109 Python Problems for CCPS 109

This document contains the specifications for the **graded lab problems** for the course **CCPS 109 Computer Science I**, as taught by [Ilkka Kokkarinen](). All computer science teachers are free to use and adapt this material for their own courses as they see fit, with proper attribution. The instructor has collected these problems from a gallimaufry of sources ranging from the lab and exam problems of his old Java version of this course to a multitude of programming puzzles and challenge sites such as [LeetCode](), [CodeAbbey](), [Stack Exchange Code Golf](), and [Wolfram Programming Challenges](). Many recreational mathematics columns by Martin Gardner and his spiritual disciples have also inspired many of these problems.

For the purposes of submission and grading, you **must implement all these functions in a single Python source code file** that **must** be named `labs109.py`. **Do not submit the individual functions as you go, but submit these labs all at once at the end of the course**. (Until then, make sure to regularly back up your work in at least two separate locations!)

You can run the `tester109.py` script at any time to execute the tests for the functions that you have completed so far. These tests are executed in the same order that your functions appear in the `labs109.py` file. The test battery for each individual function should take at most a couple of seconds to complete. If some test takes minutes to finish, your code is too slow and its logic desperately needs to be streamlined, usually by eliminating one level of nesting from your loops.

[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**.

Your functions may assume that the arguments given by the automated tester are legal and valid as given in the problem specification. Your functions do not need to perform any error detection or recovery from nonsensical arguments. Note that zero integers and empty lists are often legitimate arguments, and your functions must handle them correctly in such cases.

Past students **Shu Zhu Su**, **Mohammed Waqas** and **Zhouqin He** deserve thanks for going far above and beyond the call of duty in submitting their solutions that either revealed several errors in the original problem specifications and private model solutions, or raised the level of belief to the correctness of the original solution now independently verified. All remaining errors, ambiguities and inconsistencies are the sole responsibility of Ilkka Kokkarinen. Report them to `ilkka.kokkarinen@gmail.com` as you find them to inch 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 for the course, calculate and return the letter grade that would appear in the Ryerson grades transcript, as defined on the page [Ryerson Grade Scales](). The letter grade should be returned as a string that consists of the uppercase letter followed by the possible modifier `'+'` or `'-'`. The 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 just to use an **if-else ladder**.)

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

Ascending list

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

Determine whether the sequence of `items` is **strictly ascending** so that each element is **strictly larger** (and not just merely equal to) than the element that precedes it. Return `True` if that is the case, 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. Many people find this highly counterintuitive or even paradoxical.)

Riffle

def riffle(items, out = True):

Given a list of `items` that is guaranteed to contain an even number of elements (note that the integer 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, 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 the alternating card is taken first.

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 (or 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 accordingly return either `True` or `False`.

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

(As an extra challenge, 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. Note that dividing an integer by 10 effectively chops off its last digit, whereas the remainder operator % can be used to extract the last digit.)

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, in the list `[42, 7, 12, 9, 13, 5]`, the elements 42, 13 and 5 are dominators.

Before starting to write any code for this function, please read and 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), the same way that Shlemiel's painting and running task will increase as the fence gets longer.

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

Extract increasing integers from digit string

def extract_increasing(digits):

Given a string of digits guaranteed to consist of ordinary integer digit characters 0 to 9 only, create and return the list of increasing integers acquired from reading these digits in order. 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 discarded.

digits	Expected result
'0'	[0]
'045349'	[0, 4, 5, 34]
'777777777777777777777777'	[7, 77, 777, 7777, 77777, 777777]
'122333444455555666666'	[1, 2, 23, 33, 44, 445, 555, 566, 666]
'1234567890987654321'	[1, 2, 3, 4, 5, 6, 7, 8, 9, 98, 765, 4321]
'314159265358979323846264 3383279502884'	[3, 14, 15, 92, 653, 5897, 9323, 84626, 433832, 795028]
'271828182845904523536028 7 47135266249775724709369995 9574966967627724076630353 5 475945713821785251664274 27 466391932003059921817413 59 662904357290033429526 0'	[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 345678912345678912345678 91

Words that contain given letter sequence

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

This would be a good place for 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. On the other hand, a **subsequence** consists of characters taken in order, but not necessarily from consecutive positions. For example, `"ompu"` is both a substring and subsequence of `"computer"`, whereas `"cper"` is a subsequence but not a substring of that same word. The concepts of **sublist** and **subsequence** defined for lists are similar.

Given a list of `words` sorted in alphabetical order, and a string of required `letters`, find and return the list of precisely those words that contain these `letters` as a subsequence.

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 (0, 0) of the two-dimensional integer grid, initially heading north. It then executes the given sequence of moves, a string made up of 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 according to its 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)
'LLFLFLRLFR'	(1, 0)
'FR' * 1000	(0, 0)
'FFLLLFRLFLRFRLRRL'	(3, 2)

(As an aside, why do these problems always seem to take place in Manhattan instead of, say, Denver where the street grid is rotated 45 degrees from the main compass axes to equalize the amount of daily sunlight on streets of both orientations? That should make an interesting variation to many problems of this spirit. Of course, diagonal moves always maintain the parity of the coordinates, which makes it impossible to reach any coordinates of the opposite parity, quite like 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 particular problem with its countless variations is a classic when modified so that you must minimize the total number of returned coins. The greedy approach will then no longer produce the optimal result for all possible coin denominations. For example, for simple coin denominations of [50, 30, 1] kopecs and the amount of sixty kopecs to be exchanged, the greedy solution [50, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] uses eleven coins, whereas the optimal solution [30, 30] needs only two! A more advanced **recursive** algorithm would be needed to make the "take it or leave it" decision for each coin, with the intermediate results of this recursion then **memoized** to avoid blowing up the running time exponentially.

Blocks in pyramid

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

A solid pyramid structure (although here in the [ancient Mesoamerican](#) than the more famous ancient Egyptian style) is built from layers, each layer consisting of a rectangle of identical cubic blocks. The top layer of the pyramid consists of `n` rows and `m` columns of such blocks. The layer immediately below each layer contains one more row and one more column, all the way to the bottom layer of the pyramid. If the entire pyramid consists of `h` such layers, how many blocks does this pyramid contain in total?

This problem could be solved in a straightforward **brute force** fashion by simply looping through the `h` layers and adding up all the blocks along the way in each layer. With the current version of the automated tester, solving the test cases would take roughly one minute to complete, since all of these three arguments can get pretty big. However, you can come up with an analytical closed form formula for the result and compute the answers *much* faster that way.

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

There is an important general principle in this for you to later ponder on your own once you have done more of these problems. There is a reason why they call it **brute force**…

Rooks on a rampage

def safe_squares_rooks(n, rooks):

On a generalized *n*-by-*n* chessboard, there are some number of **rooks**, each one 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 being computer programmers instead of normal people, 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 as that rook. 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. The result is simply the product of these two counts. (Permuting the rows and columns does not change the answer to this question. We can therefore imagine all the safe rows and columns to have been permuted in a way that has all safe squares form a rectangle at the top left corner of the board, and the area of a rectangle is of course the product of its width and height.)

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 pancakes that rest on top of that spatula, a **pancake flip** of order k done for the given `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):
```

Let the shape of the given word of length *n* be a list of *n* - 1 integers, each one either -1, 0 or +1, indicating whether the next letter following the letter in that position comes later (+1), the same (0) or earlier (-1) in alphabetical order of English letters. For example, the shape of `'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 that operate on lists of words specified in this document, should not try to read the wordlist file `words_sorted.txt`, even if Python makes this possible with just a couple of lines of code. The tester script already reads in the wordlist and builds the list of words from there. Your function should use this list of `words` given to it as a parameter 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, 1]`	`['aegilops']`

Motivated students can take on as an extra challenge for each possible word length *n* ranging from 3 to 20, find the shape of length *n* - 1 that matches the largest number of words. 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? How about the shortest such shape that does not contain any zeroes?

Running median of three

```
def running_median_of_three(items):
```

Given a list of `items` that are all guaranteed to be integers, create and return a new list whose first two elements are the same as they were in original `items`, after which each element equals the **median** of the three elements in the original list ending in that position. (If two out of these three elements are equal, then that element is the median of those three.)

items	Expected result
[5, 2, 9, 1, 7, 4, 6, 3, 8]	[5, 2, 5, 2, 7, 4, 6, 4, 6]
[1, 2, 3, 4, 5, 6, 7]	[1, 2, 2, 3, 4, 5, 6]
[3, 5, 5, 5, 3]	[3, 5, 5, 5, 5]
[22, 77]	[22, 77]
[42]	[42]

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 example program. In trick-taking card games such as **bridge**, the players in turn each play one card to the trick. The unique 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 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. Given the list of `cards` played to the trick, return the card that wins the trick.

cards	trump	Expected result
[('trey', 'spades'), ('ace', 'diamonds'), ('jack', 'spades'), ('eight', 'spades')]	None	('jack', 'spades')
[('ace', 'diamonds'), ('ace', 'hearts'), ('ace', 'spades'), ('deuce', 'clubs')]	'clubs'	('deuce', 'clubs')
[('deuce', 'clubs'), ('ace', 'diamonds'), ('ace', 'hearts'), ('ace', 'spades')]	None	('deuce', '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, but so that the elements of every odd-numbered row are listed in descending order. However you choose to enforce that discipline is up to you.

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]]

(The five-dollar word of the day is "boustrophedon".)

Detab

```
def detab(text, n = 8, sub = ' '):
```

In the **detabbing** process of converting tab characters `'\t'` to ordinary whitespaces, each tab character is replaced by a suitable number of whitespace characters so that the next character is placed at a position that is exactly divisible by n. However, if the next character is already in a position that is divisible by n, another n whitespace characters are appended to the result so that at least one substitution character will appear to replace any tab character.

For demonstration purposes, since whitespace characters might be difficult to visualize during the debugging stage, the substitution character that fills in the tabs can be freely chosen with the named argument `sub` that defaults to whitespace. This function should create and return the detabbed version of its parameter `text`.

text	n	sub	Expected result
`'Hello\tthereyou\tworld'`	8	`'$'`	`'Hello$$$thereyou$$$$$$$$world'`
`'Ilkka\tMarkus\tKokkarinen'`	4	`'-'`	`'Ilkka---Markus--Kokkarinen'`
`'Tenser,\tsaid\tthe\ttensor'`	5	`'+'`	`'Tenser,+++said+the++tensor'`

People vary greatly on their preference for the value of n, which is why we make it a named argument in this problem. Some people prefer n = 4, others like the wider berth of n = 8, whereas your instructor likes the tight n = 2 best. To each his or her own.

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 mere two. A natural extension of the knight's move that keeps all moves symmetric with respect to all these dimensions is to define the possible moves as a k-tuple of strictly decreasing nonnegative integers. 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 that is. All these moves are considered to be performed together as a single jump that does not visit any of the intermediate squares. Given the `start` and `end` positions as k-tuples of integer coordinates, determine whether the generalized `knight` can jump from `start` to `end` in a single move.

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 the 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 move, whereas a 6-dimensional knight could move into $6! * 2^6 = 46080$ different neighbours in one jump. Since the number of moves emanating from each position to its neighbours grows exponentially with respect to the number of dimensions k, everything ends up being close to everything else in high-dimensional spaces. (Density in general is known to have both advantages and disadvantages in all walks of life...)

Fulcrum

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

Each item in the list of `items` is now considered to be a physical weight, and therefore guaranteed to be a positive integer. Your task is to find and return a **fulcrum** position in this list 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 both sides, and therefore does not participate in the torque calculation.)

As taught in any introductory physics textbook, the torque of an item with respect to the fulcrum equals its weight times its 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. That one, by the way, would be an interesting but a more advanced problem normally suitable for a third year computer science course... but in Python, this algorithm could easily be built around this function by using the generator `permutations` in the 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 needing the fancy **backtracking** algorithms from third year and up.)

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

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...

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 there exist multiple palindromes with the same largest possible 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'`	`'aabababababaa'`
`'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 remaining coins are grouped into exactly out coins in each group, where out is guaranteed to be a positive integer greater than one. The remainder n % out coins that do not fit into any group are taken aside and recorded. From each complete group of out coins, exactly ins coins are added back in the pile, and the rest of the coins of that group are discarded. Repeat this until the entire pile becomes empty, which must eventually happen whenever out > ins. Return a list that says 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 seen in the first three rows, this method produces the digits of n in base out in reverse order. So this entire setup turns 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 the integer 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 a real 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 sure would have been some "New Math" for the frustrated parents in the sixties who had enough trouble balancing their checkbooks in the familiar base ten already!

Last man standing

`def josephus(n, k):`

In the ancient times back "🎶 when men were made of iron and their ships were made of wood 🎶", as seen in "300", "Spartacus", "Game of Thrones" and similar historical docudramas of swords and sandals, a group of zealots (yes, *literally*) was surrounded by the overwhelming Roman enemy. To avoid capture and slow humiliating death by crucifixion, in their righteous zeal these men chose to commit mass suicide in a way 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 the shrinking grim circle. (Being normal people instead of computer scientists, they always start counting from one instead of zero, the concept of which didn't even exist for them back then anyway!) This continues until only one man remains, expected to honorably fall on his own sword and join his fallen 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 last two survivors.

n	k	Expected result
4	1	[1, 2, 3, 4]
4	2	[2, 4, 3, 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])

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 this definition depends on whether a particular number is already part of the previously generated part of the sequence.

To make your function execute in a speedy fashion even when generating a sequence that contains millions of elements, you should use a `set` to remember 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. Your better technique can create this list arbitrarily far in linear time, and should therefore be blazingly fast even for millions of elements.

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 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 own species, growls if there are more dogs than cats following that position in the direction that the animal is facing. Given a list of such `animals`, return the count of how many of them 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 pretty high 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 unsee that now. For some reason, this problem is somehow more tricky than it seems in a fair and just world.)

Bulgarian solitaire

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

You are given a row of `piles` of pebbles and a positive integer k so that the total number of pebbles in these piles equals k*(k+1)//2, a formula that just so happens to equal the sum of all positive integers from 1 to k. As an apt metaphor for the bleak daily life behind the Iron Curtain, all pebbles are identical and you don't have any choice in the moves. Each move must pick up exactly one pebble from every pile (even if doing so makes that pile disappear), and create a new pile from these collected pebbles. For example, starting with `piles = [7, 4, 2, 1, 1]`, the first move would turn this into `[6, 3, 1, 5]`. The next move turns this into `[5, 2, 4, 4]`, which then turns into `[4, 1, 3, 3, 4]`, and so on.

This function should count how many moves are needed to convert the given initial `piles` into some permutation of first k integers so that each number from 1 to k appears as the size of **exactly one pile** in the current configuration, and return that count. These numbers from 1 to k do not need to appear in sorted order inside the list. Applying the basic move to this goal state no longer changes the configuration. (In mathematical lingo, this goal state is said to be a **fixed point** of the transformation function between these configurations of pebbles.)

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 was inspired by the old [Martin Gardner](#) column "Bulgarian Solitaire and Other Seemingly Endless Tasks" where it was used as an example of what we would here call a while-loop where, in the same spirit as the Collatz problem in the lecture examples, it is not immediately obvious that the loop must eventually reach its goal and terminate. However, a combinatorial proof establishes that this one simple move can never get stuck, but will always reach the goal state from any starting configuration 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".

Thanks to your cunning, your nemesis in life is trapped inside a devious game, and for a refreshing change from the usual way of things, you now get to be the Final Boss. (Everyone is the hero inside their own movie, and all that.) This final level consists of a one-dimensional platform that reaches n-1 steps to both directions, with your two deadly new friends Scylla and Charybdis hungrily waiting below at each end of the platform at distance n.

Your nemesis starts at the center of this platform at position zero. She has to immediately commit to the entire sequence of moves, here as a string that consists of characters '+' ("🎵 just a step to the ri-i-i-i-ight" 🎵) and '-' (move one step to the left) only, and no other characters. Your **adversarial** task is to find a positive integer k so that when executing precisely every k:th step of these moves (the subsequence therefore starting from position k-1 and then taking in every k:th element), your nemesis ends up at least n steps away from the starting point and falls into one of the two possible dooms that they indirectly chose.

This function should find and return the value of k that makes your hated nemesis fall off the platform after the smallest number of moves. To ensure the existence of some solution, the given sequence of moves is guaranteed to end with the **suffix** of 2*n* consecutive steps to the right, so k=1 will always work whenever no larger step size leads to the doom faster. If several values of k work equally well, you must return the smallest such k.

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

Arithmetic progression

```
def arithmetic_progression(elems):
```

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 list of `elems` guaranteed to consist 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 that the answer is unique for automated testing, if there exist several progressions of the same length, return the one with the smallest `start`. If there exist several progressions of equal length from the same `start`, return the progression with the smallest `stride`.

elems	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)

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 a clever **approximation algorithm** from the seventies to quickly find something reasonably close to the **median element** of the given unsorted list. In this problem, the median element of the list is defined to be the element that would end up in the middle position, were that list sorted.

Tukey's algorithm itself is simple. Conceptually split the list into sublists of three elements each, and find the median element for each of these triplets. Collect these medians-of-three into a new list (whose length therefore equals one third of the length of the original list), and repeat this 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 list that results from applying one round of Tukey's algorithm to the list in the next row.

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 clever algorithm for approximating the median element is extremely robust, as can be appreciated by giving it a whole bunch of randomly shuffled lists to operate on, and plotting the resulting distribution of results and admiring how is heavily centered around the true median it ends up being. (For example, 15 honestly, for grown up realsies, is the median of the last example list above.) Assuming that all `items` are distinct, the returned element can **never** be from the true top or bottom third of the sorted elements, thus eliminating all risk of accidentally using some outlier element as your estimate of the true median and by doing so make your statistical calculations go all funky.

Whenever they zig, you gotta zag

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

Let us define that a positive integer n is 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. The step from the first digit to second can be either positive or negative. The function should determine whether its parameter 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
104175101096715	False
49573912009	False

Crack the crag

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

Crag (see the Wikipedia page for the rules and the scoring table needed in this problem) is a dice game similar in style and spirit but much simpler than the more popular and familiar games of Yahtzee and Poker dice. The players repeatedly roll three dice and assign the resulting patterns to categories of their choosing, so that once assigned, the same category cannot be used again for the future rolls.

Given the list of pips of the three dice that were rolled, this function should compute and return the highest possible score available for those dice, under the simplifying assumption that this is the first roll in the game so that **all categories of the scoring table are still available for you to choose from** and you just want to maximize the points for this first roll. This problem should therefore be a straightforward exercise in writing if-else ladders combined with simple sequence management. Note that some examples on the Wikipedia scoring table show the score that some dice would score **in that particular category**, but this is not necessarily equal to the maximum score that is in principle attainable with those dice. For example, `[1, 1, 1]` given in the category "Ones" would indeed score only three points in that category, whereas it would score 25 points in the category "Three of a kind".

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

There exist only 6^3 = 216 possible argument value combinations, so your function ought to answer them swiftly in total time that rounds down to `0.000`. However, surely you will design your if-else ladder to handle entire equivalence classes of cases together in a single step, so that your ladder consists of far fewer than 216 separate steps...

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 exactly three** separate `items` that together add up **exactly** to the given positive integer `goal`. Return `True` if three such integer items exist, and `False` otherwise.

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 grows larger, and the automated tester will make those lists larger.

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 that demonstrates how to quickly determine whether the sorted list contains two elements that 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 later in the lectures as a recursion example, 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`.)

Count all sums and products

```
def count_distinct_sums_and_products(items):
```

Given a list of distinct integers guaranteed to be in sorted ascending order, count how many different numbers can be constructed by either adding or multiplying two numbers from this list. (The two chosen numbers do not need to be distinct.) For example, given the list [2, 3, 4], this function would return 8, since there exist a total of eight distinct numbers that can be constructed this way from 2, 3 and 4. Some of them can be constructed in several different ways, such as 6 that is either 4 + 2 = 3 + 3 = 3 * 2, but the number 6 should still be counted only once in the total tally.

This problem can be solved by simply looping through all possible ways to choose a and b from the given list. Maintain a set that remembers all sums a+b and products a*b that you have seen so far, and after having iterated through all possible such pairs, return the final size of that set.

items	Expected result
[]	0
[42]	2
[2, 3, 5, 7, 9, 11]	29
[x for x in range(1, 101)]	2927
[x*x for x in range(1, 101)]	6533

(This problem was inspired by the article "How a Strange Grid Reveals Hidden Connections Between Simple Numbers" in *Quanta Magazine*.)

Sum of two squares

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

Some positive integers can be expressed as a sum of two integers that each are squares of some positive integer greater than zero. For example, $74 = 49 + 25 = 7^2 + 5^2$. This function should find and return a tuple of two positive integers greater than zero 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 give the larger of its two numbers first. Furthermore, if some integer can be broken down to a sum of squares in several different ways, return the way 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 therefore 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 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.

Hint: 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 that is 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

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 well as it handles 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 after them in the lexicographical ordering. If your `count_dominators` does not pass this hurdle, go back and edit it to have 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. This definition makes the domination a **partial ordering** so that, for example, the word `'dog'` dominates the word `'cat'`, but neither word `'dog'` and `'arg'` dominates the other. 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 possibly 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 like to ascribe deep significance to numerical coincidences so that especially consecutive repeated digits, such as a digital clock blinking 11:11, seem special and personal to them. Such people then naturally find numbers with repeated digits to be more valuable and important than all the ordinary and pedestrian numbers without any obvious pattern, so that getting into a taxicab with an exciting number 1234 is far more instagrammable than into a cab marked with some dull number such as 1729.

Let us assume that some such person assigns 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 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
999999777777444448887222	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 exact same length so that each element is replaced with the nearest element in the original list whose value is smaller. If no smaller elements exist, the element in the result list should simply remain as it were in the original list. If there exist smaller elements to both directions that are equidistant from that element, you should resolve this by using the smaller of these two elements to make the results testable.

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.

(Hint: it is actually much easier to determine that the two squares do **not** intersect, and then 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.)

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

Keep doubling

```
def double_until_all_digits(n, giveup = 1000):
```

Given a positive integer n, keep multiplying it by two until the current number contains each of the digits 0 to 9 at least once. Return the number of doublings that were necessary to reach this goal. If the number has been multiplied giveup times without reaching this goal, the function should give up and return -1.

n	giveup	Expected result
1	1000	68
1234567890	1000	0
555	10	-1

Ignore each item after its *k*:th occurrence

```
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, updating this count as you go and appending each element to the `result` list only if its count is still less than or equal to k. Note also the counterintuitive but still completely legitimate edge case of k==0 that has a well defined answer of an empty list!

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	[]

First item that is preceded by *k* smaller items

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

Given a list of `items`, find and return the value of the first element that has at least *k* preceding smaller elements in `items`, in the sense of the ordinary Python order comparison `<` applied to these items. If there is no such element anywhere in `items`, this function should return `None`. Note that the k smaller items do not need to be consecutive immediate predecessors of the current item, but can be any k items from the initial prefix of the list 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, consider the following operation: Find the smallest number from those that still remain in the list, and remove that number and the larger of its current immediate neighbours from the list. The function should keep doing this repeatedly until the largest number in the original list gets eliminated, and 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, the same operation would now 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.

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'`

(This particular operation, when executed on a list of items instead of a string, is usually called "[run-length encoding](#)". When executed on a string of numerical digits, it also has the following cute little puzzle associated with it. Given the empty string, this function returns an empty string, and given the string `'22'` that contains two twos, this function returns the same string `'22'`. Are there any other strings of digits for which this function returns the exact same string as it was given? Either find another such digit string, or prove that no such digit string can exist.)

Bishops on a binge

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

On a generalized *n*-by-*n* chessboard, there are some number of **bishops**, each bishop represented as a tuple (`row, column`) of the row and the column of the square that contains that bishop. (The rows and columns are numbered from 0 to *n* - 1.) 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 empty squares that are safe, that is, are not covered by any bishop.

To quickly check whether two squares (`r1, c1`) and (`r2, c2`) are in some diagonal with each other, you can use the test `abs(r1-r2) == abs(c1-c2)` to determine whether the horizontal distance of 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 logic separately for each of the four diagonal directions, but one 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

Up for the count

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

The **counting series "1234567891011121314151617181920212223"**... is an infinitely long string of digits 0-9 that consists of the positive integers written in ascending order without any separators between the individual numbers. This function should return the integer digit that is in the position n of the counting series, with positions starting from 0 as usual in computing.

Of course, the automated tester will again try out values of n large enough that anybody trying to solve this problem by constructing the counting series as an explicit 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 regular and predictable structure allows you to skip 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

Has the digit that is waiting for you in position n been there waiting for you since the dawn of time, or did it come to existence the moment that somebody first solved this problem up to the position n? What is your preferred stance on this thorny and ancient philosophical issue? 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 some mathematical truths created by social agreement the same way that, say, the characters of Donald Duck and James Bond were brought into existence by the conscious choices of their creators? Something to ponder before you come back to the practical plane of solving these problems.

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 that same position. For example, reversing the vowels of `'Ilkka'` should produce `'Alkki'` instead of `'alkkI'`.

(Hint: one straightforward way to solve this starts with collecting all vowels of `text` into a separate list. 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. Otherwise, take the character from the same position of the original `text`.)

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

Everybody do the Scrooge Shuffle

def spread_the_coins(piles, left, right):

Some identical coins have been placed on the integer number line (this time infinite both ways to both positive and negative direction) so that position `i` initially contains `piles[i]` coins for the positions `i` inside the list that satisfy `0 <= i < len(piles)`, and all other positions initially contain zero coins. After this, any position `i` that still contains at least `left + right` coins is said to be **unstable**, and to rectify that, exactly `left` coins from that pile spill into position `i-1`, and exactly `right` coins spill into the position `i+1`. Total number of coins in this system therefore never changes as these coins merely spill into other positions along the integer line (including negative positions) until the coins have spread far and wide enough to stabilize everywhere.

The unique end state of this back-and-forth dance where all positions are finally stable will be reached independently of the order in which the unstable positions are processed. This function should return the final position as the tuple `(start, coins)` so that `start` is the smallest position index that contains at least one coin, and coins is the list of coins starting from position `start` up to the highest position that contains at least one coin.

To make this function faster, note how whenever some pile contains `k*(left+right)` coins for some `k > 0`, you might as well scoop `k*left` coins into the previous pile and `k*right` coins into the next pile in one move. This will be much faster than sloughing through k separate little moves that individually move only `(left + right)` coins each time to achieve the same end result. The difficulty of this problem is handling the counts for negative positions. A list can be **prepended** from the front, yes, but that operation can get rather inefficient since it has to make room for the new element with a linear time reorganization. However, ordinary lists are basically special cases of dictionaries with indices restricted to consecutive natural numbers...

piles	left	right	Expected result
[20]	3	2	(-3, [5, 3, 3, 3, 6])
[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])

Calkin-Wilf sequence

`def calkin_wilf(n)`

The levels of the [Calkin-Wilf tree](#) in **level order**, when each level is read from left to right, produces the linear sequence of all possible **positive integer fractions** so that almost as if by magic, every positive integer fraction appears exactly once in this sequence. Even more wonderfully, each fraction is guaranteed to appear in its simplest reduced form! To perform the following calculations, you need to `import` the handy data types `Fraction` and [deque](#) from the standard library modules [fractions](#) and [collections](#).

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. Return the fraction object that was popped in the final round. (Actually, once you reach `n//2+1`, you could stop pushing in any new values and save some significant memory, since at that point the queue already contains the final result for you to pop out...)

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

Kempner series

```
def kempner(n):
```

As we have learned in various math courses, the n:th [harmonic number](#) is equal to the sum of the reciprocals of the first n positive integers, that is, $H_n = 1/1 + 1/2 + 1/3 + ... + 1/n$. As n approaches infinity, so does the harmonic number H_n without limit, although only with logarithmic order of growth. It is not even necessary to include all positive integers to this series to make it diverge without bound, but various infinite subsets of positive integers such as the prime numbers will still make this sum grow without limit.

A whimsical variation known as [Kempner series](#) adds up only those fractions that **do not contain the digit nine** anywhere in them. This rule eliminates enough terms from the infinite sum to make it converge to a finite limit, since terms that do not contain the digit nine get exponentially rare as n increases. Your function should compute and return K_n, the n:th term of the Kempner series by adding up the reciprocals of the first n positive integers that do not contain the digit nine. For example, to compute K_{10}, you would need to add up $1/1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/10 + 1/11$, the first ten such reciprocals.

Instead of approximating the result inaccurately with fixed precision **floating point** numbers, you must perform this computation with perfect accuracy using `Fraction` objects from the [fractions](#) module. However, since the numerators and denominators of these fractions grow pretty quickly as n increases, the result should be returned as a `Fraction` object given by the approximation produced by calling the method `limit_denominator(1000)` to approximate the true value of the result as the nearest fraction whose denominator is less than one thousand.

n	Expected result (as `Fraction`)
4	25/12
100	431/87
500	5743/914
1000	6723/98

Hippity hoppity, abolish loopity

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

A frog moving on the infinite two-dimensional 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 exactly 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 even for large 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 position. If these frogs never jump into the same square, return None.

This function should not contain any loops whatsoever, but the result should be calculated using conditional statements and integer arithmetic. Perhaps the best way to get started is by first solving 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 even for all the possible edge cases, use it to solve for t separately for the *x*- and *y*-dimensions in the original problem, and combine those two one-dimensional answers into the 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):
```

Given a list of `items`, suppose that you repeated the following operation n times: remove the first element from `items`, 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 that we perform?

Sure, this problem could be solved by actually performing that operation n times, but the point of this question is to come up with an analytical solution to compute the result much faster than going through that whole rigmarole. Of course, the automated tester is designed so that anybody trying to **brute force** their way through this problem by actually performing all n operations one by one will run out of time and memory long before receiving the answer. 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.

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 real reason why you take courses on discrete math and combinatorics is to become familiar with techniques to derive analytical solutions to problems of this nature so that you don't have to brute force their answers in a time that is prohibitively long to be feasible.)

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.) 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 one. 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	100000000000000000000000000000000000000000000000000004163 32753518329478897755794704334003003544212420356

The simplest way to make this function efficient is to harness the power of **repeated halving** to pull your wagon with an 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 two with `a = 1` and `b = 2`, and keep squaring `b` until the number in that position is 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 more final tells you which element is 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, remove 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 programming language can be built around the idea of 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 and computer scientists, is best known for [The Game of Life,](#) not to be confused with the board game of the same name, and in reality an automaton, not even a puzzle or a game. (Puzzles have one player who gets to make all decisions, games have more than one player, whereas automata make no decisions but follow their hardcoded rules.) That one achievement eclipsed basically all other wacky and original creations of Conway (and then some), so we should do our best to give his less appreciated creations at least an occasional turn in the limelight and paparazzi cameras.

This lab asks you to write a simple interpreter for the [esoteric programming language](#) called [FRACTRAN,](#) named as a pun of the word "fraction" and the FORTRAN programming language. (Everything used to be upper case back in the old days, with occasional dollar signs interspersed.) 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 the starting 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. If `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 termination... or I guess we should say *halting* (considering that 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]

Power wash those floors!

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

For positive integers `a` and `b`, the operation `a**b` can be implemented as repeated multiplication. Start with `result=1`, and multiply `a` into the `result` for a total of `b` times. No problemo. In the variation of **floor power**, the result of each multiplication is immediately truncated to its **floor**, which for positive numbers equals its whole integer part. This makes no difference when `a` is an integer, but such intermediate **quantization error** matters greatly when `a` is some `Fraction`. For example, `Fraction(7,3)**3` equals 343/27, whereas `floor_power(Fraction(7,3),3)` equals 28/3 once you implement that function. (The same phenomenon can be observed with computations that have to round their intermediate results to integers instead of using a wider type to contain the decimal places that will make a difference in the future.)

This function should solve the equation `n == floor_power(x, k)` for positive unknown x, given the positive integers `n` and `k`. Since these solutions are in general irrational numbers (just like they already are for solving roots of ordinary powers), this function should return the integer **floor** of the solution. This function has to work correctly and efficiently for arbitrarily large values of n, so don't even dream about using floating point arithmetic anywhere in these computations, but do all computations using exact `Fraction` objects. You should again use the principle of **repeated halving** to maintain two `Fraction` values `a` and `b` as lower and upper bounds for the unknown value of `x` so that `a <= x <= b`. Compute the midpoint `m` as the average of `a` and `b`, and compute `floor_power(m, k)` at that midpoint. Compare the result to n, and move either endpoint `a` or `b` to the middle depending on the result. Once `floor(b)-floor(a) < 2`, at most one more comparison tells you which one of `floor(a)` and `floor(b)` is the correct answer.

n	k	Expected result
100	2	10
1399	4	6
2018	4	6 (inspiration)
10**100	37	504

Squaring off

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

Two players play a game called "Subtract a square" with a positive integer. On their turn to play, each player can subtract any square number (1, 4, 9, 16, 25, 36, 49, ...) from the current number as long as the result would not become negative. Under the **normal play convention**, the player who makes the last move by moving to zero wins, leaving his opponent stuck with no possible moves. (In the **misère** version of the game that has otherwise the same rules but you win by losing the original game, these definitions would be adjusted accordingly.)

These kinds of 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 but not on which player has to make the next move. A state is called "hot" ("winning") if the current player can make a move that guarantees the win regardless of the opponent's response, under the natural assumption that the current player will then also continue making the best moves for the rest of the game. The game state is "cold" ("losing") if no such move exists.

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 heat values of previous states that it needs.

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. (Hint: start from goal 1 and perform the transitions in reverse.)

shape	Expected result
`'ududududdddduddd'`	15
`'dudududdudddudddd'`	14
`'uduuudddd'`	None
`'d'`	2
`'uuududdddduuuuuuuddddddd'`	None
`'duuudddddddd'`	None

Reverse ascending sublists

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

Create and return a new list that contains the same elements as the argument list `items`, 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, this function should not modify the contents of the original list, but create and return a new list object that contains the result, no matter how tempting it might be to perform this operation right there in the original list.

(In the table below, different colours are used to highlight the strictly ascending sublists for the human reader, and are not part of the actual argument given to the function. It's not like this is *Mathematica* or some symbolic computation system that deals directly with symbolic expressions so that you could do things like that...)

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):
```

The task of combining the first names of celebrity couples into a catchy shorthand for media consumption turns out to be surprisingly simple to automate. Start by counting how many **groups** of consecutive vowels (*aeiou*, since to keep this problem simple, the letter *y* is always a consonant) there are inside the first name. For example, `'brad'` and `'ben'` have one group, `'sheldon'` and `'britain'` have two, and `'angelina'` and `'alexander'` have four. Note that a vowel group can contain more than one vowel, as in `'juan'` or `'queueing'`.

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 simple 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 would cover a wider variety of name combinations and special cases. Some truly advanced set of rules that recognizes the semantic content of words would even combine `'donald'` and `'hillary'` into `'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-element tuple of *x*- and *y*-coordinates, for example `(2, 5)` or `(10, 1)`. [As originally postulated by Euclid](#), for any two distinct points on the plane, there exists **exactly one** straight line that goes through both points. (In other spaces that have very different overall shapes, [different rules and their consequences apply](#).) Of course, this same line, being infinite in both directions, will also go through an infinity of other points on the plane.

Given a list of `points` on the integer grid, find the line that contains the largest number of points from this list. To facilitate automated testing and guarantee that the answer for each test case is unambiguous, 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 all points in the list are distinct, but the points are otherwise not given in any sorted order.

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. To simplify your logic, consult the example program `geometry.py` for the **cross product** function that can be used to quickly check whether three points are collinear.

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 each row is strictly ascending, you can find this out without having to compute an infinite number of elements.) To compute the second element of that row, let (a, 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, all by itself, guarantees that the Wythoff array is an **interspersion** of positive integers; every positive integer will appear **exactly once** in this infinite grid, with no gaps or duplicates! (This result, as many other combinatorial problems, highlights nicely the deeper combinatorial importance of the deceptively simple Fibonacci numbers as building blocks of other numbers and sequences.)

The difficulty in this problem is determining the first two elements in each row after the first row, since once you know those first two values, the rest of the row is utterly trivial to generate as far as you need. This function should return the position of n inside the Wythoff array as a tuple of form (row, column), both the 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)

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](). To bring these two opposites briefly together, let us look at positive integers that consist of some solid sequence of sevens, followed by some (possibly empty) solid sequence of zeros. For example 7, 77777, 7700000, 77777700, or 70000000000000000. A surprising theorem proves for any positive integer `n`, there exists an infinity of integers restricted to this seven-zero form that are divisible by `n`, that is, equal `k*n` for some positive integer `k`. This function should find and return the **smallest** such seven-zero integer for the given `n`.

This exercise is all about efficiently generating and iterating through the numbers of the constrained form of sevens and zeros, doing so in strictly ascending order to guarantee finding the smallest possible working number. Your outer loop should iterate through the number of digits in the current number. For the current digit length `d`, the inner loop should then go through all possible ways over `k` to create a number that starts with `k` sevens, followed by `d-k` zeroes.

n	Expected result
17	7777777777777777
42	7770
101	7777
103	777777777777777777777777777777777777
2**50	70000000000000000000000000000000000000000000000000
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 the value of such solution. Furthermore, as proven in that same book, whenever `n` is not divisible by either 2 or 5, the smallest such number will always consist of a solid sequence of sevens with no zero digits after them. This handy fact can speed up your search massively for such friendly values of `n`.

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 has set up this feast has recently escalated to a whole new level of severity. To allow the Cookie Monster to scarf down his crumbly fill, The Count insists that The Cookie Monster must eat these cookies in the shortest possible sequence of moves. Each move consists of the Cookie Monster choosing an integer `p` that must be one of the remaining pile sizes. The chosen move removes `p` cookies from every pile that contains at least `p` cookies, and leaves all the smaller piles as they were before the move. This function should compute and return **the smallest number of moves** that allows Cookie Monster to eat 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 that the moves in the optimal solution can be performed in any order, which is rare for combinatorial problems of this spirit. Without loss of generality, it suffices to examine only strictly descending move sequences. Even then, [this problem is far deeper than it might initially seem](#). Various **greedy strategies** that would seem natural for the poor 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" do not always lead to the shortest possible sequence of moves.

Autocorrect for stubby 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. Since many people, such as your instructor, have stubby fingers, many typos are caused by the user pressing a virtual key next to the intended key. For example, when the non-existent word is `"cst"`, the intended word is far more likely to be `"cat"` than `"cut"`, assuming that the text was entered with an ordinary QWERTY keyboard.

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 there exist several equidistant words, return the first word in the dictionary order.

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

Advanced autocorrect algorithms use statistical techniques based on the surrounding context to suggest the best replacement, since 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.)

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 sterod 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?)

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 (`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 height zero.

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`. During each interval, only the tallest active tower has any effect on the computation.

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

(The more complex versions of this classic chestnut ask for the silhouette outline as a list of polygons, as on the linked page, and also lose the restriction that all towers must lie on the same ground level line, or even be **axis-aligned**.)

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 uses only integer arithmetic whenever all individual components are integers. And no square roots or some other nasty irrational numbers.)

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 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 between that disk and only the disks presently 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 ordinals

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

Sorting can be done using arbitrary 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 will be explained in the discrete math course, the above relation is transitive and antisymmetric, and therefore a completely legal ordering relation for integers. Outside its entertainment valu 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 and tackle a problem so simple that its solution needs only a couple of conditions, but not even any loops, let alone 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 and return how many integers between `start` and `end`, inclusive, are divisible by n. Sure, you *could* solve this problem with the list comprehension

```
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 and brutish way will soon run out of both time and space! So you should use no loops but integer arithmetic and conditional statements only, and 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 has to list the suits in the required order.

hand	Expected result
[('eight', 'spades'), ('king', 'diamonds'), ('ten', 'diamonds'), ('trey', 'diamonds'), ('seven', 'spades'), ('five', 'diamonds'), ('deuce', 'hearts'), ('king', 'spades'), ('jack', 'spades'), ('ten', 'clubs'), ('ace', 'clubs'), ('six', 'diamonds'), ('trey', 'hearts')]	[4, 2, 5, 2]
[('ace', 'spades'), ('six', 'hearts'), ('nine', 'spades'), ('nine', 'diamonds'), ('ace', 'diamonds'), ('trey', 'diamonds'), ('five', 'spades'), ('four', 'hearts'), ('trey', '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 again represented as tuples of form (rank, suit) as in the 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 offense.)
- 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'), ('deuce', 'hearts'), ('trey', 'diamonds'), ('seven', 'diamonds'), ('four', 'diamonds'), ('deuce', 'diamonds'), ('four', 'clubs')]	'diamonds'	8
[('trey', '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
[('trey', 'spades'), ('seven', 'spades'), ('deuce', 'spades'), ('trey', 'hearts'), ('queen', 'hearts'), ('nine', 'hearts'), ('ten', 'diamonds'), ('six', 'diamonds'), ('queen', 'diamonds'), ('ace', 'diamonds'), ('nine', 'clubs'), ('four', 'clubs'), ('five', 'clubs')]	'notrump'	7

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 a positive integer m > n that is **the next higher zigzag number** so that none of the numbers strictly between n and m is 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 gap between two consecutive zigzag numbers can be arbitrarily large, so you need some more analytical approach based on the digit pattern found at the end of the number. Unfortunately, this digit pattern might also be arbitrarily long before it allows you to decide where the next zigzag number lies...

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

Count consecutive summers

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

Positive integers can be expressed as sums of **consecutive** positive integers in various ways. For example, 42 can be expressed as 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. How would one characterize the "most polite" numbers that allow more ways to represent as sums of consecutive integers than any number less than them?)

Hitting integer powers

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

The integer powers of the given base such as 7 in the series 7, 49, 393, 2401, 16807, 117649, ... quickly become too large to be useful for our daily lives. Outside time and space, all alone with no human mind to watch over them, these sequences of powers probe through the infinite space of positive integers with exponentially increasing gaps that eventually contain entire universes inside them. Python lets us play around with integer powers whose millions of digits make them unimaginable for us and yet its mechanical binary decisions inerrantly probe through and dissect these behemoths that our minds could not begin to visualize.

Except when one of the numbers `a` and `b` is already a power of the other, the [Fundamental Theorem of Arithmetic](#) dictates with its iron hand that never the twain shall meet: the integer powers `a**pa` and `b**pb` can never be equal for any integer exponents `pa` and `pb`. However, in spirit of "close enough for government work", such powers "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. For example, `tolerance=100` requires their difference to be at most 1%. (This definition intentionally uses no division, so that everything happens with integer arithmetic.)

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 the above 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 "[Can you get another haircut already?](#)" of [Riddler](#), an excellent source of problems of this general kind of spirit.)

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 best not to reinvent the wheel, but 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]`

Bridge hand shorthand form

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

In contract bridge literature, hands are often given in abbreviated form that makes them 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 in the end.

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'), ('deuce', 'hearts'), ('trey', 'diamonds'), ('seven', 'diamonds'), ('four', 'diamonds'), ('deuce', 'diamonds'), ('four', 'clubs')]	`'xx QJxxxx xxxx x'`
[('trey', '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'`
[('trey', 'spades'), ('seven', 'spades'), ('deuce', 'spades'), ('trey', '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'), ('trey', 'spades'), ('deuce', 'diamonds')]	`'AKQJxxxxxxx - x -'`

Losing trick count of a bridge hand

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

The [Milton Work point count](#) that we saw in the earlier problem is the first baby step in estimating the playing power of a bridge hand. Once the partnership has found out that they have a good trump fit, hand evaluation continues more accurately using some system of [losing trick count](#). For example, a small slam in spades with hands `'AKxxxxx - Kxxx xx'` and `'xxxx xxxxx AQx -'` is a lock despite possessing only 16 of the 40 high card points in the deck, whereas any slam is hopeless with the hands `'QJxxx xx AKx QJx'` against `'AKxxx QJ QJx AKx'` despite the combined powerhouse of "quacky" 33 points with a horrendous duplication of useful cards.

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 a suit cannot have more losers than there are cards in that suit, and never more than three losers if there were ten cards of that suit in their hand. 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'), ('deuce', 'clubs'), ('five', 'clubs'), ('queen', 'hearts'), ('four', 'spades'), ('trey', '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'), ('trey', '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 each 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 guess this secret number by repeatedly guessing a four-digit number, the first player responding to each guess with 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 zero bulls and zero cows.

Given a list of `guesses` that have been completed so far, each individual guess given as a three-tuple (`guess, bulls, cows`), create and return the list of 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`.

(Hint: 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, 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 (for example, [the same game played with English words](#)) can be solved in more clever and efficient ways than the above **brute force** enumeration, but that would be a topic for a later, more advanced algorithms course.

Sort list by element frequency

```
def frequency_sort(elems):
```

Sort the given list of integer `elems` so that its elements end up in the order of **decreasing frequency**, that is, the number of times that they appear in `elems`. 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 would be to render to Caesar what is Caesar's, or "Let George do it", or whichever way you wish to put it by having the Python `sort` function perform the sorting against custom sorting criterion, as done in the example script countries.py. Create a dictionary to keep track of how many times each element occurs inside the array, and then use the counts stored in that dictionary as the **sorting key** of the array elements, breaking ties on the frequency by using the actual element value. (If you happen to remember that the order comparison of Python tuples is lexicographic, you don't even need to do any of this tie-breaking work yourself...)

elems	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 said to be 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 that.) 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 positive integer n given as argument is some perfect power. To do this, your code needs to somehow iterate through a sufficient number of possible combinations of b and e that could work, returning True as soon as it finds some b and e that satisfy b**e == n.

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 determine the answer. Achieving this efficiency is the central educational point of this particular problem.

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

(The automated tester for this problem is based on the mathematical theorem about perfect 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 would not need to try out all possible ways to express the number as an integer power to know right away that the behemoth 1234**5678 - 1 is not a perfect power.)

The MD problem

```
def md(a, b, n):
```

Consider an infinite series that starts with the value 1 at position 0. (In this problem we are again wearing the hats of computer scientists and our counting from zero, instead of starting counting from one like those dastardly mathematicians.) Let `v` be the most recent element in the sequence. If the value of the integer division `v // a` is nonzero and has not appeared in the sequence so far, the next element of the sequence equals `v // a`. Otherwise the next element equals `b * v`. For example, when trying out parameter values `a = 2` and `b = 3` in spirit of the Collatz sequence, this sequence would start off as `[1, 3, 9, 4, 2, 6, 18, 54]`.

Whenever the parameters `a` and `b` are positive and have no common factors higher than 1, this sequence can be proven to produce every positive integer exactly once, the values bouncing up and down in a chaotic spirit again very much reminiscent of the Collatz sequence. This function should return the position in the sequence where `n` makes its appearance.

a	b	n	Expected result
2	3	1	0
2	3	18	6
3	2	18	20
3	13	231	181
2	3	100000	175221

This problem was taken from the page "Unsolved Problems and Rewards" by Clark Kimberling. The "unsolved" part there was proving that every positive integer will appear exactly once, but then some guy proved that back in 2004 so that we all can just rely on that fact from now on. The goal of object-oriented design and programming is for us to become "mathematicians in spirit" so that **no problem would ever need to be solved twice**. Once some thorny problem has been successfully solved by somebody, it is sufficient to **reduce** your current problem into that problem to also declare your problem now solved. Furthermore, even the same problem can be solved any number of times, instead of merely once, by putting a sufficiently strong for-loop around the code to solve it once...

Collapse positive integer intervals

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

This function is the inverse of the earlier 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 in 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'`

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 elapse, the state of that hourglass will be (u-min(u,m), l+min(u,m)) so that the total amount of sand u+l remains constant inside the same hourglass and neither chamber can ever become negative. 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 the measurement is not allowed. Once the time in the chosen hourglass runs out, you may instantaneously flip **any subset** of these glasses (note that you don't have 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](#))
[(7, 0), (4, 0)]	9	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

(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 kept using them! Let us therefore showcase a more amazing property of this famous sequence: **every positive integer can be expressed exactly one way as a sum of non-repeated Fibonacci numbers** so that no two consecutive Fibonacci numbers appear in this sum. (After all, if the sum contains two consecutive Fibonacci numbers F_i and F_{i+1}, these two can always be replaced by F_{i+2} without affecting the total. Doing this replacement from highest terms down guarantees that F_{i+2} is not already being used, since with the value F_{i+1} assumed to exist, these two would have been turned into F_{i+3} already and we could not be having the problem that we currently have.)

To produce this unique sequence for n, the simple **greedy algorithm** can be proven to work fine. Always add into the list the largest possible Fibonacci number f that is less than or equal to n. Then convert the rest of the number n - f in the same manner, until only zero remains. This list of Fibonacci numbers must be returned in sorted descending order.

n	Expected result
10	[8, 2]
100	[89, 8, 3]
10**6	[832040, 121393, 46368, 144, 55]

Rooks with friends

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

Those dastardly rooks have 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 riotous wrath. Each rook is again represented as a tuple (`row, column`) of the coordinates of the square that it is standing on. However, in this version of the problem, some of these rooks are now 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 threaten those squares, so that 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 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)]	3200

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 list of words that match the `pattern` in the sense of the game of Hangman. First, the word and the `pattern` must be the same length. In places where the `pattern` contains some letter, the word must also contain that exact same letter. In places where the `pattern` contains an asterisk, the word must contain some character that may not be any of the letters that explicitly occur anywhere in the `pattern`. (In the game of Hangman, any such letter would have already been revealed in the earlier round when it was the current guess, along with all its other occurrences in `pattern`.)

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 `'g'` or `'d'` that appears inside the given `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']`
`'a**s**a'`	`['acystia', 'acushla', 'anosmia']`
`'*a*e*i*o'`	`['patetico']`

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 their 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 then afterwards dividing it down 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 that contains a total of 168 terms, the first five of which are `[(2, 994), (3, 498), (5, 249), (7, 164), (11, 98)]`)
10000	(a list that contains a total of 1229 terms, the first five of which are `[(2, 9995), (3, 4996), (5, 2499), (7, 1665), (11, 998)]`)

Aliquot sequence

```
def aliquot_sequence(n, giveup = 100):
```

The **proper divisors** of a positive integer n are those positive integers that exactly divide n and are strictly less than n. For example, the proper divisors of 12 are 1, 2, 3, 4 and 6. The [Aliquot sequence](#) for the positive integer n starts from n, after which each term equals the sum of the proper divisors of the previous term. For example, starting from 12, the next term in this sequence would be 1 + 2 + 3 + 4 + 6 = 16. The next term after that is computed by adding up the proper divisors of 16, giving us 1 + 2 + 4 + 8 = 15, and so on.

The Aliquot sequence terminates either when it reaches zero, or at the appearance of some term that has already shown up earlier in that same sequence, after which the sequence would simply continue forever in that same cycle. Both possibilities can be achieved with a suitable choice of n. In the same spirit as the more famous [Collatz sequence](#) previously seen in the `mathproblems.py` in-class example, it is currently unknown whether there exists some starting number n for which this sequence will go on forever without ever repeating itself. So just to be safe, this function should stop once the sequence length becomes equal to `giveup`.

n	giveup	Expected result
12	100	[12, 16, 15, 9, 4, 3, 1, 0]
34	100	[34, 20, 22, 14, 10, 8, 7, 1, 0]
34	2	[34, 20]

Lattice paths

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

You are standing on the point (x, y) in the grid of pairs of nonnegative integers, and wish to make your way to the **origin** point (0, 0). At any given point, you are only allowed to move either one step left or one step down at the time. This function should add up the number of different paths that lead from the point (x, y) to the origin (0, 0), under the constraints that each step must be taken either left or down, and also are not allowed to enter any points included in the tabu list.

This 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. 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) is 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 computing position [x][y], the positions [x-1][y] and [x][y-1] have already been computed. (This idea to solve recursions with loops 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

Count squares on integer grid

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

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

You are 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 are members of this set of `points`, and return this count. Note that the squares don't need to be **axis-aligned** so that their sides would be expected to be horizontal and vertical, as long as all four sides are the exact same length.

Hint: every square has **bottom left corner** point `(x, y)` and **direction vector** `(dx, dy)` towards the upper left corner point so that `dx >= 0` and `dy > 0`, and the three points `(x+dx, y+dy)`, `(x+dy, y-dx)` and `(x+dx+dy, y-dx+dy)` must be the top left, bottom right and top right corners of that square, respectively, all included in `points`. You can therefore loop through all possibilities for the bottom left point `(x, y)` and direction vector `(dx, dy)` to find all squares in the grid. Try to again be economical in these loops to make this function fast.

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)` so that `a` and `b` are positive integers. A rectangle can be cut in two smaller rectangles with the same total area with either a straight horizontal or a straight vertical cut along the integer axis of your choice. For example, the rectangle `(5, 8)` might be cut into pieces `(2, 8)` and `(3, 8)` in one direction, or into rectangles `(5, 4)` and `(5, 4)` in the other direction. Many other cuts would also be possible. These smaller pieces can then be cut further into smaller pieces, as long as the side being cut has length of two or more. However, you can only cut one piece at the time, and cannot cut multiple pieces in one swoop, regardless of the shape of these pieces.

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 that are necessary to achieve this.

This problem is best solved with recursion. If the perimeter of the current 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 that you see for `1 + m1 + m2`. Since this branching recursion will visit its subproblems an exponential number of times, you might want to sprinkle some `@lru_cache` memoization on it to rein it in.

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

Sum of distinct cubes

def sum_of_distinct_cubes(n):

Determine whether it is possible to express the given positive integer n as a sum of **distinct** cubes of **positive** integers. If such a feat is possible, return the list that contains the cubed integers in descending order. For example, if n = 1456, this function would return [11, 5], since $11^3 + 5^3 =$ 1456. If such breakdown of n into distinct cubes is impossible, return None.

Unlike in the earlier, far simpler question of expressing an integer as a sum of exactly two squares, the result list can now contain any number of elements as long as they are all distinct and their cubes add up to n. If the number n allows several such breakdowns into sums of distinct cubes, return the **lexicographically largest** list, that is, starting with the largest possible first number a, followed by the lexicographically largest representation of n-a*a*a. For example, when called with n = 1072, this function should return [10, 4, 2] instead of [9, 7].

This problem is best solved with recursion. If n itself is a cube, return the singleton list [n]. 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 make this function efficient even for large n, it might be good 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 so that the number of possibilities that we need to iterate through remains finite. Since the cube of a negative number is also negative, such a finite upper bound no longer exists once negative numbers are allowed, and breaking even a small number into a sum of exactly three cubes [becomes a highly nontrivial problem](#).

Fractional fit

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

You are given a list of integer two-tuples `(a, b)` so that `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`, your task is to find the length of the longest possible list that could be constructed from these 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*... 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 `[(3, 7), (1, 4), (8, 9)]` does not, 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

In the Martin Gardner column mentioned in the earlier Bulgarian Solitaire problem, this puzzle was used as an example of an opposite situation: initially seemingly unlimited process where it turns out that even if you were allowed to freely choose your list of fractions, every such sequence will inevitably paint itself into a corner after at most 17 steps. No matter how you twist and turn, some two of these fractions will lie on the same 1/18th of the unit interval, thus making the next chosen fraction, if you pardon a dad joke pun better suited for a "Jumble" newspaper puzzle, a "moot point".

Followed by its square

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

Unlike our previous functions that have received entire lists as parameters, this function receives some Python **iterator** that is guaranteed to produce some finite sequence of **strictly increasing positive integers**, but there are no guarantees of how long that sequence will be and how large the numbers inside it or the gaps between them will grow to be. This iterator **is not a list** that would allow you **random access** to any element based on its position and this way lets you jump back and forth as your heart desires. Therefore, processing the elements given by this iterator must be done in a strictly sequential fashion, although you can 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):
```

The point $(x1, y1)$ on the two-dimensional plane is said to **dominate** another point $(x2, y2)$ if it lies both to the right and above from it, that is, $x1 > x2$ and $y1 > y2$. A point inside the given list of `points` is defined to be **maximal** if it is not dominated by any other point in the 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 the list of points $[(1, 5), (8, 1), (3, 10), (2, 1), (9, 2)]$.

Given a list of `points` from the upper right quadrant of the two-dimensional plane so that all coordinates are guaranteed to be nonnegative, this function should compute how many times one would need to perform the operation of removing every point in the maximal layer from the list (and then compute the new maximal layer for the remaining points for the next round) for every point to eventually become a maximal point and removed, 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 actual educational point and motivation is making this function fast enough to finish within a reasonable time even for a big list of points, by not doing too much more work than would be necessary to identify and remove the maximal points. Start by noticing that each point can potentially be dominated only by those points whose distance from the origin (0, 0) is strictly larger. It doesn't even matter which distance metric is used...

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 piece** currently stands at the coordinates (x, y), and the parameter `pieces` is a Python `set` that contains the positions of all of the enemy pieces. This function should compute and return the maximum number of pieces that your king could capture in a single move.

In our variant of checkers, a king can capture an opponent's piece that sits one step away from it to any of the four **diagonal directions** (the list `[(-1, 1), (1, 1), (1, -1), (-1, -1)]` might come in handy in your code) so that the square behind the opponent piece in that diagonal direction is vacant. Your king can then capture that opponent piece by jumping into that vacant square, **immediately removing that opponent piece from the board**. However, unlike in chess where each move can capture at most one piece, the capturing chain in checkers can and will continue from the square that your piece jumps into, provided that some diagonal neighbour of that square also contains an opponent piece with a vacant square behind it.

This problem is best solved recursively. The base case of the problem is when there are no neighbouring opponent pieces that could be captured, returning the answer zero. Otherwise, loop through all four diagonal directions. For each such direction that contains an opponent piece that can be captured, remove that piece from the board and recursively compute the number of pieces that can be captured from the vacant square that your piece jumps into, and add one to that number from the piece that you first captured. Return the largest number that can be achieved this way.

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 [Collatz conjecture](#) we have encountered in some earlier problems, in this problem the parity of n does not restrict 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 a later algorithms course that explains such 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.

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 how any integers that have already made an appearance in some 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 can see that the terms start making repeated appearances, especially the term zero.

In the same spirit as in the earlier problem that asked you to generate terms of the Recaman 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 generated sequence to look for the most recent occurrence of each term that it generates, but rather use a Python dictionary to remember the terms that have already been seen in the sequence, along with the positions of their most recent appearances. With this dictionary, you don't need to explicitly store the sequence, but only its relevant previous terms.

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

Reversing the reversed

```
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 necessarily have to use *recursion* to solve this problem. The base case handles any argument that is not a list. When the parameter `items` 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 touch the contents of the original 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]`

Prime factors of integers

`def prime_factors(n):`

As the [fundamental theorem of arithmetic](#) again reminds us, every positive integer can be broken down into the product of its prime factors exactly one way, disregarding the order of listing these factors. Given positive integer `n > 1`, return the list of its prime factors in sorted ascending order, each prime factor included in the list as many times as it appears in the prime factorization of `n`.

During the testing, the automated tester is guaranteed to produce the pseudo-random values of `n` in ascending order. The difficulty in this problem is making the entire test to finish in reasonable time by caching the prime numbers that you discover along the way the same way as was done in `primes.py` example script, but also **caching** the prime factorization of numbers for quick access later. To be able to quickly reconstruct the prime factorization of `n` that you have already computed earlier, you don't need to cache all of the prime factors of `n`. Storing any single one of its prime factors, let's call that one `p`, is enough, since the rest of the prime factorization of `n` is the same as the prime factorization of the remaining number `n//p`. Furthermore, if the smallest prime factor `p` is small enough that it can be found quickly later anyway, it does not need to be cached at all, and this way save memory for the non-trivial factors of the more difficult values of `n`.

n	Expected result
42	[2, 3, 7]
10**6	[2, 2, 2, 2, 2, 2, 5, 5, 5, 5, 5, 5]
1234567	[127, 9721]
99887766	[2, 3, 11, 31, 48821]
10**12 - 1	[3, 3, 3, 7, 11, 13, 37, 101, 9901]
10**15 - 3	[599, 2131, 3733, 209861]

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 the computational complexity truly begins.

We normally represent integers in base 10, each digit giving the coefficient of some power of 10. Since every positive integer can equally well be uniquely represented as a sum of powers of two, computers internally encode those same integers in the simpler (for machines) **binary**. Both of these schemes need [special tricks to represent negative numbers](#) since we cannot use an explicit negation sign. The [balanced ternary](#) representation of integers uses the base three instead of two, with **signed** coefficients from -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, unlike the bases of ten or two, the balanced ternary representation is perfectly symmetric so that the representation for *-n* can be constructed by flipping the signs of the terms of the representation of *+n*. (See the examples below.)

Given an integer n, return the unique list of signed powers of three that add up to n, listing these powers in the descending order of their absolute values. This problem can be solved in a couple of different ways. The page "[Balanced ternary](#)" shows one way by first converting the number to unbalanced ternary (base three using coefficients 0, 1 and 2) and converting from there. Another way 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 to the result and convert $n - p$ for the rest, or take $3p$ to 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 reroll 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 would 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 theoretically 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 the 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 test case in 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 random ones, fours and sixes than the probabilities of fair dice would normally give. There is no law of either nature or man against that kind of tomfoolery.)

Optimal crag score

def optimal_crag_score(rolls):

Way back near when we started doing these problems, one particular 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 real, again aided with the same gift of perfect foresight as in the previous problem "Lords of Midnight" so that given the knowledge of the entire future sequence of rolls, this function should return the highest possible score that could be achieved with those rolls under the constraint that **the same category cannot be used more than once**.

This problem will require recursion to solve, 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 are 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 rest of the categories that are still available.

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 recursion to work out.

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)

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 linked 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.

Your task is to write a function that computes the k:th character of the infinite Fibonacci word, with the position counting done 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'