Draft Problems for CCPS 109

This document contains the specifications for additional problems for the collection "109 Python Problems" by Ilkka Kokkarinen. **These problems are still in the draft stage** in that they may contain bugs. So I need your help to debug them to make them suitable for wider consumption! This version of draft problems is from **April 5, 2020**.

The tester script `tester109.py` of the main problem collection contains automated testers also for these draft problems, their expected checksums and the first 300 expected results computed from the private model solutions of the author. Any student who implements one of these functions so that it passes the automated tester should contact the author at `ilkka.kokkarinen@gmail.com` to let me know about that. I will mark the problems that have been independently solved by at least one student with a green checkmark. Eventually such verified problems will migrate to the main collection to replace some weaker problems that are not pulling their weight.

✔️ Pull down your neighbour

`def eliminate_neighbours(items):`

Given a list of integer items that are guaranteed to be some permutation of positive integers from 1 to `n` where `n` is the length of the list, consider the following operation: Find the smallest number that currently remains in the list, and remove from the list that number and the larger of its current immediate neighbours. For example, given the list `[5, 2, 1, 6, 4, 3]`, the operation would remove element `1` and its current larger neighbour `6`, resulting in the list `[5, 2, 4, 3]`. Applied again, the same operation would now remove `2` and its current larger neighbour `5`, turning the list into `[4, 3]`. The function should keep doing this repeatedly until the largest number in the original list gets eliminated, and return the number of operations that were needed to achieve this goal.

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

The bottleneck of the running time is computing the new list that results from removing two elements from it. Try to think up a way to solve this problem that avoids this expensive operation.

All your fractions are belong to base

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

A pile of `n` identical coins is repeatedly subjected to the following move. These coins are grouped into exactly `out` coins for each group, where `out` is a positive integer greater than one. The remainder `n%out` coins that do not fit into any group are taken aside and recorded. From each complete group of `out` coins, exactly `in` coins are added back in the pile, and the rest of the coins of that group are discarded. Repeat this until the entire pile becomes empty, which must eventually happen whenever `out > in`. Return a list that says how many coins were taken aside in each step.

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

As seen in the first three rows, this method produces the digits of `n` in base `out` in reverse order. So this entire setup turns out to be the cleverly disguised algorithm to construct the representation of integer `n` in base `out`. However, an improvement over the standard algorithm for integer base conversion is that this version works not only for integer bases, but allows any fraction `out/in` to be used as the base, provided that `out > in` and `gcd(out, in) == 1`. For example, the familiar integer 42 would be written as 323122 in base 4/3. Whoa. Just imagine trying to do basic arithmetic in such a system... that would have been some "New Math" for the frustrated parents in the sixties!

Duplicate digit bonus

`def duplicate_digit_bonus(n):`

Some people like to ascribe deep significance to numerical coincidences so that especially consecutive repeated digits, such as the clock blinking 11:11, seem special to them. Such people then naturally find numbers with repeated digits to be more valuable and important than ordinary pedestrian numbers that seem randomly created.

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

n	Expected result
43333	200
2223	1000
777777777	20000000
3888882277777731	11001
2111111747111117777700	12002
99999977777774444488872222	21210
1234**5678	15418

Count word dominators

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

If you solved the earlier `count_dominators` problem correctly, notice that even though the problem was stated for lists of integers, the logic of domination did not depend on this fact, but only on that these elements can be order-compared with each other. For example, the call `count_dominators(['dog', 'emu', 'cat', 'bee'])` is perfectly meaningful and should correctly return 3, since `'emu'`, `'cat'` and `'bee'` dominate all words after them in the sense of string order comparison. Hopefully, you also avoided being a "Shlemiel" by looping through elements in reverse order from right to left and comparing the current element only to the maximum of remaining elements, and you kept that maximum in a local variable that you updated every time you found a new dominator.

However, things become even more interesting if we define domination between words of equal length so that a word dominates another word if for more than half of the positions, the character in the first word is strictly greater than the character in the same position in the other word. This definition makes the domination a **partial ordering** so that, for example, `'dog'` dominates `'cat'`, but neither word `'dog'` and `'arg'` dominates the other. Note also the intentional phrasing of "more than half" to break ties in words of even length such as `'aero'` and `'tram'`.

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

Nearest polygonal number

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

Any positive integer $s > 2$ defines an infinite sequence of ***s*-gonal numbers** whose i:th element is given by the formula $((s - 2)\ i^2 + (s - 4)\ i) / 2$, as explained on the Wikipedia page "Polygonal Number". (In this formula, positions start from 1, not 0.) For example, the sequence of "octagonal numbers" that spring forth from setting $s = 8$ starts with 1, 8, 21, 40, 65, 96, 133, 176...

Given the number of sides `s` and an arbitrary integer `n`, this function should find and return the *s*-gonal integer that is nearest to `n`. If `n` falls exactly halfway between two *s*-gonal numbers, return the smaller one. This function must be efficient even for gargantuan values of `n`.

`n`	`s`	Expected result
`5`	`3`	`6`
`27`	`4`	`25`
`450`	`9`	`474`
`10**10`	`42`	`9999861561`
`10**100`	`91`	`1000000000000000000000000000000000000000000000000041633275351832947889775579470433400300354421242035 6`

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

Power wash those floors

`def floor_power_solve(n, k):`

For positive integers `a` and `b`, the operation `a**b` can be implemented as repeated multiplication. Start with `result=1`, and multiply `a` into the `result` for a total of `b` times. No problemo. In the variation of **floor power**, the result of each multiplication is immediately truncated to its **floor**, which for positive numbers equals the integer part. This makes no difference when `a` is an integer, but matters greatly when `a` is some `Fraction`. (For example, `Fraction(7,3)**3` equals `343/27`, whereas `floor_power(Fraction(7,3), 3)` equals `28/3`, once you implement that function.)

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

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

Multidimensional knight moves

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

From the square that it is currently standing on, an ordinary chess knight on a two-dimensional board of squares can make an "L-move" into up to eight possible neighbours. However, we can generalize the entire chessboard into the heights of k dimensions from mere two. The natural extension of the knight's move that keeps all moves symmetric with respect to all these dimensions is to define the possible moves as a k-tuple of strictly decreasing nonnegative integers. Each one of these k offsets must be used for exactly one dimension of your choice during the move, either as a positive or a negative version.

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

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

A quick combinatorial calculation reveals that this allows exactly $k! * 2^k$ possible neighbours reachable in a single move. In this notation, the ordinary chess knight is a `(2, 1)`-knight that can reach $2! * 2^2 = 4 * 2 = 8$ possible neighbouring squares in one move, whereas some 6-dimensional knight could move into $6! * 2^6 = 46080$ different neighbours in one jump. (Everything is close to everything else in high-dimensional spaces.)

Squaring off

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

Two players play a game called "**Subtract a square**" with a pile of identical pebbles on the table. On their turn to play, each player can remove any square number (1, 4, 9, 16, 25, 36, 49, ...) of pebbles from those currently on the table. The player who removes the last pebble wins the game.

In these sorts of impartial games where the moves available to each player on his turn are the same (as opposed to "partisan games" such as chess, where the player can move only pieces of his own colour), a game state is called "cold" if the player whose turn it is to play will lose no matter what move he plays, provided that the opponent keeps playing optimally. A state is "hot" if there exists at least one move in it so that leads to a cold state.

These kinds of combinatorial games can be modelled and solved with recursive equations. Since the possible states of this game are the nonnegative integers, the base case is state 0 that is cold, since the player whose turn it is to now move has lost the game. For `n > 0`, the state `n` is hot if and only if there exists at least one integer move `m` so that the state `n-m*m` is cold.

Given a list of states as `queries`, this function should return a list of results for those queries so that `True` means that the state is hot, and `False` means that it is cold. You should compute the heat values of states into a single list that you build up in order from left to right, so that the computation of heat in each state can read from this list the heat values of previous states.

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

Split within perimeter bound

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

Rectangles are represented as tuples `(a, b)` so that `a` and `b` are positive integers. A rectangle can be cut in two smaller rectangles with the same total area with either a straight horizontal or a straight vertical cut along the integer axis of your choice. For example, the rectangle `(5, 8)` might be cut into pieces `(2, 8)` and `(3, 8)` in one direction, or into rectangles `(5, 4)` and `(5, 4)` in the other direction. Many other cuts would also be possible. These smaller pieces can then be cut further into smaller pieces, as long as the side being cut has length of two or more.

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

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

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

✔️ Up for the count

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

The **counting series** `"123456789101112131415161718192021222 3"`... is defined as an infinitely long string that consists of the positive integers written in ascending order without any separators between the individual numbers. This function should return the integer digit that is in the position `n` of the counting series, with positions starting from 0 as usual.

Of course, the automated tester will again try out values of `n` large enough that anybody trying to solve this problem by constructing the counting series as an explicit string would run out of time and space long before receiving the answer. Instead, you should observe that the structure of this infinite sequence is quite straightforward (the sequence starts with 9 single-digit numbers, followed by 90 two-digit numbers, followed by 900 three-digit numbers, and so on), thus allowing you to skip prefixes of this series in exponentially widening leaps and bounds, until you reach the position `n` and find out which digit is waiting for you there.

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

Om nom nom

`def cookie(piles):`

The beloved Sesame Street character Cookie Monster has stumbled upon a table with `piles` of cookies, each pile given as a positive integer so that no integer is repeated in `piles`. However, the obsessiveness of The Count who has set up this feast has recently escalated to a whole new level of severity. To allow Cookie Monster to eat his fill, The Count insists that he must eat these cookies in the shortest possible sequence of moves. Each move consists of the Cookie Monster choosing an integer `p` that must be one of the remaining pile sizes. The chosen move removes `p` cookies from every pile that contains at least `p` cookies, and leaves all the smaller piles as they were before the move. This function should compute and return **the smallest number of moves** that allows Cookie Monster to eat these cookies.

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

This problem has a curious property that the moves in the optimal solution can be performed in any order, so without loss of generality, you can examine only strictly descending move sequences. Even then, this problem is far deeper than it might seem. Various **greedy strategies** that would seem natural for the addiction-riddled brain of the Cookie Monster such as "choose `p` to maximize the number of cookies eaten at this move" or "choose `p` to eliminate the largest possible number of piles" (eating `p` cookies from a pile of size `p+m` makes that pile effectively disappear for the purposes of this problem, whenever there is another pile of size `m` on the table) do not always produce the shortest possible sequence of moves.

Hippity hoppity, abolish loopity

`def frog_collision_time(frog1, frog2):`

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

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

This function should not contain any loops whatsoever, but the result should be calculated using conditional statements and integer arithmetic. Perhaps the best way to get started is by first solving a simpler version of this problem with one-dimensional frogs restricted to hop along the one-dimensional line of integers. Once you get that function working correctly even for all the possible edge cases, use it to solve for `t` separately for the *x*- and *y*-dimensions in the original problem, and combine those two one-dimensional answers into the final answer.

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

✔️ Whenever they zig, you gotta zag

`def is_zigzag(n):`

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

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

`n`	Expected result
`7`	`True`
`25391`	`True`
`908172635454637281850`	`True`
`16329`	`False`
`104175101096715`	`False`
`495739120 09`	`False`

The MD problem

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

Consider the infinite series that starts with the value 1 at the position 0. Let `v` be the most recent element in the sequence. If the value `v // a` is nonzero and has not appeared in the sequence so far, the next element of the sequence equals `v // a`, otherwise the next element equals `b * v`. For example, for the parameter values `a = 2` and `b = 3` in spirit of the Collatz sequence, this sequence would start off as `[1, 3, 9, 4, 2, 6, 18, 54]`.

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

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

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

Everybody do the Scrooge Shuffle

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

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

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

To make this function run faster, start by noticing that if some pile contains `k * (left + right)` coins for some `k > 0`, you can scoop `k * left` coins into the previous pile and `k * right` coins into the next pile in one move, instead of having to slough through `k` separate little moves that individually move only `(left + right)` coins each.

piles	left	right	Expected result
`[20]`	`3`	`2`	`(-3, [5, 3, 3, 3, 6])`
`[8, 4]`	`3`	`2`	`(-2, [3, 1, 3, 3, 2])`
`[111, 12, 12]`	`19`	`6`	`(-6, [19, 13, 13, 13, 13, 13, 10, 23, 18])`

✔️ Wythoff array

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

Wythoff array is an infinite two-dimensional grid of integers that starts with values one and two in the first row. In each row, each element equals the sum of the previous two elements in that row, the same way as in the Fibonacci series, so the first row contains precisely the Fibonacci numbers.

The first element of each later row is **the smallest integer `c` that does not appear anywhere in the previous rows**. To compute the second element of that row, let `(a, b)` be the first two elements of the previous row. If the difference `c-a` equals two, the second element of that row equals `b+3`, and otherwise that element equals `b+5`. This construction automatically guarantees that the Wythoff array is an **interspersion** of positive integers, meaning that **every positive integer will appear exactly once in this infinite grid**. (Whoa!)

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

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

✔️ Next higher zigzag

`def next_zigzag(n):`

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

This problem could in principle be solved by counting up from `n+1` and using your previous `is_zigzag` function to terminate when you get to the next higher zigzag number. Unfortunately, as you can see in the table below, the gap between two consecutive zigzag numbers can be arbitrarily long...

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

Flip of time

`def hourglass_flips(glasses, t):`

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

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

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

`glasses`	`t`	Expected result
`[(7, 0), (11, 0)]`	`15`	`2` (see here)
`[(7, 0), (4, 0)]`	`9`	`2` (see here)
`[(4, 0), (6, 0)]`	`11`	`None`
`[(7, 0), (4, 0), (11, 0)]`	`20`	`3`
`[(16, 0), (21, 0)]`	`36`	`6`
`[(5, 0), (8, 0), (13, 0)]`	`89`	`7`

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