

Ryerson letter grade

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

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

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

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

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

Ascending list

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

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

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

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

In the same spirit, note how every possible universal claim made about the elements of an empty sequence is trivially true! For example, if `items` is the empty sequence, the two claims “All elements of `items` are odd” and “All elements of `items` are even” are both equally true, as is also the claim “All elements of `items` are [colourless green ideas that sleep furiously](#)”

Riffle shuffle kerfuffle

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

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

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

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

Even the odds

```
def only_odd_digits(n):
```

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

To extract the lowest digit of a positive integer n , use the expression $n \% 10$. To chop off the lowest digit and keep the rest of the digits, use the expression $n // 10$. Or, if you don't want to be this fancy, you can first convert the number into a string and work from there with string operations.

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

Cyclops numbers

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

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

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

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

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

Domino cycle

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

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

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

Colour trio

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

This problem was inspired by the [Mathologer](#) video "[Secret of Row 10](#)". To start, assume the existence of three values called "red", "yellow" and "blue". These names serve as colourful (heh) mnemonics and could as well have been 0, 1, and 2, or "foo", "bar" and "baz"; no connection to actual physical colours is implied. Next, define a rule to mix such colours so that mixing any colour with itself gives that same colour, whereas mixing any two different colours always gives the third colour. For example, mixing blue to blue gives that same blue, whereas mixing blue to yellow gives red, same as mixing yellow to blue, or red to red.

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

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

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

Count dominators

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

We shall define an element of a list of `items` to be a **dominator** if every element to its right (not just the one element that is immediately to its right) is strictly smaller than that element. By this definition, the last item of the list is automatically a dominator. This function should count how many elements in `items` are dominators, and return that count. For example, the dominators of the list `[42, 7, 12, 9, 13, 5]` would be its elements 42, 13 and 5. The last element of the list is trivially a dominator, regardless of its value, since nothing greater follows it.

Before starting to write code for this function, you should consult the parable of "[Shlemiel the painter](#)" and think how this seemingly silly tale from a simpler time relates to today's computational problems performed on lists, strings and other sequences. This problem will be the first of many that you will encounter during and after this course to illustrate the important principle of using only one loop to achieve in a tiny fraction of time the same end result that Shlemiel achieves with two nested loops. Your workload therefore increases only **linearly** with respect to the number of `items`, whereas the total time of Shlemiel's back-and-forth grows **quadratically**, that is, as a function of the **square** of the number of items.

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

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

Beat the previous

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

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

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

[illegible]

Subsequent words

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

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

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

Now that you know all that, given a list of words sorted in alphabetical order, and a string of required letters, find and return the list of words that contain letters as a *subsequence*.

| letters | Expected result (using the wordlist words_sorted.txt) |
|-----------|---|
| 'klore' | ['booklore', 'booklores', 'folklore', 'folklores', 'kaliborite', 'kenlore', 'kiloampere', 'kilocalorie', 'kilocurie', 'kilogramme', 'kilogrammetre', 'kilolitre', 'kilometrage', 'kilometre', 'kilooersted', 'kiloparsec', 'kilostere', 'kiloware'] |
| 'brohiic' | ['bronchiectatic', 'bronchiogenic', 'bronchitic', 'ombrophilic', 'timbrophilic'] |
| 'azaz' | ['azazel', 'azotetrazole', 'azoxazole', 'diazoaminobenzene', 'hazardize', 'razzmatazz'] |

Taxi \mathbb{Z} um \mathbb{Z} um

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

A lone taxicab cruising the street grid of the dusky Manhattan that we know and love from classic hardboiled *film noir* works such as “[Blast of Silence](#)” starts its journey at the origin $(0, 0)$ of the infinite two-dimensional integer grid, denoted by \mathbb{Z}^2 . Fitting in the gaunt and angular spirit of the time that tolerates few deviations or grey areas, the taxicab is at all times headed straight in one of the four main axis directions, initially north.

This taxicab then executes the given sequence of moves, given as a string of characters 'L' for turning 90 degrees left while standing in place (just in case we are making a turn backwards, in case you spotted some glad rags or some out-of-town palooka looking to be taken for a ride), 'R' for turning 90 degrees right (ditto), and 'F' for moving one block forward to current heading. This function should return the final position of the taxicab on this infinitely spanning Manhattan. (Before the reader objects to the idea of Manhattanization of everything, perhaps the rest of the world lazily simulates this infinite street grid with mirrors?)

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

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

Exact change only

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

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

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

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

Rooks on a rampage

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

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

To achieve this in reasonable time and memory, you should count separately how many rows and columns on the board are safe from any rook. Because **permuting** the rows and columns does not change the answer to this question, you can imagine all these safe rows and columns to have been permuted to form an empty rectangle at the top left corner of the board. The area of that safe rectangle is then obviously the product of its known width and height.

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

Try a spatula

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

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

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

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

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

Chirality

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

Even though this has no effect on fairness, pips from one to six are not painted on dice just any which way, but so that [pips on the opposite faces always add up to seven](#). (This convention makes it easier to tell when someone tries to use crooked dice with certain undesirable pip values replaced with more amenable ones.) In each of the $2^3 = 8$ corners of the cube, exactly one value from each pair of forbidden opposites 1-6, 2-5 and 3-4 meets two values chosen from the other two pairs of forbidden opposites. You can twist and turn any corner of the die to face you, and yet two opposite sides never come together into simultaneous view.

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

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

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

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

Do you reach many, do you reach one?

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

An ordinary [chess knight](#) on a two-dimensional board of squares can make an “L-move” into up to eight possible neighbours. However, as has so often been depicted in various works of space opera, higher beings can generalize the chessboard to k dimensions from our puny two. A natural generalization of the knight's move while maintaining its spirit is to define the possible moves as some k -tuple of **strictly decreasing** nonnegative integer offsets. Each one of these k offsets must be used for exactly one dimension of your choice during the move, either as a positive or a negative version, to determine the square where the k -knight will teleport as the unseen hand of the player moves it there through the dimension $k + 1$.

Given the `start` and `end` positions as k -tuples of integer coordinates, determine whether the knight could legally move from `start` to `end` in a single teleporting jump.

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

A quick combinatorial calculation reveals that exactly $k! \cdot 2^k$ possible neighbours are reachable in a single move, minus the moves that jump outside the board. In this notation, the ordinary chess knight is a (2, 1)-knight that reaches $2! \cdot 2^2 = 8$ neighbours in one jump. A 6-dimensional knight could reach a whopping $6! \cdot 2^6 = 46080$ different neighbours in one jump! Since the number of moves emanating from each position to its neighbours grows exponentially with respect to k , pretty much everything ends up being close to almost everything else in high-dimensional spaces, giving predators too much of an edge over the prey for life to evolve there.

Sevens rule, zeros drool

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

Seven is considered a lucky number in Western cultures, whereas [zero is what nobody wants to be](#). Let us briefly bring these two opposites together without letting it become some kind of emergency by looking at positive integers that consist of some solid sequence of sevens, followed by some (possibly empty) solid sequence of zeros. Examples of integers of this form are 7, 7700, 77777, 777777700, and 70000000000000. A surprising theorem proves that for any positive integer n , there exist infinitely many integers of such seven-zero form that are divisible by n . This function should find the smallest such seven-zero integer.

Even though discrete math and number theory help, this exercise is not about coming up with a clever symbolic formula and the proof of its correctness. This problem is about iterating through the numbers of this constrained form of sevens and zeros efficiently and correctly in strictly ascending order, so that the function can mechanistically find the smallest working number of this form without having any idea of *why* that number is the smallest working. (Or having an idea of *anything*, for that matter. Do computers dream of electric sheep?)

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

[illegible]

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

Fulcrum

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

Each element in `items` is a positive integer, in this problems semantically considered to be a [physical weight](#), as opposed to, say, the current count of chickens. Your task is to find a **fulcrum** position in this list of weights so that when balanced on that position, the total [torque](#) of the items to the left of that position equals the total torque of the items to the right of that position. The item on the fulcrum is assumed to be centered symmetrically on the fulcrum, and does not participate in the torque calculation.



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

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

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

Fail while daring greatly

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

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

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

| n | k | Expected result |
|----|---------|---|
| 4 | 1 | [1, 2, 3, 4] |
| 4 | 2 | [2, 4, 3, 1] |
| 10 | 3 | [3, 6, 9, 2, 7, 1, 8, 5, 10, 4] |
| 8 | 7 | [7, 6, 8, 2, 5, 1, 3, 4] |
| 30 | 4 | [4, 8, 12, 16, 20, 24, 28, 2, 7, 13, 18, 23, 29, 5, 11, 19, 26, 3, 14, 22, 1, 15, 27, 10, 30, 21, 17, 25, 9, 6] |
| 10 | 10**100 | [10, 1, 9, 5, 2, 8, 7, 3, 6, 4] |

All your fractions are belong to base

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

A pile of n identical coins lies on the table. Each move consists of three stages. First, the coins in the remaining pile are arranged into groups of exactly out coins per group, where out is a positive integer greater than one. Second, the $n\% \text{out}$ leftover coins that did not make a complete group of out elements are taken aside and recorded. Third, from each complete group of out coins taken out, exactly ins coins are collected to together create the new single pile, the rest of the coins again put aside for good.

Repeat this three-stage move until the entire pile becomes empty, which must eventually happen whenever $\text{out} > \text{ins}$. Return a list of how many coins were taken aside in each move.

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

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

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

Count the balls off the brass monkey

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

Mysteries of the pyramids have fascinated humanity through the ages. Instead of packing your machete and pith helmet to trek through deserts and jungles to raid the hidden treasures of the ancients like some Indiana Croft, or by gaining visions of enlightenment by intensely meditating under the apex set over a square base like some Deepak Veidt, this problem deals with something a bit more mundane; truncated [brass monkeys](#) of layers of discrete uniform spheres, in spirit of that [spherical cow running in a vacuum](#).

Given that the **top** layer of the truncated brass monkey consists of n rows and m columns of spheres, and each solid layer immediately below the one above it always contains one more row and one more column, how many spheres in total make up this truncated brass monkey that has h layers?

This problem could be solved in a straightforward **brute force** fashion by mechanistically tallying up the spheres iterated through these layers. However, the just reward for naughty boys and girls who take such a blunt approach is to get to watch the automated tester take roughly a minute to terminate! Some creative use of discrete math and summation formulas gives an **analytical closed form formula** that makes the answers come out faster than you can snap your fingers simply by plugging the values of n , m and h into this formula.

| n | m | h | Expected result |
|------------|------------|------------|---------------------|
| 2 | 3 | 1 | 6 |
| 2 | 3 | 10 | 570 |
| 10 | 11 | 12 | 3212 |
| 100 | 100 | 100 | 2318350 |
| $10^{**}6$ | $10^{**}6$ | $10^{**}6$ | 2333331833333500000 |

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

Count growlers

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

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

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

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

Best one out of three

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

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

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

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

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

Collecting numbers

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

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

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

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

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

Three summers ago

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

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

Sure, you could solve this problem with three nested loops to go through all possible ways to choose three elements from `items`, checking for each triple whether it adds up to the `goal`. However, iterating through all triples of elements would get pretty slow as the length of the list increases, seeing that the number of such triples to pick through grows proportionally to the **cube** of the length of the list! Of course the automated tester will make those lists large enough to make such solutions reveal themselves by their glacial running time.

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

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

Sum of two squares

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

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

The returned tuple must present the larger of its two numbers first. Furthermore, if some integer can be expressed as a sum of two squares in several ways, return the breakdown that maximizes the larger number. For example, the integer 85 allows two such representations $7^2 + 6^2$ and $9^2 + 2^2$, of which this function must therefore return `(9, 2)`.

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

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

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

Carry on Pythonista

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

Two positive integers a and b can be added with the usual integer column-wise addition algorithm that we all learned as wee little children so early that most people don't even think of that mechanistic operation as an algorithm! Instead of the sum $a+b$ that the language would already compute for you anyway, this problem asks you to count how many times there will be a **carry** of one into the next column during this mechanistic addition. Your function should be efficient even for behemoths that consist of thousands of digits.

To extract the lowest digit of a positive integer n , use the expression $n\%10$. To extract all other digits except the lowest one, use the expression $n//10$. You can use these simple integer arithmetic operations to traipse through the steps of the **column-wise integer addition** where you don't care about the actual result of the addition, but only tally up the carries produced in each column as **proof of work** of you actually labouring through the steps of the column-wise addition.

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

Expand positive integer intervals

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

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

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

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

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

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

Collapse positive integer intervals

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

This function is the inverse of the previous problem of expanding positive integer intervals. Given a nonempty list of positive integer `items` guaranteed to be in sorted ascending order, create and return the unique description string where every **maximal** sublist of consecutive integers has been condensed to the notation `first-last`. Such encoding doesn't actually save any characters when `first` and `last` differ by only one. However, it is usually more important for the encoding to be uniform than to be pretty. As a general principle, uniform and consistent encoding of data allows the processing of that data to also be uniform in the tools down the line.

If some maximal sublist consists of a single integer, it must be included in the result string all by itself without the minus sign separating it from the now redundant `last` number. Make sure that the string returned by your function does not contain any whitespace characters, and that it does not have a silly redundant comma hanging at the end.

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

That's enough of you!

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

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

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

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

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

Count consecutive summers

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

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

The number of ways that a positive integer n can be represented as a sum of consecutive integers is called its [politeness](#), and can also be computed by tallying up the number of odd divisors of that number. However, note that the linked Wikipedia definition includes only nontrivial sums that consist of at least two components, so according to that definition, the politeness of 42 would be 3, not 4, due to its odd divisors being 3, 7 and 21.

| n | Expected result |
|----|-----------------|
| 42 | 4 |
| 99 | 6 |
| 92 | 2 |

Powers of two are therefore the least polite of all numbers. Perhaps these powers being the fundamental building blocks of all numbers in **binary** representation also made them believe their own hype and balloon up to be too big for their britches. (Fame tends to do that even to the most level-headed of us.) As an exercise in combinatorics, how would you concisely characterize the opposite extreme of “most polite” numbers that can be represented as sums of consecutive integers in more ways than any number less than them?

That's enough for you!

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

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

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

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

What do you hear, what do you say?

```
def count_and_say(digits):
```

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

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

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

Bishops on a binge

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

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

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

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

Revorse the vewels

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

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

Along with many possible ways to perform this dance, one straightforward way to reverse the vowels starts by appending the vowels of `text` into a separate list, and initializing the `result` to an empty string. Then, loop through all characters of the original `text`. Whenever the current character is a vowel, `pop` one from the end of the list of the vowels. Convert that vowel to either upper- or lowercase depending on the case of the vowel that was originally in that position, and add it to `result`. Otherwise, add the character from the original `text` to the `result` as it was.

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

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

Brangelin-o-matic for the people

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

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

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

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

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

Uambcsrln the wrod

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

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

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

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

```

def ryerson_letter_grade(n):
    if n < 50:
        return 'F'
    elif n > 89:
        return 'A+'
    elif n > 84:
        return 'A'
    elif n > 79:
        return 'A-'
    tens = n // 10
    ones = n % 10
    if ones < 3:
        adjust = "-"
    elif ones > 6:
        adjust = "+"
    else:
        adjust = ""
    return "DCB"[tens - 5] + adjust

```

```

def safe_squares_bishops(n, bishops):
    safec = 0
    for i in range(n):
        for a in range(n):
            safe=True
            for b in bishops:
                if abs(i - b[0]) == abs(a - b[1]):
                    safe=False
            if safe:
                safec+=1
    return safec

```

```

def first_preceded_by_smaller(items, k=1):
    leng=len(items)
    for i in range(leng):
        count = 0
        for a in range(i):
            if items[i]>items[a]:
                count += 1
        if k<= count:
            return items[i]

```

```

def count_consecutive_summers(n):
    count = 0
    for i in range(1, n+1):
        s = 0
        for a in range(i, n+1):
            s += a
            if s == n:
                count += 1
    return count

```

```

def collapse_intervals(items):
    final = ''
    i = 0
    leng=len(items)
    while i < leng:
        fir = i
        final += str(items[fir])
        while fir+1 < leng and items[fir]+1 == items[fir+1]:

```

```

        fir += 1
    if fir != i:
        i = fir+1
        final += '-' + str(items[fir])
    else:
        i += 1
    if i != leng:
        final += ','
return final

```

```

def seven_zero(n):
    digit = 1
    final = 0
    while True:
        if n%2 == 0 or n%5 == 0:
            a = 1
            while digit >= a:
                value = int(a * '7' + (digit-a) * '0')
                if value%n == 0:
                    final = value
                    a += 1
            else:
                value = int(digit * '7')
                if value%n == 0:
                    final = value
                else:
                    value=0
            digit += 1
        if final > 0:
            return final

```

```

def remove_after_kth(items, k=1):
    final = []
    for i in items:
        if k > final.count(i):
            final.append(i)
    return final

```

```

def count_carries(a, b):
    count=0
    carry=0
    maxi=max(a,b)
    mini=min(a,b)
    while mini>0 or carry>0:
        x=mini%10
        y=maxi%10
        total=x+y+carry
        if total>9:
            count+=1
            carry=total//10
        else:
            carry=0
        mini//=10
        maxi//=10
    return count

```

```

def group_and_skip(n,out,ins):
    final=[]

    while(n>0):
        rem=n%out

```

```

        final.append(rem)
        group=n//out
        n=group*ins
    return final

def brangelina(first, second):
    i,vowels= 0,'aeiou'
    while second[i] not in vowels:
        i+=1
    second=second[i:]
    groups, in_group=[], False
    for j in range(len(first)):
        if first[j] in vowels:
            if not in_group:
                groups.append(j)
                in_group=True
        else:
            in_group=False
    if len(groups)==1:
        first= first[:groups[0]]
    else:
        first= first[:groups[-2]]
    return first+second

#helper funcction for collect_numbers
def find_inverse(perm):
    leng=len(perm)
    inv = [0]*leng
    for i in range(leng):
        inv[perm[i]] = i
    return inv
def collect_numbers(perm):
    inverse = find_inverse(perm)
    mx=-1
    lengt= len(inverse)
    c=1
    for i in range(lengt):
        if(inverse[i]>mx):
            mx=inverse[i]
        else:
            c+=+1
            mx = inverse[i];
    return c

def pyramid_blocks(n,m,h):
    final=0;
    for i in range(h):
        final+=(n*m)
        n+=1
        m+=1
    return final

def taxi_zum_zum(moves):
    x, y = 0, 0
    direction = 'N'
    for i in moves:
        if i=='F':
            if direction=='S':y-=1
            elif direction=='N': y+=1
            elif direction=='W':x-=1
            elif direction=='E':x+=1

        elif i=='L':

```



```

        if direction=='E':direction='N'
        elif direction=='N': direction='W'
        elif direction=='S':direction='E'
        elif direction=='W':direction='S'

    elif i=='R':
        if direction=='E':direction='S'
        elif direction=='N': direction='E'
        elif direction=='S':direction='W'
        elif direction=='W':direction='N'

    return (x,y)
def words_with_letters(words, letters):
    final = []
    for i in words:
        count_a = 0
        count_b = 0

        while count_b<len(i) and count_a<len(letters):
            if i[count_b] == letters[count_a]:
                count_a += 1
                count_b += 1
            if count_a==len(letters):
                final.append(i)
    return final

def is_cyclops(n):
    n= str(n)
    if len(n)% 2 == 1:
        if n.count("0")>1:
            return False
        if n[((len(n)+1)//2)-1]=="0":
            return True
        else:
            return False
    else:
        return False

def is_ascending(items):
    if len(items) == len(set(items)):
        if sorted(items)== items:
            return True
        else:
            return False
    else:
        return False

def riffle(items, out=True):
    if items==[]:
        return items
    final = []
    half= len(items)//2
    if out==True:
        half1= items[:half]
        half2= items[half:]
    else:

```

```

    half1= items[half:]
    half2= items[:half]
i=0
while i < half:
    final.append(half1[i])
    final.append(half2[i])
    i=i+1
return final

def only_odd_digits(n):
    if n<= 0:
        return False
    list= []
    while n >0:
        list.append(n%10)
        n//=10
    list=list[::-1]

    for n in list:
        if all(int(n) % 2 == 1 for n in list):
            return True
        else:
            return False

def domino_cycle(tiles):
    a = len(tiles)
    if a == 0:
        return True
    elif(tiles[0][0] == tiles[-1][1]):
        for i in range(a-1):
            if tiles[i][1] != tiles[i+1][0]:
                return False
        else:
            return True
    else:
        return False

def pancake_scramble(text):
    a= text
    b=len(text)
    for c in range(2,b+1):
        a= a[:c][::-1]+a[c:]
    return a

def count_dominators(items):
    l=len(items)
    if l == 0:
        return 0
    maxim = items[-1]
    count = 1
    for i in range(-2, -len(items) - 1, -1):
        if items[i] > maxim:
            count +=1
        if maxim < items[i]:
            maxim = items[i]

    return count

```

```

def m_of_3(a,b,c):
    return sorted([a,b,c])[1]
def tukeys_ninthers(items):
    while len(items)>1:
        medians= []
        for i in range(0,len(items),3):
            m=m_of_3(items[i], items[i+1], items[i+2])
            medians.append(m)
        items= medians
    return items[0]

def count_and_say(digits):
    if len(digits) == 0:
        return ""
    else:
        final = ""
        fir = digits[0]
        count = 1
        for i in digits[1:]:
            if i == fir:
                count += 1
            else:
                final += str(count)
                final += fir
                count = 1
            fir = i
        final += str(count)
        final += fir
        return final

def unscramble(words,word):
    final=[]
    leng= len(word)
    for i in words:
        if leng ==len(i) and i[-1]==word[-1] and i[0]==word[0] :
            if sorted(list(word[1:-1]))==sorted(list(i[1:-1])):
                final.append(i)
    return final

def is_left_handed(pips):
    combo=[(1,2,3), (2,3,1), (3,1,2), (1,4,2), (2,1,4), (4,2,1), (1,3,5), (3,5,1), (5,1,3),
(1,5,4), (4,1,5), (5,4,1), (2,6,3), (3,2,6), (6,3,2), (2,4,6), (4,6,2), (6,2,4), (3,6,5),
(5,3,6), (6,5,3), (4,5,6), (5,6,4), (6,4,5)]
    if pips in combo:
        return True
    else:
        return False

def extract_increasing(digits):
    final = []
    current = 0
    previous = -1
    for i in range(len(digits)):
        d = int(digits[i])
        current = 10 * current + d
        if current > previous:
            final.append(current)
            previous = current
            current = 0
    return final

def josephus(n,k):

```

```

    sold=[(i+1) for i in range(n)]
    st=0
    lis=[]
    sz=n
    while len(sold)>1:
        st=(st+(k-1))%sz
        sz-=1
        lis.append(sold[st])
        del sold[st]
    lis.append(sold[0])
    return lis

def expand_intervals(intervals):
    intr = intervals.split(",")
    lst = []
    if len (intervals) == 0:
        return lst
    for num in intr:
        num = num.strip()
        if "-" in num:
            first = int(num.split("-")[0])
            last = int(num.split("-")[1])
            for i in range(first, last + 1):
                lst.append(i)
        else:
            lst.append(int(num))
    return lst

#helper funtion for three_summers
def two_summers(items, goal,i=0):
    j= len(items)-1
    while i< j:
        s = items[i]+items[j]
        if s ==goal:
            return True
        elif s <goal:
            i+=1
        else:
            j-=1
    return False

def three_summers(items,goal):
    for k in range(len(items)):
        if (two_summers(items,goal-items[k],k+1)):
            return True
    return False

def reverse_vowels(text):
    result, vowels = "", 'aeiouAEIOU'
    vowelList= [c for c in text if c in vowels]
    for c in text:
        if c not in vowels:
            result=result+c
        else:
            v= vowelList.pop()
            v= v.lower() if c.islower() else v.upper()
            result=result+v
    return result

def sum_of_two_squares(n):
    square = int(n ** 0.5)
    for i in range(square, 0, -1):
        tem = n - i * i
        if int(tem ** 0.5) ** 2 == tem and int(tem ** 0.5) > 0:

```

```
    return i, int(tem ** 0.5)
```

```
def can_balance(items):  
    for i in range(len(items)):  
        left = 0  
        right = 0  
        for j in range(i):  
            left += items[j]*(i-j)  
        for j in range(i+1, len(items)):  
            right += items[j]*(j-i)  
        if left == right:  
            return i  
    else:  
        return -1
```

```
def colour_trio(colours):  
    length=len(colours)  
    while len(colours) > 1:  
        final = ""  
  
        for i in range(len(colours)-1):  
            colour_1 = colours[i]  
            colour_2= colours[i+1]  
  
            if colour_1 == colour_2:  
                final += colour_1  
  
            else:  
                final += "ybr".replace(colour_1,"").replace(colour_2,"")  
        else:  
            colours = final  
  
    return colours
```

```
def give_change(amount, coins):  
    final = []  
    ind = 0  
    while amount > 0:  
        if amount >= coins[ind]:  
            final.append(coins[ind])  
            amount = amount - coins[ind]  
        else:  
            ind+=1  
  
    return final
```

```
def count_growlers(animals):  
  
    dogs = []  
    cats = []  
    dog_count = 0  
    cat_count = 0  
    for animal in animals:  
        if(animal == 'dog' or animal == 'god'):  
            dog_count += 1  
            dogs.append(dog_count)  
            cats.append(cat_count)  
        else:  
            cat_count += 1  
            cats.append(cat_count)  
            dogs.append(dog_count)
```

```

final = 0
a = 0
while a < len(animals):
    if animals[a] == 'dog' or animals[a] == 'cat':
        if a > 0 and dogs[a-1] > cats[a-1]:
            final += 1

        elif animals[a] == 'god' or animals[a] == 'tac':
            if (dogs[len(dogs)-1] - dogs[a]) > (cats[len(cats)-1] - cats[a]):
                final += 1
    a += 1

return final

```

```

def safe_squares_rooks(n,rooks):
    unsafe_r=set()
    unsafe_c=set()
    for rook in rooks:
        unsafe_r.add(rook[0])
        unsafe_c.add(rook[1])
    count_r=n-len(unsafe_r)
    count_c=n-len(unsafe_c)
    return count_r*count_c

```

```

def knight_jump(knight, start, end):
    new = []
    for i in range(len(knight)):
        new.append(knight[i])
    for i in range(len(knight)):
        val = abs(start[i] - end[i])
        if val in new:
            new.remove(val)
            continue
    else:
        return False
    return True

```