109 Python Problems for CCPS 109

This document contains the specifications for the **graded lab problems** for the course CCPS 109 Computer Science I, as taught by Ilkka Kokkarinen. All computer science teachers are free to use and adapt this material for their own courses as they see fit, with proper attribution. The instructor has collected these problems from a gallimaufry of sources ranging from the lab and exam problems of his old Java version of this course and more recent original problems to a multitude of programming puzzles and challenge sites such as LeetCode, CodeAbbey, Stack Exchange Code Golf, and Wolfram Programming Challenges, and a host of others forgotten over the years.

You can download the necessary tester and data files either from GitHub or repl.it as a zip file to work locally on your computer using IDLE, Spyder or any similar interactive environment. To test your completed functions as you write them, simply run the `tester109.py` to execute the tests for precisely those functions that you have implemented so far, in the order in which your functions appear in your source code. Each test should usually take at most a few seconds to complete. If some test takes several minutes to finish, your code is too slow so that your logic does more work than it really ought to. Streamlining your solutions is an important part of this learning process.

For the purposes of submission and grading, you **must implement all these functions in a single Python source code file** that **must** be named `labs109.py`. **Do not submit the individual functions as you go, but submit these labs all at once at the end of the course**.

SILENCE IS GOLDEN. When called, none of your functions should ever print anything on the console, but silently return the expected result. You can do some debugging outputs during the development, but remember to comment them out before submission. Your `labs109.py` file can also have an `if __name__ == '__main__'` block for your own private test cases for debugging.

All your functions may assume that their pseudo-randomly generated arguments coming from the automated tester are legal and valid as the problem specification guarantees. Your functions do not need to perform any error detection or recovery from nonsensical arguments.

All of the following lab problems have been independently solved by at least one student in the course, producing a solution whose generated checksum equaled the one computed from the instructor's private model solution. To achieve such a blessed state of affairs, students Shu Zhu Su, Mohammed Waqas and Zhouqin He deserve credit and great thanks for going far above and beyond the call of duty in submitting their solutions that revealed several errors in the original solutions and problems specifications. All remaining errors and inconsistencies are the sole responsibility of Ilkka Kokkarinen.

Ryerson letter grade

`def ryerson_letter_grade(pct):`

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

(Same as all other programming problems that follow this problem, this can be solved in various different ways. At this point of your studies, the simplest way to solve this problem would probably be just to use an **if-else ladder**.)

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

Ascending list

`def is_ascending(items):`

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

Note that the empty sequence is ascending, as is every one-element sequence, so be careful that your function returns the correct answers also 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 of that particular sequence being ascending?)

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

(In the same spirit, note how every possible universal claim made about the elements of an empty sequence is trivially true. For example, if `seq` is the empty sequence, the two claims "All elements of `seq` are odd" and "All elements of `seq` are even" are both equally true, which seems highly counterintuitive.)

Riffle

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

Given a list of `items` that is guaranteed to contain an even number of elements (note that the integer zero is an even number), create and return a list produced by performing a perfect **riffle** to the `items` by interleaving the items of the two halves of the list in an alternating fashion.

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

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

Only odd digits

`def only_odd_digits(n):`

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

Hint: to extract the lowest digit of a positive integer `n`, use the expression `n % 10`. To extract all other digits except the lowest one, use the expression `n // 10`. Or, if you don't want to be this fancy, first convert the number into a string and work there.

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

Blocks in pyramid

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

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

Here you can solve this problem in a straightforward fashion by simply looping through the `h` layers and adding up all the blocks along the way in each layer. However, if you happen to know some discrete math and combinatorics, you can come up with an analytical closed form formula for the result and compute the answers much faster that way. (There is an important general principle in this for you to ponder later on your own.)

`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`

Cyclops numbers

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

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

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

(As an extra challenge, try to solve this problem using only loops, conditions and integer arithmetic operations, without first converting the integer into a string and working from there. Note that dividing an integer by 10 effectively chops off its last digit, whereas the remainder operator `%` can be used to extract the last digit.)

Domino cycle

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

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

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

Count dominators

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

An element of `items` is said to be a **dominator** if every element to its right is strictly smaller than it. This function should count how many elements in the given list of `items` are dominators, and return that count. For example, in the list `[42, 7, 12, 9, 2, 5]`, the elements 42, 12, 9 and 5 are dominators. By this definition, the last item of the list is automatically a dominator.

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

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

Extract increasing integers from digit string

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

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

`digits`	Expected result
`'0'`	`[0]`
`'045349'`	`[0, 4, 5, 34]`
`'77777777777777777777777'`	`[7, 77, 777, 7777, 77777, 777777]`
`'122333444455555666666'`	`[1, 2, 23, 33, 44, 445, 555, 566, 666]`
`'1234567890987654321'`	`[1, 2, 3, 4, 5, 6, 7, 8, 9, 98, 765, 4321]`
`'3141592653589793238462643383279502884'`	`[3, 14, 15, 92, 653, 5897, 9323, 84626, 433832, 795028]`
`'27182818284590452353602874713526624977572470936999595749669676277240766303535475945713821785251664274274663919320030599218174135966290435729003342952605'`	`[2, 7, 18, 28, 182, 845, 904, 5235, 36028, 74713, 526624, 977572, 4709369, 9959574, 96696762, 772407663, 3535475945, 7138217852, 51664274274, 66391932003, 599218174135, 966290435729]`
`'123456789' * 100`	A list that contains 75 elements, the last one of which equals `34567891234567891234567891`

Words that contain given letter sequence

`def words_with_letters(words, letters):`

Given a list of `words` sorted in alphabetical order, and a string of required `letters`, find and return the list of precisely those words that contain the letters inside them in the exact order given, but not necessarily in consecutive positions.

`letters`	Expected result (using the wordlist `words_alpha.txt`)
`'antmneic'`	`['antiferromagnetic', 'antimagnetic', 'antimnemonic', 'aquopentamminecobaltic', 'pantomnesic', 'phantasmogenetic']`
`'unskit'`	`['underskirt', 'underskirts', 'unshockability', 'unshrinkability', 'unsinkability', 'unskaithd', 'unskaithed', 'unskirted', 'unspeakability']`
`'rupih'`	`['frumpish', 'frumpishly', 'frumpishness', 'grumpish', 'grumpishness', 'porcupinish', 'rupiah', 'rupiahs', 'trumpetfish', 'trumpetfishes']`
`'reeomy'`	A list of 54 words, the first three of which are `['adrenalectomy', 'arteriectomy', 'arteriophlebotomy']` and the last three are `['ureterotomy', 'urethrectomy', 'vertebrectomy']`

Taxi zum zum

`def taxi_zum_zum(moves):`

A Manhattan taxicab starts at the origin point `(0, 0)` of the two-dimensional integer grid, initially heading north. It then executes the given sequence of `moves`, a string made up of characters `'L'` for turning 90 degrees left (while standing in place), `'R'` for turning 90 degrees right (ditto), and `'F'` for moving one step forward according to its current heading. This function should return the final position of the taxicab in the integer grid coordinates of Manhattan.

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

(As an aside, why do these problems always seem to take place in Manhattan instead of, say, Denver where the street grid is rotated 45 degrees from the main compass axes to equalize the amount of daily sunlight on streets of both orientations? That should make interesting variation to many problems of this spirit. Of course, diagonal moves always maintain the parity of the coordinates, which makes it impossible to reach any coordinates of the opposite parity, quite like in that old joke with the punchline "Gee... I don't think that you can get there from here.")

Count growlers

`def count_growlers(animals):`

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

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

(I admit that I was pretty high when I originally thought up this problem, at least high enough to perceive the letter 't' as a tail of a happy cat held up high, and 'd' as the snout and stand-up ears of a curious dog, perhaps some kind of spitz or a similar breed. Yeah, good luck trying to unsee that now. For some reason, this problem is somehow more tricky than it seems.)

Bulgarian solitaire

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

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

This function should count how many moves are needed to convert the given initial `piles` into some permutation of first *k* integers so that each number from `1` to `k` appears as the size of **exactly one pile** in the current configuration, and return that count. (Once in this goal state, the move does not change the configuration.)

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

(This problem was inspired by the old Martin Gardner column "Bulgarian Solitaire and Other Seemingly Endless Tasks" where it was used as an example of what we would here call a while-loop where it is not immediately obvious that the loop will ever reach its goal and get to terminate. However, a clever combinatorial proof shows that this one crude move can never get stuck, but will always reach the goal after at most $k(k-1)/2$ iterations from any starting configuration.)

Scylla or Charybdis?

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

This problem was inspired by the article "A Magical Answer to the 80-Year-Old Puzzle". Your opponent is standing at the center of a one-dimensional platform that reaches `n-1` steps to both directions, with Scylla and Charybdis hungrily waiting at each end of the platform at distance `n`. The opponent gives you a `sequence` of his planned moves as a string made of two symbols `'+'` ("🎶 just a step to the ri-i-i-i-ight" 🎶) and `'-'` (move one step left). Your **adversarial** task is to find a positive integer `k` so that when executing precisely every `k`:th step of the sequence (that is, the first step taken is `sequence[k-1]`, the next step taken is `sequence[2*k-1]`, and so on), your opponent ends up at least `n` steps away from the starting point, falling to his indirectly chosen doom.

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

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

Arithmetic progression

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

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

Given a list of `elems` guaranteed to consist of positive integers listed in strictly ascending order, find and return the longest arithmetic progression whose all values exist somewhere in that sequence. Return the answer as a tuple `(start, stride, n)` of the values that define the progression. To ensure that the answer is unique for automated testing, if there exist several progressions of the same length, return the one with the smallest `start`. If there exist several progressions of equal length from the same `start`, return the progression with the smallest `stride`.

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

Tukey's ninther

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

Back in the day when computers were far slower and had a lot less RAM for our programs to burrow into, special techniques were necessary to achieve many things that are trivial today with a couple of lines of code. In this spirit, "Tukey's ninther" is a clever **approximation algorithm** from the seventies to quickly find the **median element** of an unsorted list, or at least find some element that is close to the true median with high probability. In this problem, the median element of the list is defined to be the element that would end up in the middle position, were that list sorted.

Tukey's algorithm itself is simple. Conceptually split the list into sublists of three elements each, and find the median element for each of these triplets. Collect these medians-of-three into a new list (whose length therefore equals one third of the length of the original list), and repeat this until only one element remains. (For simplicity, your function can assume that the length of `items` is always some power of three.) In the following table, each row contains the list that results from applying one round of Tukey's algorithm to the list in the next row.

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

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

Suppressed digit sum

`def suppressed_digit_sum(n):`

Given an integer `n`, compute and return the sum of all integers that can be constructed from `n` by removing one digit. For example, when called with the four-digit integer `n = 1234`, this function should add up the four terms 123 + 134 + 124 + 234, and therefore return 615. However, each of these integers should be added only once to the total, even if that integer could be constructed from `n` in more than one way. For example, given `n = 333`, this function should return 33 instead of 66.

`n`	Expected result
`1`	`0`
`42`	`6`
`7771777`	`2326731`
`123456789`	`123456780`
`4444444444`	`444444444`

Crack the crag

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

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

Given the list of pips of the three dice that were rolled, this function should compute and return the highest possible score available for those dice, under the simplifying assumption that this is the first roll in the game so that **all categories of the scoring table are still available for you to choose from**. This problem should therefore be a straightforward exercise in writing if-else ladders combined with simple sequence management.

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

(For this problem, there exist only $6^3 = 216$ possible argument value combinations, so the automated tester should finish in time that rounds down to `0.000`. However, surely you will design your if-else ladder to handle entire equivalence classes of cases together in a single step, so that your ladder consists of far fewer than 216 separate steps.)

Three summers ago

`def three_summers(items, goal):`

Given a list of positive integer `items` guaranteed to contain at least three elements with all of its **elements in sorted ascending order**, determine whether there **exist exactly three** separate `items` that together add up **exactly** to the given positive integer `goal`. Return `True` if three such integer items exist, and `False` otherwise.

You could, of course, solve this problem with three nested loops to go through all possible ways to choose three elements from `items`, checking for each triple whether it adds up to `goal`. However, this approach would get rather slow as the number of elements in the list grows larger, and the automated tester will make those lists larger.

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

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

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

Count all sums and products

`def count_distinct_sums_and_products(items):`

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

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

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

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

Sum of two squares

`def sum_of_two_squares(n):`

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

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

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

Try a spatula

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

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

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

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

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

Carry on Pythonista

`def count_carries(a, b):`

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

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

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

First missing positive integer

`def first_missing_positive(items):`

Given a list of `items` that are all guaranteed to be integers, but given in any order and with some numbers potentially duplicated, find and return the smallest positive integer that is missing from this list. (Zero is neither positive or negative, which makes **one** the smallest positive integer.)

This problem could be solved with two nested loops, the outer loop counting up through all positive numbers in ascending order, and the inner loop checking whether the current number exists inside the list. However, such crude **brute force approach** would be very inefficient as these lists get longer. Instead, perhaps the built-in `set` data type of Python could be used to remember the numbers you have seen so far, to help you determine the missing integer quickly in the end. Or some other way might also work in an equally swift fashion, if not even faster...

`items`	Expected result
`[7, 5, 2, 3, 10, 2, 99999999999, 4, 6, 3, 1, 9, 2]`	`8`
`list(range(100000, 2, -1))`	`1`
`random.shuffle(list(range(1000000)))` (of course you must `import random` before this call)	`1000000`
`[6, 2, 42, 1, 7, 9, 99999999999, 8, 4, 64]`	`3`

Check your permutation

`def is_permutation(items, n):`

Given a list of `items` that is guaranteed to contain exactly `n` integers as its elements, determine whether this list is a **permutation** of integers from 1 to `n`, that is, it contains every integer from 1 to `n` exactly once. If some number found in `items` is less than 1 or greater than `n`, or if some number occurs more than once in `items`, return `False`. Otherwise the given list truly is a permutation, and your function should return `True`.

In a similar spirit to the previous problem of finding the first missing positive integer from the given list of `items`, this problem could easily be solved using two nested loops, the outer loop iterating through each expected number from 1 to `n`, and the inner loop checking whether the current number occurs somewhere in `items`. However, this approach would again get very inefficient for large values of `n`. A better solution once again trades more memory to spend less time, using some data structure to remember the expected numbers that it has already seen before, which allows this function to make its later decisions more efficiently.

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

Tribonacci

```
def tribonacci(n, start = (1, 1, 1)):
```

The famous and important Fibonacci series is defined to start with values `(1, 1)` and from there, each element in the sequence equals the sum of the previous two elements. However, a cute variation aptly called the **Tribonacci series** works otherwise the same way, but starts with a triple of values `(1, 1, 1)` and from there onwards, each element equals the sum of the previous **three** elements. Tribonacci numbers grow quite a bit faster than Fibonacci numbers, the first twenty being 1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, 2209, 4063, 7473, 13745, 25281, and 46499.

Given the element position `n` and optionally some other starting triple of values than `(1, 1, 1)`, compute and return the element in position `n` of the resulting Tribonacci series. As usual, the position numbering in a sequence starts from zero. Since `n` can get pretty large, you can't solve this problem with recursion, since your function would crash with a stack overflow once the recursion limit is exceeded. Instead, you need to remember the three most recent values, and in a loop, use them to compute the next value of the sequence.

Make sure that your function works correctly in the edge cases `n == 0` and `n == 1`, especially since the three elements of `start` are not necessarily equal.

`n`	`start`	Expected result
`0`	`(1, 1, 1)`	`1`
`10`	`(1, 1, 1)`	`193`
`10`	`(4, 3, 2)`	`542`
`100`	`(2, 2, 2)`	`2541432357740055042988699 62`
`1000`	`(1, 1, 1)`	(a positive integer that consists of a total of 265 digits, the first five of which are `19433` and the last five are `16217`)
`100000`	`(1, 1, 1)`	(a positive integer that consists of a total of 26465 digits, the first five of which are `38322` and the last five are `33281`)

Nearest smaller element

`def nearest_smaller(items):`

Given a list of integer `items`, create and return a new list of the exact same length so that each element is replaced with the nearest element in the original list whose value is smaller. If no smaller elements exist, the element in the result list should simply remain as it were in the original list. If there exist smaller elements to both directions that are equidistant from that element, you should resolve this by using the smaller of these two elements to make the results testable.

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

Interesting, intersecting

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

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

(Hint: it is actually much easier to determine that the two squares do **not** intersect, and then negate that answer. Two squares do not intersect if one of them ends in the horizontal direction before the other one begins, or if the same thing happens in the vertical direction.)

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

Keep doubling

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

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

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

Remove each item after its *k*th occurrence

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

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

Hint: loop through the `items`, maintaining a dictionary that remembers how many times you have already seen each element, updating this count as you go and appending each element to the `result` list only if its count is still less than or equal to `k`. Note also the counterintuitive but still completely legitimate edge case of `k==0` that has a well defined answer of an empty list!

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

First item that is preceded by *k* smaller items

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

Given a list of `items`, find and return the value of the first element that has at least *k* preceding smaller elements in `items`, in the sense of the ordinary Python order comparison `<` applied to these items. If there is no such element anywhere in `items`, this function should return `None`. Note that the `k` smaller items do not need to be consecutive immediate predecessors of the current item, but can be any `k` items from the initial prefix of the list before the current element.

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

Maximum difference sublist

```
def maximum_difference_sublist(items, k = 2):
```

Given a list of integers, find and return the sublist of *k* consecutive elements where the difference between the smallest element and the largest element is the largest possible. If there are several sublists of *k* elements in `items` so that all these sublists have the same largest possible difference, return the sublist that occurs first.

`items`	`k`	Expected result
`[42, 17, 99, 12, 65, 77, 11, 26]`	`4`	`[42, 17, 99, 12]`
`[42, 17, 99, 12, 65, 77, 11, 26]`	`5`	`[99, 12, 65, 77, 11]`
`[36, 14, 58, 11, 63, 77, 46, 32, 87]`	`1`	`[36]`
`[36, 14, 58, 11, 63, 77, 46, 32, 87]`	`5`	`[14, 58, 11, 63, 77]`
`list(range(10**6))`	`5`	`[0, 1, 2, 3, 4]`

(As you can see in the example in the fourth row, the largest element of the list `87` is not necessarily part of the sublist `[14, 58, 11, 63, 77]` that contains the maximum difference. Ditto for the smallest element in the list.)

What do you hear, what do you say?

`def count_and_say(digits):`

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

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

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

And sometimes why

```
def disemvowel(text):
```

To "disemvowel" some piece of `text` to obfuscate its meaning but still potentially allowing somebody to decipher it provided that they really want to put in the effort to unscramble it, we simply remove every vowel character from that string, and return whatever remains. For example, disemvoweling the string `'hello world'` would produce `'hll wrld'`. The letters *aeiou* are vowels as usual, but to make this problem more interesting, we will use a crude approximation of the correct rule to handle the problematic letter *y* by defining that the letter *y* is a vowel whenever the text contains one of the proper vowels *aeiou* immediately to the left or to the right of that *y*.

For simplicity, it is guaranteed that `text` will contain only lowercase letters (although possibly accented) and punctuation characters, but no uppercase characters. To allow the automatic tester to produce consistent results, your disemvoweling function should not try to remove any accented vowels.

`text`	Expected result
`'may the yellow mystery force be with you!'`	`'m th llw mystry frc b wth !'`
`'my baby yields to traffic while riding on her bicycle'`	`'my bby lds t trffc whl rdng n hr bcycl'`
`'sydney, kay, and yvonne met boris yeltsin in guyana'`	`'sydn, k, nd yvnn mt brs ltsn n gn'`
`'yay'`	`''`

(As the above examples illustrate, handling the letter *y* correctly in all cases would require pronunciation information above and beyond what we can possibly have available that would allow us to determine whether some letter *y* represents a vowel or a consonant sound. Those of you who are interested about this sort of stuff can check out this Stack Exchange discussion thread.)

Rooks on a rampage

`def safe_squares_rooks(n, rooks):`

On a generalized *n*-by-*n* chessboard, there are some number of **rooks**, each rook represented as a two-tuple (`row, column`) of the row and the column that it is in. (The rows and columns are numbered from 0 to *n* - 1.) A chess rook covers all squares that are in the same row or in the same column as that rook. Given the board size `n` and the list of `rooks` on that board, count the number of empty squares that are safe, that is, are not covered by any rook.

(Hint: count separately how many rows and columns on the board are safe from any rook. The result is simply the product of these two counts.)

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

Bishops on a binge

`def safe_squares_bishops(n, bishops):`

On a generalized *n*-by-*n* chessboard, there are some number of **bishops**, each bishop represented as a tuple `(row, column)` of the row and the column of the square that contains that bishop. (The rows and columns are numbered from 0 to *n* - 1.) A chess bishop covers all squares that are on the same diagonal with that bishop arbitrarily far into any of the four diagonal compass directions. Given the board size `n` and the list of `bishops` on that board, count the number of empty squares that are safe, that is, are not covered by any bishop.

(Hint: to quickly check whether two squares `(r1, c1)` and `(r2, c2)` are in some diagonal with each other, you can use the test `abs(r1 - r2) == abs(c1 - c2)` to determine whether the horizontal distance of those squares equals their vertical distance, which is both necessary and sufficient for those squares to lie on the same diagonal. This way you don't need to write the logic separately for each of the four diagonal directions, but one test can handle all four diagonal directions in one swoop.)

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

The hand that's hard to get

`def hand_is_badugi(hand):`

In the exotic **draw poker** variation of badugi, the four cards held by the player form the titular and much sought after **badugi** if no two cards in that hand have the same suit or the same rank. Given the `hand` as a list of four elements so that each card is encoded as a tuple of strings `(rank, suit)` the exact same way as in the `cardproblems.py` example program, determine whether that hand is a badugi.

hand	Expected result
`[('queen', 'hearts'), ('six', 'diamonds'), ('deuce', 'spades'), ('jack','clubs')]`	`True`
`[('queen', 'hearts'), ('six', 'diamonds'), ('deuce', 'spades'), ('queen','clubs')]`	`False`
`[('queen', 'hearts'), ('six', 'diamonds'), ('deuce', 'spades'), ('jack','spades')]`	`False`

Revorse the vewels

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

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

(Hint: one straightforward way to solve this starts with collecting all vowels of `text` into a separate list. After that, iterate through all positions of the original `text`. Whenever the current position contains a vowel, take one from the end of the list of the vowels. Convert that vowel to either upper- or lowercase depending on the case of the vowel that was originally in that position. Otherwise, take the character from the same position of the original `text`.)

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

Scrabble value of a word

```
def scrabble_value(word, multipliers = None):
```

Compute the point value of the given `word` in Scrabble, using the standard English version letter points. The parameter `multipliers` is a list of integers of the same length as `word`, telling how much the value of the letter in each position gets multiplied by. It is guaranteed that `word` consists of lowercase English letters only. If `multipliers == None`, your function should use the default multiplier value of one for each position.

To spare you from error-prone and tedious typing, the following dictionary should come handy: `{'a':1, 'b':3, 'c':3, 'd':2, 'e':1, 'f':4, 'g':2, 'h':4, 'i':1, 'j':8, 'k':5, 'l':1, 'm':3, 'n':1, 'o':1, 'p':3, 'q':10, 'r':1, 's':1, 't':1, 'u':1, 'v':4, 'w':4, 'x':8, 'y':4, 'z':10 }`

`word`	`multipliers`	Expected result
`'hello'`	`None`	`8`
`'world'`	`[1, 3, 1, 1, 1]`	`11`
`'hexahydroxycyclohexane'`	`None`	`66`

When I zig, you gotta zag

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

This function creates **a list of lists** that represents a two-dimensional grid with the given number of `rows` and `cols`. This grid should contain the integers from `start` to `start + rows * cols - 1` in ascending order, but so that the elements of every odd-numbered row are listed in descending order, so that when read in ascending order, the numbers zigzag through the two-dimensional grid.

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

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

Calkin-Wilf sequence

`def calkin_wilf(n)`

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

Your function should return the `n`:th element of this sequence. First, create a new instance of `deque` and `append` the first fraction `1/1` to "prime the pump", so to speak, to initiate the production of the values of this sequence. Then, repeat the following procedure `n` times. Pop the fraction currently in front of the queue using the `deque` method `popleft`, extract its numerator and denominator `p` and `q`, and push the two new fractions `p / (p + q)` and `(p + q) / q` to the back of the queue. Return the fraction object that was popped in the final round. (Actually, once you reach `n//2+1`, you could stop pushing in any new values, since at that point, the queue already contains the final result...)

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

Kempner series

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

As we have learned in various math courses, the *n*:th harmonic number is equal to the sum of the reciprocals of the first *n* integers, that is, $H_n = 1/1 + 1/2 + 1/3 + ... + 1/n$. As *n* approaches infinity, so does the harmonic number H_n, although quite slowly, but with all kinds of useful properties.

A whimsical variation known as Kempner series works otherwise the same way, but adds up only those fractions that **do not contain the digit nine** anywhere in them. Your function should compute and return K_n, the *n*:th term of the Kempner series by adding up the reciprocals of the first *n* positive integers that do not contain the digit nine. For example, to compute K_{10}, you would need to add up $1/1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 + 1/10 + 1/11$, the first ten such reciprocals.

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

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

That fifth overtrick

```
def bridge_score(suit, level, vul, dbl, made):
```

In the card game of duplicate bridge, the score gained from each successfully made **contract** depends on five parameters: the `suit` and `level` of the contract (for example, "three clubs" or "six notrump"); whether the pair is currently **`vul`**nerable; whether the contract has been **doubled** or **redoubled** (here indicated by whether parameter `dbl` equals `''`, `'X'` or `'XX'`); and how many tricks were `made` (above the book of six tricks after which the trick counting actually begins). To keep this simple, your function only needs to handle successful contracts with `made >= level`.

This problem is an exercise of executing a series of decisions that lead to the correct result in all possible situations. The exact details of this scoring calculation are explained in section 2 of the Wikipedia page "Bridge Scoring" that you can design and write your logic from. (Rubber and honour bonuses are **not** part of duplicate bridge scoring.) The handy page "Scoring table" by the bridge writer Richard Pavlicek lists the precomputed scores for every possible successfully made contract.

Note that the scoring always works the same for both **minor suits** of clubs and diamonds, so you can handle both with the same logic. The same holds for both **major suits** of hearts and spades. This table is split into two sections, the first for **partscore contracts** (the raw trick values without overtrick bonuses add up to less than a hundred) and the second for **game contracts** (the raw trick values add up to hundred or more, earning a bonus of 500 points if vulnerable, 300 if not. (Note also that in bridge you only get these game and slam bonuses only if you actually bid that game or slam, and otherwise it would not really help you much to take even all thirteen tricks!)

suit	level	vul	dbl	made	Expected result
`'diamonds'`	`1`	`False`	`''`	`1`	`70`
`'hearts'`	`3`	`False`	`''`	`4`	`170`
`'hearts'`	`4`	`True`	`''`	`4`	`620`
`'notrump'`	`4`	`True`	`'X'`	`4`	`790`
`'clubs'`	`5`	`False`	`''`	`6`	`480`
`'diamonds'`	`5`	`False`	`'X'`	`6`	`650`
`'notrump'`	`7`	`True`	`'XX'`	`7`	`2980`

Double trouble

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

Given a list of `items`, suppose that you repeated the following operation `n` times: remove the first element from `items`, and append that same element twice to the end of `items`. Which one of the `items` would be removed and copied in the last operation that we perform?

Sure, this problem could be solved by actually performing that operation `n` times, but the point of this question is to come up with an analytical solution to compute the result much faster than going through that whole rigmarole. Of course, the automated tester is designed so that anybody trying to solve this problem by actually performing all `n` operations one by one will run out of time and memory long before receiving the answer. To come up with this analytical solution, tabulate some small cases (you can implement the brute force function to compute these) and try to spot the pattern that generalizes to arbitrarily large values of `n`.

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

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

Giving back change

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

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

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

(This particular problem with its countless variations is a classic when modified so that you must minimize the total number of returned coins. The greedy approach will then no longer produce the optimal result for all possible coin denominations. For example, for simple coin denominations of `[50, 30, 1]` and `60` to be exchanged, the greedy solution `[50, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]` uses eleven coins, whereas the optimal solution `[30, 30]` needs only two! A more advanced **recursive** algorithm would be needed to make the "take it or leave it" decision for each coin, with the intermediate results of this recursion then **memoized** to avoid blowing up the running time exponentially.)

All cyclic shifts

```
def all_cyclic_shifts(text):
```

Given a `text` string, create and return the list of all its possible *cyclic shifts*, that is, strings of the same length that can be constructed from `text` by moving some prefix of that string to the end of that string. For example, the cyclic shifts of `'hello'` would be `'hello'`, `'elloh'`, `'llohe'`, `'lohel'`, and `'ohell'`. Note that every string is trivially its own cyclic shift by moving the empty prefix to the end. (Always keep in mind the important principle that unlike most systems and processes designed for humans in the real world, in computer programming **zero is a legitimate possibility that your code must be prepared for**, usually by doing nothing, since doing nothing is the correct thing to do in that situation. But as we know, sometimes it can be so very hard in life to sit back and do nothing...)

However, to make this problem more interesting rather than just one linear loop and some sublist slicing, the cyclic shifts that are duplicates of each other should be included in the answer only once. Furthermore, to make the expected result unique to facilitate automated testing, the returned list should contain these cyclic shifts sorted in ascending alphabetical order, instead of the order in which they were generated by your consecutive cyclic shift offsets.

`text`	Expected result
`'01001'`	`['00101', '01001', '01010', '10010', '10100']`
`'010101'`	`['010101', '101010']`
`'hello'`	`['elloh', 'hello', 'llohe', 'lohel', 'ohell']`
`'xxxxxxxxx'`	`['xxxxxxxxx']`

Postfix interpreter

`def postfix_evaluate(items):`

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

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

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

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

(By adding more operators and another auxiliary stack, an entire programming language can be built around the idea of postfix evaluation. See the Wikipedia page "Forth", if interested.)

Fractran interpreter

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

The esoteric programming language called FRACTRAN is one of the wackier inventions of John Conway, who is quite a character among mathematicians and computer scientists. A program written in such mysterious and primitive form consists of nothing but a list of positive integer fractions, in this problem given as tuples of the numerator and the denominator. (Of course, you are allowed to use the `Fraction` data type of the Python `fractions` module to simplify the computations inside your function.)

Given a positive integer `n` as the starting state, the next state is the product `n*f` for the first fraction listed in `prog` for which `n*f` is an integer. That number then becomes the new state for the next round. If `n*f` is not an integer for any of the fractions `f` listed in `prog`, the execution terminates. Your function should compute and return the sequence of integers produced by the given FRACTRAN program, with a forced termination... or I guess we should say *halt* (considering that Conway is British) taking place after `giveup` steps, if the execution has not halted by then.

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

Bingo bango bongo, I don't want to leave the Python

```
def contains_bingo(card, numbers, centerfree = True):
```

A two-dimensional grid of elements is represented as a list whose elements are themselves lists that represent the individual rows of the grid. This representation generalizes easily to structures of arbitrary dimensionality *k* by using a list whose elements represent structures of dimension *k* - 1. In this problem, your task is to determine whether the given 5-by-5 bingo card contains a bingo, that is, all the numbers of some row or column are included in the list of drawn `numbers`. The bingo can also be in the main diagonal (from upper left to lower right) or in the main anti-diagonal (from upper right to lower left). If the parameter `centerfree` is `True`, the center square is considered an automatic match regardless of whatever number is there.

`card`	`numbers`	`centerfree`	Expected result

`[`
`[38, 93, 42, 47, 15],`
`[90, 13, 41, 10, 56],`
`[54, 23, 87, 70, 6],`
`[86, 43, 48, 40, 92],`
`[71, 24, 44, 1, 34]`
`]`
	`[1, 2, 3, 4, 6, 8, 12,`
	`13, 15, 16, 19, 21,`
	`22, 24, 28, 34, 38,`
	`40, 41, 42, 43, 45,`
	`47, 49, 51, 53, 55,`
	`57, 58, 62, 65, 66,`
	`69, 70, 72, 82, 83,`
	`84, 86, 88, 95, 97]`
		`True`	`True`

`[`
`[89, 23, 61, 94, 67],`
`[19, 85, 90, 70, 32],`
`[36, 98, 57, 82, 20],`
`[76, 46, 25, 29, 7],`
`[55, 14, 53, 37, 44]`
`]`
	`[1, 2, 3, 4, 5, 6, 7,`
	`8, 9, 10, 11, 13, 14,`
	`16, 18, 19, 20, 21,`
	`22, 23, 24, 25, 27,`
	`28, 29, 31, 33, 35,`
	`36, 37, 38, 39, 41,`
	`42, 44, 45, 46, 47,`
	`48, 49, 51, 52, 55,`
	`56, 57, 58, 59, 60,`
	`61, 62, 63, 64, 65,`
	`68, 70, 71, 73, 75,`
	`76, 77, 79, 81, 82,`
	`84, 85, 86, 87, 88,`
	`89, 90, 91, 94, 98]`
		`False`	`True`

Group equal consecutive elements into sublists

```
def group_equal(items):
```

Given a list of elements, create and return a list whose elements are lists that contain the consecutive runs of equal elements of the original list. Note that elements that are not duplicated in the original list will still become singleton lists in the result, so that every element will get included in the resulting list of lists.

items	Expected result
['bob', 'bob', 7, 'bob']	[['bob','bob'], [7], ['bob']]
[1, 1, 4, 4, 4, 'hello', 'hello', 4]	[[1,1],[4,4,4],['hello','hello'],[4]]
[1, 2, 3, 4]	[[1], [2], [3], [4]]
[1]	[[1]]
[]	[]

Recamán's sequence

`def recaman(n):`

Compute and return the first `n` terms of the Recamán's sequence, starting from the term $a_1 = 1$. See the linked definition as defined on Wolfram Mathworld, noting how this definition depends on whether a particular number is already part of the previously generated part of the sequence, so your algorithm needs to use some suitable data structure to allow efficient testing for that.

To make your function execute in a speedy fashion even when generating a sequence that contains millions of elements, you should use a `set` to remember which values are already part of the previously generated sequence, so that you can generate each element in constant time instead of having to iterate through the entire previously generated list like some "Shlemiel" would do, your better technique thus allowing you to create this entire list in linear time and thus be blazingly fast even for millions of elements.

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

ztalloc ecneuqes

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

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

This function should, given the step `shape` as a string that is guaranteed to consist of only letters `u` and `d`, determine which starting value for Collatz sequence produces that `shape`. However, this function must also recognize that some shape strings are impossible as entailed by the transition rules of Collatz problem, and correctly return `None` for all such shapes. (Hint: start from the goal 1 and perform the transitions in reverse.)

`shape`	Expected result
`'ududududdddduddd'`	`15`
`'dudududduddduddd'`	`14`
`'uduuudddd'`	`None`
`'d'`	`2`
`'uuududddduuuuuuuddddd'`	`None`

Running median of three

`def running_median_of_three(items):`

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

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

Detab

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

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

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

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

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

A profound lack of history

`def safe_squares_knights(n, knights):`

On a generalized *n*-by-*n* chessboard there are some number of **knights**, the position of each knight represented as a tuple `(row, column)` of the row and the column that it is in. (The rows and columns are numbered from 0 to *n* - 1.) A chess knight covers the eight squares inside the board that can be reached from its position with a jump of **knight's move** making a **two-and-one L-shape**. Given the board size `n` and the list of `knights` on that board, count the number of empty squares that are safe, that is, are not covered by any knight and thus live to tell the tale. Remember also that a chess knight can jump over any pieces placed between them and their target square.

Hint: The following list might come handy in solving this problem:

```
knight_moves = [(2,1),(1,2),(2,-1),(-1,2),(-2,1),(1,-2),(-2,-1),(-1,-2)]
```

n	knights	Expected result
4	`[(2,3), (0,1)]`	8
8	`[(1,1), (3,5), (7,0), (7,6)]`	44
2	`[(1,1)]`	3
6	`[(0,0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]`	14
100	`[(row, (row*row) % 100) for row in range(100)]`	9191

Reverse every ascending sublist

`def reverse_ascending_sublists(items):`

Create and return a new list that contains the same elements as the argument list `items`, but reversing the order of the elements inside every maximal **strictly ascending** sublist. This function should not modify the contents of the original list, but create and return a new list object that contains the result.

(In the table below, different colours are used for illustrative purposes to visualize the strictly ascending sublists in this document, and of course are not part of the actual argument given to the function. It's not like this is *Mathematica* or some symbolic computation system that operate directly on expressions and their forms so that you could just do things like that.)

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

Brangelina

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

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

If the first name has only one vowel group, keep only the consonants before that group and throw away everything else. For example, `'ben'` becomes `'b'`, and `'brad'` becomes `'br'`. Otherwise, if the first word has $n > 1$ vowel groups, keep everything before the **second last** vowel group $n - 1$. For example, `'angelina'` becomes `'angel'` and `'alexander'` becomes `'alex'`. Concatenate that with the string you get by removing all consonants at the beginning of the second name.

All names given to this function are guaranteed to consist of the 26 lowercase English letters only, and each name will have at least one vowel and one consonant somewhere in it.

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

(These simple rules do not always produce the best possible result. For example, `'ross'` and `'rachel'` meld into `'rachel'` instead of `'rochel'`, and `'joey'` and `'phoebe'` meld into `'joebe'` instead of `'joeybe'`. The reader is invited to think up more advanced rules that would cover a wider variety of name combinations and special cases.)

Line with most points

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

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

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

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

This problem can be brute forced with three nested loops, but the point (heh) of this problem is not to do too much more work than you really need to. To simplify your logic, consult the example program `geometry.py` for the cross product function that can be used to quickly check whether three points are collinear.

Count distinct lines

```
def count_distinct_lines(points):
```

Two distinct points on a plane define the unique line that passes through both points, same way as originally defined by Euclid in the previous question "Line with most points" that asked you to find the line that goes through the largest number of points in the given list. Whenever three or more points are collinear, any two of them define the exact same infinitely long line. In this problem, the function should count how many distinct lines are defined by the given list of `points`.

You should not use **floating point arithmetic** of decimal numbers to compute the solution, but perform all calculations using only exact integer arithmetic so that your code will handle correctly coordinates even in the order of one googol and more. Make sure that your function does not crash due to division by zero whenever the line defined by the two points is **axis-aligned**, that is, either vertical or horizontal. The cross product function from `geometry.py` example script will again come handy here.

`points`	Expected result
`[(1, 42), (99, 17)]`	`1`
`[(2, 2), (7, 10), (3, 99)]`	`3`
`[(2, 3), (5, 5), (8, 7)]`	`1`
`[(0, 5), (5, 0), (0, 0), (5, 5)]`	`6`
`[(x, x*x) for x in range(100)]`	`4950`
`[(x, y) for x in range(10) for y in range(10)]`	`2306`
`[(x**10, x**20) for x in range(1000)]`	`499500`

Sevens rule, zeros drool

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

Seven is considered to be a lucky number in Western culture, whereas zero is what nobody wants to be. To bring these two opposites briefly together, let us look at only those positive integers that consist of some solid sequence of sevens, followed by some (possibly empty) solid sequence of zeros. For example 7, 77777, 7700000, 77777700, or 70000000000000000. The excellent MIT Open Courseware online textbook "*Mathematics for Computer Science*" (PDF link to the 2018 version for anybody who is interested) proves that for any positive integer `n`, there exists at least one positive integer of consecutive sevens followed by consecutive zeros that is divisible by `n`. This integer can get pretty humongous, but infinitely many such numbers will necessarily exist for any integer `n`.[1] This function should find and return the **smallest** such integer for the given `n`.

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

Furthermore, as proven in that same book, whenever `n` is not divisible by either 2 or 5, the smallest such number will always consist of a solid sequence of sevens with no zero digits after them. This handy fact can be used to speed up this search massively for such friendly values of `n`.

`n`	Expected result
`17`	`7777777777777777`
`42`	`7770`
`101`	`7777`
`103`	`77777777777777777777777777777777`
`2**50`	`70000000000000000000000000000000000000000000000000`
`12345`	(a behemoth that consists of a total of 822 sevens, followed by a single zero)

[1] If *s* is any number made of sevens and zeros that is divisible by *n*, the number *s*' that is made of sevens and zeros and is divisible by *sn* is also divisible by *n*, but is guaranteed to be larger than *s*. Therefore, the maximal number made of sevens and zeros that is divisible by *n* cannot exist, which means that there exist infinitely many such numbers.

Six degrees of separation

`def connected_islands(n, bridges, queries):`

An archipelago nation consists of a total of `n` islands numbered `0`, ..., `n-1`. Initially every one of these islands is, well, as the saying goes, an island. To rectify this isolation, the government of this archipelago nation decides to build a series of `bridges` to connect these islands to each other. Each bridge is given as a tuple `(a, b)` for the two islands `a` and `b` that it connects, and can be crossed in either direction. Connecting any two islands with a bridge immediately creates a pairwise route between any other islands that were previously connected to either one of these islands.

Once all these `bridges` have been built, we wish to perform a series of `queries` so that each query `(a, b)` asks whether there exists some route between these islands. The function should return a list of truth values for the answers to all these queries in one swoop.

n	bridges	queries	Expected result
3	`[(0, 1), (1, 2)]`	`[(2, 0)]`	`[True]`
7	`[(1, 2), (0, 1), (4, 5), (2, 1), (0, 5), (4, 0)]`	`[(3, 1), (6, 4), (0, 2), (4, 6), (3, 0), (1, 0), (1, 3)]`	`[False, False, True, False, False, True, False]`
19	`[(12, 2), (8, 14), (9, 0), (11, 4), (8, 1), (12, 13), (3, 1), (17, 2), (15, 11), (16, 1), (9, 17), (1, 10), (3, 7), (2, 5), (16, 18), (6, 5)]`	`[(13, 1), (17, 5), (13, 11), (17, 2), (12, 5), (1, 15), (3, 16), (1, 6), (18, 7), (10, 14), (14, 8), (5, 12), (0, 9), (18, 8), (15, 17), (7, 2), (2, 16), (0, 12), (3, 0)]`	`[False, True, False, True, True, False, True, False, True, True, True, True, True, True, False, False, False, True, False]`
1000	`[(a, a+1) for a in range(999)]`	`[(a, 0) for a in range(1000)]`	(a list of one thousand `True` elements)

Autocorrect for stubby fingers

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

In this day and age all you whippersnappers are surely familiar with **autocorrect** that replaces a non-word with the "closest" real word in the dictionary. Many techniques exist to make guessing what the user meant to say more accurate. Since many people, such as your instructor, have stubby fingers, many typos emerge from pressing a virtual key next to the intended key. For example, when the non-existent word is `"cst"`, the original word is far more likely to have been `"cat"` than `"cut"`, assuming that the text was entered with an ordinary QWERTY keyboard.

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

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

(Advanced autocorrect algorithms use statistical techniques based on the surrounding context to suggest the best replacement, since not all words are equally likely to be the intended word, given the rest of the sentence. For example, the misspelling `'urc'` should probably be corrected very differently in the sentence "The gateway consists of an urc of curved blocks" than in "The brave little hobbit swung his sword at the smelly urc.")

Uambcsrlne the wrod

`def unscramble(words, word):`

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

`wrod`	Etecxepd rssuelt (unsig the wrosldit `words_alpha.txt`)
`'mniister'`	`['minister', 'misinter']`
`'mnutiy'`	`['munity', 'mutiny']`
`'creavn'`	`['carven', 'cavern', 'craven']`
`'tninrpey'`	`['tripenny']`

(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 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?)

Substitution words

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

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

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

`pattern`	Expected result (using the wordlist `words_alpha.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']`
`'CBEGBHGCD'`	`['sabianism', 'variative']`
`'AFGHCFECH'`	`['capitasti', 'capotasto', 'corevolve', 'craterlet', 'nauseates', 'prolarval', 'schnecken']`
`'BDADCAF'`	`['cerebra', 'cerebri', 'ciliola', 'crureus', 'danaine', 'decency', 'detects', 'detents', 'feretra', 'kotoite', 'macauco', 'miliola', 'minions', 'moronry', 'mutuate', 'orarian', 'pereira', 'pinions', 'recency', 'salable', 'salably', 'silicle', 'terebra', 'vacance', 'vacancy', 'vitiate']`

Manhattan skyline

`def manhattan_skyline(towers):`

This classic problem in computational geometry is best illustrated by pictures and animations such as those on the page "The Skyline problem", so check it out first to get an idea of what is going on.

Given an unsorted list of rectangular `towers` as tuples of the form `(sx, ex, h)` where `sx` and `ex` are the start and end *x*-coordinates (naturally, `ex > sx`) and `h` is the height of that tower, compute and return the **total visible area of the towers**, being careful not to double count any towers that are partially overlapping. All towers share the same flat ground baseline at height zero.

The classic solution illustrates the important sweepline technique that starts by creating a list of precisely those *x*-coordinate values where something relevant to the problem takes place. In this problem, the relevant *x*-coordinates are those where some tower either starts or ends. Next, loop through this list in ascending order, updating your computation for the interval between the current relevant *x*-coordinate and the previous one. In this particular problem, you need to maintain a **list of active towers** so that each tower `(sx, ex, h)` gets added as a member when `x == sx`, and gets removed as a member when `x == ex`. During each interval, only the tallest active tower has any effect on the computation.

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

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

Count overlapping disks

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

Right on the heels of the previous Manhattan skyline problem, another classic problem of similar spirit that is here best solved with a sweepline algorithm. Given a list of `disks` on the two-dimensional plane as tuples `(x, y, r)` so that `(x, y)` is the center point and `r` is the radius of that disk, count how many pairs of disks intersect each other in that their areas, including the edge, have at least one point in common. To test whether two disks `(x1, y1, r1)` and `(x2, y2, r2)` intersect, use the Pythagorean formula `(x2-x1)**2 + (y2-y1)**2 <= (r1+r2)**2`. (Note again how this precise formula uses only integer arithmetic whenever all individual components are integers. And no square roots or some other nasty irrational numbers.)

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

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

Fulcrum

`def can_balance(items):`

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

As taught in any introductory physics textbook, the torque of an item with respect to the fulcrum equals its weight times its distance from the fulcrum. If a fulcrum position exists, return that position, otherwise return -1 to indicate that the given `items` cannot be balanced, at least not without rearranging them. (That one, by the way, would be an interesting but a more advanced problem normally suitable for a third year computer science course... but in Python, this algorithm could easily be built around this function by using the generator `permutations` in the module `itertools` to try out all possible permutations in an outer loop until you find one permutation that works. In fact, quite a few problems of this style can be solved with this "**generate and test**" approach without needing the fancy **backtracking** algorithms from third year and up.)

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

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

Sort integers by their digit counts

`def sort_by_digit_count(items):`

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

`items`	Expected result
`[98, 19, 4321, 9999, 73, 241, 111111, 563, 33]`	`[111111, 33, 241, 4321, 563, 73, 19, 98, 9999]`
`[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]`)

Count divisibles in range

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

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

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

but of course the automated tester is designed so that anybody trying to solve this problem in such a blunt and brutish way will soon run out of both time and space! So you should use no loops but integer arithmetic and conditional statements only, and be careful with various edge cases and off-by-one pitfalls lurking in the bushes. Note that either `start` or `end` can be negative or zero, but `n` is guaranteed to be greater than zero.

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

Bridge hand shape

`def bridge_hand_shape(hand):`

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

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

Milton Work point count

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

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

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

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

Count consecutive summers

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

Positive integers can be expressed as sums of **consecutive** positive integers in various ways. For example, 42 can be expressed as such a sum in four different ways: (a) 3 + 4 + 5 + 6 + 7 + 8 + 9, (b) 9 + 10 + 11 + 12, (c) 13 + 14 + 15 and (d) 42. As the last solution (d) shows, any positive integer can always be trivially expressed as a **singleton sum** that consists of that integer alone. Given a positive integer `n`, determine how many different ways it can be expressed as a sum of consecutive positive integers, and return that count.

`n`	Expected result
`42`	`4`
`99`	`6`
`92`	`2`

(As an aside, how would you concisely characterize the positive integers that have exactly one such representation? On the other side of this issue, which positive integer between one and one billion has the largest possible number of such breakdowns?)

The card that wins the trick

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

Playing cards are again represented as tuples of `(rank, suit)` as in the cardproblems.py example program. In trick-taking card games such as bridge, the players in turn each play one card to the trick. The winner of the trick is determined by the following rules:

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

Given the `cards` played to the trick as a list, return the card that wins the trick.

`cards`	`trump`	Expected result
`[('trey', 'spades'), ('ace', 'diamonds'), ('jack', 'spades'), ('eight', 'spades')]`	`None`	`('jack', 'spades')`
`[('ace', 'diamonds'), ('ace', 'hearts'), ('ace', 'spades'), ('deuce', 'clubs')]`	`'clubs'`	`('deuce', 'clubs')`
`[('deuce', 'clubs'), ('ace', 'diamonds'), ('ace', 'hearts'), ('ace', 'spades')]`	`None`	`('deuce', 'clubs')`

Iterated removal of consecutive pairs

`def iterated_remove_pairs(items):`

Given a list of elements, create and return a new list that contains the same elements except all occurrences of pairs of consecutive elements have been removed. However, this operation must continue in iterated fashion so that whenever removing some pair of consecutive elements causes two equal elements that were originally apart from each other to end up next to each other, that new pair is also removed, and so on, until nothing more can be removed.

`items`	Expected result
`[1, 2, 3, 4, 5, 5, 4, 3, 2, 1]`	`[]`
`['bob', 'tom', 'jack', 'jack', 'bob', 42]`	`['bob', 'tom', 'bob', 42]`
`[42, 42, 42, 42, 42]`	`[42]`
`[42, 5, 8, 2, 99, 99, 2, 7, 7, 8, 5]`	`[42]`

Expand positive integer intervals

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

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

Given a string that contains one or more such comma-separated interval descriptions, guaranteed to be given in sorted ascending order and never overlap with each other, create and return the list that contains all the integers contained inside these intervals.

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

Bridge hand shorthand form

`def bridge_hand_shorthand(hand):`

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

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

Losing trick count of a bridge hand

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

The Milton Work point count that we saw in the earlier problem is the first baby step in estimating the playing power of a bridge hand. Once the partnership has found out that they have a good trump fit, hand evaluation continues more accurately using some system of losing trick count. (For example, a small slam in spades with hands `'AKxxxxx - Kxxx xx'` and `'xxxx xxxxx AQx -'` is a lock despite possessing only 16 of the 40 high card points in the deck, whereas any slam is hopeless with the hands `'QJxxx xx AKx QJx'` against `'AKxxx QJ QJx AKx'` despite the combined powerhouse of "quacky" 33 points with a horrendous duplication of useful cards.[2])

In this problem, we compute the basic losing trick count as given in step 1 of "Methodology" section of the Wikipedia page "Losing Trick Count" without any finer refinements. Keep in mind that a suit cannot have more losers than there are cards in that suit, and never more than three losers if there were ten cards of that suit in their hand. The following dictionary (composed by student Shu Zhu Su during the Fall 2018 term) might also come handy for the combinations whose losing trick count differs from the string length, once you convert each `J` of the shorthand form into an `x` :

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

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

[2] Actually, as I later realized after writing that, 6NT has a chance of success against opposition that is smart enough to fall for an outrageous bluff. Since West did not double and then chortle while cashing the heart ace and king, East must hold one of these top honors, and West hopefully holds the other. Cross to dummy with a diamond, and without revealing anything with your mannerisms or voice, call for a low heart. A competent East will not go up with his honor since it looks like you are finessing with either AQ or KQ in hand, and after you play your queen, West will similarly hold up his ace or king, hoping that you will repeat the finesse. This ruse will not work against bad players who automatically fly up with their honor and thus stumble into their two natural heart tricks. In all games that feature private information, it is possible to bluff only those opponents who can be trusted to draw reliable statistical inferences from your observed actions, and who also have the guts to act on their inferences. (I apologize for this interruption as this is supposed to be *Python 101* instead of *Declarer Play in Bridge in Style of Hideous Hog*, but your instructor, being a notorious "Notrump Hog" himself, was unable to resist the temptation to get to write a tiny bridge column!)

Bulls and cows

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

In the old two-player game of "Bulls and Cows" (more recently reincarnated with pretty colours under the name "Mastermind") the first player thinks up a four-digit secret number whose each digit must be different, for example 8723 or 9425. (For simplicity, we will not use the digit zero in this problem.) The second player tries to guess this secret number by repeatedly guessing a four-digit number, the first player responding to each guess with how many "bulls", the right digit in the right position, and "cows", the right digit but in the wrong position, that guess contains. For example, if the secret number is 1729, the guess 5791 contains one bull (the digit 7) and two cows (the digits 9 and 1). The guess 4385, on the other hand, contains zero bulls and zero cows.

Given a list of `guesses` that have been completed so far, each individual guess given as a three-tuple `(guess, bulls, cows)`, create and return the list of four-digit numbers that are consistent with all these guesses, sorted in ascending order. Note that it is very much possible for the result list to be empty, if no four-digit integer is consistent with all of the `guesses`.

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

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

This problem and its myriad generalizations (for example, the same game played with English words) can be solved in more clever and efficient ways than the above **brute force** enumeration, but that would be a topic for a later, more advanced algorithms course.

Longest palindrome substring

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

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

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

(The real challenge in this problem is making the function much faster than the trivial solution of looping through all possible substrings and checking which ones are palindromes, remembering the longest palindrome that you have seen. Those who are interested in this kind of algorithmic tweaking can check out the Wikipedia page "Longest Palindromic Substring".)

Words with given shape

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

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

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

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

Sort list by element frequency

`def frequency_sort(elems):`

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

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

(Hint: create a dictionary to count how many times each element occurs inside the array, and then use the counts stored in that dictionary as the **sorting key** of the array elements, breaking ties on the frequency by using the actual element value. If you happen to remember that the order comparison of Python tuples is lexicographic, you don't even need to do any of this tie-breaking work yourself...)

Calling all units, B-and-E in progress

`def is_perfect_power(n):`

A positive integer `n` is said to be a perfect power if it can be expressed as the power `b**e` for some two integers `b` and `e` that are both **greater than one**. (Any positive integer `n` can always be expressed as the trivial power `n**1`, so we don't care about that.) For example, the integers 32, 125 and 441 are perfect powers since they equal `2**5`, `5**3` and `21**2`, respectively. This function should determine whether the positive integer `n` given as argument is some perfect power. To do this, your code needs to somehow iterate through a sufficient number of possible combinations of `b` and `e` that could work, returning `True` as soon as it finds some `b` and `e` that satisfy `b**e == n`.

Since `n` can get pretty large, your function should not examine too many combinations of `b` and `e` above and beyond those that are necessary and sufficient to determine the answer. Achieving this efficiency is the central educational point of this particular problem.

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

(The automated tester for this problem is based on the mathematical theorem about perfect powers that says that after the special case of two consecutive perfect powers 8 and 9, whenever the positive integer `n` is a perfect power, `n-1` cannot be a perfect power. This theorem makes it really easy to generate random test cases with known correct answers, both positive and negative. For example, we would not need to try out all possible ways to express the number as an integer power to know right away that the humongous integer `1234**5678 - 1` is not a perfect power.)

Sum of highest n scores for each player

```
def highest_n_scores(scores, n = 5):
```

Each player plays the game some number of times, each time resulting in a tuple `(name, score)` for how much that player scored in the game. Given all the games played as a list of such tuples, add up the *n* highest scores for each individual player as the total score for that player. Create and return a list that contains tuples `(name, total)` for the players and their total scores, sorted in ascending order by `name`. If some player has played fewer than `n` times, just add up the scores of however many games that player has played.

scores	n	Expected result
`[('bill', 10), ('jack', 6), ('sheldon', 3), ('tina', 2), ('amy', 3), ('sheldon', 6), ('tina', 7), ('jack', 2), ('bob', 3), ('bob', 4), ('bill', 3), ('bill', 9), ('sheldon', 5), ('amy', 2), ('jack', 7), ('sheldon', 5), ('sheldon', 7), ('bill', 1), ('bill', 9), ('sheldon', 5), ('bill', 2), ('bill', 6), ('jack', 6), ('bob', 4), ('tina', 5), ('sheldon', 4), ('sheldon', 2), ('amy', 6), ('bob', 7), ('jack', 2), ('bob', 5), ('sheldon', 9), ('jack', 5), ('amy', 9), ('bob', 7), ('tina', 6), ('tina', 2), ('amy', 7), ('jack', 10), ('tina', 4), ('bob', 5), ('jack', 10), ('bob', 7), ('jack', 5), ('amy', 4), ('amy', 8), ('bob', 4), ('bill', 8), ('bob', 6), ('tina', 6), ('amy', 9), ('bill', 4), ('jack', 2), ('amy', 2), ('amy', 4), ('sheldon', 1), ('tina', 3), ('bill', 9), ('tina', 4), ('tina', 9)]`	`3`	`[('amy', 26), ('bill', 28), ('bob', 21), ('jack', 27), ('sheldon', 22), ('tina', 22)]`

Collapse positive integer intervals

`def collapse_intervals(items):`

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

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

Distribution of abstract bridge hand shapes

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

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

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

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

For example, if it were somehow possible to give to your function the list of all 635,013,559,600 possible bridge hands and not run out of the heap memory in the Python virtual machine, the returned dictionary would contain 39 entries for the 39 abstract 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.)

Sort words by typing handedness

```
def sort_by_typing_handedness(words):
```

When typing an English word on the standard QWERTY keyboard, the left hand types in the letters *q* to *y* in the top row, *a* to *h* in the middle row, and *z* to *b* in the bottom row. Other letters are typed in with the right hand. For the purposes of this problem, we define the score of a word as the number of its left hand characters, minus the number of its right hand characters. For example, the score for the word `'repetition'` would be 5 - 5 = 0.

This function should sort the given list of `words` in the *descending* order of their score, and return the entire sorted list. To make the resulting list unique for the purposes of automated testing, words that have an equal score should en up in their ascending dictionary order.

You should let Python do the sorting with the `sort` method that the Python lists already have built in. You merely need to give this function a suitable `key` function that converts each word into a two-tuple of the handedness tally and the word itself. For example, `'hello'` is sorted as `(-1, 'hello')`, and `'computer'` is sorted as `(0, 'computer')`. Since the order comparison of Python tuples is lexicographic, this produces the correct result using the handedness tally as the primary sorting key, and the word itself as the secondary sorting key whenever the primary keys are equal.

Sorting the words in `words_alpha.txt` in this manner, the first five words are

```
['regeneratoryregeneratress', 'tessarescaedecahedron', 'cabbageheadedness', 'archpresbyterate', 'bathyhyperesthesia']
```

and the last five end up being

```
['unmonopolising', 'unmonopolizing', 'humuhumunukunukuapuaa', 'kinnikinnik', 'nonillumination'].
```

Fibonacci sum

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

Fibonacci numbers are a cliche in introductory computer science, especially in teaching recursion where this famous combinatorial series is mainly used to reinforce the belief that recursion is silly... but all cliches became cliches in the first place because they were so good that everyone and their brother kept using them! Instead of that silly recursion example to compute Fibonacci numbers in exponential time instead of using a loop like a reasonable person, let us rather showcase a more amazing property of this famous sequence: **every positive integer can be expressed exactly one way as a sum of non-repeated Fibonacci numbers** so that no two consecutive Fibonacci numbers appear in this sum. (After all, if the sum contains two consecutive Fibonacci numbers F_i and F_{i+1}, these two can always be replaced by F_{i+2} without affecting the total.)

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

Oh yeah, and one more thing: your function has to be able to handle values of `n` up to `10**10000`. The correct answer for that monstrosity is way too long to print here, but it consists of 13,166 terms, the largest of which has 10,000 digits. (Ain't Python and its integers of unlimited size just grand as they let us tickle the belly of some of these eternal behemoths in their slumber previously undisturbed by humans?)

`n`	Expected result
`10`	`[8, 2]`
`100`	`[89, 8, 3]`
`10**6`	`[832040, 121393, 46368, 144, 55]`
`10**20`	`[83621143489848422977, 12200160415121876738, 2880067194370816120, 1100087778366101931, 160500643816367088, 37889062373143906, 117669030460994, 27777890035288, 4052739537881, 1548008755920, 365435296162, 2971215073, 24157817, 3524578, 196418, 75025, 10946, 4181, 610, 233, 89, 21, 3, 1]`

(Mathematically minded students can try to prove rigorously using **mathematical induction** that this greedy algorithm really works for all integers, so that this function always creates and returns the unique list of non-consecutive Fibonacci numbers whose sum equals `n`.)

Rooks with friends

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

Those dastardly rooks have again gone on a rampage on a generalized *n*-by-*n* chessboard, just like in the earlier problem of counting how many squares were safe from their wrath. Each rook is again represented as a tuple `(row, column)` of the coordinates of the square that it is standing on. However, in this version of the problem, some of these rooks are now your **friends** (same colour as you) while the others are your **enemies** (the opposite colour from you). Friendly rooks protect the chess squares by standing between them and any enemy rooks that might threaten those squares, so that an enemy rook can attack only those squares in the same row or column that do not enjoy the protection of any friendly rook standing between them. Given the board size `n` and the lists of `friends` and `enemies`, count how many empty squares on the board are safe from the enemies.

n	friends	enemies	Expected result
4	`[(2,2), (0,1), (3,1)]`	`[(3,0), (1,2), (2,3)]`	2
4	`[(3,0), (1,2), (2,3)]`	`[(2,2), (0,1), (3,1)]`	2
8	`[(3,3), (4,4)]`	`[(3,4), (4,3)]`	48
100	`[(r, (3*r+5) % 100) for r in range(1, 100, 2)]`	`[(r, (4*r+32) % 100) for r in range(0, 100, 2)]`	3200

Possible words in Hangman

`def possible_words(words, pattern):`

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

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

`pattern`	Expected result (using wordlist `words_alpha.txt`)
`'***dg*'`	`['abedge', 'aridge', 'bludge', 'bridge', 'cledge', 'cledgy', 'cradge', 'fledge', 'fledgy', 'flidge', 'flodge', 'fridge', 'kludge', 'pledge', 'plodge', 'scodgy', 'skedge', 'sledge', 'slodge', 'sludge', 'sludgy', 'smidge', 'smudge', 'smudgy', 'snudge', 'soudge', 'soudgy', 'squdge', 'squdgy', 'stodge', 'stodgy', 'swedge', 'swidge', 'trudge', 'unedge']`
`'a**s**a'`	`['acystia', 'acushla', 'anosmia']`
`'*a*e*i*o'`	`['patetico']`

Factoring factorials

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

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

Since `n` will get into the thousands in the automated tester, you should accumulate the prime factors of `n!` separately for each individual term of the factorial as you go, rather than first computing the entire `n!` and then afterwards dividing it down into its prime factors.

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

Aliquot sequence

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

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

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

The tester for this problem is pretty heavy, so you might want to optimize the calculating of proper factors of the given integer with **memoization**. Maintain an auxiliary dictionary in which you store the lists of proper factors of integers as you compute them. The next time that you need the proper factors of some integer, check if this dictionary already contains them, in which case you just look up those factors instead of computing them again all hard way from scratch.

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

Last man standing

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

In the ancient times "🎶 when men were made of iron and their ships were made of wood 🎶", as seen in "300", "Spartacus", "Game of Thrones" and similar historical docudramas of swords and sandals, a group of zealots (yes, *literally*) was surrounded by the overwhelming Roman enemy. To avoid capture and slow death by crucifixion, in their zeal these men chose to commit mass suicide in a way that prevented any one of them from changing his mind. The zealots arranged themselves in a circle and used lots to choose a random step size `k`. Starting from the first man, they repeatedly count `k` men ahead and quickly kill that man, removing his corpse from the decreasing circle. (Being normal people instead of computer scientists, they always start counting from one instead of zero, the concept of which didn't even exist for them back then anyway!) This continues until only one man remains, expected to honorably fall on his own sword and join his fallen brothers. Josephus would very much prefer to be this last man since he has other ideas of surviving. Help him and his secret confederate survive with a function that, given `n` and `k`, returns the list of the execution order so that these men know which places let them be the last two survivors.

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

Lattice paths

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

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

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

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

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

Count squares on integer grid

`def count_squares(points):`

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

You are given a set of `points`, each point a two-tuple `(x, y)` where `x` and `y` are positive integers. This function should count how many **squares** these points define so that all four corners of the square are members of this set of `points`, and return this count. Note that the squares don't need to be **axis-aligned** so that their sides would be expected to be horizontal and vertical, as long as all four sides are the exact same length.

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

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

Split digit string to minimize sum

`def minimize_sum(digits, k):`

A string that is guaranteed to consist of nothing but digit characters `'0'` to `'9'` needs to be split into **exactly** `k` nonempty pieces (it is guaranteed that `k <= len(digits)`) in a way that minimizes the sum of those pieces when added together as integers. For example, the best way to split the string `'12345'` into exactly three pieces would be `['12', '34', '5']`, resulting in the sum 12 + 34 + 5 = 51. Any other split would produce a larger sum, as you may verify.

This problem is best solved with recursion. The base case `k = 1` has the simple answer of returning that digit string as an integer. Otherwise, loop through all possible positions of the digit string that you could use to split off the first piece while leaving in at least `k - 1` digits, and recursively find the smallest sum that you can get by splitting the remaining digits into exactly `k - 1` pieces. Return the smallest sum produced by the best splitting place that you found. Note from `'90210'` given first in the table below how the zero digits are always best used as leading digits of a piece...

Since the above recursion keeps recomputing the same subproblems exponentially many times, it would be a great idea to **memoize** it using the `@lru_cache` decorator from `functools` module so that each result gets automatically looked up the second time it comes up, instead of your code having to waste time to recompute that same result from scratch through an exponential thicket of possible cutting point combinations. These memoized results will then also speed up all later test cases that (by some amazing coincidence, I bet) just so happen to contain some of the very same subproblems inside them...

digits	k	Expected result
'90210'	2	219
'123456789'	3	1368
'123456789'	9	45
'100020003000'	2	13002
'100020003000'	3	3003
str(3**100)	6	123119910
str(2**1000)	10	342863442431016700276808245531 6

Sum of distinct cubes

`def sum_of_distinct_cubes(n):`

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

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

This problem is best solved with recursion. If `n` itself is a cube, return the singleton list `[n]`. Otherwise, loop down through all possible values of the first element `a` in the result list, and for each such `a`, break down the remaining number `n-a*a*a` into a sum of distinct cubes using only integers that are smaller than your currently chosen `a`. Again, to make this function efficient even for large `n`, it might be good to prepare something that allows you to quickly determine whether the given integer is a cube, and whenever it is not, to find the largest integer whose cube is smaller...

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

This problem is intentionally restricted to positive integers so that the number of possibilities that we need to iterate through remains finite. Since the cube of a negative number is also negative, such finite upper bound no longer exists once negative numbers are allowed, and breaking even a small number into a sum of exactly three cubes becomes a highly nontrivial problem.

Fractional fit

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

You are given a list `fs` of integer two-tuples `(a, b)` so that `0 <= a < b`. When interpreted as exact integer fraction `f = a / b`, the values therefore fall within the unit interval, that is, `0 <= f < 1`. Given the list `fs`, your task is to find the longest possible list of these fractions that has the following curious property:

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

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

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

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

Followed by its square

`def square_follows(it)`:

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

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

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

Count maximal layers

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

The point `(x1, y1)` on the plane is said to **dominate** another point `(x2, y2)` if it lies to the right and above from it, that is, `x1 > x2` and `y1 > y2`. A point inside the given list of `points` is defined to be **maximal** if it is not dominated by any other point in the list. Unlike in one dimension, a list of points on the two-dimensional plane can contain any number of maximal points, which then form the **maximal layer** for that particular list of points. For example, the points `(3, 10)` and `(9, 2)` are the maximal layer for the list of points `[(1, 5), (8, 1), (3, 10), (2, 1), (9, 2)]`.

Given a list of `points` whose coordinates are guaranteed to be nonnegative, this function should compute how many times one would need to perform the operation of removing every point in the maximal layer from the list (and then compute the new maximal layer for the remaining points for the next round) for that entire list to become empty, and return that count.

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

This is again one of those problems whose actual educational point and motivation is making this function fast enough to finish within a reasonable time even for a big list of points, by not doing too much more work than would be necessary to identify and remove the maximal points. Start by noticing that each point can potentially be dominated only by those points whose distance from the origin (0, 0) is strictly larger...

Maximum checkers capture

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

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

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

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

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

Collatzy distance

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

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

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

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

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

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

Van Eck sequence

`def van_eck(n):`

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

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

The sequence begins with 0, 0, 1, 0, 2, 0, 2, 2, 1, 6, 0, 5, 0, 2, 6, 5, 4, 0, 5, 3, 0, 3, 2, 9, ... so you can see that the terms start making repeated appearances, especially the term zero.

In the same spirit as in the earlier problem that asked you to generate terms of the Recaman sequence, unless you want to spend the rest of your day waiting for the automated tester to finish, this function should not repeatedly loop backwards through the generated sequence to look for the most recent occurrence of each term that it generates, but rather use a Python dictionary to remember the terms that have already been seen in the sequence, along with the positions of their most recent appearances. With this dictionary, you don't need to explicitly store the sequence, since this problem does not ask for you to return the entire sequence, but only one particular term.

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

Reversing the reversed

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

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

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

Prime factors of integers

```
def prime_factors(n):
```

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

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

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

Balanced ternary

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

The integer two and its powers abound all over computing whereas the number three seems to be mostly absent, at least until we get to the **theory of computation** and **NP-complete problems** where the number three turns out to be a very different thing from two not only quantitatively, but qualitatively. To rectify this injustice, we shall give this good little boi its turn in the limelight.

We normally represent integers in base 10, each digit giving the coefficient of some power of 10. Since every positive integer can equally well be uniquely represented as a sum of powers of two, computers internally encode those same integers in simpler (for machines) **binary**. Both of these schemes need special tricks to represent negative numbers since we cannot use an explicit negation sign. The balanced ternary representation of integers uses the base three instead of two, with **signed** coefficients from -1, 0 and +1. Every integer (positive or negative) can be broken down into sum of signed powers of three exactly one way. Furthermore, unlike the bases of ten or two, the balanced ternary representation is perfectly symmetric so that the representation for -*n* can be constructed by flipping the signs of the terms of the representation of +*n*. (See the examples below.)

Given an integer `n`, return the unique list of signed powers of three that add up to `n`, listing these powers in the descending order of their absolute values. This problem can be solved in a couple of different ways. The page "Balanced ternary" shows one way by first converting the number to unbalanced ternary (base three using coefficients 0, 1 and 2) and converting from there. Another way is to first find *p*, the highest power of 3 that is less than equal to *n*. Then, depending on the value of *n* - *p*, you either take *p* to the result and convert *n* - *p* for the rest, or take 3*p* to the result and convert *n* - *3p* for the rest. All roads lead to Rome.

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

Lords of Midnight

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

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

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

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

(To make the test cases more interesting, the automated tester is programmed to produce fewer random ones, fours and sixes than the probabilities of fair dice would normally give. There is no law of either nature or man against that kind of seeming tomfoolery.)

Optimal crag score

`def optimal_crag_score(rolls):`

Way back near when we started doing these problems, one particular problem asked you to compute the best possible score for a single roll in the dice game of crag. In this problem, we will now play the game for real, again aided with the same gift of perfect foresight as in the previous problem "Lords of Midnight" so that given the knowledge of the entire future sequence of `rolls`, this function should return the highest possible score that could be achieved with those rolls under the constraint that **the same category cannot be used more than once**.

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

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

Forbidden substrings

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

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

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

Infinite Fibonacci word

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

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

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

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