

# CCPS 109 Computer Science I Labs

Ilkka Kokkarinen

Chang School of Education, Ryerson University

Version of October 3, 2021

# General requirements

This document contains specifications for the lab problems in the course [CCPS 109 Computer Science I](#), as taught by [Ilkka Kokkarinen](#) for Chang School of Continuing Education, Ryerson University, Toronto, Canada. It contains 109 individual problems for you to choose your battles from, presented roughly in order of increasing complexity.

All problems are designed to allow solutions **using only the core computation structures introduced during the first five weeks of this course**, assuming that the student has completed enough practice problems at the excellent [CodingBat Python](#) interactive training site, or acquired the equivalent knowledge from earlier studies. Except for a handful of problems that explicitly point out the standard library modules necessary to solve them, no knowledge of any specialized Python libraries is required. However, **you are allowed to use anything in the Python 3 standard library** that you find useful in expressing your program logic. (In computer science and programming, [no problem should ever have to be solved twice!](#))

**Each correctly solved problem is worth the exact same one point to your course grade.** Your grade starts from the baseline of 40 points, not itself quite enough to pass the course. Solving ten problems makes your course grade reach 50 points, which corresponds to the lowest possible passing course grade of D minus. Solving fifty problems (yes, *any* fifty) takes you up to 90 points where the highest possible course grade of A plus awaits.

**You must implement all these functions in a single source code file** named `labs109.py`. This allows you to run the [tester109.py](#) script at any time to validate the functions you have completed so far, so you always know exactly where you are standing on this course. These tests are executed in the same order that your functions appear inside the `labs109.py` file.

Your functions may assume that **the arguments given to them are as promised in the problem specification**, so that these functions never have to handle and recover from arguments whose value or type is invalid. Your functions must especially handle zero integers and empty lists correctly whenever they can be legitimate argument values.

**The test for each individual function should take at most a couple of seconds to complete.** If some test takes a minute or more to complete, your code is too slow and its logic desperately needs to be streamlined. This is usually achieved by shaving off one level of nesting from the structured loops, occasionally with the aid of a `set` or a `dict` to remember the stuff that your function has seen and done so far.

[Silence is golden.](#) **None of your functions should print anything on the console**, but return the expected result silently. 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, **your function should never change the contents of the lists that they receive as arguments**. If your function really has to do this for a list of `items`, make sure to first create a local copy of that list with the assignment `items = items[:]` and operate on that local copy.

This specification document and the automated tester are released under GNU Public License version 3 so that they can be adapted and distributed among all teachers and students of computer science. The author compiled these problems over time from a gallimaufry of sources ranging from the original lab and exam problems of his old Java version of this course to a multitude of programming puzzle and code challenge sites such as [LeetCode](#), [CodeAbbey](#), [Stack Exchange Code Golf](#), and [Wolfram Programming Challenges](#). Classic recreational mathematics ([yes, that is a real thing](#)) columns by Martin Gardner and his spiritual disciples also inspired many problems.

The author has tried to dodge not just the Scylla of the well-worn problems that you can find in almost all textbooks and problem collections, but also the Charybdis of pointless make-work drudgery that doesn't have any inherent meaning or purpose on its own above the finger practice to provide billable "jobs for the boys" and maintain the illusion that the System is working as intended. Some of these seemingly simple problems also touch the deeper issues of computer science that you will encounter in your later undergraduate and graduate courses, and occasionally even relate to entire other fields of human endeavour and that way have nontrivial worldview implications, both philosophical and practical.

This problem set also has an official associated subreddit [r/ccps109](#) for students to discuss the individual problems and associated issues. This discussion should take place at the level of ideas, so that no solution code should be posted or requested. Any issues with course management and the course schedule in any particular semester should be kept out of this subreddit to give a more permanent and fundamental nature. Furthermore, no student should at any time be stuck with any one problem. Once you embark on solving these problems, you should read ahead to find at least five problems that interest you. Keep them simmering on your mental back burner as you go about your normal days. Then when you get on an actual keyboard, work on the problem where you feel yourself being closest to a working solution.

The author wishes to thank past students **Shu Zhu Su**, **Rimma Konoval**, **Mohammed Waqas** and **Zhouqin He** for going above and beyond the call of duty in completing solutions that either revealed errors in the original problem specifications or their private model solutions, or independently agreed with the private model solution and this way raised the confidence to the correctness of both. A small legion of individual students who pointed out ambiguities, errors and inconsistencies in particular problems blazed this trail and made this ride smoother for future students, and the members of that horde know who they are. All remaining errors, ambiguities and inconsistencies both logical and extracurricular remain the sole responsibility of Ilkka Kokkarinen. Report any and all such errors as discovered to [ilkka.kokkarinen@gmail.com](mailto:ilkka.kokkarinen@gmail.com).

# List of Problems

Ryerson letter grade	10
Ascending list	11
Riffle shuffle kerfuffle	12
Even the odds	13
Cyclops numbers	14
Domino cycle	15
Colour trio	16
Count dominators	17
Beat the previous	18
Subsequent words	19
Taxi $\mathbb{Z}$ um $\mathbb{Z}$ um	20
Exact change only	21
Rooks on a rampage	22
Try a spatula	23
Words with given shape	24
Chirality	25
The card that wins the trick	26
Do you reach many, do you reach one?	27
Sevens rule, zeros drool	28
Fulcrum	29
Fail while daring greatly	30
All your fractions are belong to base	31

Recamán's sequence	32
Count the balls off the brass monkey	33
Count growlers	34
Bulgarian solitaire	35
Scylla or Charybdis?	36
Longest arithmetic progression	37
Best out of three	38
Whenever they zig, you gotta zag	39
Crack the crag	40
Three summers ago	41
Sum of two squares	42
Carry on Pythonista	43
As below, so above	44
Expand positive integer intervals	45
Collapse positive integer intervals	46
Prominently featured	47
Like a kid in a candy store, except without money	48
Dibs to dubs	49
Nearest smaller element	50
Interesting, intersecting	51
So shall you sow	52
That's enough of you!	53
Brussel's choice	54
Count consecutive summers	55
McCulloch's second machine	56
That's enough for you!	57
Crab bucket list	58

What do you hear, what do you say?	59
Bishops on a binge	60
Dem's some mighty tall words, pardner	61
Up for the count	62
Reverse the vowels	63
Everybody on the floor, do the Scrooge Shuffle	64
Rational lines of action	65
Verbos regulares	66
Hippity hoppity, abolish loopity	67
Where no one can hear you bounce	68
Nearest polygonal number	69
Don't worry, we will fix it in the post	70
Fracran interpreter	71
Bulgarian cycle	72
Permutation cycles	73
Whoever must play, cannot play	74
ztalloc ecneuqes	75
The solution solution	76
Reverse ascending sublists	77
Brangelin-o-matic for the people	78
Line with most points	79
Om nom nom	80
Autocorrect for sausage fingers	81
Uambcsrlne the wrod	82
Substitution words	83
Manhattan skyline	84
Count overlapping disks	85

Ordinary cardinals	86
Count divisibles in range	87
Bridge hand shape	88
Milton Work point count	89
Never the twain shall meet	90
Bridge hand shorthand form	91
Points, schmoints	92
Bulls and cows	93
Frequency sort	94
Calling all units, B-and-E in progress	95
Lunatic multiplication	96
Distribution of abstract bridge hand shapes	97
Flip of time	98
Fibonacci sum	99
Wythoff array	100
Rooks with friends	101
Possible words in Hangman	102
All branches lead to Rome	103
Be there or be square	104
Split within perimeter bound	105
Sum of distinct cubes	106
Fractional fit	107
Followed by its square	108
Count maximal layers	109
Maximum checkers capture	110
Collatzy distance	111
Van Eck sequence	112

Tips and trips	113
Balanced ternary	114
Lords of Midnight	115
Optimal crag score	116
Forbidden substrings	117
Go for the Grand: Infinite Fibonacci word	118
Bonus problem 110: Reverse the Rule 110	119





## Ryerson letter grade

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

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

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

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

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

# Ascending list

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

Determine whether the sequence of `items` is **strictly ascending** so that each element is **strictly larger** (not just merely **equal to**) than the element that precedes it. Return `True` if the list of `items` is strictly ascending, and return `False` otherwise.

Note that the empty sequence is ascending, as is also every one-element sequence, so be careful that your function returns the correct answers in these seemingly insignificant **edge cases** of this problem. (If these sequences were not ascending, pray tell, what would be the two elements that violate the requirement and make that particular sequence not be ascending?)

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

In the same spirit, note how every possible universal claim made about the elements of an empty sequence is trivially true! For example, if `items` is the empty sequence, the two claims “All elements of `items` are odd” and “All elements of `items` are even” are both equally true, as is also the claim “All elements of `items` are [colourless green ideas that sleep furiously](#).” Many people initially find this to be highly counterintuitive and paradoxical, but it is a consequence of the fact that any logical formula of the form “If *A*, then *B*” (or for that matter, “*A* entails *B*”, “*A* implies *B*”, or even “*B* is a consequence of *A*” depending on which level of meta-logic you feel most comfortable living in) is satisfied whenever *A* is false.

## Riffle shuffle kerfuffle

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

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

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

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

## Even the odds

```
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.

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, you can first convert the number into a string and work from there with string operations.

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

# Cyclops numbers

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

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

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

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

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

# 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

# Colour trio

```
def colour_trio(colours):
```

This problem was inspired by the fun little [Mathologer](#) video “[Secret of Row 10](#)”. To start, define a simple **algebraic structure** made of three elements imaginatively named “red”, “yellow” and “blue”. These names are intended to serve as colourful (heh) mnemonics for us to conveniently talk about abstract elements that could just as well have been named “foo”, “bar” and “baz”, so no connection to actual physics of colour is intended.

Next, define a simple “addition table” between these colours with a rule that says that whenever any colour is added to itself, the result is that same colour, whereas adding two different colours always gives the third. For example, adding blue to blue gives you that same blue, whereas adding blue to yellow would give you red.

Given the first row of `colours` as a string of lowercase letters to denote these colours, this function should repeatedly construct the rows below the first row according to the following discipline. The  $i$ :th element of each row is calculated by adding the two colours in positions  $i$  and  $i + 1$  of the previous row above the current row. This process terminates when the current row contains only one element, returned as the final answer. For example, the first row `'rybyr'` leads to `'brrb'`, which leads to `'yry'`, which leads to `'bb'`, which leads to `'b'` as our final answer, Regis.

colours	Expected result
'y'	'y'
'bb'	'b'
'rybyry'	'r'
'brybbr'	'r'
'rbyryrrbyrbb'	'y'
'yrbbbbryyrybb'	'b'



# Count dominators

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

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

Before starting to write code for this function, you should consult the parable of "[Shlemiel the painter](#)" and think how this seemingly silly tale from a simpler time relates to today's computational problems performed on 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 achieves with two nested loops. Your workload therefore increases only **linearly** with respect to the number of `items`, whereas the total time of Shlemiel's back-and-forth grows **quadratically**, that is, as a function of the **square** of the number of items.

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

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



# Subsequent words

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

This problem is an excuse to introduce some general discrete math terminology that helps make many later problem specifications less convoluted and ambiguous. A **substring** of a string consists of characters taken **in order** from consecutive positions. Contrast this with the similar concept of **subsequence** of characters still taken in order, but not necessarily at consecutive positions. For example, each of the five strings `' '`, `'e'`, `'put'`, `'ompu'` and `'computer'` is both a substring and subsequence of the string `'computer'`, whereas `'cper'` and `'out'` are subsequences, but not substrings.

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

Now that you know all that, given a list of words sorted in alphabetical order, and a string of required letters, find and return the list of words that contain letters as a *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', 'kiloostered', 'kiloparsec', 'kilostere', 'kiloware']
'brohiic'	['bronchiectatic', 'bronchiogenic', 'bronchitic', 'ombrophilic', 'timbrophilic']
'azaz'	['azazel', 'azotetrazole', 'azoxazole', 'diazoaminobenzene', 'hazardize', 'razzmatazz']

# Taxi $\mathbb{Z}$ um $\mathbb{Z}$ um

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

A taxicab cruising the street grid of the dusky Manhattan that we know and love from classic *film noir* works such as [“Blast of Silence”](#) starts its journey at the origin  $(0,0)$  of the infinite two-dimensional integer grid, denoted by  $\mathbb{Z}^2$ . The taxicab has a **heading** that is always one of the four main compass directions, initially heading north. The taxicab then faithfully executes the given sequence of moves given as a string of characters 'L' for turning 90 degrees left while standing in place (just in case we are making a turn backwards, in case you spotted some glad rags or some out of town palooka looking to be taken for a ride), 'R' for turning 90 degrees right (ditto), and 'F' for moving one step forward to the current heading. This function should return the final position of the taxicab in on this infinitely spanning Manhattan. (Perhaps the rest of the world lazily simulates this infinite street grid with mirrors; how would you even know?)

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

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

## Exact change only

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

Given the amount of money (expressed as an integer as the total number of [kopecks](#) of Poldavia, Ruritania, Montenegro or some other such fictional vaguely Eastern European country) and the list of available denominations of `coins` (similarly expressed as kopecks), create and return a list of coins that add up to the amount using the **greedy approach** where you use as many of the highest denomination coins when possible before moving on to the next lower denomination. The list of coin denominations is guaranteed to be given in descending sorted order, as should your returned result also be.

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

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

## Rooks on a rampage

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

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

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

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

## Try a spatula

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

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

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

text	Expected result
'hello world'	'drwolhel ol'
'ilkka'	'akilk'
'pancake'	'eanpack'
'abcdefghijklmnopqrstuvwxyz'	'zxvtrpnljhdbacegikmoqsuwy'
'this is the best of the enterprising rear'	're nsrrteetf sbets ithsi h eto h nepiigra'

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 to sort the given list.

## Words with given shape

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

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

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

shape	Expected result (using wordlist <code>words_sorted.txt</code> )
[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 a recreational challenge to find the shape of length  $n - 1$  that matches the largest number of words, for the possible values of  $n$  from 3 to 20. Alternatively, try to count how many possible shapes of length  $n - 1$  do not match any words of length  $n$  at all. What is the shortest possible shape that does not match any words?



# Chirality

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

Even though this has no effect on fairness, pips from one to six are not painted on dice just any which way, but so that [pips on the opposite faces always add up to seven](#). (This convention makes it easier to tell when someone tries to use gaffed dice without certain undesirable pip values.) In each of the  $2^3 = 8$  corners of the cube, exactly one value from each pair of forbidden opposites 1-6, 2-5 and 3-4 meets two values chosen from the other two pairs of opposites. You can twist and turn any corner of the die to face you, and yet two opposite sides never spread in simultaneous view.

This discipline still allows for two distinct ways to paint the pips. In the corner shared by the faces 1, 2, and 3, if these numbers read out **clockwise** as 1-2-3, that die is **left-handed**, whereas if they read out as 1-3-2, that die is **right-handed**. Analogous to a pair of shoes made separately for the left and right foot, left- and right-handed dice are in certain sense identical, and yet again no matter how you twist and turn, you can't seriously put either shoe in the other foot than the one it was designed for. (Well, at least not without taking that three-dimensional pancake “Through the Looking-Glass” by flipping it around in the fourth dimension!)

The three numbers read around any other corner stamp the three numbers in the unseen opposite sides, and therefore determine the handedness of that entire die just as firmly. Given the three-tuple of pips read clockwise around some corner, determine whether that die is left-handed. There are only  $2^3 \cdot 3! = 8 \cdot 6 = 48$  possible pip combinations to test for, so try to exploit as many of the four two-fold symmetries to simplify your code. (This problem would certainly make for an interesting exercise [code golf](#), a discipline that we otherwise frown in this course as *the* falsest economy.)

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

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

# The card that wins the trick

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

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

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

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

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

## Do you reach many, do you reach one?

```
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, the entire chessboard can be generalized into  $k$  dimensions from just the puny two. A natural extension of the knight's move to keep moves symmetric with respect to these dimensions is to define the possible moves as some  $k$ -tuple of **strictly decreasing** nonnegative integer offsets. Each one of these  $k$  offsets must be used for exactly one dimension of your choice during the move, either as a positive or a negative version.

For example, the three-dimensional  $(4, 3, 1)$ -knight makes its way by first moving four steps along any one of the three dimensions, then three steps along any other dimension, and then one step along the remaining dimension, whichever dimensions that was. These steps are considered to be performed together as a single jump that does not visit or is blocked by any of the intermediate squares. Given the `start` and `end` positions as  $k$ -tuples of integer coordinates, determine whether the knight can get from `start` to `end` in a single jump.

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

A quick combinatorial calculation reveals that exactly  $k! * 2^k$  possible neighbours are reachable in a single move, minus the moves that jump outside the board. In this notation, the ordinary chess knight is a  $(2, 1)$ -knight that reaches  $2! * 2^2 = 8$  neighbours in one jump. A 6-dimensional knight could reach a whopping  $6! * 2^6 = 46080$  different neighbours in one jump! Since the number of moves emanating from each position to its neighbours grows exponentially with respect to  $k$ , pretty much everything ends up being close to almost everything else in high-dimensional spaces. (Density in general is known to have both advantages and disadvantages in all walks of life.)

# Sevens rule, zeros drool

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

Seven is considered a lucky number in Western cultures, whereas [zero is what nobody wants to be](#). Let us briefly bring these two opposites together by looking at positive integers that consist of some solid sequence of sevens, followed by some (possibly empty) solid sequence of zeros. Examples of such integers might be 7, 77777, 7700000, 77777700, or 7000000000000000. A surprising theorem proves that for any positive integer  $n$ , there exist infinitely many integers of such seven-zero form that are divisible by  $n$ . This function should return the **smallest** such seven-zero integer.

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

[illegible]

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

# Fulcrum

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

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

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

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



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

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

(Physics side quiz: seeing that a helium balloon rises and stays up in the air, wouldn’t a vacuum balloon do the same in an even more buoyant manner?)

# Fail while daring greatly

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

The ancient world “ when men were made of iron and their ships were made of wood ” could occasionally be [an entertainingly violent place](#), at least if we believe [historical docudramas](#) of swords and sandals such as “300”, “Spartacus” and “Rome” marred by dust and sweat and blood. (Historical records from those eras are spotty at best anyway.) During [one particularly memorable incident](#), a group of [zealots](#) (yes, Lana, *literally*) found themselves surrounded by overwhelming Roman forces. To avoid capture and arduous death by crucifixion, in their righteous zeal these men committed themselves to mass suicide in a way that ensured each man’s commitment to this shared fate. They arranged themselves in a circle, and used lots to choose a step size  $k$ . From the first man, repeatedly count  $k$  men ahead, kill that man and remove his corpse from this grim circle.

Being [normal people instead of computer scientists](#), this deadly game of eeny-meeny-miney-moe one-based counting continues until the last man standing falls on his own sword to complete the circle. [Josephus](#) would very much prefer to be this last man, since he has other ideas of surviving. Help him survive with a function that, given  $n$  and  $k$ , returns the list of the execution order so that these men know which places let them be the survivors who get to walk away from this grim circle. A [cute mathematical solution](#) instantly determines the survivor for  $k = 2$ . Unfortunately  $k$  can get arbitrarily large, even far exceeding the current number of men... if only to briefly excite us cold and timid souls, hollow men without chests, the rictus of our black lips gaped in grimace that sneers at the strong men who once stumbled. (If only to lighten up this dramatic lament, note the [feline generalization of this problem](#) that practically begs to be turned into an adorable viral video.)

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 ]
10	10**100	[ 10, 1, 9, 5, 2, 8, 7, 3, 6, 4 ]

# All your fractions are belong to base

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

A pile of  $n$  identical coins lies on the table. Each move consists of three stages. First, the coins in the remaining pile are arranged into groups of exactly  $out$  coins each, where  $out$  is a given positive integer greater than one. Second, the  $n \% out$  leftover coins that did not make a complete group of  $out$  elements are taken aside and recorded. Third, from each complete group of  $out$  coins taken out, exactly  $ins$  coins are collected to together create the new single pile, the rest of the coins again put aside. Repeat this three-stage move until the entire pile becomes empty, which must eventually happen whenever  $out > ins$ . Return a list of how many coins were taken aside in each move.

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

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

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

# Recamán's sequence

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

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

Note how the definition for the current element depends on whether a particular number can be found in the previously generated part of the sequence. To allow your function to execute in a sure and speedy fashion even for the prefix of millions of elements, you should use a set to keep track of which values are already part of that sequence. This way you can generate each element in constant time, instead of having to iterate through the entire previously generated list like some "[Shlemiel](#)" would have done. The better technique that saves time by using more memory can create this list arbitrarily far in linear time, and should therefore be reasonably fast even for millions of elements and upwards, at least until the process runs out of memory. Hey, nothing is perfect, and [everything is a tradeoff](#).

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



# Count the balls off the brass monkey

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

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

This problem could be solved in a straightforward **brute force** fashion by mechanistically tallying up the spheres iterated through these layers. However, the just reward for naughty boys and girls who take such a blunt approach is to get to watch the automated tester take roughly a minute to terminate! Some creative use of discrete math and summation formulas gives an **analytical closed form formula** that makes the answers come out faster than you can snap your fingers simply by plugging the values of  $n$ ,  $m$  and  $h$  into this formula. (Seeing that the brass monkey is a Mesoamerican pyramidal shape discretized into a pyramidal integer grid instead of the cubic  $\mathbb{Z}^3$  integer grid, formulas for [pyramidal numbers](#) might be a good starting point...)

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

# Count growlers

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

Let the strings 'cat' and 'dog' denote that kind of animal facing left, and 'tac' and 'god' denote that same kind of animal facing right. Since in this day and age this whole setup sounds like some kind of a meme anyway, let us somewhat unrealistically assume that each individual animal, regardless of its own species, growls if it sees **strictly more dogs than cats** to the direction that the animal is facing. Given a list of such `animals`, return the count of how many of these animals are growling. In the examples listed below, the growling animals have been highlighted in green.

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 am the first to admit that I was high as a kite when I thought up this problem.)

# Bulgarian solitaire

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

You are given a row of `piles` of identical pebbles, and a positive integer `k` so that the total number of pebbles in these piles equals  $k * (k+1) // 2$ , the `k`:th **triangular number** that equals the sum of the positive integers from 1 to `k`. (Quickly, what is the [sum of integers from 1 to 100?](#))

An apt metaphor for the bleak daily life behind the Iron Curtain, all pebbles are identical and you don't have any choice in your moves in this solitaire game. Each move picks up exactly one pebble from every pile (making piles with only one pebble vanish), and creates a new pile from this handful. For example, the first move from `[7, 4, 2, 1, 1]` turns into `[6, 3, 1, 5]`. The next move turns into `[5, 2, 4, 4]`, which then turns into `[4, 1, 3, 3, 4]`, and so on.

This function should count how many moves lead from the initial `piles` to the **steady state** where each number from 1 to `k` appears as the size of **exactly one pile**, and return that count. These numbers from 1 to `k` may appear in any order, not necessarily sorted. (Applying this move to this steady state will simply lead right back to that same steady state, hence the name.)

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

This problem comes from the [Martin Gardner](#) column “*Bulgarian Solitaire and Other Seemingly Endless Tasks*”. It was used there as an example of a while-loop where it is not immediately obvious that this loop will always eventually reach its goal and terminate, analogous to the behaviour of the **Collatz 3x+1 problem** seen in [mathproblems.py](#). However, unlike in that still annoyingly open problem, Bulgarian solitaire can be proven to never get stuck, but reach the steady state from any starting configuration of triangular numbers after at most  $k(k-1)/2$  steps.

## Scylla or Charybdis?

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

This problem was inspired by the article "[A Magical Answer to the 80-Year-Old Puzzle](#)" in [Quanta Magazine](#). Thanks to your recent cunning stunts, your [nemesis](#) in life finds herself trapped inside a devious game where, for a refreshing change from the usual way of things, you get to play the Final Boss. (Everyone is the hero in their own story until they become a villain in somebody else's, after all.) Her final showdown resembles a one-dimensional 8-bit video game platform that reaches  $n-1$  discrete steps from its center to both directions, with beep boop sound effects to complete the mood of that era. At each end of this platform, exactly  $n$  steps from the center, your lethal friends [Scylla and Charybdis](#) are already licking their lips in anticipation of a tasty morsel to fall in over the edge.

Your nemesis starts at the center of this platform, and must begin to immediately committing to her entire sequence of future moves as a string of '+' ("♪ just a step to the ri-i-i-ight ♪") and '-' (move one step to the left). Your task is to find and return a positive integer  $k$  so that executing every  $k$ :th step of moves (so this subsequence starts from position  $k-1$  and includes every  $k$ :th element from then on) makes your nemesis fall into either one of the two possible dooms  $n$  steps away from her starting point at the center.

If several values of  $k$  do the job, return the smallest value of  $k$  among those that minimize the number of steps to the downfall. Otherwise, those freaking Tolkien eagles will once again swoop down to rescue your nemesis. She will then return in a third year course on analysis of algorithms as an **adversary** to put your functions through a wringer of wits where she gets to wield the same **second-mover advantage** that you got to enjoy this time.

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

# Longest arithmetic progression

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

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

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

items	Expected result
<code>[42]</code>	<code>(42, 0, 1)</code>
<code>[2, 4, 6, 7, 8, 12, 17]</code>	<code>(2, 2, 4)</code>
<code>[1, 2, 36, 49, 50, 70, 75, 98, 104, 138, 146, 148, 172, 206, 221, 240, 274, 294, 367, 440]</code>	<code>(2, 34, 9)</code>
<code>[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]</code>	<code>(7, 87, 9)</code>
<code>range(1000000)</code>	<code>(0, 1, 1000000)</code>

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

## Best out of three

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

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

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

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

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

# Whenever they zig, you gotta zag

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

A positive integer  $n$  is a **zigzag number** (also called an “alternating number” in some combinatorics materials) if the series of differences between its consecutive digits read from left to right strictly alternates between positive and negative steps. The step from the first digit to the second may be either positive or negative to start this dance. This 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** is a dice game similar to the more popular games of [Yahtzee](#) and [Poker dice](#) in style and spirit, but with much simpler combinatorics of roll value calculation due to this game using only three dice. Players repeatedly roll three dice and assign the resulting patterns to **scoring categories** so that once some roll has been assigned to a category, that category is considered to have been spent and cannot be used again for any future roll. These tactical choices between safety and risk-taking give this game a more tactical flair on top of merely relying on the favours of Lady Luck for rolling the bones. See [the Wikipedia page](#) for the scoring table used in this problem.

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

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

dice	Expected result
<code>[1, 2, 3]</code>	20
<code>[4, 5, 1]</code>	5
<code>[3, 3, 3]</code>	25
<code>[4, 5, 4]</code>	50
<code>[1, 1, 1]</code>	25
<code>[1, 1, 2]</code>	2



## Three summers ago

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

Given a sorted list of positive integer `items`, determine whether there exist **precisely three** separate `items` that together exactly add up to the given positive integer `goal`.

Sure, you could 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, iterating through all triples of elements would get pretty slow as the length of the list increases, seeing that the number of such triples to pick through grows proportionally to the **cube** of the length of the list! Of course the automated tester will make those lists large enough so that such solutions reveal themselves with their glacial running time.

Since `items` are known to be sorted, a better technique will find the answer significantly faster. See the function `two_summers` in the example program [listproblems.py](#) to quickly find two elements in the given sorted list that together add up to the given `goal`. You can 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 some two elements that add up to `goal-x`.

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

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

## Sum of two squares

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

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

To facilitate automated testing, the returned tuple must present the larger of its two numbers first. Furthermore, if some integer can be expressed as a sum of two squares in several ways, return the breakdown that maximizes the larger number. For example, the integer 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**, as previously seen in the function `two_summers` in the [listproblems.py](#) example program, directly works also on this problem! The two indices start from both ends of the sequence, respectively, inching towards each other until they eventually either find a working solution, or meet somewhere before ever finding one.

n	Expected result
1	None
2	(1, 1)
50	(7, 1)
8	(2, 2)
11	None
$123^2 + 456^2$	(456, 123)
$5555^2 + 6666^2$	(77235, 39566)

(In **binary search**, one of these indices would jump halfway towards the other in every round, making the execution time **logarithmic** with respect to  $n$ . However, we are not in such a lucky situation with this setup.)

# Carry on Pythonista

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

Two positive integers *a* and *b* can be added with the usual integer column-wise addition algorithm that we all learned as wee little children. Instead of the sum *a+b* that the language would already compute for you anyway, this problem makes you count how many times there will be a **carry** of one into the next column during this mechanistic addition. Your function should be efficient even for behemoths that consist of thousands of digits.

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 traipse through the steps of the **column-wise integer addition** where you don't care about the actual result of the addition, but only tally the carries produced in each column as **proof of work** of you actually labouring through the steps of the column-wise addition.

a	b	Expected result
0	0	0
99999	1	5
1111111111	2222222222	0
123456789	987654321	9
2**100	2**100 - 1	13
10**1000 - 123**456	123**456	1000

## As below, so above

```
def leibniz(heads, positions):
```

Regardless of your stance on the Pascal's wager, betting that rows of the famous [Pascal's triangle](#) will be generated at some point during a computer science intro course is safe as houses. Even without any consideration to the [interesting combinatorial sequences inside this infinite table](#), merely tabulating its entries makes for a splendid exercise of nested loops.

However, our version looks at **Leibniz triangles**, a curious “upside-down” variation of Pascal's triangle where each entry equals the sum of the two entries immediately **below** it. Unlike the well-coiffed recursive definition of the Pascal's triangle where every recursive call is directed towards the Holy Trinity of Absolute Units that begat the rest of this table, Leibniz triangles have no base cases at all! Instead, they form infinite pyramids of turtles standing on each other's backs all the way down to never-never-land from where the Adversary has infinitely many ways to fill in this triangle.

To allow a finite usurper to reside at the top of this infinite pyramid, the Good Lord discovered fractions and nonpositive numbers. As it turns out, merely defining the heads, the leading elements of each row, sufficiently constrains the recursion to have a unique solution everywhere in the triangle! The original [Leibniz harmonic triangle](#) is one possible solution for this set of recursive equations, generated from using consecutive **unit fractions** as heads.

This function should generate as many rows of the Leibniz triangle as there are heads given, using two nested loops to generate the entries. However, to simplify automated testing, this function needs to return only the entries in given `positions` of the last generated row. Once those are correct, we can safely assume your intermediate rows to also have been correct, without actually forcing you to “show your work”. (Even though you do have to loop through the rows, you don't actually need to explicitly store all of them, but only the current row and its previous row.)

heads	positions	Expected result
[3, 2]	range(2)	[2, 1]
[1, -1, 1, -1]	range(4)	[-1, 2, -4, 8]
range(6)	range(6)	[5, -1, 0, 0, 0, 0]
[Fraction(1, n) for n in range(1, 6)]	[4, 2]	[Fraction(1, 5), Fraction(1, 30)]

## 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 intentionally restricted to positive integers so that there will be no ambiguity between the minus sign character used as a separator and an actual unary minus sign tacked in front of a digit sequence.) For example, the interval that contains the integers 5, 6, 7, 8, 9 can be more concisely described as '5-9'. Multiple intervals can be described together by separating their descriptions with commas. A singleton interval with only one value is given as that value.

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

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

## Collapse positive integer intervals

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

This function is the inverse of the previous problem of expanding positive integer intervals. Given a nonempty list of positive integer `items` 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`. Such encoding doesn't actually save any characters when `first` and `last` differ by one. However, it is usually more important for the encoding to be uniform than to be pretty. (As a general rule, uniform and consistent encoding of data allows its processing down the line to also be uniform.)

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 that it does not have a silly redundant comma hanging at the end.

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

## Prominently featured

```
def prominences(height):
```

The one-dimensional silhouette of an island is given as a `height` list of raw **elevation** values at each position, measured from the baseline at the sea level. This rocky and volcanic island is guaranteed to not contain any **plateaus**, consecutive positions of equal elevation. Positions outside the `height` list are assumed to lie at sea level.

This function should return the list of **peaks** of this island, that is, local positions whose immediate left and right neighbours lie at a lower elevation. (In the examples below, peaks have been highlighted in green.) Each peak is given as a tuple (`position`, `elevation`, `prominence`). The `position` and `elevation` of each peak come directly from the `height` list. As explained on the Wikipedia page [“Topographical prominence”](#), the `prominence` of a peak is the minimum vertical descent necessary to get to any higher peak, even if you can choose this higher peak and the route freely to minimize this descent. The lowest valley along this best route is the **key col** of the original peak, and the higher peak reached through the key col is its **parent peak**. Since the highest peak of the island has no key col, its prominence is defined to equal its raw elevation.

This problem will [soon get tricky for two-dimensional height fields](#) with all that blasted freedom of movement to go around things. Fortunately, in this far simpler one-dimensional version, you can first construct the list of all the peaks and valleys with one linear pass through `height`, as these are the only entries that are relevant for the calculation of prominences. For each peak, proceed to the left until you reach some higher peak, keeping track the lowest elevation seen along the way. Then do the same thing again, but this time going right from that same starting peak. The higher of these two lowest valleys is the key col that lets you compute the prominence of the peak.

height	Expected result
[42]	[(0, 42, 42)]
[1, 3]	[(1, 3, 3)]
[1, 3, 2, 5, 1]	[(1, 3, 1), (3, 5, 5)]
[4, 2, 100, 99, 101, 2]	[(0, 4, 2), (2, 100, 1), (4, 101, 101)]
[3, 5, 9, 12, 4, 3, 6, 11, 2]	[(3, 12, 12), (7, 11, 8)]

## Like a kid in a candy store, except without money

```
def candy_share(candies):
```

A group of children is sitting around in a circle so that each child has some identical candies in front of them. To synchronize these children to move as a group in a logically simultaneous fashion, the teacher carries a bell that she rings to mark the start of each round. At each ring, each child who presently has at least two pieces of candy must pass one piece to the left, and pass one piece to the right. Those children who currently have zero or one pieces at that moment will simply sit out that round, hoping from candies to travel to their position.

Same as in other [chip-firing games](#) of this character, the total number of candies remains fixed throughout the process. As shown in “Mathematics Galore”, [James Tanton’s](#) delightful collection of recreational mathematical problems that this problem was adapted from, whenever the initial state contains strictly fewer candies than children, this process must eventually reach a stable terminal state where no child has more than one piece of candy. Given the initial distribution of candies, this function should return the number of rounds needed to reach this stable state.

candies	Expected result
[1, 0, 1, 1, 0, 1]	0
[3, 0, 0, 0]	1
[4, 0, 0, 0, 0, 1]	6
[5, 1, 0, 0, 0, 0, 0, 1, 0]	10
[0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	21

When the initial state contains more candies than children, simple application of the pigeonhole principle shows why this process can never terminate. However, interesting chaos breaks loose in the borderline case with exactly as many candies as children! Some states such as [0, 3, 0] reach a stable state while others such as [2, 1, 0] are destined to race forever around some **limit cycle** of states that keep passing the same candy around the circle. It is still an open problem in computer science to classify the starting states significantly faster than merely mechanistically iterating this process. (Falling into a limit cycle can be detected efficiently with the famous [tortoise and hare algorithm](#), as seen in the “Bulgarian cycle” problem later in this collection.)



# Dibs to dubs

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

Some of us ascribe deep significance to numerical coincidences, so that consecutive repeated digits or other low description length patterns, such as a digital clock blinking 11:11, seem special and personally meaningful to such people. They will then find numbers with repeated digits to be more valuable than the ordinary numbers that have no obvious patterns. For example, getting inside a taxicab flashing an exciting number such as 1234 or 6969 would be far more instatokkable than a taxicab adorned with some more pedestrian number such as 1729.

Assume that some such person assign a meaningfulness score to every positive integer so that every maximal block of  $k$  consecutive digits with  $k > 1$  scores  $10^{(k-2)}$  points for that block. A block of two digits thus scores one point, three digits score ten points, four digits score a hundred points, and so on. However, if only to make this more interesting, a special rule is in effect saying that whenever a block of digits lies at the lowest end of the number, that block scores double the points it would score in any other position. This function should compute the meaningfulness score of the given positive integer  $n$  as the sum of its individual block scores.

n	Expected result
43333	200
2223	10
77777777	20000000
3888882277777731	11001
2111111747111117777700	12002
9999997777774444488872222	21210
1234**5678	15418

## Nearest smaller element

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

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

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

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

(Most truths about computing don't actually assume a computer as the underlying media, but continue to be true even if you move out of the chassis of the part of the reality that you consider a "computer" in your mental model of reality.)

## Interesting, intersecting

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

An **axis-aligned** 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** by having at least one point in common, even if that one point is the shared corner point of two squares placed kitty corner. (The area of the common intersection of two squares can therefore be zero, if the intersection consists of parts of the one-dimensional edges.) This function **should not contain any loops or list comprehensions of any kind**, but should compute the result using only integer comparisons and conditional statements.

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

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

# So shall you sow

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

The African board game of [Oware](#) is one of the most popular [Mancala variations](#). After appreciating the simple structure of **sowing games** that allow us to toy with **emergent complexity** with minimal equipment of holes and stones that are amply available in all places on this planet that the human hand would ever voluntarily set down its foot in, we will first display hearty mathematical gusto by generalizing the game mechanics for arbitrary number of  $k$  houses. The complete **minimax** function to find the best move in the given situation will unfortunately have to wait until CCPS 721 *Artificial Intelligence*. However, we already have the means to implement a small but essential part of this algorithm; the **move executor** to play out the given move in the given situation.

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

After sowing, capturing commences from the house that the last seed was sown into. Capturing continues backwards as long as the current house is on the opponent's side and contains two or three seed. For simplicity, we ignore the edge situations from the **grand slam rule** and the gentlemanly meta-rule to always leave at least one seed for the opponent to move.

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

## That's enough of you!

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

Given a list of `items`, some of which may be duplicated, this function should 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 the first `k` are discarded.

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

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

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

# Brussel's choice

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

This problem is adapted from another jovial video "[The Brussel's Choice](#)" of [Numberphile](#), a site so British that you just know there must be a Trevor and a Colin somewhere. You should watch its first five minutes to get an idea of what is going on, even if the mathematical properties of such numbers is not necessary for you to complete the coding. (The nice thing about not just computing but all machine is that *how* they work from the point of view of an outsider does not require your conscious knowledge of *why* they work on the inside.) This function should compute the list of all numbers that the positive integer  $n$  can be converted to by treating it as a string and replacing some substring  $m$  of its digits with the new substring of either  $2*m$  or  $m/2$ , the latter substitution allowed only when  $m$  is even so that dividing it by two produces an integer.

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

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

## Count consecutive summers

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

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

The number of ways that a positive integer  $n$  can be represented as a sum of consecutive integers is called its [politeness](#), and can also be computed by tallying up the number of odd divisors of that number. However, note that the linked Wikipedia definition includes only nontrivial sums that consist of at least two components, so according to that definition, the politeness of 42 would be 3, not 4, 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. Perhaps these powers being the fundamental building blocks of all numbers in **binary** representation also made them believe their own hype and balloon up to be too big for their britches. (Fame tends to do that even to the most level-headed of us.) As an exercise in combinatorics, how would you concisely characterize the opposite extreme of “most polite” numbers that can be represented as sums of consecutive integers in more ways than any number less than them?

## McCulloch's second machine

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

[Esoteric programming languages](#) should be appreciated as works of conceptual performance art. These tar pits are not designed to serve as practical programming tools, but to demonstrate in a tongue-in-cheek manner how some deceptively simple mechanism unexpectedly grants its wielder the powers of **universal computation**.

Introduced by [Raymond Smullyan](#) in one of his clandestine brain teasers on logic and computability, [McCulloch's second machine](#) is a **string rewriting** system that receives a string of `digits` between one and nine. The rewrite rule applied to the rest of the `digits` (denoted below by `X`) depends on the first digit. Notice how rule for the leading digit 2 is applied to `X` itself, whereas rules for the leading digits 3 to 5 are applied to `Y=mcclulloch(X)`. This function should also return `None` whenever either `X` or `Y` is `None`.

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

[illegible]



## That's enough for you!

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

Find and return the first element of the given list of `items` that is preceded by at least `k` smaller elements in the list. These required `k` smaller elements can be positioned anywhere before the current element, not necessarily consecutively immediately before that element. If no element satisfying this requirement exists in the list, this function should return `None`.

Since the only operation performed for the individual `items` is their order comparison, and especially no arithmetic occurs at any point during execution, this function should work for lists of any types of elements, as long as those elements are pairwise comparable with each other.

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

## Crab bucket list

```
def eliminate_neighbours(items):
```

Given a list of integer `items` guaranteed to be some **permutation** of positive integers from 1 to `n` where `n` is the length of the list, keep performing the following step until the largest number in the original list gets eliminated; Find the smallest number still in the list, and remove from this list both that smallest number and the larger one of its current immediate left and right neighbours. (At the edges, you have no choice which neighbour you remove.) Return the number of steps that were needed to remove the largest element `n` from the list.

For example, given the list `[5, 2, 1, 4, 6, 3]`, start by removing the element 1 and its current larger neighbour 4, resulting in the list `[5, 2, 6, 3]`. The next step will then remove the element 2 and its larger neighbour 6, thus reaching the goal in two steps.

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

Removing an element from the middle of the list is expensive, and will surely form the bottleneck in the straightforward solution of this problem. We can just shake our heads to such mediocrity. However, since the `items` are known to be the integers 1 to `n`, it suffices to merely **simulate** the effect of these expensive removals without actually performing them! Define two auxiliary lists `left` and `right` to keep track of the current **left neighbour** and the current **right neighbour** of each element. These two lists can be easily initialized with a single loop through the positions of the original `items`. To simulate the removal of any item `i` in constant time, make its current left and right neighbours `left[i]` and `right[i]` figuratively “join hands” with symmetric assignments `right[left[i]]=right[i]` and `left[right[i]]=left[i]`, as in “Joe, meet Moe; Moe, meet Joe”. This noble law of hatchet, axe and saw keeps the elimination times equal for all trees, regardless of their original position and standing in the forest. (No tree, no problem; all will ultimately be equal in the mulch from the equal bite of the wood chipper.)

# 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, the digits '222274444499966' would be read out loud as “four twos, one seven, five fours, three nines, two sixes”, combined to produce the result '4217543926'.

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

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

# Bishops on a binge

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

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

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

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

## Dem's some mighty tall words, pardner

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

The **length** of a word is easy enough to define by tallying up its characters. Taking the road less traveled, we define the **height** of the given word with a **recursive** rule for the height of the given word to follow from the heights of two words whose concatenation it is.

First, any character string that is not one of the actual words automatically has zero height. Second, an actual word that cannot be broken into a concatenation of two nonempty actual words has the height of one. Otherwise, the height of an actual word equals one plus the **larger** of the heights of the two actual words whose combined concatenation it can be expressed as. To make these heights unambiguous for words that can be split into two non-empty subwords in several ways, this splitting is always done the best way that produces the tallest final height.

Since the list of words is sorted, you can use **binary search** (already available as the function `bisect_left` in the `bisect` module) to quickly determine whether some subword is an actual word. Be mindful of the return value of that function whenever the parameter string is not an actual word, and also the edge case of subwords that start with two or more copies of the letter `z`.

word	Expected result (using the wordlist <code>words_sorted.txt</code> )
'hxlllo'	0
'chukker'	1
'asteroid'	7
'pedicured'	2
'enterprise'	6
'antidisestablishmentarianism'	11
'noncharacteristically'	13

In the giant wordlist `words_sorted.txt` every individual letter seem to be a word of its own, which makes these word heights much larger than we expect them to be. Narrowing the dictionary to cover only everyday words would make most of these word heights topple. Finally, words such as 'mankind' steadfastly thumb their noses at humanity by splitting only into nonsense such as 'mank' and 'ind'. [The meaning of those words is a mystery, and that's why so is mankind...](#)

# Up for the count

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

The [Champernowne word](#) 1234567891011121314151617181920212223..., also known as the **counting series**, is an infinitely long string of digits made up of all positive integers written out in ascending order without any **separators**. This function should return the digit at position  $n$  of the Champernowne word. Position counting again starts from zero for us budding computer scientists.

Of course, the automated tester will throw at your function values of  $n$  huge enough that those who construct the Champernowne word as an explicit string will run out of both time and space long before receiving the answer. Instead, note how the fractal structure of this infinite sequence starts with 9 single-digit numbers, followed by 90 two-digit numbers, followed by 900 three-digit numbers, and so on. This observation gifts you a pair of [seven league boots](#) that allow their wearer skip prefixes of this series in exponential leaps and bounds, instead of having to crawl their way to the desired position one step at the time with the rest of us. Once you reach the block of numbers that contains the position  $n$ , the digit waiting there is best determined with integer arithmetic, perhaps aided by a conversion to a string for easy access to the digit in a particular position.

n	Expected result
0	1
10	0
100	5
10000	7
$10^{**}100$	6

This favourite problem of many past students segues into a more difficult but also more awesome side quest problem. Define “shy” integers to appear inside the Champernowne word for the first time only at their first “official appearance”, whereas “eager” integers make earlier appearances in the digit sequence as Frankennumbers stitched together from pieces of lesser numbers. Can you think up the rule that establishes at a glance that the eight-digit integer 92021222 is *muy* eager, whereas its shy successor 92021223 does not entertain any public visitors before her *quinceañera*?

# 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 'aeiouAEIOU') and reversing their order, while keeping all other characters exactly as they were in their original positions. Furthermore, the capitalization of each position must remain as it was in the original `text`. For example, reversing the vowels of 'Ilkka' should produce 'Alkki' instead of 'alkkI'. Results of applying this operation to perfectly ordinary English sentences often comically resemble pig latin imitations of other language families.

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

text	Expected result
'Revorse the vewels'	'Reverse the vowels'
'Bengt Hilgursson'	'Bongt Hulgirssen'
'Why do you laugh? I chose the death.'	'Why da yee leogh? I chusa thu dooth.'
'These are the people you protect with your pain!'	'Thisa uro thi peoplu yoe protect weth year peen!'
"Who's the leader of the club that's made for you and me? T-R-I-C-K-Y M-O-U-S-E! Tricky Mouse! TRICKY MOUSE! Tricky Mouse! TRICKY MOUSE! Forever let us hold our Hammers high! High! High! High!"	"Whi's thi liider af thu clob thot's mude fer yeo end mu? T-R-O-C-K-Y M-I-E-S-U! Trocky Miesu! TROCKY MIESU! Trocky Miesu! TROCKY MIESA! Furovor let as hald uer Hommers hagh! Hegh! Hegh! Hogh!"

# Everybody on the floor, do the Scrooge Shuffle

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

Some positions of the integer line  $\mathbb{Z}$  initially contain a pile of gold doubloons, listed in `coins` starting from the origin position of zero. All other positions on the set of integers  $\mathbb{Z}$ , both positive and negative, are initially empty. Any position that contains at least `left+right` doubloons is **unstable**. As long as some position `i` is still unstable, exactly `left` doubloons from that position `i` spill into its predecessor position `i-1`, and exactly `right` doubloons spill into its successor position `i+1`. Rinse and repeat until every position has become stable.

The unique terminal state of this back-and-forth dance will necessarily be reached after a finite number of steps, independent of the order of processing the unstable positions in the interim. (Discrete math enthusiasts may want to prove this; first show that this process is guaranteed to terminate, then show that the resulting end state is unique.) This function should return this terminal state as tuple `(start, coins)` where `start` is the leftmost position that contains at least one doubloon. The second component lists the `coins` that ended in each position, this list spanning the positions from `start` up to the last non-empty position.

Whenever some pile contains `k*(left+right)` doubloons, you can speed things up by scooping `k*left` coins into the predecessor and `k*right` doubloons into the successor pile as a single move. The first difficulty of this problem is handling the negative positions during computation, the second being quickly finding some unstable position to process. (A dictionary can easily keep track of the doubloon counts at arbitrary positions, even those with a negative index.)

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



# Rational lines of action

```
def calkin_wilf(n)
```

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

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

n	Expected result (as Fraction)
10	3/5
10000	11/39
100000	127/713

Once you reach the position  $n//2+1$ , the queue already contains the result you need, so you can save a good chunk of memory by not actually pushing in any new values. The linked Wikipedia page and other sources also provide shortcuts to jump into the given position faster than sloughing your way there the hard way one element at a time.

# Verbos regulares

```
def conjugate_regular(verb, subject, tense):
```

Conjugating verbs *en español* is [significantly more complex](#) than in English where the same information is conveyed with an explicit subject and additional words around that *pinche* infinitive. Frustrated students can only exclaim “¡Dios mío!” or “¡No me gusta!” and raise their eyes and hands dramatically to the heavens. After obviously watching one too many episodes of *Narcos* to make the lockdown time pass by, this author decided to spend some of this trying time pretending to prepare himself for the coming borderless cyberpunk utopia that we are slouching towards together, and *tomo el toro por los cuernos* by finally learning some basic Spanish, one of the most meme-worthy languages devised for both men and their machines.

Fortunately, regular Spanish verbs whose infinitive ends with either [-ar](#), [-er](#) or [-ir](#) follow systematic rules that make a good exercise in Python string processing and multiway logic. Given the infinitive of the verb, the subject as one of *yo, tú, él, ella, usted, nosotros, vosotros, ellos, ellas* and *ustedes*, and the tense as one of four simple indicative tenses *presente, pretérito, imperfecto* and *futuro* to keep this problem manageable, this function should return that conjugate of the verb. Since it is no longer the year 1985 when every ASCII character was stored in a single byte, enough on our daily uphill journey both ways in rain or shine (and that is how we liked it without complaining), today's Unicode strings of Python 3 handle [accented characters](#) no different from any other characters. Therefore, these accents should also be correct in the returned answer.

verb	subject	tense	Expected result
'añadir'	'ellos'	'presente'	'añaden'
'cantar'	'yo'	'presente'	'canto'
'bailar'	'tú'	'pretérito'	'bailaste'
'decidir'	'usted'	'futuro'	'decidirá'
'meter'	'ustedes'	'imperfecto'	'metían'
'romper'	'ella'	'pretérito'	'rompió'
'escribir'	'nosotros'	'imperfecto'	'escribíamos'

(Este problema es básicamente un gran árbol de decisiones. Sin embargo, las regularidades en las reglas de conjugación del español te permiten combinar casos equivalentes para manejarlos con la misma lógica.)

# Hippity hoppity, abolish loopity

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

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

Given two frogs `frog1` and `frog2` that initially stand on different squares, return the time when both frogs hop into the same square. If these two frogs never hop into the same square at the same time, this function should return `None`.

**This function definitely should not contain any loops whatsoever.** The result should be calculated using conditional statements and integer arithmetic. To get cracking, first solve a simpler version of this problem of one-dimensional frogs restricted to hop along the one-dimensional line of integers. Once you get that function working correctly, including all its possible edge cases such as both frogs jumping at the exact same speed, or one or both frogs staying put in the same place with zero speed, use that method as a subroutine to solve for  $t$  separately for the  $x$ - and  $y$ -dimensions in the original problem. Combine these two one-dimensional answers for 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

# Where no one can hear you bounce

```
def reach_corner(x, y, n, m, aliens):
```

A lone chess bishop finds himself standing on the square  $(x, y)$  of a curious  $n$ -by- $m$  chessboard floating in the outer space, covered with completely frictionless magic ice. (True story. Yeah, and again, zero-based indexing.) This board is not necessarily square, but can be any rectangle of integer dimensions. The board is surrounded from all sides with bouncy walls similar to an air hockey table. Some of the squares on the board are `aliens` that use both big and little mouths to make this game to be over for any man who enters.

No knife tricks will help the bishop here, but he must instead reach any one of the four possible exits at the four corners of the board. Similar to patiently watching the screensaver of an old DVD player, the bishop must reach any one of these corners in a single move. Bishop can initially propel himself to any of the four diagonal directions, but after this initial impulse to get him going, the bishop can no longer control his movements on the frictionless ice, but will keep sliding towards whatever end the present direction vector and bouncing from the walls take him. It is even possible for the bishop to forever repeat the same cycle of squares. (Your function, unknowingly doing this dance on the far more complex surface of computer state configurations, should not follow suit, but nuke the whole computation from the orbit, just to be sure.)

This function should determine whether there exists at least one starting direction that leads the bishop to the safety of any one of the four corners, avoiding all the `aliens` along the way.

x	y	n	m	aliens	Expected result
0	2	5	5	[ ]	False
4	4	9	9	[ (0, 0), (8, 8), (8, 0), (8, 8) ]	False
1	1	1000	2	[ (0, 0), (0, 1), (999, 0) ]	True
1	1	1000	2	[ (0, 0), (0, 1), (999, 1) ]	False
3	2	4	4	[ (1, 2), (0, 1) ]	False
3	2	5	4	[ (2, 2), (1, 4) ]	True

## 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 that was obviously not devised by a computer scientist but some normal person, positions start from 1, not 0. Furthermore, the letter  $i$  denotes the position here, since the letter  $n$  already means something else. For example, the sequence of "[octagonal numbers](#)" that springs forth from  $s = 8$  starts with 1, 8, 21, 40, 65, 96, 133, 176...

Given the number of sides  $s$  and an arbitrary integer  $n$ , this function should return the  $s$ -gonal integer closest to  $n$ . If  $n$  falls exactly halfway between two  $s$ -gonal numbers, return the smaller one.

n	s	Expected result
5	3	6
27	4	25
450	9	474
10**10	42	9999861561
10**100	91	100000000000000000000000000000000000000000000000 00000041633275351832947889775579470433400300 3544212420356

As you can see from the last row of the previous table, this function must be efficient even for gargantuan values of  $n$ . The simplest way to get there is to harness the power of **repeated halving** to pull your wagon with a clever application of **binary search**. Start with two integers  $a$  and  $b$  wide enough that they satisfy  $a \leq i \leq b$  for the currently unknown position  $i$  that the nearest polygonal number is stored in. (Just initialize these as  $a=1$  and  $b=2$ , and keep squaring  $b$  until the  $s$ -gonal number in that position gets too big. It's not like these initial bounds need to be super 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 final comparison tells you the correct answer.

## Don't worry, we will fix it in the post

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

When arithmetic expressions are given in the familiar infix notation  $2 + 3 * 4$ , parentheses force the different evaluation order to be different from the usual **PEMDAS** order determined by *precedence* and *associativity*. Writing arithmetic expressions in *postfix* notation (also known as [Reverse Polish Notation](#), for all you “simple Poles on a complex plane”) may look strange to us humans accustomed to the conventional *infix* notation. However, postfix notation is easier for machines, since it allows *any* evaluation order to be expressed unambiguously without 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 `'/'` to denote the four arithmetic operators. To evaluate a postfix expression using a simple linear loop, use an ordinary list as an initially empty **stack**. Loop through the `items` one by one, from left to right. Whenever the current item is an integer, just **append** it to the end of the list. Whenever the current item is one of the four arithmetic operations, **pop** two items from the end of the list to perform that operation on, and **append** the result back to the list. Assuming that `items` is a legal postfix expression, which is guaranteed in this problem so that you don't need to perform any error detection or recovery, once all items have been processed, the lone number still 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 from dividing by zero, this problem comes with an artificial (yet mathematically [perfectly sound](#)) rule that division by zero gives a zero result, 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, '-', '/', 42, '+']	$3 / (3 - 3) + 42$	42
[7, 3, '/']	$7 / 3$	2
[1, 2, 3, 4, 5, 6, '*', '*', '*', '*', '*']	$1 * 2 * 3 * 4 * 5 * 6$	720

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

# Fractran interpreter

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

The late great [John Conway](#) is best known for [The Game of Life](#), not to be confused with the family board game with the same name. That one achievement eclipsed basically all other [wacky and original creations](#) of Conway, so we ought to give his less appreciated creations an occasional spin in the paparazzi lights and the red carpet fame.

This function works as an **interpreter** for the simple [esoteric programming language](#) called [FRACTRAN](#), a pun combining “fraction” with the [FORTRAN programming language](#). (Things used to be all in uppercase back in when real scientists wore horn-rimmed glasses came to work driving cars were equipped with tail fins, with occasional dollar signs interspersed as visual separators to keep the perfumed princes of the military-industrial complex happy.) A program written in such very mysterious form consists 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 required to use the `Fraction` data type in the `fractions` module to perform these computations exactly.

Given a positive integer  $n$  as its start state, the next state is the product  $n*f$  for the first fraction listed in `prog` for which  $n*f$  is an exact integer. That integer then becomes the new state for the next round. Once  $n*f$  is not an integer for any of the fractions  $f$  listed in `prog`, the execution terminates. Your function should compute the sequence of integers produced by the given FRACTRAN program, with a forced termina... (“Oi, mate, Ol’ Connie was British!”), sorry, forced **halting** taking place after `giveup` steps, should the execution have not halted by itself 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 in previous row)	20	[9, 495, 435, 1155, 1015, 2695, 3185, 595, 546, 102, 38, 23, 95, 385, 455, 85, 78, 66, 58, 154, 182]

# Bulgarian cycle

```
def bulgarian_cycle(piles):
```

When the total number of pebbles in the piles is the  $k$ :th **triangular number**, iteration in the earlier **Bulgarian solitaire** problem is guaranteed to converge to the **steady state** whose pile sizes are exactly the positive integers 1 to  $k$  in some order. When started with some other number of pebbles, the iteration can never reach a steady state. Instead, it must eventually re-enter some state that it has visited earlier, and from there be doomed to eternally repeat the same **limit cycle** of states. This function should compute the length of this limit cycle reached from the given starting `piles`.

Note that we consider two states to be equal if they contain the same pile sizes in some order, not necessarily the same order. Also, since the starting state is not necessarily part of the limit cycle, we need to somehow recognize that the current state has already been visited in the iteration. No-one can be faulted for assuming that it must be necessary to store **all** states seen along the way into a set, but surprisingly, this is not so! The [tortoise and hare algorithm](#) credited to Robert Floyd for the problem of general [cycle detection](#) ingeniously [trades time for plenty of space](#). (Usually we make such tradeoffs to the opposite direction to optimize for running time, but space is at premium this time.) By iterating the sequence longer than is strictly necessary, the algorithm needs to remember only the current positions of tortoise and hare, independent of the limit cycle and the path there!

Both tortoise and hare begin at the same state of `piles`. For every step taken by the tortoise, the hare takes two steps. In this version of the famous morality tale, the hare has learned his bitter lesson and does not stop to nap or showboat along the way. Akin to two long distance runners racing towards a stadium located some unknown distance ahead to then run around, the tortoise will eventually reach the stadium, and will be inevitably overtaken by the hare already at the stadium. The position where these two animals meet again must be part of the limit cycle. Then, allow the hare take his victory nap to mark the position while the tortoise goes around the stadium once more, this time keeping count of its steps to tally the length of this limit cycle.

piles	Expected result
[1, 3, 2]	1
[5, 7]	5
[4, 4, 3, 3, 2, 1]	6
[17, 99, 42]	18
[n*n for n in range(1, 51)]	293



# Permutation cycles

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

In this problem, a **permutation** is a list of  $n$  elements that contains every integer from 0 to  $n - 1$  exactly once. For example, `[5, 2, 0, 1, 4, 3]` is one of the  $6! = 720$  permutations for  $n = 6$ . Each permutation can be considered a **bijective function** that **maps** each position 0 to  $n - 1$  into some unique position so that no two positions map to the same position. For example, the previous permutation maps the position 0 into 5, position 1 into 2, 2 into 0, and so on.

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

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

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

# Whoever must play, cannot play

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

Two players play a game of "[Subtract a square](#)" starting from a positive integer. On his turn to play, each player must subtract some **square number** (1, 4, 9, 16, 25, 36, 49, ...) from the current number so that the result does not become negative. Under the **normal play convention** of these games, the player who moves to zero wins the match by making his opponent unable to move.

This and many similar [combinatorial games](#) can be solved with recursive equations. This game is an example of an [impartial game](#) since identical moves are available to the player whose turn it is to move, as opposed to "partisan games" such as chess where each player may legally only move pieces of one colour. The base case of recursion is that the terminal state zero where no moves are possible is **cold** ("losing"). After that, a state is **hot** (or "winning", perhaps said in the exuberant voice of a beloved celebrity) if at least one move available in that state leads to a cold state. If every move available in the given state leads to a hot state, that state is cold for the unlucky player whose turn it is to play from that state.

Since the heat value of each state  $n$  is determined by the heat values of states lower than  $n$ , we might as well combine all these heat computations of states into a single **batch job**. This function should return a list of results for the given `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 heat value computation in each state can simply read from this list the already computed heat values of lower-numbered states.

queries	Expected result
[7, 12, 17, 24]	[False, False, False, True]
[2, 3, 6, 10, 14, 20, 29, 39, 57, 83, 111, 149, 220, 304, 455, 681]	[False, True, True, False, True, False, True, False, False, True, True, True, True, True, True]
[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, False, False, True, True, True, True, True, True, True, True, True, True]

(In the **misère** version of the game that has otherwise identical rules but where you win by losing the original game by getting to zero, these definitions would be adjusted accordingly.)

## ztalloc ecneueques

```
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 it cannot be known beforehand how many steps are needed to reach the goal from the given starting value, and in fact, no finite upper bounds are even known. (The existence of *any* such upper bound, regardless of how loose and pessimistic, would immediately affirm the Collatz conjecture.) The answer was given as the list of integers that the sequence visits before terminating at its goal. However, this sequence can also be viewed 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 letters 'u' and 'd', respectively. For example, starting from  $n=12$ , the sequence `[12, 6, 3, 10, 5, 16, 8, 4, 2, 1]` would have the step shape `'ddududddd'`.

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

shape	Expected result
'ududududdddduddd'	15
'dudududdduddd'	14
'uduuuuddd'	None
'd'	2
'uuududddduduuuuuuddddd'	None
'duuuddddddd'	None

# The solution solution

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

Despite the chemical inspiration behind this problem, its solution is not an overly alkaline pH balance. This problem was inspired by yet another [Numberphile](#) video, this time "[The Centrifuge Problem](#)" that you can first watch to get an idea what this problem is about, especially since the lovely Dr. Krieger surely is a far better lecturer than this grumpy old instructor in most ways that matter. Those whose eyes can read faster than their ears can hear can alternatively consult the blog post "[The Balanced Centrifuge Problem](#)" by Matt Baker.

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

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

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

# Reverse ascending sublists

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

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

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

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

# Brangelin-o-matic for the people

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

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

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

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

These rules do not always produce the best possible result. For example, 'ross' and 'rachel' combine into 'rachel' instead of the more informative 'rochel'. The reader can later think up more advanced rules that cover a wider variety of name combinations and special cases. (Some truly advanced set of rules, perhaps trained with **deep learning** techniques to also implicitly recognize the **semantic** content might combine 'donald' and 'hillary' into either 'dollary' or 'dillary', depending on the intended tone.)

# Line with most points

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

A point on the two-dimensional integer grid  $\mathbb{Z}^2$  is given as a two-tuple of  $x$ - and  $y$ -coordinates, for example  $(2, 5)$  or  $(10, 1)$ . [As originally postulated by Euclid](#), any two distinct points on the plane define **exactly one line** that goes through both points. (In differently shaped spaces made of finger-quotes “points” and “lines”, [different rules and their consequences apply](#).) Of course, this unique line, being shamelessly infinite to both directions, will also pass through an infinite number of other points on that same plane... or will it? (Can you prove this?)

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

To get started, consult the example program [geometry.py](#) for the **cross product** function that can be used to quickly determine whether three given points on the plane are **collinear**. Using this operation as a subroutine, the rest of the algorithm can operate at a higher level of abstraction.

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

This problem could always be inefficiently **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. (As long as we are not creating a **moral hazard** by externalizing the costs of laziness on others who did not consent to those costs, being lazy is a great virtue in programming, and as such ought to more fittingly be called “elegant” or “optimal”.)

# Om nom nom

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

The beloved [Cookie Monster](#) from *Sesame Street* has stumbled upon a table with `piles` of cookies, each pile a positive integer. However, the monomaniacal obsessiveness of [The Count](#) who set up this fiesta has recently escalated to a whole new level of severity. The Count insists that these cookies must be eaten in the smallest possible number of moves. Each move chooses one of the remaining pile sizes `p`, and removes `p` cookies from every pile that contains at least `p` cookies (eradicating all piles with exactly `p` cookies), and leaves all smaller piles untouched as they were.

Since The Count is also obsessed with order and hierarchies, he expects the moves to be done in decreasing order of values of `p`, as seen in the third column of the table below. This function should return the smallest number of moves that allows Cookie Monster to scarf down these cookies.

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

The typical formulation of this problem lets Cookie Monster choose any subset of remaining piles, and remove the same number of cookies from each pile in the chosen subset. Interested readers can check out the article “[On the Cookie Monster Problem](#)” about the subtle complexities of the general problem formulation. For example, contrary to intuition, various **greedy strategies** that might seem natural for the addiction-riddled brain of the Cookie Monster, such as “choose `p` to maximize the number of cookies eaten at the current move” or “choose `p` to eliminate the largest possible number of piles” (yep, think about that one), do not always produce the shortest sequence of moves.



# Autocorrect for sausage fingers

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

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

Given a word, a list of words, and a **distance** function df that tells how “far” the first key is from the second (for example,  $df('a', 's')=1$  and  $df('a', 'a')=0$ ), return the word in the list of words that have the same number of letters that minimizes the total distance, measured as the sum of the distances between the keys in same positions. If several words are equidistant from word, return the first of these words in the dictionary order.

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

More advanced autocorrect techniques use the surrounding context to suggest the best replacement, seeing that not all words are equally likely to be the intended word, given the rest of the sentence. For example, the misspelling 'urc' should probably be corrected very differently in the sentence “The gateway was an urc made of granite blocks” than in the sentence “The brave little hobbit swung his sword at the smelly urc.” Similarly, even though 'lobe' is a perfectly valid word by itself, “I lobe you!” is most likely a typo; and whoever wrote just one word 'hellonthere' probably meant to write two but missed the space bar at the bottom row... or it could even be three words, assuming that the space bar detection was somehow broken.

# Uambcsrln the wrod

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

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

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

Writing the filter to transform the given plaintext to the above scrambled form is also a good little programming exercise in Python. This leads us to a useful estimation exercise: how long would the plaintext have to be for you to write such a filter yourself instead of scrambling the plaintext by hand as you would if you needed to scramble just one word? In general, how many times do you think you need to solve some particular problem until it becomes more efficient to design, write and debug a Python script to do it? Would your answer change if it turned out that millions of people around the world also have that same problem, silently screaming for their Strong Man saviour to ride in on his mighty horse to automate this repetitive and error-prone task? Could the very reader of this sentence perhaps be The One who the ancient prophecies have merely hinted at, coming to solve all our computational problems?

# 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 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. To make this problem interesting, this substitution must be **injective**, so that no two different uppercase letters in the pattern map to the same lowercase letter.

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 both pattern characters A and D would have to be non-injectively mapped to the same letter a.

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']
'ABCDEFGHIJKLM'	243 words, first three of which are ['absorptively', 'adjunctively', 'adsorptively']

# Manhattan skyline

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

This classic problem in [computational geometry](#) (essentially, geometry using only integer arithmetic without any of those nasty transcendental functions; yes, that is an actual place and I can lead you into that paradise) is best illustrated by pictures and animations such as those on the page "[The Skyline problem](#)". All towers share the same flat ground baseline. Given a list of rectangular towers as tuples  $(s, e, h)$  where  $s$  and  $e$  are the start and end  $x$ -coordinates with  $e > s$  so that tenement is not fit only for immigrants fresh off the Flatland plane (Rodgers and Hammerstein would have surely turned this backstory into a delightful ditty) and  $h$  is the height of that tower, compute the total visible area of the towers. Be careful not to **double count** the area of partially overlapping towers.

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

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

towers	Expected result
<code>[(0, 5, 2), (1, 3, 4)]</code>	14
<code>[(3, 4, 1), (1, 6, 3), (2, 5, 2)]</code>	15
<code>[(-10, 10, 4), (-8, -6, 10), (1, 4, 5)]</code>	95
<code>[(s, 1000-s, s) for s in range(500)]</code>	249500
<code>[(s, s+(s*(s-2))%61+1, max(1,s*(s%42))) for s in range(100000)]</code>	202121725069

## 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 to be solved 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 so that their areas have at least one point in common. Two disks  $(x_1, y_1, r_1)$  and  $(x_2, y_2, r_2)$  intersect if and only if they satisfy the **Pythagorean** inequality  $(x_2 - x_1)^2 + (y_2 - y_1)^2 \leq (r_1 + r_2)^2$ . Note again how this precise formula needs only integer arithmetic whenever its arguments are integers, and how no square roots or any other irrational numbers gum up the works with their noise of decimals. (This formula also uses the operator `<=` instead of `<` to count two **kissing** disks as an intersecting pair.)

For this problem, crudely looping through all possible pairs of disks would work, but also become horrendously inefficient as the list grows larger. However, a sweep line algorithm can solve this problem not just **effectively** but also **efficiently** (a crucial but often overlooked “eff-ing” distinction) by looking at a far fewer pairs of disks. Again, sweep through the space from left to right for all relevant  $x$ -coordinate values and maintain **the set of active disks** at the moment. Each individual disk  $(x, y, r)$  enters the active set when the vertical sweep line reaches the  $x$ -coordinate  $x - r$ , and leaves the active set when the sweep line reaches  $x + r$ . When a disk enters the active set, you need to look for its intersections only in this active set.

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

# Ordinary cardinals

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

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

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

As explained in your friendly local discrete math course, the above relation is transitive and antisymmetric, and therefore an ordering relation for integers. Outside its absurdist entertainment value it is nearly useless, though. The ordinary ordering of ordinals that we tend to assume to be some sort of law of nature is more useful because it plays along nicely with useful arithmetic operations such as addition (for example that if  $a < b$ , then  $a + c < b + c$ ) and by doing so makes these operations immensely more powerful in a symbiotic relationship.

## Count divisibles in range

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

Let us take a breather by tackling a problem simple enough that its solution needs only a couple of conditional statements and some arithmetic, but not even one loop or anything even more fancy. The difficulty is coming up with the conditions that cover all possible cases of this problem exactly right, including all of the potentially tricky **edge** and **corner cases**, without being **off-by-one**. Given three integers `start`, `end` and `n` so that `start <= end`, count how many integers between `start` and `end`, inclusive, are divisible by `n`. Sure, the distinguished gentleman *could* solve this problem as a one-liner 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 with such a blunt tool will find themselves running out of both time and space! Your code should have **no loops at all**, but use only integer arithmetic and conditional statements to root out the truth. Also, be careful with various edge cases and off-by-one ticks lurking in the bushes. Note that either `start` or `end` can be negative or zero, but `n` is guaranteed to be greater than zero.

start	end	n	Expected result
7	28	4	6
-77	19	10	9
-19	-13	10	0
1	$10^{12} - 1$	5	199999999999
0	$10^{12} - 1$	5	200000000000
0	$10^{12}$	5	200000000001
$-10^{12}$	$10^{12}$	12345	162008911

## Bridge hand shape

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

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

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



# Milton Work point count

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

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

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

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

# Never the twain shall meet

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

Powers of integers rapidly blow up too large to be useful in our daily lives. Outside time and space, all alone with no human minds to watch over their journey that is paradoxically both infinite and instantaneous, these sequences probe through the space of positive integers with exponentially increasing gaps that eventually contain entire universes inside them. Fortunately, Python allows us to play around with integer powers of millions of digits. Its mechanical decisions will reliably and unerringly dissect these slumbering behemoths that our mortal minds could not begin to visualize, even as our logic tickles their bellies in a way that hopefully feels sufficiently pleasant as not to anger them.

Except when the prime factors of  $a$  and  $b$  fit nicely, the iron hand of the [Fundamental Theorem of Arithmetic](#) dictates that the integer powers  $a^{p_a}$  and  $b^{p_b}$  can never be equal for any two positive integer exponents  $p_a$  and  $p_b$ . However, in the jovial spirit of "[close enough for the government work](#)", we define such powers to "hit" if their difference  $\text{abs}(a^{p_a} - b^{p_b})$  multiplied by the `tolerance` is at most equal to the smaller of those powers. (This definition intentionally avoids division to keep it both fast and accurate for arbitrarily large integers.) For example, `tolerance=100` requires the difference to be within 1%. For the positive integers  $a$  and  $b$ , return the smallest integer exponents  $(p_1, p_2)$  that satisfy the `tolerance` requirement.

a	b	tolerance	Expected result
9	10	5	(1, 1)
2	4	100	(2, 1)
2	7	100	(73, 26)
3	6	100	(137, 84)
4	5	1000	(916, 789)
10	11	1000	(1107, 1063)
42	99	100000	(33896, 27571)

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

# Bridge hand shorthand form

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

In literature on the game of **contract bridge**, hands are often given in abbreviated form that makes their relevant aspects easier to visualize at a glance. In this abbreviated shorthand form, suits are always listed **in the exact order of spades, hearts, diamonds and clubs**, so no special symbols are needed to show which suit is which. The ranks in each suit are listed as letters from 'AKQJ' for **aces and faces**, and all **spot cards** lower than jack are written out as the same letter 'x' to indicate that its exact spot value is irrelevant for the play mechanics of that hand. These letters must be listed in descending order of ranks AKQJx. If some suit is **void**, that is, the hand contains no cards of that suit, that suit is abbreviated with a minus sign character '-'. The shorthand forms for the individual suits are separated using single spaces, with no trailing whitespace.

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

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

# Points, schmoints

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

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

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

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

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

# Bulls and cows

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

In the two-player game of "[Bulls and Cows](#)", reincarnated after the seventies [Rural Purge](#) under the more flashy and urban title "[Mastermind](#)" in colourful plastic, the first player thinks up a four-digit secret number with all digits different, such as 8723 or 9425. (For simplicity, the digit zero is not used in this problem.) The second player tries to pinpoint this secret number by repeatedly guessing some four-digit numbers. After each guess, the first player reveals how many **bulls**, the right digit in the right position, and **cows**, the right digit but in the wrong position, that particular guess contains. For example, if the secret number is 1729, the guess 5791 contains one bull (the digit 7) and two cows (the digits 9 and 1). The guess 4385, on the other hand, contains no bulls or cows, thus neatly eliminating four digits from the search space!

This function should determine which numbers could still be the secret number according to the results of the list of `guesses` completed so far. Each individual guess is given as a tuple `(guess, bulls, cows)`. This function should return the list of such consistent four-digit numbers, sorted in ascending order. Note that it is very much possible for this result list to be empty, if the results of the `guesses` are inherently contradictory. (Good functions ought to be **robust** so that they do the right thing even when given syntactically correct but semantically meaningless data.)

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

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

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

# Frequency sort

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

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

The best way to solve this problem is to “Let George do it”, or whichever way you wish to put this concept of passing the buck to a hapless underling, by assigning the role of George on the Python `sort` function. We are going to tell the `sort` function to perform this sorting according to a custom sorting criterion, as was done in the lecture example script [countries.py](#).

Start by creating a dictionary to keep track of how many times each element occurs inside the array. Then, use these counts stored in that dictionary as the **sorting key** of the actual array elements, breaking ties on the frequency by using the actual element value. (If you also then remember that the order comparison between Python tuples is **lexicographic**, you don't even have to burden yourself with the work needed to break these ties between two equally frequent values...)

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

## Calling all units, B-and-E in progress

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

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

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

n	Expected result
42	False
441	True
469097433	True
$12^{34}$	True
$12^{34} - 1$	False

A tester for this problem can be easily built on [Catalan's conjecture](#), these days actually a proven mathematical theorem that says that after the special case of the two consecutive perfect powers 8 and 9, whenever the positive integer  $n$  is a perfect power,  $n-1$  can never be a perfect power. This theorem makes it easy to generate pseudorandom test cases with known answers, both positive and negative. For example, we don't have to slog through all potential ways to express the number as an integer power to know from the get-go that  $12345^{67890}-1$  is not a perfect power. This also illustrates the common asymmetry between performing a computation to opposite directions. Given some chunky integer such as 4922235242952026704037113243122008064, but not the formula that originally produced it, it is not that easy to tell whether that number is a perfect power, or some perfect power plus minus one.

# Lunatic multiplication

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

This problem was inspired by another jovial [Numberphile](#) video "[Primes on the Moon](#)" by Neil Sloane ([as in](#)) that you should watch first to get an idea of how this wacky “lunar arithmetic” works. Formerly known as “dismal arithmetic”, addition and multiplication of natural numbers are redefined so that **adding** two digits means taking their **maximum**, whereas **multiplying** two digits means taking their **minimum**. For example,  $2 + 7 = 7 + 2 = 7$  and  $2 * 7 = 7 * 2 = 2$ . Unlike ordinary addition, there can never be a carry to the next column of digits, no matter how many individual digits are added together in that column. For numbers with several digits, addition and multiplication work exactly as you learned back in grade school, except that the shifted digit columns from lunar multiplication of the individual digits are added in the same lunatic fashion.

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

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



# Distribution of abstract bridge hand shapes

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

This is a continuation of the earlier problem to compute the shape of the 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 only look at the sorted counts of the suits, but don't care about the order of 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 far less balanced shapes 5-4-3-1 and 5-4-2-2 that one might have intuitively assumed to be less frequent. ([Understanding why randomness tends to produce variance rather than converging to complete uniformity](#) is a great aid in understanding many other counterintuitive truths about the behaviour of random processes in computer science and mathematics.)

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

# Flip of time

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

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

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

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

glasses	t	Expected result
$[(7, 0), (11, 0)]$	15	2 ( <a href="#">see here</a> )
$[(4, 0), (6, 0)]$	11	None
$[(7, 0), (4, 0), (11, 0)]$	20	3
$[(16, 0), (21, 0)]$	36	6
$[(5, 0), (8, 0), (13, 0)]$	89	7

(The generator `itertools.product([0,1], repeat=n)` produces all  $2^n$  possible  $n$ -element tuples of  $[0, 1]$  to represent the possible subsets of  $n$  individual `glasses` to be flipped.)

# 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. However, all clichés became clichés in the first place because they were so just darn good that every Tom, Dick and Harry kept using them! Let us therefore traipse around the [Zeckendorf's theorem](#), the more amazing property of these famous numbers: **every positive integer can be expressed exactly one way as a sum of distinct non-consecutive Fibonacci numbers**.

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

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

Note how this greedy construction guarantees that the chosen Fibonacci numbers are distinct and cannot contain two consecutive Fibonacci numbers  $F_i$  and  $F_{i+1}$ . Otherwise their sum  $F_{i+2}$  would also have fit inside  $n$  and been used instead of  $F_{i+1}$  and thus would have prevented the use of  $F_{i+1}$  with the same argument involving the next higher Fibonacci number  $F_{i+3}$  ...

# Wythoff array

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

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

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

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

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

## Rooks with friends

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

Those dastardly rooks have again gone on a rampage on a generalized  $n$ -by- $n$  chessboard, just like in the earlier problem of counting how many squares kept their occupants from being pulverized into dust under these lumbering juggernauts. Each individual rook is again represented as a two-tuple `(row, column)` of the coordinates of the square it stands on. However, this version of the problem introduces a pinch of old-fashioned ethnic nepotism to the mix so that some of these rooks are your **friends** (same colour as you) while the others are your **enemies** (the opposite colour from you). Trapped in this hellscape, you can only scurry along and hope to survive until the dust settles.

Friendly rooks protect the chess squares by standing between them and any enemy rooks that want to invade those squares. An enemy rook can attack only those squares in the same row or column that do not enjoy the protection of such a friendly rook standing between them. Given the board size  $n$  and the lists of `friends` and `enemies` (these two lists are guaranteed disjoint so that no rooks act as either fence-sitters or turncoats), count how many empty squares on the board are safe from the enemy rooks.

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

# Possible words in Hangman

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

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

Each `pattern` can only match words of the exact same length. In the positions where the `pattern` contains some letter, the word must contain that very same letter. In the positions where the `pattern` contains an asterisk, the word character in that position can be anything except any of the letters that already occur inside the pattern. In an actual game of Hangman, all occurrences of such a letter would have already been revealed in the earlier round with letter as the current guess.

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

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

## All branches lead to Rome

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

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

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

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

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

# Be there or be square

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

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

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

To identify four points that constitute a square, note how every square has **bottom left corner** point  $(x, y)$  and **direction vector**  $(dx, dy)$  towards its upper left corner point that satisfies the constraints  $dx \geq 0$  and  $dy > 0$ , so that the three points  $(x+dx, y+dy)$ ,  $(x+dy, y-dx)$  and  $(x+dx+dy, y-dx+dy)$  for top left, bottom right and top right corners of that square, respectively, are also included in points. You can therefore iterate through all possibilities for the bottom left point  $(x, y)$  and the direction vector  $(dx, dy)$  and be guaranteed to find all squares in the grid. But do try to be economical in these loops so that your function will sweep up these squares more swiftly than even Tom Swift himself.

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



## Split within perimeter bound

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

Rectangles are represented as tuples  $(a, b)$  of two positive integer side lengths. A rectangle can be cut into 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, one way among all possible ways to cut the rectangle  $(5, 8)$  would be to cut it into  $(2, 8)$  and  $(3, 8)$  in the first dimension. Another way would be to cut it into  $(5, 4)$  and  $(5, 4)$  in the second dimension. These smaller pieces can then be further cut into smaller pieces, as long as the length of the side being cut is at least two. Also, at the risk of disappointing any followers of Edward de Bono reading this, you are not allowed to cut multiple pieces in one motion of the blade by stacking those pieces on top of each other. (But please do keep telling us once again that delightful story about those square tires!)

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

This problem is best solved with recursion. If the perimeter of the piece is within the limit  $p$ , return zero. Otherwise, loop through the 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  $m_1$  and  $m_2$  moves, respectively. Return the smallest value for  $1+m_1+m_2$  that you find. Since this branching recursion will visit its subproblems an exponential number of times, you might want to sprinkle on some `@lru_cache` memoization to downgrade that exponential tangle into a quadratic one.

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

# Sum of distinct cubes

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

Positive integers can sometimes be broken down into sums of **distinct** cubes of **positive** integers, sometimes in multiple different ways. This function should find and return the list of distinct cubes whose sum equals the given positive integer  $n$ . Should  $n$  allow several breakdowns into sums of distinct cubes, this function must return the **lexicographically highest** solution that starts with the largest possible first number  $a$  where it is followed by the lexicographically highest representation of the rest of the number  $n - a^3$ . For example, when called with  $n=1072$ , this function should return `[10, 4, 2]` instead of `[9, 7]`. Whenever it is impossible to break the parameter  $n$  into a sum of distinct cubes, return `None`.

Unlike in the earlier, much simpler question of expressing  $n$  as a sum of exactly two squares, the result can now contain any number of distinct terms. This problem, like almost all such problems that cause an exponential blowup of two-way “take it or leave it” decisions, is usually best solved with recursion that examines both branches of each such decision and returns the one that gives a better result. The base case is when  $n$  itself is a cube. Otherwise, loop down through all possible values of the first element  $a$  in the result list, and for each such  $a$ , break down the remaining number  $n - a^3$  into a sum of distinct cubes using only integers that are smaller than your current  $a$ . Again, to let your function efficiently gleam these cubes even for large  $n$ , it might be a good idea to prepare something that allows you to quickly determine whether the given integer is a cube, and whenever it is not, to find the largest integer whose cube is smaller.

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

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

# Fractional fit

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

When interpreted as integer fractions `Fraction(a,b)`, two-tuples  $(a,b)$  for which  $0 \leq a < b$  all fall inside the **unit interval**. Given the list `fs` of available fractions, find the length of the longest possible list that can be constructed from those fractions under the following series of constraints:

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

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

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

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

## Followed by its square

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

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

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

it	Expected result
<code>iter([3, 4, 6, 8, 11, 13, 18])</code>	<code>[]</code>
<code>iter([2, 3, 4, 5, 8, 9, 11, 16])</code>	<code>[3, 4]</code>
<code>iter(range(1, 10**6))</code>	(a list that consists of exactly 998 elements, the first five of which are <code>[2, 3, 4, 5, 6]</code> and the last five are <code>[995, 996, 997, 998, 999]</code> )
<code>iter([x*x for x in range(1, 1000)])</code>	<code>[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]</code>
<code>iter([x**10 for x in range(1, 100)])</code>	<code>[1024, 59049, 1048576, 9765625, 60466176, 282475249, 1073741824, 3486784401]</code>

## Count maximal layers

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

A point  $(x_1, y_1)$  on the two-dimensional plane **dominates** another point  $(x_2, y_2)$  if both inequalities  $x_1 > x_2$  and  $y_1 > y_2$  hold. A point inside the given list of `points` is **maximal** if it is not dominated by any other point in the list. Note that unlike in boring old one dimension, some lists of `points` may even contain maximal points that do not dominate *any* other point! 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  $(9, 2)$  form the maximal layer for  $[(1, 5), (8, 1), (3, 10), (2, 1), (9, 2)]$ .

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

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

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

## Maximum checkers capture

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

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

Some variants of checkers allow **leaping kings** that can capture pieces across arbitrary distances. For simplicity, in this problem kings are more subdued so that they capture a piece only one step away to the four diagonal directions (the list  $[(-1,1), (1,1), (1,-1), (-1,-1)]$  might come handy) assuming that the square behind the opponent piece in that diagonal direction is vacant. Your king can then capture that piece by jumping over it into the vacant square, **immediately removing that captured piece from the board**. However, unlike in chess where each move can capture at most one enemy piece, the chain of captures continues from the new square, potentially capturing all the pieces in one dramatic swoop like in movies.

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

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

# Collatz distance

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

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

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

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

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

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

# Van Eck sequence

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

The [Van Eck sequence](#) problem was making wide rounds at the time while your author was updating these lab specifications. He had to read the problem only halfway through to already know that it was going to be one of the best problems ever! The first term in the first position  $i = 0$  equals zero. The term in later position  $i$  is determined by the term  $x$  in the previous position  $i - 1$ :

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

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

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

n	Expected result
0	0
10	0
1000	61
$10^{**}6$	8199
$10^{**}8$	5522779



## Tips and trips

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

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

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

pattern	Expected result (using the wordlist <code>words_sorted.txt</code> )
'**q'	'aeq'
'*yo***'	'iyobaa'
'*m*gv**o'	None
'*ef***da**s'	'befloadabas'
'**l**a*l*ai*'	'ablobaalcaid'
'*e'*8	'lekereseyewemeve'
'*'*100	'aaahabaalabbloadabcdfthadcabdeaddtdryadebbsfm tgnubcfibabeamablschaeraboafbibobaccmlxxiseana ambdspace'

# 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 **theory of computation** and its **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 usually springs forth new computational complexity.

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

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

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

# Lords of Midnight

```
def midnight(dice):
```

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

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

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

To make the test cases more interesting, the tester is programmed to produce fewer ones, fours and sixes than the probabilities of perfectly fair dice would yield. A tactical placement of thumb on the scales occasionally helps the raw randomness to hit hidden targets more effectively.

## Optimal crag score

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

An earlier problem asked you to compute the best possible score for a single roll in the dice game of [crag](#). In this problem, the game is played for multiple rolls, again aided with the same gift of perfect foresight as in the previous “Lords of Midnight” problem. Given the entire sequence of `rolls`, this function should return the highest possible score that can be achieved with those `rolls` under the constraint that **the same category cannot be used more than once**, as dictated by the rules of the actual game of crag.

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

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

# Forbidden substrings

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

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

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

# Go for the Grand: Infinite Fibonacci word

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

[Fibonacci words](#) are strings defined analogously to [Fibonacci numbers](#). The recursive definition starts with two base cases  $S_0 = '0'$  and  $S_1 = '01'$ , followed by the recursive rule  $S_n = S_{n-1}S_{n-2}$  that concatenates the two previous Fibonacci words. See the Wikipedia page for more examples. Especially importantly for this problem, the length of each Fibonacci word equals that particular Fibonacci number. Even though we cannot actually generate the entire Fibonacci word  $S_\infty$  because it would be infinitely long, we can construct the character at any particular position  $k$  of  $S_\infty$  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  $S_\infty$ , must start with the exact same prefix  $S_n$  and therefore contain that same character in the position  $k$ .

This function should compute the  $k$ :th character of the infinite Fibonacci word, the position counting done again starting from zero. 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 the arbitrary laws of imposed by men as you, little starling, fly boldly higher to bid and make this grand slam, your function must be able to work for such monstrously 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 string  $S_n$  for such a large  $n$  to allow you to look up its  $k$ :th character. Instead, use the recursive formula and the list of Fibonacci numbers that you will dynamically compute as far as needed, and apply the [self-similar fractal nature of the infinite Fibonacci word](#).

k	Expected result
0	'0'
1	'1'
10	'0'
$10^{10}$	'0'
$10^{100}$	'0'
$10^{100} + 1$	'1'
$10^{10000}$	'0'

## Bonus problem 110: Reverse the Rule 110

```
#for Suzanne: I think that I finally got it now
def reverse_110(current):
```

In the field of [elementary cellular automata](#), [Rule 110](#) stands tall and proud for being provably **computationally universal**. Standing at [the borderline between chaos and order](#) where all the interesting patterns can temporarily exist, the repeated iterations of this rule produce emergent fractal patterns.

The function to iterate an arbitrary elementary cellular automaton would have been an excellent problem earlier in this collection. But we didn't come all the way here for something that trivial. The function required in this bonus problem **must operate in reverse** by computing the `previous` state from which the Rule 110 would produce the `current` state when applied one step forward!

So that every position has a left and a right neighbour, the left and right edges of each state are **cyclically** wrapped together. To make the returned answers unambiguous for automated testing, this function should return the **lexicographically smallest** of the `previous` states that Rule 110 takes to the same `current` state. For the `current` states that are impossible to produce from any `previous` state using Rule 110, this function should return `None`.

Even straightforward computations can suddenly become complicated when performed backwards from the expected results to the arguments that would produce those results when executed in the forward direction. This problem should be solved using a separate **recursive backtracking** utility function to fill in the `previous` state from beginning to end. The body of the recursion should first check for conflicts between the `previous` and the `current` states, and then try appending the two values 0 and 1 one at the time to recursively fill in the rest of the `previous` state...

current	Expected result
[1, 0, 1, 1]	[1, 0, 0, 1]
[0, 0, 1, 1, 0]	[0, 0, 0, 1, 0]
[0, 1, 0, 1, 0, 1, 1, 1]	None
[1, 0, 1, 1, 0, 0, 0, 0]	[1, 0, 0, 1, 1, 1, 1, 1]