

# CCPS 109 Computer Science I Labs

Ilkka Kokkarinen

Chang School of Continuing Education  
Toronto Metropolitan University

Version of August 28, 2022

# 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, Toronto Metropolitan University, Canada. It contains 109 individual problems (plus seven bonus problems) to choose your battles from, listed 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 that are useful to solve them, no knowledge of any specialized Python libraries is required. Despite this, **you are allowed to use anything in the Python 3 standard library** that you find useful in constructing your program logic. In computer science and all of 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, by itself not quite enough to pass this 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, yes, for grownup realties, those fifty problems can be *any* fifty) takes you up to 90 points where the highest possible grade of A+ 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** when executed on an off-the-shelf desktop from the past couple of years. 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. The automated tester imposes a twenty second cutoff for each test to complete, after which the test is forcefully terminated.

[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 all of them out before submission.

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](#). The classic recreational mathematics ([yes, that is a real thing](#)) by Martin Gardner and his spiritual successors also inspired many problems.

The author has tried to dodge not just the Scylla of the well-worn problems that you can find in basically all textbooks and online problem collections, but also the Charybdis of pointless make-work drudgery that doesn't have any inherent meaning or purpose on its own, at least above the blunt finger practice to provide billable "jobs for the boys" to maintain the illusion that The Machine is still churning 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. Occasionally these problems even relate to entirely separate fields of human endeavour, which makes them 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**, **Zhouqin He** and **Karel Tutsu** for going above and beyond the call of duty in solving these problems in ways that either revealed errors in the original problem specifications or the instructor's private model solutions, or more fortunately, independently agreed with the instructor's private model solution and by this agreement massively raised the instructor's confidence to the correctness of both. In their wake follows the Green Checkmark Gang, a legion of individual students who pointed out ambiguities, errors and inconsistencies in particular problems, and this way effectively acted together as a giant Zamboni to plane the ice for future students to skate over a little bit easier. Members of that horde, you all know who you are. All remaining errors, ambiguities and inconsistencies, whether aesthetic, logical or 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

Count the balls off the brass monkey	32
Count growlers	33
Bulgarian solitaire	34
Scylla or Charybdis?	35
Longest arithmetic progression	36
Best one out of three	37
Collecting numbers	38
Between the soft and the NP-hard place	39
Count Troikanoff, I presume	40
Crack the crag	41
Three summers ago	42
Sum of two squares	43
Carry on Pythonista	44
As below, so above	45
Expand positive integer intervals	46
Collapse positive integer intervals	47
Prominently featured	48
Like a kid in a candy store, except without money	49
Dibs to dubs	50
Nearest smaller element	51
Interesting, intersecting	52
So shall you sow	53
That's enough of you!	54
Brussel's choice	55
Cornered cases	56
Count consecutive summers	57
McCulloch's second machine	58

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

Count overlapping disks	86
Ordinary cardinals	87
Count divisibles in range	88
Bridge hand shape	89
Milton Work point count	90
Never the twain shall meet	91
Bridge hand shorthand form	92
Points, schmoints	93
Bulls and cows	94
Frequency sort	95
Calling all units, B-and-E in progress	96
Lunatic multiplication	97
Distribution of abstract bridge hand shapes	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
Flip of time	105
Bulgarian cycle	106
Square it out amongst yourselves	107
Sum of distinct cubes	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
Painted into a corner	117
Go for the Grand: Infinite Fibonacci word	118
Bonus problem 110: Reverse the Rule 110	119
Bonus problem 111: Aye, eye, I	120
Bonus problem 112: Count domino tilings	121
Bonus problem 113: Invaders must die	122
Bonus problem 114: Stepping stones	123
Bonus problem 115: Ex iudiciis, lux	124
Bonus problem 116: Flatland golf	125





## Ryerson letter grade

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

Before its name change, Toronto Metropolitan University was known as Ryerson University. 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
<code>[]</code>	<code>True</code>
<code>[-5, 10, 99, 123456]</code>	<code>True</code>
<code>[2, 3, 3, 4, 5]</code>	<code>False</code>
<code>[-99]</code>	<code>True</code>
<code>[4, 5, 6, 7, 3, 7, 9]</code>	<code>False</code>
<code>[1, 1, 1, 1]</code>	<code>False</code>

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](#)”

## Riffle shuffle kerfuffle

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

Given a list of `items` whose length is guaranteed to be even (note 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 even 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 whack it 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. (In practice these division and remainder operations are often further condensed into equivalent but faster **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 `tiles` forms 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 [Mathologer](#) video “[Secret of Row 10](#)”. To start, assume the existence of three values called “red”, “yellow” and “blue”. These names serve as colourful (heh) mnemonics and could as well have been 0, 1, and 2, or “foo”, “bar” and “baz”; no connection to actual physical colours is implied. Next, define a rule to mix such colours so that mixing any colour with itself gives that same colour, whereas mixing any two different colours always gives the third colour. For example, mixing blue to blue gives that same blue, whereas mixing blue to yellow gives red, same as mixing yellow to blue, or red to red.

Given the first row of `colours` as a string of lowercase letters, this function should construct the rows below the first row one row at the time according to the following discipline. Each row is one element shorter than the previous row. The  $i$ :th element of each row comes from mixing the colours in positions  $i$  and  $i + 1$  of the previous row. Rinse and repeat until only the singleton element of the bottom row remains, returned as the final answer. For example, starting from the first row 'rybyr' leads to 'brrb', which leads to 'yry', which leads to 'bb', which leads to 'b' for the final answer, Regis. When the Python virtual machine laughingly goes 'brrrrr', that will lead to 'yrrrr', 'brrr', 'yrr', and 'br' for the final answer 'y' for “Yes, please!”

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

(Today's five-dollar power word to astonish your friends and coworkers is "[quasigroup](#)".)



## Count dominators

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

Define an element of a list of `items` 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 that element. Note how 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, the dominators of the list `[42, 7, 12, 9, 13, 5]` would be the elements 42, 13 and 5. The last element of the list is trivially a dominator, regardless of its value, since nothing greater follows it.

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 (and this includes Python built-ins such as `max` and list slicing) and pretending that this somehow makes those inner loops take a constant time will only summon the Gods of Compubook Headings to return with tumult to claim their lion's share of execution time.

## Beat the previous

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

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

This problem can be solved with a for-loop through the `digits` that looks at each digit exactly once regardless of the position of that digit in the beginning, end or middle of the string. Keep track of the `current` number (initially zero) and the `previous` number to beat (initially equal to minus one). Each digit `d` is then processed by pinning it at the end of `current` number with the assignment `current=10*current+int(d)`, updating the `result` and `previous` as needed.

digits	Expected result
'600005'	[ 6 ]
'045349'	[ 0, 4, 5, 34 ]
'7777777777777777777777777777'	[ 7, 77, 777, 7777, 77777, 777777 ]
'122333444455555666666'	[ 1, 2, 23, 33, 44, 445, 555, 566, 666 ]
'2718281828459045235360287471352662497757247093699959574966967627724076630353547594571382178525166427427466391932003059921817413596629043572900334295260'	[ 2, 7, 18, 28, 182, 845, 904, 5235, 36028, 74713, 526624, 977572, 4709369, 9959574, 96696762, 772407663, 3535475945, 7138217852, 51664274274, 66391932003, 599218174135, 966290435729 ]
'1234' * 100	A list with 38 elements, the last one equal to 3412341234123412341234123
'420' * 420	A list with 56 elements, the last one equal to 420420420420420420420420420420420420420420420420

## Subsequent words

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

This problem serves as an excuse to introduce some general discrete math terminology that will help 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” might mean a subset that would be a subsequence, were the members of that set were written out sequentially in sorted order.

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', 'kilooersted', '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 lone taxicab cruising the street grid of the dusky Manhattan that we know and love from classic hardboiled *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  $\mathbb{Z}^2$ . Fitting in the gaunt and angular spirit of the time that tolerates few deviations or grey areas, the taxicab is at all times headed straight in one of the four main axis directions, initially north.

This taxicab then 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 block forward to current heading. This function should return the final position of the taxicab on this infinitely spanning Manhattan. (Before the reader objects to how the Manhattanization of everything continues apace, while the rest of the world just lazily simulates this infinite street grid with mirrors?)

moves	Expected result
'RFRL'	$(1, 0)$
'LFFLF'	$(-2, -1)$
'LLFLFLRLFR'	$(1, 0)$
'FR' * 1729	$(0, 1)$
'FFLLLLFRLFLRFRLRRL'	$(3, 2)$

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 vaguely Eastern European fictional country that *Tintin* and similar hearty fellows would occasionally visit for an adventure, followed by the list of available denominations of `coins` also expressed as kopecks, this function should 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]

Along with its countless variations, this problem 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 healthy and 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.

To achieve this in reasonable time and memory, you should 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, you can 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 then obviously the product of its known width and height.

n	rooks	Expected result
10	[ ]	100
4	[(2, 3), (0, 1)]	4
8	[(1, 1), (3, 5), (7, 0), (7, 6)]	20
2	[(1, 1)]	1
6	[(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'	'zxvtrpnljhfdbacegikmoqsuwy'
'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 in our wordlist?



# 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 crooked dice with certain undesirable pip values replaced with more amenable ones.) 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 forbidden opposites. You can twist and turn any corner of the die to face you, and yet two opposite sides never come together into simultaneous view.

This discipline still allows for two distinct ways to paint the pips. If the numbers in the corner shared by the faces 1, 2, and 3 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 one sense perfectly 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. (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 a 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 feel free to exploit these 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 upon 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 one, 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, as has so often been depicted in various works of space opera, higher beings can generalize the chessboard to  $k$  dimensions from our mere puny two. A natural generalization of the knight's move while maintaining its spirit 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, to determine the square where the  $k$ -knight will teleport as the unseen hand of the player moves it there through the dimension  $k + 1$ .

Given the `start` and `end` positions as  $k$ -tuples of integer coordinates, determine whether the knight could legally move from `start` to `end` in a single teleporting 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, excepting the moves that jump outside the board. In this notation, the ordinary chess knight is a (2, 1)-knight that reaches up to  $2! * 2^2 = 8$  neighbours in one jump. A 6-dimensional knight could reach up to 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, which gives predators too much of an edge over the prey for life to ever evolve there.

## 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 without letting it become some kind of emergency by looking at positive integers that consist of some solid sequence of sevens, followed by some (possibly empty) solid sequence of zeros. Examples of integers of this form are 7, 7700, 77777, 777777700, and 70000000000000. 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 find the smallest such seven-zero integer.

Even though discrete math and number theory help, this exercise is not about coming up with a clever symbolic formula and the proof of its correctness. This problem is about iterating through the numbers of this constrained form of sevens and zeros efficiently in strictly ascending order, so that the function can mechanistically find the smallest working number of this form. However, to speed up the search, we accept the result that 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  $n$ .

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.

# Fulcrum

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

Each element in `items` is a positive integer, in this problems semantically considered to be a [physical weight](#), as opposed to, say, the current count of chickens. Your task is to find 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 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 the inner loop finds one permutation that balances. (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.)

# Fail while daring greatly

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

The ancient world of swords and sandals “ when men were made of iron and their ships were made of wood ” could occasionally also be [an entertainingly violent place](#), at least according to popular [historical docudramas](#) such as “300”, “Spartacus” and “Rome”. 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 unwavering commitment to this shared fate. They arranged themselves in a circle, and used lots to choose a step size  $k$ . Then repeatedly count  $k$  men ahead, kill him and remove his corpse from this grim circle.

Being [normal people instead of computer scientists](#), this deadly game of eeny-meeny-miney-moe is one-based, and 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 hammy 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 per group, where  $out$  is a 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 for good.

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 famous 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 (hopefully something higher than winning a bar bet), 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 swinging sixties for whom balancing their chequebooks in the familiar base ten was already an exasperating ordeal!

# 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 some Indiana Croft, or by gaining visions of enlightenment by intensely meditating under the apex set over a square base like some Deepak Veidt, this problem deals with something a bit more mundane; 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.

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

As an unrelated note, as nice as Python can be for casual coding by liberating us from all that low level nitty gritty, [Wolfram is another great language](#) with [great online documentation](#). You can play around with this language for free on [Wolfram Cloud](#) to try out not just all the cool one-liners from the tutorials and documentation pages, but evaluate arbitrary mathematical expressions of your own, for example `Sum[(n+i)(m+i), {i, 0, h-1}]`, in a [fully symbolic](#) fashion.



## 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', 'cat', 'dog', 'god', 'cat', 'tac']	11
['god', 'tac', 'tac', 'tac', 'tac', 'dog', 'dog', 'tac', 'cat', 'dog', 'god', 'cat', 'dog', 'cat', 'cat', 'tac']	0

(I will be the first to admit that I was high as a kite when I thought up this problem. Sometimes problems and their solutions can be hard to tell apart, as that distinction depends on the angle from which you view the whole mess from.)

# Bulgarian solitaire

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

In front of you are  $k*(k+1)//2$  identical pebbles, given as `piles`. (These pebbles total the  $k$ :th **triangular number** that is equal to the sum of the positive integers from 1 to  $k$ , for every budding little Gauss out there to speed up their computations.) Bored out of your mind, you lazily arrange these pebbles around. Eventually you make a game out of it; seeing how many times you can arrange these pebbles into neat rows in a day, then trying to break that record.

An apt metaphor for the bleak daily life behind the Iron Curtain, in this solitaire game all pebbles are identical and you don't have any choice in your moves. 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 the move to this steady state simply leads 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 unpredictable 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 play the Final Boss. (Everyone is the hero in their own story until they become a villain in someone else's.) Her final showdown resembles an 8-bit video game one-dimensional 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 henchmen [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 and lift your nemesis to safety. She is then nearly guaranteed to 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, 7, 10, 13, 16]` is an arithmetic progression of length 5, starting from value 4 with the stride of three.

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, do you see the difference between two deceptively similar concepts of “leftmost of longest” and “longest of leftmost”?

## Best one 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 it.

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**. This 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 realties. 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 exercise: prove this), thus eliminating all risk of using some funky outlier as the approximate median.

# Collecting numbers

```
def collect_numbers(perm):
```

This problem is adapted from the problem "[Collecting Numbers](#)" in the [CSES Problem Set](#). Your adversary has shrunk you to a microscopic scale and trapped you inside the computer. You are currently standing at the first element of `perm`, some **permutation** of integers from 0 to  $n-1$ , each such number appearing in this list exactly once. The rules of this game dictate that you are only allowed to move right and advance only one element at the time, but never step to the left or jump over any elements. This list is treated as **cyclic** so that whenever you would step past the last element, you again find yourself back at the beginning, having gone one more round around the list.

To get out, you must tell your adversary as soon as possible how many rounds it would take you to complete the task of collecting the elements from 0 to  $n-1$  in the exact ascending order. Following the movement rules, you would keep going right until you find the element 0. After that, you would keep going right until you find the next element 1. Whenever you come to the end, you start a new round from the beginning. Eventually, you will have collected all the numbers this way in order and be done. For example, given the permutation `[2, 0, 4, 3, 1]`, you collect 0 and 1 during the first round. In the second round, you collect both 2 and 3. In the third round, you finally get to collect the remaining 4 and call it a day, for the final answer of three rounds.

The important lesson of this problem is that a function that simulates some other system as a **black box** does not *literally* have to follow the rules of that system, but may use any computational shortcut whatsoever, as long as the final answer is correct! Therefore, beat your adversary with flair by first constructing the **inverse permutation** of `perm`. This inverse permutation `inv` is another permutation of integers from 0 to  $n-1$  that satisfies `inv[i]==j` whenever `perm[j]==i`. For example, the inverse of the previous permutation is `[1, 4, 0, 3, 2]`. The `inv` list allows you to quickly look up the location that any particular element is stored at in the original `perm`. Once your function has first constructed `inv`, you will be able to compute the required number of rounds with a single for-loop over `range(0, n)` that does not even glance at the original `perm` during its execution, since this loop gets all the information that it needs from the `inv` list alone.

perm	Expected result
<code>[0, 1, 2, 3, 4, 5]</code>	1
<code>[2, 0, 1]</code>	2
<code>[0, 4, 3, 5, 2, 1]</code>	4
<code>[0, 2, 4, 1, 3, 5]</code>	3
<code>[8, 6, 9, 5, 4, 11, 2, 0, 3, 10, 12, 1, 7]</code>	7
<code>list(range(10**6, -1, -1))</code>	1000001

## Between the soft and the NP-hard place

```
def verify_betweenness(perm, constraints):
```

Immediately following the previous problem of collecting numbers, here is another problem about permutations where computing the inverse permutation first will make the computing the actual result a breeze. Define that the permutation satisfies the **betweenness** constraint given as a three-tuple  $(a, b, c)$ , if in that permutation the middle element  $b$  lies somewhere between the boundary elements  $a$  and  $c$ . The elements  $a$  and  $c$  can appear either order in the permutation. For example, the permutation  $[2, 1, 3, 0]$  satisfies the betweenness constraints  $(1, 2, 3)$  and  $(0, 3, 2)$ , but violates the betweenness constraints  $(1, 2, 3)$  and  $(3, 0, 2)$ . This function should determine whether the given permutation satisfies all of the given betweenness constraints.

perm	constraints	Expected result
$[2, 0, 3, 1]$	$[(0, 2, 1), (1, 3, 0), (1, 3, 2)]$	False
$[0, 2, 3, 5, 4, 1]$	$[(0, 3, 1), (4, 2, 0)]$	True
$[5, 2, 0, 3, 1, 4]$	$[(4, 3, 5), (2, 0, 4), (2, 1, 4), (1, 2, 5), (4, 0, 5)]$	True
$[6, 1, 0, 3, 2, 4, 5]$	$[(2, 3, 4), (4, 0, 3)]$	False

This problem is far deeper than it might initially appear. Even though we can verify relatively quickly whether a given permutation satisfies the given constraints, determining whether the given constraints could be satisfied by some permutation, literally any one permutation that you totally get to choose for yourself, turns out to be [NP-complete](#)! The worst-case running time of all known algorithms guaranteed to find such a permutation whenever one exists grows exponentially with respect to the length of the permutation. The fundamental asymmetry between how easy it is to verify that the solution that some hinky guy in a dark alley offers to sell you satisfies the given constraints, versus the exponential difficulty of constructing a working solution for the given constraints, seems to be built into the "physics of the abstract" of our shared reality itself.

## Count Troikanoff, I presume

```
def count_troikas(items):
```

Define three positions  $i < j < k$  of the `items` list to constitute a **troika** if the elements in all three positions are equal, and the positions themselves are spaced apart in an **arithmetic progression**. More succinctly, these positions satisfy `items[i]==items[j]==items[k]` and `j-i==k-j`. This function should count how many troikas exist in the given list of `items`. For example, the list `[42, 17, 42, 42, 42, 99, 42]` contains four troikas, each using the value 42.

This problem could be solved by inefficiently juggernauting through all equally spaced position triples  $(i, j, k)$ . (You only need to loop through  $i$  and  $j$ , since those nail down  $k$ .) However, you can do better by building a dictionary whose keys are the elements that appear somewhere in `items`. Each value is a sorted list of positions where that particular key appears in `items`. For the previous example list, this dictionary would be `{42: [0, 2, 3, 4, 6], 17: [1], 99: [5]}`.

Then, loop through the keys of this dictionary. For each key, loop through every position pair  $i < j$  into its value list (the decorator `itertools.combinations` again goes *brrrrr*, ha ha), and compute the assumed third position with the formula  $k = j + (j - i)$ . If `items[k]==items[i]`, increment your troika counter. Searching for troikas in this manner will typically spend an order of magnitude less time churning out the answer than the "Shlemiel" method of brute force. The more distinct values these `items` cover, the sharper the edge of this more sophisticated method.

<code>items</code>	Expected result
<code>[5, 8, 5, 5]</code>	0
<code>[-8, -8, -8, -18, -8, 13, -8]</code>	3
<code>[3, 6, 3, 6, 3, 6, 6, 6, 3]</code>	5
<code>[23, 23, 23, 23, 23, 23, 16, 23]</code>	8
<code>[21, 21, 21, 21, 21, 21, 21, -41, 21, 76, 21, -71, 47]</code>	15
<code>[42 for _ in range(100)]</code>	2450



## Crack the crag

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

**Crag** is an old 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 to make such solutions reveal themselves by 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`.

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

For the general **subset sum problem** that was 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):
```

Many positive integers can be expressed as a sum of exactly two squares of positive integers, both possibly equal. 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 the integer  $n$  cannot be so expressed as a sum of two squares.

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 pointers**, as previously seen in the function `two_summers` in the [listproblems.py](#) example program, directly works also on this problem! The two positions start from both ends of the sequence, respectively, from where they inch towards each other in a way that guarantees that neither can ever skip over a solution. This process must eventually come to a definite outcome, since one of the two things must eventually happen: either a working solution is found, or the positions meet somewhere before ever finding a solution.

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, causing the execution time to be **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 so early that most people don't even think of that mechanistic operation as an algorithm! Instead of the sum  $a+b$  that the language would already compute for you anyway, this problem asks you to 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 this time you don't care about the actual result of the addition, but only tally up 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
11111111111	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 some rows of the famous [Pascal's triangle](#) will be generated during the first two years of a computer science undergraduate program is safe as houses. Even without consideration to the [interesting combinatorial sequences inside this infinite table](#), merely tabulating its entries is a good little exercise of using nested loops.

This problem looks at **Leibniz triangles**, a curious “upside-down” variation of the Pascal's triangle where each entry equals the sum of the two entries immediately **below** it, instead of **above** it the way they do in Pascal's triangle. The immediate problem is that such upside-down recursion has no base cases that would ever allow it to terminate. Instead of Pascal's base case of the unity bit at the top whose union with the zero bit of the nonexistent void outside then fills in the rest of the triangle, the Leibniz triangle branches out into an cacophonous pyramid of turtles all the way down to Dante's ten circles, whence the Adversary can fill it in uncountable (yes, Lana, *literally*) ways, each more diabolical than the next.

Fortunately for us clouds of dust dancing in the great wind that carries us all, merely defining the `heads`, the leading elements of every row, sufficiently constrains this recursion to rein in the rest of the triangle. Should a dispute about some value emerge, very well then, let us simply take out our [mechanistic step reckoners](#) and calculate! One possible solution to this recursion is the original [Leibniz harmonic triangle](#) that comes from using consecutive **unit fractions** as heads.

This function should immanentize as many rows of the Leibniz triangle as there are `heads` given, using two nested loops to compute these entries. However, this function needs to return only the entries in given `positions` of the final generated row. Having verified those to be 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 starting from the top, 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)]

(Extra gold stars are awarded for students whose code Ol' Leiby himself would have grokked.)

## 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>

For purely aesthetic reasons, the problem is restricted to positive numbers. The above notation would still be unambiguous even if the very same minus sign character served two masters by acting as both the element separator and the unary minus sign. The internal state of the parser always decides what each character, like, finger-quotes "means" in the current position.

We also note how the different lengths of dashes and quotation marks pointing in different directions tend not be used in computer text, despite the fact that they are Unicode symbols just the same as any old-timey ASCII symbol. (Perhaps ASCII stands for "ass-kiss" here. Go figure.)

## 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 only one. However, it is usually more important for the encoding to be uniform than to be pretty. As a general principle, uniform and consistent encoding of data allows the processing of that data to also be uniform in the tools down the line.

If some maximal sublist consists of a single integer, it must be included in the result string all 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. To simplify things, 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. Each peak is represented as a tuple (`position`, `elevation`, `prominence`). The `position` and `elevation` of each peak are taken 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 **strictly higher** peak, when you get to choose this higher peak and the route freely to minimize this descent. (In technical terms, you must do **maximin optimization** in choosing the best route to *maximize* the height of its *minimum* point.) The lowest valley along this best route is the **key col** of the original peak, and the next strictly higher peak reached through the key col is its **parent peak**. Since the highest peak of the island has no key col, its prominence simply equals its raw elevation.

This problem will [soon get tricky for two-dimensional height fields](#) with all the ensuing freedom 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)]
[1, 10, 8, 10, 5, 11]	[(1, 10, 5), (3, 10, 5), (5, 11, 11)]
[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 sit around in a circle so that each child has some number of identical candies in front of them. To synchronize these children to move as a group in a logically simultaneous fashion, the teacher rings a bell to mark 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, waiting 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 will 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]	21

With 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 where there are 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 forever passing the same candy around the circle. It is still an unsolved mathematical problem to classify the starting states significantly faster than mechanistically iterating this process! (Falling into a limit cycle can be detected efficiently with the famous [tortoise and hare algorithm](#), as you will see for yourself with 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 into a taxicab flashing an exciting number such as 1234 or 6969 would be more instatokkable than getting into 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 ambiguity by always using the smaller of these two elements.

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]

Side question for any combinatorics enthusiasts: starting from a random permutation of  $n$  distinct elements, what is the expected number of rounds that the above algorithm needs to be iterated for all elements to become equal?

## 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 intersection of two squares can have zero area, 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 general 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 its foot down, we will first display hearty mathematical gusto by generalizing the game mechanics for arbitrary number of  $k$  houses. The complete **minimax algorithm** to find the best move in the given situation will 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 act 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 machines is that understanding *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 ])

## Cornered cases

```
def count_corners(points):
```

On two-dimensional integer grid, a **corner** is three points of the form  $(x, y)$ ,  $(x, y + h)$  and  $(x + h, y)$  for some  $h > 0$ . Such points form sort of a **carpenter's square** or a **chevron** pointing southwest so that the point  $(x, y)$  serves as the **tip** and  $(x, y + h)$  and  $(x + h, y)$  define its axis-aligned **wings** of equal length. For example, the points  $(2, 3)$ ,  $(2, 7)$  and  $(6, 3)$  form a corner as  $x = 2, y = 3$  and  $h = 4$ . On the other hand, the points  $(2, 3)$ ,  $(2, 1)$  and  $(0, 3)$  do not form a corner, since their group orientation, albeit well-intentioned but unfortunately facing the wrong way, would give the matching with  $h = -2$ . Also note that any number of corners can share a common point.

Given a list of `points` sorted by their  $x$ -coordinates, ties broken by  $y$ -coordinates, this function should count how many corners altogether exist among these `points`. Again, this problem *could* be solved in "Shlemiel" fashion by **brute forcing** through all three-element subsets of `points`, either the manly way of using three levels of nested for-loops, or calling the `itertools.combinations` sequence decorator for a good time. Instead, you should iterate through the individual points  $(x, y)$  to determine if they can serve as a tip of some chevron. Since the  $x$ -coordinate of the upwing is the same as that of the tip, the inner while-loop can examine points sequentially from the current point. For each candidate point  $(x, y + h)$  to try as this upwing, you need to rapidly determine whether the right wing  $(x + h, y)$  that completes this corner is among the remaining `points`...

points	Expected result
<code>[(2, 2), (2, 5), (5, 2)]</code>	1
<code>[(0, 3), (0, 8), (2, 2), (2, 5), (5, 2), (5, 3)]</code>	2
<code>[(1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (3, 3), (4, 3)]</code>	3
<code>[(0, y) for y in range(1000)] + [(x, 999-x) for x in range(1, 1000)]</code>	999
<code>[(x, y) for x in range(200) for y in range(200)]</code>	2646700

As for the corners themselves and how they relate to each other, [a curious theorem](#) sets a hard upper bound to how many points can be chosen from an  $n$ -by- $n$  integer grid without creating any corners. Try this out for yourself for small  $n$ , if only to get the feel of the tight wiggle room that will soon see you having painted yourself in yet another corner in the twisty maze of possibilities. Conversely, how many corners could you create by placing  $n$  points on the plane?



## 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](#) cry out to be appreciated as works of conceptual performance art. These tar pits are not designed to serve as practical programming tools, but to point out in a ha-ha-only-serious manner how some deceptively simple mechanism is secretly some ancient demon sword that 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` computed from the call `Y=mcculloch(X)`. This function should 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
4X	Y[ ::-1 ]	reversal
5X	Y + Y	concatenation
anything else	None	N/A

Recursive application of these rules produces the following example results:

digits	Expected result
'329 '	'929 '
'53231 '	'3123131231 '
'4524938 '	'83948394 '
'343424859355 '	'4859355248593552485935524859355 '
'433342717866 '	'7178662717866271786627178662717866271786627178662717866 '

## 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 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 `[5, 2, 6, 3]`. The next step will remove 2 and its larger neighbour 6, reaching the goal in two steps.

Removing an element from the middle of the list is expensive, and will surely form the bottleneck in the straightforward solution of this problem. 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 ever actually mutating `items`! 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 remove `i`, make its left and right neighbours `left[i]` and `right[i]` figuratively join hands with two 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 standing in the forest. (No tree, no problem; all will ultimately be equal in the mulch from the equal bite of the wood chipper.)

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

## 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 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 from inches below the surface, but also 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):
```

The generalized  $n$ -by- $n$  chessboard has this time been taken over by some [bishops](#), each represented as a tuple `(row, col)` of the coordinates 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 check whether two squares `(r1, c1)` and `(r2, c2)` are reachable from each other in a single bishop move, the expression `abs(r1-r2)==abs(c1-c2)` checks that 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 one test handles four diagonals in one swoop.

n	bishops	Expected result
10	<code>[]</code>	100
4	<code>[(2, 3), (0, 1)]</code>	11
8	<code>[(1, 1), (3, 5), (7, 0), (7, 6)]</code>	29
2	<code>[(1, 1)]</code>	2
6	<code>[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]</code>	18
100	<code>[(row, (row*row) % 100) for row in range(100)]</code>	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 multiple ways, this splitting is done the best way that produces the tallest final height.

Since the list of words is known to be sorted, you can use **binary search** (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 the poor souls who construct the Champernowne word as an explicit string will run out of both time and space long before receiving the answer. Instead, notice how 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  $k$ -digit numbers that contains the position  $n$ , the digit in that position is best determined with integer arithmetic, perhaps aided by an `str` conversion for access to digit positions.

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 rewarding side quest. Define “shy” integers as those that appear inside the Champernowne word for the first time 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 a rule that establishes at a glance that the eight-digit integer 92021222 is *mu*y eager, as is also the number 456123? What does your intuition say about the respective **densities** of eager and shy integers among all integers?



## Reverse the vowels

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

Given a `text` string, create and return a new string constructed by finding all its **vowels** and reversing their order, while keeping all other characters exactly as they were in their original positions. To make the result more presentable, the capitalization of each position must remain the same as it was in the original `text`. For example, reversing the vowels of `'Ilkka'` should produce `'Alkki'` instead of `'alkkI'`. For this problem, vowels are the usual `'aeiouAEIOU'`.

Along with many possible ways to perform this dance, one straightforward way to reverse the vowels starts by appending the vowels of `text` into a separate list, and initializing the `result` to an empty string. Then, loop through all characters of the original `text`. Whenever the current character is a vowel, `pop` 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 add it to `result`. Otherwise, add the character from the original `text` to the `result` as it was.

text	Expected result
'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!'
'We had to sacrifice a couple of miners to free Bolivia.'	'Wa hid ti socrefeco e ciople uf monars te frii Balovae.'
"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!"

Applying this operation to perfectly ordinary English sentences often produces pig latin imitations of other languages. Perhaps some Vietnamese-speaking reader could extend this algorithm to recognize and reverse also all diacritical versions of the vowels in that language, if only to check out whether this transformation yields some unintentional comedy or revelations, divine or mundane, when applied to everyday Vietnamese sentences.

# Everybody on the floor, come do the Scrooge Shuffle

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

Some positions on the integer line  $\mathbb{Z}$  initially contain gold coins, as given in the `coins` list starting from the origin position of zero. For example, if `coins == [3, 0, 1, 7]`, there are three coins at the position 0, one coin at the position 2, and seven coins at the position 3. All other positions on the integer line  $\mathbb{Z}$ , both positive and negative, are initially empty, as is also the position 1 in the middle.

Any position that contains at least `left+right` coins is **unstable**. For example, if `left==3` and `right==2` for the previous example piles, only the position 3 would be unstable. Given the starting configuration of `coins`, your function should repeatedly find some unstable position `i`. Move exactly `left` coins from that position `i` to the position `i-1`, and move exactly `right` coins to the position `i+1`. Performing this spill in the previous configuration at the position 3 produces the new configuration `[3, 0, 4, 2, 2]` that is stable everywhere since no position contains five coins or more, and this process terminates.

It can be proven that the terminal coin configuration of this back-and-forth dance will necessarily be reached after some finite number of steps, and is unique independent of the order that you process the unstable positions in the interim. This function should return a tuple `(start, coins)` where `start` is the leftmost position that contains at least one coin. The second component of the answer lists the `coins` that end up in each position from the position `start` up to the last position that contains at least one coin. For example, the expected answer from this function for the previous example would be `(0, [3, 0, 4, 2, 2])`. Some initial configurations will end up spilling coins to the negative part of the integer line  $\mathbb{Z}$ , so your function needs to be able to handle this.

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 make its appearance in its lowest reduced form! To perform the following calculations, you should import the data types `Fraction` and [deque](#) from the [fractions](#) and [collections](#) modules.

Your function should return the  $n$ :th element of this sequence. First, create a new instance of `deque` and append the first fraction  $1/1$  to prime the pump, so to speak, to initiate the production of the values of this sequence. 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 <code>Fraction</code> )
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 hefty 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 the hard way there 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 memorize all these mechanistic forms. After what was clearly one too many episodes of *Narcos* to help the lockdown time go by, this author decided to spend some of this trying time by finally learning some basic Spanish, one of the most meme-worthy languages devised for 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 for our daily uphill journey both ways rain or shine (and that is how we liked it, without complaining!), today's Unicode strings of Python 3 treat [accented characters](#) no differently from any other characters, appropriate for the globalized cyberpunk utopia just around the corner. 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. Su escala de decisiones debe contener mucho menos de ciento veinte escalones. Ponte a trabajar, cabrón!)*

## Hippity hoppity, abolish loopity

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

A frog hopping along on the infinite two-dimensional grid of integers is represented with a 4-tuple of the form  $(sx, sy, dx, dy)$  so that  $(sx, sy)$  is the frog's **starting position** at the time zero, and  $(dx, dy)$  is its constant **direction vector** for each successive 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 should not contain any loops whatsoever**, but result should be computed with conditional statements and integer arithmetic. First solve the 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 your one-dimensional 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 to get 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 once 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 their big mouths and little mouths to make it game over for any man who enters.

Rapid motion knife tricks won't help the bishop here. 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. The bishop can initially propel himself to any of the four diagonal directions. After this initial impulse that gets 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), (0, 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 obviously was not devised by [some computer scientist](#) but a more normal person, positions start from 1, not 0. Furthermore, we rewrote the formula to use the letter  $i$  to denote the position, since the letter  $n$  already means something else for us in this problem.

For example, the sequence of "[octagonal numbers](#)" that springs forth from  $s = 8$  shoots into infinity as 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.

[illegible]

As you can see from the last row of the expected results table, this function must be efficient even for gargantuan values of  $n$ . You should harness the power of **repeated halving** to pull your wagon to this promised land 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 nudge the different evaluation order to differ from the usual **PEMDAS** order determined by **precedence** and associativity of each operator. The alternative **postfix** notation (also known as [Reverse Polish Notation](#), for all you “simple Poles on a complex plane”) may first look weird to people accustomed to the conventional infix. However, postfix notation turns out to be easier for machines, since it allows encoding *any* intended evaluation order without any parentheses!

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 basic 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 left in the stack becomes 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, '*', '*', '*', '*' ]	$1 * 2 * 3 * 4 * 5$	120

(By adding more operators and another auxiliary stack, an entire Turing-complete programming language can be built around postfix evaluation. Interested students can sally forth to the Wikipedia page "[Forth](#)" to learn more of this ingeniously simple **concatenative programming** language.)



# 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 earlier [family board game of the same name](#). Since that achievement eclipsed basically all other [wacky and original creations](#) of Conway, we ought to rectify this by giving these less appreciated creations an occasional spin in the paparazzi lights and the red carpet fame.

This function works as an **interpreter** for the [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 esoteric 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, m8, 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
3	[(94, 51)]	10	[3]
7	[(7, 3), (12, 7)]	10	[7, 12, 28, 48, 112, 192, 448, 768, 1792, 3072, 7168]
10	[(3, 6), (9, 11), (4, 5), (6, 2), (7, 1)]	10	[10, 5, 4, 2, 1, 3, 9, 27, 81, 243, 729]
5	[(108, 125), (90, 45), (122, 57), (75, 158)]	10	[5, 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120]

# 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 from 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.

As explained in the YouTube video "[Cycle Notation of Permutations](#)" of [Socratica](#), a **cycle** 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 cyclic "[necklace](#)" fashion so that  $[1, 2, 3]$ ,  $[2, 3, 1]$  and  $[3, 1, 2]$  are **rotationally equivalent** ways to represent the same three-cycle, distinct from another possible cycle of these elements representable 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  $j=i$  and keep moving from the current position  $j$  to its successor position  $\text{perm}[j]$  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 must 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 theses cycles without any separators in between. Amazingly, this run-on **standard representation** of the permutation still allows unique reconstruction of the original permutation and its 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 gets to move to zero wins as his opponent is unable to move. This is an example of an [impartial game](#) where the exact same moves are available to the player whose turn it is to move, as opposed to "partisan games" such as chess where players may only move pieces that wear their colours.

This and similar [combinatorial games](#) can be modelled and solved with recursive equations. A state is **hot** (or "winning") if at least one legal move from that state leads to a **cold** (or "losing") state. Yes, that really is the entire definition, including the base case. (You might have to squint a bit to see it.)

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 all states as a single list built up from left to right. The heat value computation at each state can then read the previously computed heat values of the lower numbered states to which there is a legal move from the current state.

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, True, True, False, False, 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 ecneuques

```
def ztalloc(shape):
```

The famous [Collatz sequence](#) was used in the lectures as an example situation where it is necessary to use a while-loop, since it is impossible to predict in advance how many steps are needed to reach the goal from the given starting value. (In fact, no finite upper bounds are known, as 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, the Collatz 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 'u' and 'd', respectively. 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 made of letters u and d, determine which starting value for the Collatz sequence produces that exact `shape`. However, this function must also recognize that some shape strings are impossible as entailed by the Collatz transition rules, 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, you have ensure that your function does not accept moves that would be illegal in the original forward-going Collatz sequence. Especially the rule  $3x+1$  can only be applied to odd values of  $x$ , no matter how tempting the successor  $3x+1$  might be.

shape	Expected result
'ududududddddudddd'	15
'dudududddudddudddd'	14
'uduuudddd'	None
'd'	2
'uuududddddouuuuuuddddd'	None
'duuuddddd'	None

## The solution solution

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

This problem was inspired by yet another [Numberphile](#) video, this time "[The Centrifuge Problem](#)". Alternatively, students whose eyes can read faster than their ears listen can check out Matt Baker's post "[The Balanced Centrifuge Problem](#)". A **centrifuge** has  $n$  identical slots, each big enough to fit one test tube. To prevent this centrifuge from wobbling,  $k$  identical tubes must be placed into these slots so that their mutual **center of gravity** lies precisely at the center of the centrifuge.

For balancing  $k$  test tubes into  $n$  slots to be possible, **both**  $k$  and  $n-k$  must be expressible as sums of **prime factors** of  $n$ , not necessarily distinct. For example, when  $n$  equals 6 with prime factors 2 and 3, the centrifuge can hold 0, 2, 3, 4 ( $= 2 + 2$ ) or 6 ( $= 3 + 3$ ) test tubes. However, there is no safe way to balance 1 or 5 test tubes into these six slots. Even though  $5 = 2 + 3$  satisfies the first part of the rule, there is no way to counterbalance the remaining empty slot!

This function does not actually need to construct the balanced configuration of  $k$  test tubes, but determine whether some balanced configuration exists for the given  $n$  and  $k$ .

n	k	Expected result
6	3	True
7	0	True
15	8	False
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 in the previous sentence to require each element to be greater than the previous element, not merely equal to it.

As is the case with all functions found in this problem collection, this function should not modify the contents of the original `items` 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** in the original list to save memory. If you want to use the original `items` as the starting point of computation, you can always assign `items=items[:]` in the first line of your function to create a separate but equal copy of the `items`, to operate on that copy instead of the original from that point onwards.

In the table below, different colours highlight the maximal strictly ascending sublists for readability, and are not part of the actual argument object given to the Python function.

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. (A truly advanced set of rules, perhaps trained with **deep learning** techniques on a real world news corpus to implicitly recognize the **semantic** content might combine 'donald' and 'hillary' into either 'dollary' or 'dillary', depending on the intended tone and audience.)

## 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, although that claim might not be as easy to prove from first principles.

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 the pseudorandom fuzz testing, this function should not return the line itself, but merely the count of how many points lie on that line. The list of `points` is guaranteed to contain at least two points, and 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. Armed with this function, you could next numerically investigate the problem presented in the middle panel of [this XKCD comic](#). (When generating each random non-self-intersecting path, **rejection sampling** would be the easiest way to ensure that your choices among all possible such paths are unbiased.)



# 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 (thus eradicating all piles with exactly `p` cookies), and leaves all smaller piles untouched as they were.

Since the Count is also unhealthily obsessed with order and hierarchies, he expects these 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. For the given `piles`, your function should loop through its possible moves, and recursively solve the problem for the new piles that result from applying each move to `piles`. The optimal solution for `piles` is then simply the optimal one of those subproblem solutions, plus one.

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

The above version of the Cookie Monster problem has been streamlined a bit to keep this problem simple enough to solve here. The more general problem formulation allows Cookie Monster choose any subset of remaining piles, and remove the same freely chosen number of cookies from each pile in the chosen subset. Interested students can check out the article “[On the Cookie Monster Problem](#)” about the subtle complexities of the general problem formulation.

# 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 uaigntbmuivy of atematoud testing. In the vast maitjory of ceass, this list wlil catnoin only one wrod.

wrod	Etecxepd rssuelt (unsig the wrosldit wdors_sroted.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 mapped to the same letter a, violating the requirement for the mapping to be injective.

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']

# Manhattan skyline

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

[Computational geometry](#) is essentially geometry that uses only integer arithmetic without any of those nasty transcendental functions. (Not only is that an actual place, but I have come here to lead my people into that promised land.) The **Manhattan skyline problem**, a classic chestnut of computational geometry, is best illustrated by pictures and animations such as those on the page "[The Skyline problem](#)". 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 (surely Rodgers and Hammerstein would have 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. All towers share the same flat ground baseline.

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 these towers 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 is added the area summation.

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

The automated tester script will produce start and end coordinates from an exponentially increasing scale. This puts kibosh on solving this problem 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 with an inner loop, and finally sums up these position height strips for the total area.

## 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 represented 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** 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 runs on pure integer arithmetic whenever its arguments are integers, so no square roots or any other irrational numbers gum up the works with their decimal noise. (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 for larger lists. 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 meaningfully according to arbitrary comparison criteria, as long as those criteria satisfy the mathematical requirements of a **total ordering** relation. To play around with this concept to grok it, let us define a wacky ordering relation of positive integers so that for any two integers, the one that contains the digit 9 more times is considered to be larger, regardless of the magnitude and other digits of these numbers. For example,  $99 > 12345678987654321 > 10^{100000}$  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 down there 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 determines 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 useless in The Real World, though. The ordinary ordering of ordinals that we tend to grant the status of being almost some sort of law of nature is useful because it plays along nicely with useful arithmetic operations such as addition (for example,  $a + c < b + c$  whenever  $a < b$ ) to make these operations more powerful in this 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 more fancy. The difficulty is coming up with the conditions that cover all possible cases of this problem just right, including all of the potentially tricky **edge** and **corner cases**, without being **off-by-one** anywhere. 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 author's thumb tactically resting on the scale of the automated tester ensures 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 well 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 and become 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 and all their possible timelines encoded 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 Eldritch abominations 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  $b$  already co-operate, the iron hand of the [Fundamental Theorem of Arithmetic](#) dictates that the integer powers  $a^{pa}$  and  $b^{pb}$  can never be equal for any two positive integer exponents  $pa$  and  $pb$ . However, in the jovial spirit of "[close enough for the government work](#)", we define such powers to "hit" if their difference  $\text{abs}(a^{pa} - b^{pb})$  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` expects  $a^{pa}$  and  $b^{pb}$  to be within 1%. For given positive integers  $a$  and  $b$ , return the smallest integer exponents  $(pa, pb)$  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](#), generally an excellent source of puzzles of this 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 such things to astonishing precision merely 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 the 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 dramatic 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 our version of 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 pruning all four digits from future guesses!

This function should list all numbers that still could be the secret number according to the results of the list of guesses completed so far. Each guess is given as a tuple (guess, bulls, cows). This function should return the list of such consistent four-digit integers 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 argument values.)

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

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

# 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 to 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 relevant 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 built on [Catalan's conjecture](#), these days 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 wacky operators define an [algebraic ring](#) where useful and important algebraic 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 in the audience, 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 (since Python lists are mutable, 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 far less balanced shapes 5-4-3-1 and 5-4-2-2 that one might have 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.)

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.

# 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 just so 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 **interspersation** of positive integers; every positive integer will appear **exactly once** in the entire infinite grid, with no gaps or duplicates anywhere! (This result also nicely highlights the deeper combinatorial importance of the deceptively simple Fibonacci sequence as potential building blocks of integers and their 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 ( $row, col$ ), 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, col)` 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 to be disjoint so that no rook can act as either a fence-sitter or a turncoat), 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 any letter except one of the letters that 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 letter 'g' or 'd' that already appears 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', 'sudge', 'sjudgy', 'stodge', 'stodgy', 'swedge', 'swidge', 'trudge', 'unedge']
'*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 natural numbers, 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. The goal  $(0, 0)$  is never part of the `tabu` list.

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**. (Note the crucial difference between an empty path that *exists*, versus a *nonexistent* 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 the origin  $(0, 0)$  equals the sum of paths from the two neighbours  $(x-1, y)$  and  $(x, y-1)$  from which the current path can continue in a mutually exclusive fashion.

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 with `lru_cache`, or even better, not use recursion at all but build up a two-dimensional list whose entries are the individual subproblem solutions. Fill in the correct values with two `for`-loops in some order that guarantees that when these loops arrive at position `[x][y]`, the results for positions `[x-1][y]` and `[x][y-1]` needed to compute `[x][y]` are already there.

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

This technique of using `for`-loops to fill out the memoization tables instead of using a recursive formula is called **dynamic programming**. This elegant technique is truly a skeleton key that [opens thousands of doors](#) that Hercules himself would not be able to pry open with brute force.

# 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.

Given a set of points, each point a tuple  $(x, y)$  where  $x$  and  $y$  are nonnegative integers, this function should count how many **squares** exist so that all four corners are members 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. For example, the points  $(0, 3)$ ,  $(3, 0)$ ,  $(6, 3)$  and  $(3, 6)$  define a square, even if it may happen to look like a lozenge from our axis-aligned vantage point. As long as all four sides have the exact same length, be that length an 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  $dx \geq 0$  and  $dy > 0$ , so that the points  $(x+dx, y+dy)$ ,  $(x+dy, y-dx)$  and  $(x+dx+dy, y-dx+dy)$  for the 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 please do at least make an attempt to be economical in these loops, and make your function sweep up these squares more swiftly than even Tom Swift himself ever could corner them.

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

Interested students can check out the [Mathologer](#) video "[What does this prove?](#)" about this general topic, if only to see a beautiful **infinite regress** proof of why squares are the only regular polygons whose corner points can lie exactly on the points of an integer grid.



# Flip of time

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

An hourglass is represented as a tuple  $(u, l)$  (the second character is the lowercase L, not the *numero uno*) for the number of minutes in its upper and lower chambers. After  $m$  minutes have elapsed, the state of that hourglass will be  $(u - \min(u, m), l + \min(u, m))$ . The total amount of sand inside the hourglass never changes, nor will either chamber contain negative anti-sand.

Given a list of `glasses`, your task is to find an optimal sequence of moves to measure exactly  $t$  minutes, scored as the number of individual hourglass flips performed along the way. Each move consists of two stages. You must first wait for the hourglass that currently holds the lowest nonzero amount  $m$  of sand in its upper chamber to run out. When that happens, choose **any subset** of `glasses` and instantaneously flip this chosen subset. You don't have any choice in waiting in the first stage, but in the second stage you enjoy an embarrassment of riches of  $2^n$  possible moves for  $n$  glasses! Continue such two-stage moves to best measure the remaining  $t - m$  minutes.

This function should return **the smallest possible number of individual hourglass flips** needed to measure exactly  $t$  minutes, or `None` if this exact measurement is impossible. The base cases of recursion are when  $t$  equals zero or when exactly  $t$  minutes remain in some upper chamber (no flips are needed), or when  $t$  is smaller than the smallest time remaining in the upper chamber of any hourglass (no solution is possible). Otherwise, update all hourglasses to their new states after  $m$  minutes, and loop through all possible subsets of `glasses` to flip. For each such subset, recursively construct the optimal sequence of moves to measure the remaining  $t - m$  minutes, and combine these moves to the best solution.

glasses	t	Expected result
$[(7, 0), (11, 0)]$	15	2 ( <a href="#">see here</a> )
$[(4, 0), (6, 0)]$	11	None
$[(7, 1), (10, 4), (13, 1), (18, 4)]$	28	3
$[(13, 1), (14, 3)]$	22	16
$[(10, 1), (13, 4)]$	15	None

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

# Bulgarian cycle

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

When the total number of pebbles in the piles is the  $k$ :th **triangular number**, iteration of the earlier **Bulgarian solitaire** problem is guaranteed to reach a **steady state** whose individual pile sizes are the integers 1 to  $k$ , each exactly once. Started with some other number of pebbles, this iteration will never reach such a steady state. Instead, the system will eventually re-enter some state that it has already visited, and from there keep repeating the same **limit cycle** of states. For example, starting from  $[2, 3, 2]$ , the system goes through states  $[1, 2, 1, 3]$ ,  $[1, 2, 4]$ ,  $[1, 3, 3]$ , and  $[2, 2, 3]$ . Since this last state  $[2, 2, 3]$  is equal to  $[2, 3, 2]$  that we have already seen before, this completes the limit cycle of length four. Note that the cycle does not need to end at the starting state, but it can end in any one of the states later in this sequence.

This function should compute the length of the limit cycle starting from the given `piles`. However, note that this limit cycle does not necessarily include the initial `piles`, but may actually begin some number of states down the road! The ingenious [tortoise and hare algorithm](#) of Robert Floyd detects that iteration has returned to some previously seen state, without having to store the entire sequence of states explicitly in memory. This algorithm only needs to keep track of two separate current states called **tortoise** and **hare**. Both start at the same state of `piles`, and proceed through the same sequence of states. However, the hare takes two steps for every step taken by the tortoise.

Akin to two long distance runners racing towards a stadium some unknown distance ahead, even the tortoise will eventually reach the stadium. Running around the stadium track, it will be inevitably overtaken by the hare, and the position where these two animals meet must be part of the limit cycle. Then, just have the hare count its steps around the stadium one more time while the tortoise takes a nap to mark the spot, so that the hare knows when to stop counting.

piles	Expected result
$[1, 3, 2]$	1
$[2, 3, 2]$	4
$[4, 4, 3, 3, 2, 1]$	6
$[17, 99, 42]$	18
$[n*n \text{ for } n \text{ in range}(1, 51)]$	293

## Square it out amongst yourselves

```
def cut_into_squares(w, h):
```

A rectangle is defined by its width  $w$  and height  $h$ , both positive integers. We allow a rectangle to be cut into two smaller rectangles with either a straight horizontal or a straight vertical cut at any integer position. For example, one of the possible ways to cut the rectangle  $(5, 8)$  in two pieces would be to cut it into  $(2, 8)$  and  $(3, 8)$ . Another way would be to cut it into two pieces is into  $(5, 4)$  and  $(5, 4)$ . And many others, too many to list here. The resulting smaller pieces can then be further cut into smaller pieces, as long as the length of the side being cut is at least two to allow a cut. (Also, at the risk of disappointing the fans of Edward de Bono, you are not allowed to cut multiple pieces in one motion of the blade by stacking pieces on top of each other.)

Your task is to keep cutting the given  $w$ -by- $h$  rectangle into smaller pieces until each piece is a **square**, that is, the width of each piece equals its own height. This is always possible, since you could just keep cutting until the size of each piece has become 1-by-1. However, this function should return the smallest number of cuts that makes each piece a square.

This problem is best solved with recursion. Its base cases are when  $w==h$  for a piece that is already a square so you return 0, and when  $\min(w, h)==1$  allowing no further cuts on the shorter side, return  $\max(w, h)-1$ . Otherwise, loop through the possible ways to cut this piece into two smaller pieces, recursively computing the best possible way to cut up these two pieces. Return the number of cuts produced by the optimal way of cutting this piece. Since this branching recursion would visit its subproblems exponentially many times, you will surely want to sprinkle some `@lru_cache` memoization magic on it to downgrade that exponential tangle into a mere quadratic one.

w	h	Expected result
4	4	0
4	2	1
7	6	4
91	31	10
149	139	19

## Sum of distinct cubes

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

Positive integers can 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  $a$  followed by the lexicographically highest representation of the rest of the number  $n - a * a * a$ . For example, called with  $n = 1729$ , this function should return `[ 12, 1 ]` instead of `[ 10, 9 ]`. If it is impossible to express  $n$  as 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 leads to a better outcome. The base case is when  $n$  equals zero. Otherwise, loop **down** the possible values that could serve as the first element  $a$  in the result. For each such  $a$ , break down the rest  $n - a * a * a$  recursively into a sum of distinct cubes using only integers smaller than  $a$ . 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 find the largest integer whose cube is not larger than  $n$ .

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

This problem is intentionally restricted to positive integers to keep the number of possibilities 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 become a [highly nontrivial problem...](#)

## 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 points that are maximal despite that fact that they 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 our problem the royalty is more subdued so that kings capture a piece only one step in the four diagonal directions (the list `[(-1, 1), (1, 1), (1, -1), (-1, -1)]` might come handy here) 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 swoop.

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

# Collatzy distance

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

From a positive integer  $n$ , you are allowed to move into either  $3*n+1$  or  $n//2$  as a single step. Even though these formulas were obviously inspired by the wonderfully chaotic [Collatz conjecture](#) seen in the lecture on unlimited iteration, in this problem the parity of  $n$  does not tie your hands to use just one of these formulas; either formula can be applied to any integer  $n$ , regardless of its parity. This function should determine how many steps are needed to get from `start` to `goal` by taking the shortest possible route. All such problems can be solved with [breadth-first search](#), presented here as a preview for a later course that covers **graph traversal** algorithms. (Take this problem now in the same spirit as you would watch a Marvel post-credits teaser.)

The algorithm visits all integers reachable from `start` advancing in **layers**, so that each layer consists of integers at the same distance from the `start`. To begin, the zeroth layer is the singleton list `[start]`, all numbers at the distance zero from `start`. (Not too many of those!) Then, each layer  $k+1$  consists of integers  $3*n+1$  and  $n//2$  where  $n$  steps through the numbers in layer  $k$ , except that any numbers already seen in any previous layers are ignored. For example, if `start=4`, the first four layers are `[4]`, `[13, 2]`, `[40, 6, 7, 1]` and `[121, 20, 19, 3, 22, 0]` for the reachable numbers at distances from zero to three from the `start`, respectively. (As you see here, typical search problems see these layers growing exponentially along  $k$ .)

Keep generating layers until the `goal` first appears, at which point you return the current layer number as your answer. (In practical implementations of breadth-first search, a single **first-in-first-out queue** stores the nodes that the algorithm has discovered but not yet visited, instead of explicitly processing each layer in a separate inner loop.)

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

# Van Eck sequence

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

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

- If the position  $i - 1$  was the first appearance of that term  $x$  in the sequence so far, the term in the current position  $i$  equals zero.
- Otherwise, now that the previous term  $x$  has now appeared at least twice in the sequence, the term in the current position  $i$  is equal to 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 every new term.

Unless you want to spend the rest of your day waiting for the automated tester to finish, this function should not repeatedly bend over backwards to look through the already generated sequence for the most recent occurrence of each term. You should instead use a Python dictionary to remember the terms previously seen in the sequence, mapped to the positions where they made their most recent appearance. With such dictionary, you don't need to explicitly store the entire sequence, but only the previous terms and positions that are relevant for your future decisions.

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 all top level calls from the tester, `tabu` is 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	'aaahabaalabbloadabcdfthadcabdeaddtdryadebbsfmtgnubcfibabeamablschaeraboafbibobaccmlxxiseanaambdspace'

# Balanced ternary

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

The integer two and its powers abound all over computing whereas the number three seems to be mostly absent, at least until **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 often opens a fresh spring of computational complexity.

Integers are normally written out in base ten, each digit giving the coefficient of the power of ten in that position. Computers internally encode integers in the simpler (at least 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, be it positive or negative, can be broken down into a sum of signed powers of three exactly one way. Furthermore, the balanced ternary representation is symmetric around zero; the representation for  $-n$  can be trivially constructed 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 can 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**8	[129140163, -43046721, 14348907, -531441, 177147, -59049, -19683, -6561, -2187, -729, 243, -81, -9, 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 you 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. DUCY?

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 is dictated by the rules of the actual game of crag the same way as in the more famous dice game of Yahtzee.

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 here, since several rolls might fit in the same category, and yet are not equally good choices with respect to the remaining categories. Therefore, you need to process the list of rolls recursively, each level of recursion assigning some still unused category to the current roll. For each such category, recursively compute the best possible way to assign unused categories to the remaining rolls. The best category for the current roll is then the one that maximizes the sum of the value for the current roll and this recursively computed solution for the remaining rolls.

To speed up this recursion, you can keep track of the best solution that you have found so far. When the current branch of the recursive search has a chance of beating that best solution, you can treat the current branch as having a solution, as there is no point wasting perfectly good processor cycles computing the exact value of something that cannot affect the final answer.

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

## Painted into a corner

```
def cut_corners(points):
```

As in the earlier problem, for a set of points in the two-dimensional integer grid, a **corner** is three points of the exact form  $(x, y)$ ,  $(x, y + h)$  and  $(x + h, y)$  for some  $h > 0$  for its **tip** and its two **wings**. Given a list of `points` sorted by  $x$ -coordinates, ties resolved by  $y$ -coordinates, this function should return the minimum number of points whose removal from `points` makes all corners disappear. Obviously, at least one of the three points of every corner must be removed. However, whenever several corners share a common point, removing that shared point makes all those corners disappear! This function should therefore be judicious in its choices of which points to remove, as the greedy way is not always the optimal way.

This function should start by creating a list of all corners among the given `points`, each corner represented as a tuple  $(i, j, k)$  of the **position indices** of the `points` that form its tip and its wings. Then, call the recursive function that determines the minimum number of points to remove to get rid of all corners. At each level of recursion, this function should choose any one of remaining corners whose three points are still intact. It makes no difference which corner you choose, since you must remove at least one point from every corner anyway, either now or later. As a general rule in life, whenever there is no choice in some matter, it can never harm you to take the bull by the horns and just do it! Try removing each of these three points separately in a for-loop, and for each potentially removed point, recursively examine the best way to eliminate the corners that still remain in that scenario. Out of those three, use the best way that works.

Sets and dictionaries should be used to speed up the decision making at all levels of this recursion. Furthermore, if some corner contains at most one point that is shared with some other corner, there is little point (heh) in branching the search for that corner, since removing that corner can never result in a worse global outcome than removing any of the two other points.

points	Expected result
<code>[(2, 2), (2, 5), (6, 2)]</code>	0
<code>[(2, 2), (2, 5), (5, 2)]</code>	1
<code>[(1, 0), (1, 3), (1, 4), (2, 0), (2, 2), (2, 3), (4, 0), (5, 0)]</code>	3
<code>[(x, y) for x in range(5) for y in range(5)]</code>	8

## 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 again starting from zero. Since this is the last problem of this problem set, to symbolize your reach towards the infinite as you realize that you can simply ignore the arbitrary laws of imposed by society of 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 seem like you could wait it out standing on your head. Of course, our universe would not have enough atoms to encode  $S_n$  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 dynamically expand as needed, and ride the [self-similar fractal nature of the infinite Fibonacci word](#) through the finish line.

k	Expected result
0	'0'
1	'1'
10	'0'
$10^{**}6$	'0'
$10^{**}100 + 1$	'1'
$10^{**}10000$	'0'
$1234^{**}5678$	'0'

## Bonus problem 110: Reverse the Rule 110

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

In [elementary cellular automata](#), [Rule 110](#) stands tall for being the only rule proven to be **computationally universal**, along with its mirror image counterpart Rule 124. Teetering at [the borderline between chaos and order](#) where the really interesting patterns get to temporarily exist, repeated iterations of this **Turing-complete** rule produce emergent fractal patterns that could, given enough time and memory, encode all possible computations that can exist.

The function to iterate an arbitrary elementary cellular automaton would have made an excellent problem earlier in this collection. But we didn't come all the way up here for something that trivial, oh no. Instead, this function must effectively **operate in reverse** by computing the `previous` state from which the Rule 110 would produce the `current` state when applied one step forward! As much as we would like to be able to turn back time, we can only go forward with the arrow of time. Simulating the reversal of the process usually makes this task exponentially harder compared to simulating the same process in the forward direction.

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 `current` states that are impossible to produce from any `previous` state with Rule 110 (so called [Gardens of Eden](#)), this function should return `None`.

Straightforward computations can suddenly become complicated when performed backwards from the expected results to the arguments that would produce those results 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 each of the two values 0 and 1 one at the time, recursively filling in the rest of the `previous` state for each value e.

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]

## Bonus problem 111: Aye, eye, I

```
# for Liz: Every day with you was a Wednesday
def wordomino(state, words):
```

Your nemesis, unceremoniously de-platformed in an earlier problem, has crawled her way out of Scylla's janitorial end in style of *Shawshank Redemption*. Having learned her life lesson about certain dishes best served with tea and a cookie, your nemesis will now cleverly engage you in a friendly combinatorial game of "word dominos" that this author made up just for this problem.

Two players make alternate moves constrained by a shared list of legal four-letter English words. The `state` of the game so far is some string of letters. On her turn, each player must extend the current `state` with one letter from *a* to *z* so that the block of last **four** characters of this new extended state is one of the legal words. However, to guarantee the eventual termination of this match, the block of last **three** characters of this extended state may not previously appear as that block anywhere in the current `state`. The player unable to extend the `state` on her turn will fall into the formless void that separates this course from its successor CCPS 209 *Computer Science II*.

Same as in the earlier combinatorial game of **subtracting a square**, a `state` of this game is **hot (winning)** if and only if at least one legal move from that `state` leads to a **cold (losing)** successor state. Your function should therefore recursively evaluate the successor states that can be reached from the current `state` by extending it with a single letter. For a cold `state`, this function should return `None`. For a hot `state`, this function must return its **alphabetically lowest** winning move that leads to a cold successor state that gives your opponent no chance to turn the tide.

state	Expected result
'perp'	None
'dunes'	'e'
'tropens'	'e'
'omethangairedeaditalitelaneten'	None
'damahabetashagingairana'	'l'

With 7186 four-letter words in `words_sorted.txt` to choose from, the branches of the minimax search tree for this game can randomly get pretty deep. Fortunately, most of these moves tend to be pretty much forced (well, at least semi-forced) along the way. This keeps down the effective **branching factor**, and doing so keeps the eventual running time tolerable.



## Bonus problem 112: Count domino tilings

```
#for Tom: Systematic signals rein in unimaginable possibilities
def domino_tile(rows):
```

In [domino tiling](#) problem, a room of contiguous unit squares is to be **tessellated** with blank 2-by-1 domino tiles, each such tile placed either vertically or horizontally. Of course, this task can be possible only for even room sizes. This necessary condition by itself does not yet guarantee the existence of some tiling, as seen in the famous [mutilated chessboard](#) problem.

In our version of domino tiling, the shape of the room consists of vertically stacked `rows` that all share the common straight line left wall. The list of `rows` contains the widths of the individual rows, as measured in squares. This formulation therefore includes not only the usual rectangular rooms version of this problem, but even the more general [Young diagrams](#) as a special case.

This function should compute the number of ways that the room described by given `rows` can be tiled with dominoes. As with all combinatorial problems of this spirit, the solution requires careful consideration analogous to the Serenity Prayer; since you have no choice about covering every square of the room, you might as well grab the bull by the horns and recursively fill that room in order so that each level of recursion quickly finds the first uncovered square. There, recursion branches in two directions to tally the ways to complete the filling after placing a horizontal or a vertical tile on that square, depending on what your previously filled dominoes still allow. Since this recursion exponentially repeats the same subproblems all over the place, some cha-ching scheme is absolutely necessary to rein in the **combinatorial explosion**. Once you get the logic working correctly, optimize everything that you possibly can.

rows	Expected result
[ 2, 2 ]	2
[ 4, 3, 2, 1 ]	0
[ 5, 3, 4, 2 ]	7
[ 8, 8, 8, 8, 8, 9, 9 ]	1895245
[ 12, 12, 11, 11, 11, 11, 11 ]	13571985717
[ 6 ] * 100	81111224923796326743146807111807488 45564827012150625388192928607034509

The number of ways to tile the given room generally grows exponentially along the size of that room. Some of these tilings appear more aesthetic by exhibiting symmetries or other such visually pleasant forms. Enumerating all tilings under additional aesthetic constraints may either be easier or harder than enumerating the unconstrained tilings. For a good example of this phenomenon, see the Project Euler problem "[Tatami-free rooms](#)".

## Bonus problem 113: Invaders must die

```
# for Ezzat: Intuition beats book learning when the stakes get high
def laser_aliens(n, aliens):
```

Having escaped from the icy  $n$ -by- $n$  chessboard in the earlier problem "Where no one can hear you bounce", Bishop has called in reinforcements to eliminate the scourge of aliens from this system once and for all. After they arrive, the space marines take positions somewhere along the edges of this chessboard. Each marine wields a laser cannon that can blast a single shot through exactly one row or column of this board, that shot eradicating every alien positioned anywhere on that row or column. Given the positions of `aliens` on this board as a list of tuples `(row, col)`, sorted by rows and breaking ties by columns, this function should compute and return the minimum number of space marines needed to eliminate every alien.

The key of solving this kind of combinatorial search and optimization problems lies in relaxing and allowing things that you don't have a choice about to constrain your search, instead of pointlessly branching at every turn to redundantly explore an exponential number of possibilities. For any row that still contains aliens, you must either shoot a laser horizontally through that row, or shoot a laser vertically through **every** column that contains an alien in that row. This gives you a nice two-way choice for each row. Furthermore, during shooting through all the columns, clever updating and downdating of some **auxiliary data structures**, maintained on the side throughout the recursion to quickly tell you the positions of the aliens in the given rows or columns, will massively curb down branching in future rows with aliens in those same columns. You also have a choice of the order in which you process the individual rows...

n	aliens	Expected result
3	[(0, 1), (0, 2)]	1
4	[(0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]	2
5	[(0, 3), (3, 2), (3, 3), (4, 2)]	2
8	[(1, 4), (3, 6), (4, 6), (4, 7), (5, 0), (5, 4), (5, 6), (5, 7), (7, 0), (7, 7)]	4
10**6	[(42, 42), (999999, 999999)]	2

## Bonus problem 114: Stepping stones

```
# for Suzy: La dureté de deux diamants nous libérera aussi
def stepping_stones(n, ones):
```

This problem is adapted from the [Numberphile](#) video "[Stones on an Infinite Chessboard](#)" that the reader should first watch to learn how this excellent puzzle works. To keep this problem manageable, we will restrict the placement of stones into a finite  $n$ -by- $n$  chessboard whose edges the stones cannot cross into the void. As usual, rows and columns are numbered starting from zero.

Starting from stone number 2, consecutively numbered stones are placed on board in ascending order so that the moment that the stone number  $k$  is placed somewhere, the stones previously placed into its up to eight neighbouring locations add up to exactly  $k$ . Given the board size  $n$  and the initial locations of the brown ones, this function should return the highest numbered stone in the longest sequence of stones that can be placed consecutively on the board within the rules of this puzzle. For example, as demonstrated by Neil Sloane in the above video, you can snake your way up to the stone number 16 from two brown stones placed diagonally with one empty space between them, provided that the board size is large enough to contain the resulting pattern. However, no matter how you twist and turn, no placement of stones will get you any further than that.

To speed up this search, you should maintain a list of sets whose  $k$ :th element is a set of positions whose current sum of neighbouring stones equals  $k$ . When placing the stone number  $k$  on the board, your function can quickly loop through only the positions in that particular set, instead of examining the entire board each time. Placing a new stone somewhere updates the sums of the currently empty cells in its neighbourhood, making these neighbouring cells jump from one set to another, and then jump back into the original set once the recursive search to place the sequence of stones starting from the stone  $k+1$  has returned.

n	ones	Expected result
4	[(0, 0), (3, 3)]	1
5	[(0, 1), (1, 1), (2, 2)]	10
6	[(2, 0), (5, 3), (1, 3), (0, 0)]	19
7	[(4, 4), (1, 2), (6, 5), (1, 5), (4, 2), (1, 4), (4, 5)]	27
8	[(4, 4), (1, 3), (1, 2), (1, 1), (4, 6), (1, 4), (1, 0), (4, 1)]	29

## Bonus problem 115: Ex iudiciis, lux

```
#for Joanne: Light heart, light touch
def illuminate_all(lights):
```

Each individual light in the given row of `lights` has a **brightness** that determines how many adjacent positions to both left and right the illumination from that light reaches, in addition to illuminating the position of that light itself. (A light whose brightness is zero will therefore illuminate only itself, but doesn't have anything to shine on its neighbours.) Your task is to turn on as few individual lights as possible so that every position of the entire row is illuminated by at least one light. Since the smallest working subset of lights is not necessarily unique, your function should only return the number of lights needed to illuminate the entire row.

To organize this search into a workable recursion, start by noting that same as in the **subset sum** recursion example seen in the class, you have a basic "take it or leave it" two-way choice for the last light in the list. You can explore both branches of this choice recursively, and use the branch that produces a better solution. Furthermore, many individual lights can be discarded immediately since the optimal solution to the problem will never need those lights to achieve anything that it could not just as well achieve without them.

Your recursion should have a second parameter to keep track of how much light is still "owed" to the non-illuminated positions skipped from the right, or how much extra light is spilling into the positions in the list from the lit up positions skipped from the right. Since the recursion will keep repeating the same subproblems in exponentially many different ways, use some `@lru_cache` magic to eliminate the exponential blowup of running time.

lights	Expected result
[2, 3, 3, 2]	1
[1, 0, 1, 0]	2
[0, 0, 0, 1, 2, 0, 2, 0, 0]	4
[0, 0, 2, 3, 4, 1, 0, 1, 2, 2, 5, 2, 3, 0, 1, 0, 0]	3
[0, 2, 2, 2, 2, 0, 1, 2, 0, 0, 1, 1, 1, 0, 1, 0, 0, 2, 1, 1, 0, 1, 1, 2, 0, 0, 1]	8

## Bonus problem 116: Flatland golf

```
#for Sharon: First there is hair, then there is no hair, then there is
def best_clubs(holes):
```

The denizens of [Flatland](#) wish to embark on a friendly game of golf. Since their reality simply has no room for fancy golf resorts with spectacular views, Flatland golf courses are one-dimensional lines, with the hole located at some distance away from the teeing point at the origin. Each Flatland linksman has room for only two clubs in the bag. In the Flatland physics version of golf, the ball will always travel the exact same distance when swung with the same club. Each club is therefore rated for the fixed distance that it will always make the ball travel.

Since one-dimensional golf courses don't take much space, it is okay to hit the ball past the hole as far as you like, and then turn back on the next stroke. For example, carrying two clubs rated for 40 and 110 meters, the hole located 210 meters away can be reached with six strokes, three of which are done with each club. In general, a hole at distance  $d$  can be reached using the clubs  $c_1$  and  $c_2$  if and only if the distance  $d$  is divisible by the **greatest common divisor** of  $c_1$  and  $c_2$ .

This function should choose the two clubs in the bag to minimize the total score through the entire course with given distances to its holes. Since this optimal pair of clubs may not be unique, this function should compute the smallest achievable total strokes for the given holes. Sure, discrete math and number theory, especially that of linear [Diophantine equations](#), will be a big help in these computations, but the test cases can be handled within the time limit with brute force with just a touch of cleverness sprinkled on.

holes	Expected result	(best clubs)
[2, 3, 4, 5]	6	5, 2
[2, 6, 7, 9]	7	9, 2
[2, 5, 7, 10, 14]	8	7, 5
[19, 42, 51, 65, 85, 85, 104]	33	19, 14
[300, 250, 200, 325, 275, 350, 225, 375, 400]	26	125, 100

This problem comes from "[536 Puzzles and Curious Problems](#)" by [Henry Ernest Dudeney](#) and [Martin Gardner](#), two historical heavyweights of recreational mathematics. Written in a simpler time when such calculations were performed by hand (and even this very problem was titled "Queer Golf", heh), the entire puzzle was just the last row of the above table. Since we now have enough computational power under our thumbs to easily crank out more computations than the entire pre-computing world executed throughout its history, the logic of this problem can be mechanized for arbitrary golf courses instead of just this particularly tricky one.