, 2, 5, 2], [3, 6, 3, 2, 2, 5], [6, 2, 2, 6, 3, 2], [2, 2, 3, 2, 2, 2]]`	0
`[[2, 6, 2, 1, 3, 3], [2, 2, 2, 2, 2, 3], [2, 2, 4, 3, 6, 6], [4, 5, 6, 3, 2, 5], [2, 4, 2, 6, 5, 3], [2, 2, 2, 2, 2, 3]]`	17
`[[3, 4, 6, 6, 6, 2], [3, 2, 6, 2, 3, 3], [2, 2, 2, 2, 2, 3], [6, 1, 4, 2, 2, 2], [2, 2, 2, 3, 2, 3], [2, 3, 3, 3, 3, 2]]`	14
`[[2, 3, 5, 3, 2, 2], [1, 3, 2, 3, 6, 4], [3, 2, 3, 3, 3, 5], [3, 6, 4, 6, 2, 3], [2, 3, 3, 2, 3, 2], [3, 5, 3, 5, 1, 2]]`	17

To make the test cases more interesting, the automated tester is programmed to produce fewer ones, fours and sixes than the probabilities of fair dice would normally yield. No law of nature or man stands against that sort of tactical placement of your thumb on the scales.

Optimal crag score

```
def optimal_crag_score(rolls):
```

An earlier problem asked you to compute the best possible score for a single roll in the dice game of crag. In this problem, we will now play the game for multiple rolls, again aided with the same gift of perfect foresight as in the previous "Lords of Midnight" problem so that given the knowledge of the entire future sequence of `rolls`, this function should return the highest possible score that can be achieved with those rolls under the constraint that **the same category cannot be used more than once**, as in the rules of the actual game of crag.

This problem will require recursive search, and the techniques that you might come up with to speed up its execution by pruning away search branches that cannot lead to optimal solutions will be highly important for your future algorithms courses. Note that the greedy algorithm "sort the rolls in the descending order of their maximum possible individual score, and then use each roll for its highest scoring remaining category" does not work, since several rolls might fit in the same category, and yet are not equally good choices with respect to the remaining categories.

rolls	Expected result
`[(1, 6, 6), (2, 5, 6), (4, 5, 6), (2, 3, 5)]`	101
`[(3, 1, 2), (1, 4, 2)]`	24
`[(5, 1, 1), (3, 5, 2), (2, 3, 2), (4, 3, 6), (6, 4, 6), (4, 5, 2), (6, 4, 5)]`	74
`[(3, 1, 2), (1, 4, 2), (5, 2, 3), (5, 5, 3), (2, 6, 3), (1, 1, 1), (5, 2, 5)]`	118
`[(1, 5, 1), (5, 5, 6), (3, 2, 4), (4, 6, 1), (4, 4, 1), (3, 2, 4), (3, 4, 5), (1, 2, 2)]`	33

Forbidden substrings

```
def forbidden_substrings(letters, n, tabu):
```

This function should construct and return a list of all *n*-character strings that consists of given `letters` under the constraint that a string may not contain any of the substrings in the `tabu` list. The list of strings should be returned in sorted alphabetical order. This problem also needs some branching recursion to work out the potentially exponential number of possible combinations.

letters	n	tabu	Expected result
'AB'	4	['AA']	['ABAB', 'ABBA', 'ABBB', 'BABA', 'BABB', 'BBAB', 'BBBA', 'BBBB']
'ABC'	3	['AC', 'AA']	['ABA', 'ABB', 'ABC', 'BAB', 'BBA', 'BBB', 'BBC', 'BCA', 'BCB', 'BCC', 'CAB', 'CBA', 'CBB', 'CBC', 'CCA', 'CCB', 'CCC']
'ABC'	6	['BB', 'CCCA', 'CB', 'CA', 'CBBCC', 'BA']	['AAAAAA', 'AAAAAB', 'AAAAAC', 'AAAABC', 'AAAACC', 'AAABCC', 'AAACCC', 'AABCCC', 'AACCCC', 'ABCCCC', 'ACCCCC', 'BCCCCC', 'CCCCCC']
'ABCD'	7	['DBBD', 'CB', 'BBCC', 'DB']	(a list that contains a total of 6436 words)
'Z'	10**100	['ZZZ']	[ ] (and that answer needs to come out right away instead of after about 10**80 years)

Go for the Grand: Infinite Fibonacci word

```
def fibonacci_word(k):
```

[Fibonacci words](#) are strings defined analogously to [Fibonacci numbers](#). The recursive definition starts with two base cases S_0 = `'0'` and S_1 = `'01'`, followed by the recursive rule $S_n = S_{n-1}S_{n-2}$ that concatenates the two previous Fibonacci words. See the Wikipedia page for more examples. Especially importantly for this problem, notice how the length of each Fibonacci word equals that particular Fibonacci number. Even though we cannot actually generate the infinite Fibonacci word S_∞ because it would be infinitely long, we can construct the character at any particular position k of S_∞ in a finite number of steps by realizing that after generating some word S_n that is longer than k, every longer word S_m for $m > n$, and therefore also the infinite word S_∞, must start with the exact same prefix S_n and therefore contain that asked character in the position k.

This function should compute the k:th character of the infinite Fibonacci word, the position counting done again starting from zero as usual. Since this is the last problem of this entire course, to symbolize your reach towards the infinite as you realize that you can simply ignore any arbitrary limits imposed by the laws of man and **go for the grand**, your function must be able to work for such unimaginably large values of k that they make even one googol `10**100` seem like you could wait it out standing on your head. The entire universe would not have enough atoms to encode the entire string S_n for such a large n to allow you to look up its k:th character. Instead, you need to apply the recursive formula, the list of Fibonacci numbers that you will dynamically compute as far as needed, and the [self-similar fractal nature of the infinite Fibonacci word](#).

k	Expected result
0	`'0'`
1	`'1'`
10	`'0'`
10**6	`'0'`
10**100	`'0'`
10**100 + 1	`'1'`
10**10000	`'0'`