Third Collection of Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of August 1, 2025

This document contains the specifications for a third collection of Python problems created by Ilkka Kokkarinen starting from March 2023, to augment the original set of 109 Python Problems designed for the course **CCPS 109 Computer Science I** for Chang School of Continuing Education, Toronto Metropolitan University, Canada (formerly Ryerson University).

Same as in the original problem collection, these problems are listed roughly in order of increasing difficulty, but the reader shouldn't get too hung up on that, since that estimation is basically a gut feeling anyway and depends on the propensities and interests of the reader. The complexity of the solutions for these additional problems ranges from straightforward loops up to convoluted branching recursive backtracking searches that require all sorts of clever optimizations to prune the branches of the search sufficiently to keep the total running time of the test suite within the twenty second time limit.

The rules for solving and testing these problems are exactly the same as they were for the original 109 problems. You must implement all your functions in a single source code file that must be named `labs109.py`, the exact same way as you wrote your solutions to the original 109 problems. The automated test suite to check that your solutions are correct, along with all associated files, is available in the GitHub repository `ikokkari/PythonProblems`.

Python coders of all levels will hopefully find something interesting in this collection. The author believes that there is room on Earth for all races to live, prosper and get strong at coding while having a good time by solving these problems. Same as with the original 109 problems, if some problem doesn't spark joy to you, please don't get stuck with that problem, but just skip it and move on to the next problem that interests you. Life is too short to be wasted on solving stupid problems.

Table of Contents

Multiply and sort

```
def multiply_and_sort(n, multiplier):
```

The following problem, unearthed from the post "Double (or triple) and sort" in Michael Lugo's blog "God Plays Dice" ("where randomness meets divine intervention"), explores an iterated numerical transformation with curious properties and overall behaviour. Starting from a positive integer `n`, multiply the current number by the given `multiplier`, and sort the resulting digits in ascending order to produce the next number in the sequence. Leading zeros, being the shy digits that they are, quietly disappear from the front.

For example, with n = 513 and `multiplier` = 2, the product 1026 becomes 0126 after sorting, which simplifies to 126 (the zero, true to form, showing itself out). This process continues until some integer repeats itself in the sequence, at which point the sequence admits defeat and stops. Your task in writing this function is to return the largest integer encountered during this mathematical game of multiply-and-tidy-up.

`n`	`multiplier`	Expected result
`2`	`5`	`4456`
`3`	`4`	`2667`
`5`	`5`	`4456`
`20`	`7`	`15556688`
`31`	`7`	`12345678888888888889`

As one commenter of the original post pointed out, this process seems to always terminate up to multipliers up to 20, but then freezes for some multipliers from 21 onwards and behaves well for the others. Interested readers might want to further explore this behaviour and prove some necessary and sufficient conditions for termination. The automated tester will, of course, give your function only numbers for which this sequence will terminate in a reasonable time.

Approval voting

```
def approval_voting(ballots):
```

Approval voting is an alternative to the traditional first-past-the-post single winner election scheme. Instead of voting for just one favoured candidate, every voter marks every candidate in the ballot for either approval or disapproval. The candidate who gets the most approvals tallied over all the ballots wins the election.

For the purposes of this problem, assume that the *n* candidates have been numbered 0, …, *n* – 1, and each ballot is a text string of length *n* whose each character is either `'Y'` or `'N'` depending on whether that voter goes "yea" or "nay" for that candidate. This function should tally up the approvals for each candidate, and return the index of the winning candidate. If multiple candidates receive the same number of approvals, your function should resolve the tie for the candidate with the lowest position index.

`ballots`	Expected result
`['NYN', 'NYN', 'NYN', 'YNN', 'YYN']`	1
`['NYY', 'NYY', 'YYN', 'NYY', 'NNY', 'NYY']`	1
`['NYNYN', 'NYNYY', 'YYYNY', 'YYYYN', 'YNNNY', 'YNNNY', 'NYYNN', 'YNNYN']`	0

Vigenère cipher

```
def vigenere(text, key, direction):
```

The Vigenère cipher is a method of cryptography from almost five centuries ago. Simple enough to allow both encryption and decryption to be performed by hand even under battlefield conditions, this method was nearly impossible to crack with the methods of its era by someone who was not in possession of the agreed encryption `key`, at least within the time limit when the encrypted information would have made a difference in the battlefield.

Using the modern English alphabet, the Vigenère method encrypts the given `text` one character at the time, replacing each character with that character shifted as many positions to the right in the alphabet (wrapping around after `z`) as given by the corresponding character of the `key`. The character `a` denotes a shift of zero, the character `b` denotes a shift of one, and so on, up to the character `z` that denotes a shift of 25.. If the `key` is shorter than the text, the `key` is concatenated with itself as many times as needed to stretch it to sufficient length. To decrypt the `text` with the same `key`, simply shift each character that same amount in the opposite direction.

Given the `text` and `key`, in this problem both guaranteed to consist of the 26 lowercase letters of English only, this function should perform both encryption (if `direction` equals +1) and decryption (if `direction` equals –1) of the `text` with the given `key`.

`text`	`key`	`direction`	Expected result
`'locoedturse'`	`'adsum'`	`+1`	`'lruiqdwmlee'`
`'locoedturse'`	`'adsum'`	`–1`	`'llkusdqcxge'`
`'marorcotoin'`	`'kafiz'`	`+1`	`'wawwqmoywhx'`
`'marorcotoin'`	`'kafiz'`	`–1`	`'camgssoogjd'`

It is also easy to see that the Vigenère cipher contains both Caesar cipher and rot-13 as trivial special cases, the `key` being a singleton letter in both cases.

Bug in a line

```
def bug_in_a_line(board):
```

The following problem is from Peter Winkler's "Mathematical Puzzles: A Connoisseur's Collection" where this simple system is shown to have interesting mathematical properties over all possible boards. Here we shall content ourselves into merely executing the system for the given starting `board`. A smol cartoon bug stands in the position 0 of the `board` that is a string whose characters are traffic lights, either `R`ed, `Y`ellow, or `G`reen. The colour of the traffic light in the cell that the bug is at determines its next move:

- If the current light is **green**, the bug flips that light to be **yellow**, and moves one step **right**.
- If the current light is **yellow**, the bug flips that light to be **red**, and moves one step **right**.
- If the current light is **red**, the bug flips that light to be **green**, and moves one step **left**.

The bug therefore moves left roughly one third of the time, and right roughly two thirds of the time. This function should iterate the steps taken by the bug through the `board` until it falls off from either edge, and return the total number of steps taken by the bug to fall off. You will surely first convert the `board` to a `list` instead of an immutable string, so that you can locally modify the traffic light that the bug flips in constant time, regardless of the length of the `board`.

`board`	Expected result
`'GGR'`	`5`
`'RRGY'`	`1`
`'GYRGYYRRRR'`	`26`
`'GGYRRYYYYRYRYG'`	`34`

The bug cannot end up wandering forever on a finite `board`, but must eventually fall off from either end. Were the bug traversing a finite board infinitely long, it would have to visit some cells infinitely often, so let k be the lowest position index of these infinitely often visited cells. Since that cell is red during one third of these visits, the bug would also end up visiting the cell $k - 1$ infinitely often, a contradiction to the assumption that k was the lowest position index with that property. (This type of proof of contradiction is called a **proof of infinite descent**.)

Friendship paradox

```
def friendship_paradox(friends):
```

Each person in a group of n people numbered 0, ..., *n* − 1 has at least one friend in this group, with `friends[i]` giving the list of friends of person *i*. Friendship or its absence is always symmetric between any two people. (After all, unrequited friendship belongs in middle school diary entries of overly dramatic emo theatre kids, not in serious and manly study of graph theory.)

This problem investigates a curious phenomenon known in the literature as the friendship paradox: on average, an individual's friends have more friends than that individual. While this may sound like a manifestation of social anxiety and everyone else seemingly having cooler lives than you, this phenomenon has a rigorous mathematical basis in that more popular people appear as someone's friends more frequently than less popular people, which skews the average number of friends that your friends have compared to your own friendship count.

Given the list of lists of `friends` for the people in the group, this function should compute and return two numerical quantities as a two-element tuple: the average number of friends per person, and the average number of friends among the friends per the average person. Both quantities should be computed and returned as exact `Fraction` objects, because real mathematicians don't round their social calculations.

`friends`	Expected result
`[[1], [0, 2], [1]]`	`(Fraction(4, 3), Fraction(5, 3))`
`[[1, 3], [2, 0, 3], [1, 3], [1, 0, 2]]`	`(Fraction(5, 2), Fraction(8, 3))`
`[[1, 4, 3, 2], [3, 2, 4, 0], [1, 0, 3], [4, 1, 0, 2], [0, 3, 1]]`	`(Fraction(18, 5), Fraction(37, 10))`

As a coding challenge, the enjoyers of Python **generator expressions** can try to write the part of the function that computes the sum of friends that the friends of a particular person as a one-liner.

Right turn at Albuquerque

```
def albuquerque_stretch(text):
```

This cute little problem, suitable for a coding exercise for a Python intro course, is "Albuquerque challenge" recently posted in Code Golf Stack Exchange but of course here we shall aim for correctness and clarity instead of the extreme brevity of the source code. This function should first initialize the `result` to an empty string, and then loop through the characters `c` of `text` in order from left to right. At the first occurrence of that character `c`, simply append that character to the `result`. From the second occurrence onwards, append the substring of the original `text` spanning from the most recent occurrence of `c` to its current occurrence, both endpoints inclusive, to the `result`.

You can trace the steps of the first test case `'success'` either by yourself or follow its steps on the original web page linked above for a concrete explanation of these rules. It is probably easiest to use a Python dictionary to remember the characters that you have already seen in the loop, each character associated with the position of its previous occurrence in the `text`.

`text`	Expected result
`'success'`	`'succcesuccesss'`
`'albuquerque'`	`'albuququerquerquerquerque'`
`'glarrota'`	`'glarrrotarrota'`
`'dontgangseedmmf'`	`'dontgantgangangseeedontgangseedmmmf'`

Generalized Fibonacci sequence

```
def generalized_fibonacci(multipliers, n):
```

Fibonacci sequence is a dusty cliche in teaching computer science, usually presented as the silly exponential time recursion, but better implemented as a loop that produces the elements one at the time based on the previous elements. Since we are going to write this function, we might as well implement a general version that can produce not just the terms of the plain old Fibonacci sequence, but any generalized Lucas sequence where each term F_i is computed as a sum of some m immediately preceding values F_{i-1}, F_{i-2}, ... , F_{i-m}, each preceding term multiplied with the given fixed multiplier a_j applied to the term F_{i-j}. For example, the classic Fibonacci sequence is defined with the recursive equation $F_i = F_{i-1} + F_{i-2}$ where the multipliers of the two preceding terms are $a_1 = a_2 = 1$. Pell numbers would be defined with equation $F_i = 2F_{i-1} + F_{i-2}$ where the terms are $a_1 = 2$ and $a_2 = 1$.

Given the `multipliers` as a tuple of m elements, compute the term in position `n` of the sequence. Position numbering starts from 0, and the first terms in positions 0, ..., $m - 1$ of the sequence all equal 1. The order of `multipliers` is such that the last multiplier in position `-1` is applied to the term immediately preceding the current term being computed, the second last multiplier in position `-2` is applied to the term before that term, and so on.

`multipliers`	`n`	Expected result
`(1, 1)`	`5`	`8`
`(1, 2)`	`8`	`577`
`(4, 4, -3, -7, 5, 0, 7)`	`31`	`2579538296491128384859`
`(7, -7, 9, 9, -7, 1, -3, 9, 4)`	`35`	`416464779917747725480`

Bays–Durham shuffle

```
def bays_durham_shuffle(item_stream, k):
```

The quality and the **period** of the sequence of pseudorandom numbers produced by any simple random number generator can be massively improved with the **Bays–Durham shuffle algorithm** that is ingenious yet simple enough for us to implement here with a handful of lines of Python code.

This function receives a Python **iterator** named `item_stream` that produces a sequence of non-negative integers, to which this function should apply the Bays–Durham shuffle in the following fashion. Unlike in the other problems that you have seen in this and previous two Python problem collections, this iterator does not support **random access indexing** at arbitrary positions, but will produce the elements one at the time when you apply the Python `next` function to it, raising the `StopIteration` exception when it has no elements left.

Use the first `k` items from the stream to build a `table` of *k* elements, and initialize the `result` list to be empty. Then loop through the remaining *n* – *k* items from the stream, and perform the following operation for each such item *e*: replace the element *x* currently in the position (*e* mod *k)* by the item *e*, and append the removed element *x* into the result list. For example, suppose that *k* = 4 and the current `table` contains the values `[17, 42, 99, 5]`, and the next item from the stream equals 55. Compute the table position with formula 55 % 4 = 3, replace the element 5 in that position by 55, and append that replaced element 5 into the result list, giving you the new `table` with values `[17, 42, 99, 55]`. This function should return the list of *n* – *k* values computed from the stream in this manner, after the `table` of first *k* elements of the stream.

(Inquiring minds would also want to know, of course, what would happen if the Bays–Durham shuffle were iteratively applied to the random sequence already shuffled in that manner.)

Double-ended pop

```
def double_ended_pop(items, k):
```

You are given a list of `items`, each one a nonnegative integer, and are allowed to do the following operation `k` times; remove an element from either the beginning or the end of the `items` list. This function should return the largest possible sum of items that can be built up with *k* such removal operations.

`items`	`k`	Expected result
`[13, 0, 9, 11, 6, 8, 18, 13, 5, 11]`	`4`	`47`
`[7, 11, 17, 10, 5, 12, 10, 0, 1, 1, 14, 2, 12]`	`4`	`47`
`[17, 9, 0, 15, 19, 18, 7, 10, 20, 2, 0, 0, 20, 13, 3, 9, 14, 2, 2]`	`7`	`85`
`[21, 14, 24, 0, 21, 7, 21, 1, 4, 11, 18, 20, 0, 6, 0, 9, 21, 12, 16, 20]`	`5`	`95`

First singleton

```
def first_singleton(text):
```

This function should find and return the first character in the `text` that occurs in that `text` exactly once. If the `text` does not contain any such singleton characters, return `None`.

Of course your function has to look at every character of the `text` before it can return the answer with absolute confidence. However, with the aid of Python sets and dictionaries to keep track of the characters that you have seen once or more than once, your function should be a single for-loop through the `text` that processes each character in constant time, and thus be blazingly fast for thousands of strings that contain thousands of characters each.

`text`	Expected result
`'gjexx'`	`'g'`
`'bababa'`	`None`
`'dddhmsddhdsmdrdhdddnddrs'`	`'n'`
`'fzmfomfpzmzfpomxfculzmfcqm'`	`'x'`

Shell sort

```
def shell_sort(permutation, gaps):
```

Insertion sort is the first serious sorting algorithm that most computer science students learn and analyze and implement as an exercise. Insertion sort is provably optimal among all comparison sort algorithms that are constrained to compare and swap adjacent elements, as it will perform exactly as many comparisons and swaps as are necessary and sufficient to sort the list with such swaps.

Faster sorting algorithms must necessarily compare and swap elements across longer distances over the list to quickly get them closer to their final resting places. Shell sort is a generalization of insertion sort that operates in multiple rounds, each time using a different gap value from the `gaps` sequence in decreasing order. In each round, loop through positions from the current gap value g up to $n - 1$, and repeatedly compare and swap the element in that position with the element located g steps before in in the list until either the gap g would lead past the beginning of the array, or the element g steps before the current position less than or equal to the current element. Since the last gap size always equals one, the last round performs an ordinary insertion sort that is guaranteed to sort the list regardless of how the previous rounds have shaken the elements around.

This function should implement the shell sort algorithm according to the pseudocode given in the linked Wikipedia page. However, instead of returning the sorted array, as **proof of work** that you genuinely did implement the actual Shell sort algorithm without any underhanded activity, your function should instead return the two-tuple `(comparisons, swaps)` of how many element comparisons and element swaps were performed during the algorithm for the given `permutation` of integers 0, ..., $n - 1$ using the given sequence of `gaps`.

`permutation`	`gaps`	Expected result
`[5, 4, 1, 2, 0, 3]`	`[1, 3]`	`(12, 7)`
`[6, 0, 5, 1, 3, 4, 2]`	`[1, 3, 5]`	`(17, 8)`
`[9, 7, 3, 5, 0, 1, 12, 6, 4, 2, 10, 8, 11]`	`[1, 2, 3, 4, 6, 8, 9, 12]`	`(70, 15)`
`[13, 7, 11, 9, 10, 4, 8, 14, 3, 5, 0, 2, 12, 6, 1]`	`[1, 4, 9]`	`(59, 35)`

Maximum repeated suffix

```
def maximum_repeated_suffix(items):
```

This function should find the length of the **maximum repeated suffix** of the given sequence of `items`, that is, the longest suffix that is immediately preceded in that sequence by that very same suffix. This function should not use list slicing to extract the suffix, but perform the comparison in the same list by looping through the position indices, which should make this function fast even for the barrage of thousands of lists that each contain thousands of elements thrown at this function by our automated fuzz tester.

`items`	Expected result
`[1, 3, 2, 3]`	0
`[3, 0, 3, 0]`	2
`[2, 0, 1, 3, 1, 1, 3, 1]`	3
`[1, 0, 0, 3, 0, 1, 5, 0, 5, 5, 0, 3, 0, 1, 5, 0, 5, 5]`	8

Count slices with given sum

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

Given a list of `items`, each item guaranteed to be a positive integer, this function should count the number of separate **slices** in this list whose elements add up to the given `goal`. These slices are allowed to partially overlap if necessary, and can consist of a singleton element equal to the `goal`.

You could, of course, iterate through all possible starting positions of slices, and then looping from each starting position, add up elements until the total becomes greater or equal to the `goal`. However, to make this algorithm way more efficient, you should note that if you already have tallied up the sum of elements of some particular slice, you can easily compute the sum of elements of that slice without its first element simply by subtracting that first element from the tally.

This approach allows you to compute the answer in a single linear time loop through the entire list. Use two indices to mark the starting and ending positions of the current slice, both indices starting at the first element. If the sum of items in the current slice is less than `goal`, advance the end index, and otherwise advance the start index, first incrementing the tally whenever the sum of items equals `goal`. Since each round of the loop advances at least one of the indices, the algorithm is guaranteed to finish in time that is linear with respect to the number of elements in the list.

Such a **two pointers** approach of using two indices to delimit the sublist in question is a common technique to solve many problems on a list in linear instead of quadratic time. In problems like this one, these two indices act as "point man" and "rear guard", never retreating but always advancing towards the enemy until the victory has been reached. (In some other problems, for example the "two summers" problem from course examples, the indices start from opposite ends of the list and always advance towards each other until they meet somewhere in the middle.) As of this writing, the LeetCode category "Two pointers" offers 219 problems amenable to two pointers technique.

`items`	`goal`	Expected result
`[2, 1, 1, 1, 2, 1]`	`2`	`4`
`[1, 3, 2, 1, 2, 1, 3, 2, 1, 2]`	`3`	`7`
`[1, 4, 3, 1, 4, 3, 2, 1, 3, 1, 3, 1, 1, 2, 1]`	`8`	`8`
`[3, 4, 4, 1, 1, 3, 1, 4, 2, 3, 4, 1, 3, 5, 5, 4]`	`14`	`7`

First sublist whose sum is a multiple of *k*

```
def sublist_sum_k(items, k):
```

Right on the heels of the previous problem swerves in another one on the same theme; determine whether the given list of positive integer `items` contains at least one nonempty contiguous sublist whose sum of elements is an integer multiple of *k*. If at least one such sublist exists, return the start position of the earliest such sublist, otherwise return `None`.

To avoid being a "Shlemiel" who solves this problem by looping through all possible sublists of `items`, we need to brush up a little bit on elementary number theory to get to a working solution that consists of a single for-loop through the `items` aided by some additional data structures. You should keep track of the running **prefix sum** of the elements that the loop has processed so far, and at each element, compute the remainder of the current prefix sum when divided by `k`. Keeping track of all the remainders that you have seen during this loop along with the positions of these remainders allows you to quickly recognize not only such a sublist but also its starting position.

`items`	`k`	Expected result
`[1, 10, 15, 23, 35, 22]`	`25`	`1`
`[24, 14, 33, 8, 31, 4]`	`32`	`None`
`[79, 80, 40, 66, 37, 4, 27, 55, 51, 88]`	`133`	`6`
`[87, 187, 71, 125, 4, 175, 81, 110, 66, 124, 28, 66, 87, 193]`	`454`	`None`

Ones mule, zeros pool

```
def one_zero(n):
```

The original 109 Python Problems collection featured a problem “Sevens rule, zeros drool” that asked the reader to find the smallest integer *m* whose positional base-10 representation is a solid nonempty sequence of sevens followed by a solid (possibly empty) sequence of zeros so that this *m* is divisible by the given *n*. An integer *m* of this restricted form satisfying this requirement was always guaranteed to exist. In the same vein, this problem now asks you to find the smallest integer *m* whose positional base-10 representation consists of a solid nonempty sequence of ones followed by a solid (possibly empty) sequence of zeros.

The clever proof to establish the existence of such *m* also immediately gives us a method to construct it. Your function through the **repdigit** integers 1, 11, 111, 1111, 11111, ..., noting the remainder left over when each repdigit integer is divided by *n*. If you come upon a repdigit integer that is divisible by *n*, well then, that's the one that you need. Alternatively, whenever you find two repdigit integers r_1 and r_2 with the same remainder, their difference $r_1 - r_2$ is also divisible by *n*, and is trivially of the required form! Use a dictionary to map the *n* possible remainders to the repdigit numbers that produced them, so that you can quickly recognize that you have already seen the current remainder earlier in the repdigit sequence and subtract those two repdigit numbers.

The famous **pigeonhole principle** guarantees that you must hit the answer after at most *n* steps. (In fact, the equally important **birthday paradox** ensures that you will usually hit the required answer much sooner, on average roughly around the square root of *n*.)

n	Expected result
1	1
3	111
4	100
11	11
28	11111100
118	11111111111111111111111111111111111111111111111111111111111110

Poker test of randomness

```
def poker_test(sequence):
```

Statistical tests to evaluate the quality of **pseudorandom number generators** calculate various statistical quantities for the pseudorandom sequences that they generate, and compare those quantities to the quantities that a truly random sequence would be mathematically expected to have.

One such test looks at all the *n* – 4 overlapping sublists of five elements in the given `sequence` of *n* elements, and classifies each sublist according to which type of hand in **dice poker** that subsequence of five elements corresponds to. This function should return the list of counts of each type of hand in the exact order of the following seven hand types; five of a kind, four of a kind, full house, three of a kind, two pair, one pair, and high card.

`sequence`	Expected result
`[0, 3, 6, 6, 2, 6, 0, 3, 1, 1, 2, 1, 1, 4]`	`[0, 1, 0, 4, 0, 4, 1]`
`[6, 4, 6, 4, 4, 4, 4, 4, 4, 2, 4, 4, 2, 4, 2, 2, 2, 5, 1]`	`[2, 6, 5, 2, 0, 0, 0]`
`[4, 7, 1, 7, 7, 1, 3, 8, 3, 8, 8, 8, 8, 8, 8, 8, 8, 1, 1, 1, 2, 1]`	`[4, 4, 4, 2, 2, 2, 0]`
`[10, 4, 2, 0, 5, 2, 0, 6, 7, 11, 2, 3, 5, 3, 10, 6, 11, 10, 10, 7, 8, 3, 10, 2, 4, 10, 11]`	`[0, 0, 0, 1, 1, 12, 9]`

Lehmer code: encoding

```
def lehmer_code(permutation):
```

In theory of permutations, an **inversion** is a pair of positions $i < j$ in that permutation so the element in the position i is larger than the element in the position j. The count of how many inversions the permutation contains measures how "out of order" that permutation is compared to the perfectly sorted permutation whose elements are in ascending order.

The **right inversion count**, also called the Lehmer code of a permutation of n distinct elements, is itself a list of $n - 1$ elements. The element in the position i of the Lehmer code gives the count of inversions where that position appears as the first position of the pair $i < j$. Alternatively, the element in the position i of the Lehmer code counts how many elements after that position are smaller than the element in position i. Since this count would always be zero for the last position of the permutation, this redundant zero is left out of the Lehmer code.

This function should return the Lehmer code for the given `permutation` of integers 0, ..., $n - 1$.

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

Since the result of the Lehmer code can always be further interpreted as a factoradic number (see the later problem "Factoradical dudes" in this same collection), this allows the permutations of integers 0, ..., $n - 1$ to be encoded **bijectively** ("one-to-one") to the integers 0, ..., $n! - 1$. The inverse versions of these same two functions applied in reverse order will then decode these nonnegative integers back to the original permutations.

Sum of square roots

```
def square_root_sum(n1, n2):
```

One big thing that this instructor tries to do differently compared to most teachers of introductory Python courses is to train his students to instinctively avoid using floating point numbers, the same as how they would avoid following the voice coming from the sewer to tell them that they are all just "floating" down there. However, paraphrasing the life story of the reformed mob *caporegime* Michael Franzese with the popular YouTube channel, if you are going to be in the streets, then you have to be in the streets the right way. Instead of using the binary floating point type given by your computer's processor, you should use the `Decimal` type defined in the `decimal` module to perform your floating point calculations in base ten to the desired arbitrary precision.

Given two lists of positive integers, determine whether the sum of square roots of the numbers in the first list is strictly larger than the sum of square roots of the numbers in the second list. You should perform these computations using the `Decimal` type increasing the precision until the two sums of square roots are distinct enough.

n1	n2	Expected result
`[6, 9, 13]`	`[5, 10, 12]`	`True`
`[7, 13, 14, 18, 22]`	`[8, 12, 15, 17, 23]`	`False`
`[15, 20, 24, 28, 29]`	`[16, 19, 25, 27, 30]`	`False`
`[14, 17, 20, 23, 24, 28]`	`[15, 16, 21, 22, 25, 27]`	`False`

From the point of view of computational complexity, this problem is much harder than it might seem. The intricacies of integer mathematics and irrational numbers make it difficult to determine how many decimal places the addition needs to be performed to be absolutely sure about which sum is larger. The exact placement of this problem to a specific complexity class is still an open question in computability theory.

Loopless walk

```
def loopless_walk(steps):
```

Another neat one from Stack Overflow Code Golf. A series of `steps` visits some letters in the given order. However, we want to shorten this journey by repeatedly performing the following operation as many times as it is possible: find the first letter *c* that occurs a second time in `steps`, and remove every letter in all positions following first occurrence of *c*, up to and including the second occurrence. For example, this operation would turn `'baflac'` into `'bac'`. Once this operation can no longer be performed since no letter occurs twice in `steps`, return the `steps` that remain.

This operation is pretty straightforward in theory, but making it efficient for long strings might require some thinking to avoid looping through the same prefixes of the string redundantly multiple times, or having to build up new strings by extracting some small part near the front.

`steps`	Expected result
`'zzz'`	`'z'`
`'aasaa'`	`'a'`
`'wczzzv'`	`'wczv'`
`'bhkzmrvfr'`	`'bnkzmr'`
`'llpllllllnell'`	`'l'`

Fourth seat opening

```
def fourth_seat_opening(hand):
```

Sitting at the bridge table in yet another tournament instead of touching grass like a normal person, the hand sees three players pass in front of you, and you need to decide whether to open the bidding or pass out the hand altogether. A handy rule of thumb of rule of fifteen, also historically called the **Cansino count**, says to add up your raw **high card points** (4 four each ace, 3 for each king, 2 for each queen, and 1 for each jack) and the number of spade cards in your hand regardless of their ranks, and open the bidding if this count adds to 15 or more. The spade suit is special since it's the "boss suit", and you would like the opponents to rue the day afterwards "We missed our spade fit, partner, you should have known to open, herpa derpa derp" whenever you are short in that suit.)

Nothing is sure in life or bridge, but this rule has the highest expected value in this decision of uncertainty, over all possible distributions of cards for other three players that is consistent with the observed evidence of all three passing out. According to bridge author Larry Cohen, it is also profitable to open in the fourth seat with a pre-emptive bid if you hold a major suit of at least six cards or minor suit of at least seven cards, and have at least 10 high card points. With 14 high card points, you still open but at one level, since you don't want to miss your game contract when the partner has a good limit raise in your major. (Most serious partnerships don't use two level pre-empts for minors, but give the 2♣ and 2♦ opening bids a more important artificial meaning.)

Your `hand` is once again encoded the same way as in the example program `cardproblems.py`. This function should determine whether it is profitable to open the given `hand` in the fourth seat.

`hand`	Expected result
`[('ten', 'hearts'), ('three', 'diamonds'), ('king', 'hearts'), ('seven', 'hearts'), ('six', 'hearts'), ('four', 'diamonds'), ('four', 'hearts'), ('king', 'spades'), ('two', 'spades'), ('eight', 'hearts'), ('seven', 'diamonds'), ('two', 'diamonds'), ('ace', 'diamonds')]`	`True`
`[('ten', 'hearts'), ('five', 'hearts'), ('king', 'diamonds'), ('seven', 'hearts'), ('four', 'diamonds'), ('eight', 'spades'), ('ace', 'spades'), ('four', 'clubs'), ('eight', 'diamonds'), ('eight', 'hearts'), ('seven', 'diamonds'), ('four', 'spades'), ('ten', 'diamonds')]`	`False`

Convert fraction to simple continued fraction

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

Consider a rational number a/b where $0 < a < b$ so that the value of that fraction is between 0 and 1. This fraction can be expressed equivalently as $1/(b/a)$. Since $b > a$, the denominator can be written equivalently as $c + (b \bmod a)/a$, where c equals the integer part of the division b/a that will be at least equal to one. The leftover smaller fraction $(b \bmod a)/a$ can then be similarly rewritten. The remaining fraction will eventually be of the form $1/b$, giving us a simple continued fractions, an alternative way to express rational numbers as a list of positive integers. This function should return the simple continued fraction representation of the fraction a/b.

For example, consider the fraction 5/17. This can be written as 1/(17/5), which can be written as 1/(3 + 2/5). The inner fraction 2/5 can in turn be written as 1/(5/2) = 1/(2 + 1/2), giving the final complete representation for the original representation of 5/17 = 1/(3 + 1/(2 + 1/2)). Listing the integer terms of the in order that they appear in the levels of the continued fraction gives the result list `[3, 2, 2]`.

a	b	Expected result
3	13	`[4, 3]`
5	17	`[3, 2, 2]`
5	61	`[12, 5]`
11	97	`[8, 1, 4, 2]`
20086956	85050054	`[4, 4, 3, 1, 2, 8, 2, 20, 7, 4, 2]`

Simple continued fractions have all kinds of curious mathematical properties and practical applications. For example, finding the best rational approximation p/q for the given quantity x so that no fraction with a smaller denominator than q is closer to x than p/q. The simple continued fraction is always finite for any rational starting numbers and infinite for irrational numbers, this infinite sequence of terms eventually becoming periodic if and only if the original number is a **quadratic irrational**, that is, a solution of some polynomial equation of order two.

Convert simple continued fraction to fraction

```
def from_simple_continued_fraction(terms):
```

The function is the inverse of the previous problem of converting a rational number to a simple continued fraction so that this function should reverse the conversion and return the original rational number as a Python `Fraction` object. Your function should process the `terms` from end to beginning, adding each term to the accumulated fraction so far and taking the inverse of that result to be the accumulated fraction to the next round of this loop.

`terms`	Expected result (as `Fraction`)
`[4, 3]`	`3/13`
`[4, 6]`	`6/25`
`[3, 2, 2]`	`5/17`
`[6, 1, 8, 6]`	`55/379`
`[13, 6, 11, 6, 2, 4, 1, 13, 7, 6, 15, 2]`	`90669457/1193592058`
`[5, 1, 4, 6, 6, 15, 6, 5, 9, 11, 5, 1, 1]`	`103597999/601519457`

As a side note, in a situation like this where you are writing two functions that are each other's inverses, it is trivially easy to write a powerful fuzz tester to convince yourself that the functions are correct. Just generate random inputs for one function, pass the result of the first function to the second function, and compare that the result produced by the second function is identical to the original random input. Repeat this a billion times to root out all bugs from your two functions.

Sturm und drangle

```
def sturm_word(x, y):
```

On a grid paper where all the grid lines are drawn horizontally and vertically between integer points, a line segment is drawn from the origin point (0, 0) to the point (x, y) where x and y are non-negative integers so that gcd(x, y) = 1. This line segment will intercept some of the horizontal and vertical grid lines along its route from origin to the point (x, y), but will not go through any corners of the grid until reaching the destination point (x, y). DUCY?

If we denote each crossing of some vertical grid line by `'0'` and each crossing of some horizontal grid line by `'1'`, writing these crossings into a single string creates a finite Sturmian word defined by the slope y/x of the line segment. This function should return the Sturmian word collected during the journey to the endpoint (x, y).

The simplest approach to this problem is probably to step in unit steps along one axis direction, using the slope of the line to increase the position to the direction of the other axis. As much as you may be tempted, please don't use any floating point numbers in your function, but perform all calculations using exact `Fraction` values. God created integers, the rest is a work of man.

x	y	Expected result
7	19	`'1101110111011011110111011'`
19	7	`'0010001000100100010001000100'`
1	5	`'1111'`
6	29	`'11110111110111110111110111111011111'`
6	31	`'111110111111011111011111011111011111'`

Sturmian words for the grid crossings of the infinitely long lines defined by their slope have interesting mathematical properties that the reader can check out on the linked Wikipedia page.

Rational roots of a polynomial

```
def rational_roots(coefficients):
```

The `coefficients` of the polynomial $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ can be given as a tuple of $n + 1$ elements listed in increasing order of powers of x. For example, the polynomial $2x^4 - 3x^3 + x + 9$ would be given as `(9, 1, 0, -3, 2)`, the missing second order term simply having the coefficient zero. The famous rational root theorem of algebra says that if p/q is a rational root of a polynomial where p and q are reduced to lowest terms, p must be a factor of a_0, and q must be a factor of a_n. Since a known finite number of possible values of p and q fit this requirement (don't forget to consider both positive and negative values of p/q), this theorem can be used to effectively and efficiently find all rational roots of any polynomial, regardless of the order of that polynomial.

This function should return all rational roots of the polynomial with the given `coefficients` as `Fraction` objects, listed in ascending sorted order. You should return each root as a `Fraction` object even if it is an integer. For simplicity, we will also ignore the multiplicity of these roots so that you don't need to implement the polynomial division algorithm. Both the first and the last term of the `coefficients` tuple given to your function are guaranteed to be nonzero, so the polynomial will not have a root at zero.

You probably want to implement the Horner's method as a subroutine to evaluate the polynomial with the given `coefficients` at each candidate root p/q. You could, of course, implement some weak sauce polynomial evaluation method instead, but why settle for the second best when the very best method is such a concise and beautiful piece of code, and only four lines of Python?

`coefficients`	Expected result
`(-4, -5, -5, -1)`	`[Fraction(-4, 1)]`
`(1, 0, -4)`	`[Fraction(-1, 2), Fraction(1, 2)]`
`(-10, -1, 1, -4, -2, 8, 3)`	`[]`
`(728, 2658, -22225, -1335, 4262, 792)`	`[Fraction(-4, 1), Fraction(-7, 2), Fraction(-13, 99), Fraction(1, 4), Fraction(2, 1)]`

Multiple winner election

```
def multiple_winner_election(votes, seats, method):
```

Anglosphere countries tend to use voting systems where each district elects exactly one representative, typically done with the blunt first-past-the-post plurality method to determine the sole winner of each district. Many other countries use some type of party list proportional voting system where each electoral district has multiple seats to fill. Each voter casts a vote for a particular party, either directly ("closed list") or indirectly ("open list"). The `seats` that are up for grabs in each district are then dealt out to the parties in proportion to their `votes` in that district, but so that each party gets a whole integer number of seats.

Given the number of `votes` received by parties numbered from 0, ..., n – 1, this function should compute the distribution of `seats` to these n parties according to these `votes`. Seats are handed out one at the time. Each party has a **priority quantity** that depends on v, the number of votes received by the party, unchanging through this process, and s, the number of seats that have been given to that party so far, increasing through the process. The formula to compute the priority quantity depends on which one of the three methods of D'Hondt, Webster and Imperiali is used. The priority quantity of the party that received v votes and has so far been given s seats is $v/(s + 1)$ in the D'Hondt method, $v/(2s + 1)$ in the Webster method, and $v/(1 + s/2)$ in the Imperiali method. Each seat always goes to the party with the highest priority quantity at the time that seat is given out, and as a result, the priority quantity of that party will be lower as the term s in the denominator of the priority quantity increases.

You should perform all these calculations with exact integer arithmetic using the `Fraction` data type, of course. If two parties have the exact same priority quantity at the time a seat is given, you should break the tie by giving that seat to the party identified with the smaller number.

votes	seats	method	Expected result
`[128, 115, 26, 23]`	`21`	`'dhondt'`	`[10, 8, 2, 1]`
`[128, 115, 26, 23]`	`21`	`'webster'`	`[9, 8, 2, 2]`
`[128, 115, 26, 23]`	`21`	`'imperiali'`	`[10, 9, 1, 1]`
`[40, 37, 35, 15, 12]`	`23`	`'webster'`	`[7, 6, 6, 2, 2]`
`[28, 20, 20, 17, 14]`	`17`	`'dhondt'`	`[5, 4, 3, 3, 2]`

Maximum interval clique

```
def maximum_interval_clique(intervals):
```

A closed **interval** on the real number line is given as a two-tuple (*s*, *e*) of its **start** and **end** points, and contains every point *x* for which $s \leqslant x \leqslant e$. Two intervals **overlap** if they have at least one common point, that is, it is not the case that one of them ends before the other one starts.

Consider the situation of having *n* such `intervals` whose start and end points have been chosen from integers 0, ..., 2*n* – 1 so that each one of these 2*n* integers appears exactly once as a start or end point of some interval. Your task is to find the **maximum clique** of these `intervals`, that is, a largest possible subset of intervals so that every pair of intervals in this subset overlaps each other. Since the clique is not necessarily unique, this function should only return its size.

Unlike in general **graphs** whose maximal cliques can be enumerated with the **Bron–Kerbosch algorithm** also seen later in this same problem collection, the highly constrained overlap structure of real line intervals allows for a far more efficient **sweepline algorithm** to determine the size of the maximum clique in linear time. We start by noticing that not only does every pair of intervals in the clique contain some common point by definition, but this common point can be chosen to be the exact same point for all pairs of intervals in that clique. Furthermore, since the start and end points of all intervals are known to be integers from 0, ..., 2*n* – 1, the shared common point can be one of these points. Therefore, you should loop through these integer points from left to right, maintaining a running count of how many intervals contain the current point. Use a preprocessing loop through the `intervals` before this counting loop to set up an auxiliary list to tell you what happens at each point, so that you can update the running count in a constant time at every point.

`intervals`	Expected result
`[(4, 5), (1, 2), (0, 3)]`	2
`[(1, 7), (6, 8), (3, 5), (0, 4), (2, 9)]`	4
`[(8, 10), (12, 14), (2, 4), (1, 13), (5, 6), (0, 7), (9, 15), (3, 11)]`	4

Maximum overlap intervals

```
def maximum_overlap_intervals(intervals):
```

Same as in the previous problem of finding the maximum interval clique, a closed **interval** on the real number line is given as a two-tuple (*s*, *e*) of its **start** and **end** points, and contains every point *x* for which $s \leq x \leq e$. Two intervals **overlap** if they have at least one common point.

Consider again the situation of *n* such `intervals` whose start and end points have been chosen from integers 0, ..., 2*n* – 1 so that each one of these 2*n* integers appears exactly once as a start or end point of some interval. This time, your task is to find the intervals that overlap the largest number of other intervals, but unlike in the interval clique problem, those other intervals do not need to overlap each other. Since there can be several such intervals with the same overlap count, your function should return the list of all these intervals, listed in ascending sorted order of their starting points.

Instead of looping through every interval to compare it to every other interval in an inner loop, this problem can also be solved far more efficiently with the classic **sweepline algorithm** approach. Loop through the integers 0, ..., 2*n* – 1 while maintaining an **active set** of intervals that contain the current point. At the same time, maintain a **counter dictionary** to keep track of the number of intervals that each interval overlaps. When this loop comes to a point *s* that is the start point of some interval (*s*, *e*), initialize the count for that interval to the current size of the active set. Next, increment the count by one for each member of the active set, and finally add (*s*, *e*) to the active set. When this loop comes to a point *e* that is the end point of some interval (*s*, *e*), simply remove that interval from the active set. A linear preprocessing loop through the `intervals` again allows you to set up an auxiliary list to tell you exactly what to do at each point that the main loop iterates over.

`intervals`	Expected result
`[(0, 5), (1, 6), (2, 3), (4, 7)]`	`[(0, 5), (1, 6)]`
`[(0, 5), (1, 6), (2, 4), (3, 9), (7, 8)]`	`[(3, 9)]`
`[(0, 1), (2, 13), (3, 8), (4, 10), (5, 14), (6, 7), (9, 12), (11, 15)]`	`[(2, 13), (5, 14)]`

Peter Winkler's "Mathematical Puzzles: A Connoisseur's Collection" presents the following extremely surprising result; for *n* intervals constructed randomly from the endpoints 0, ..., 2*n* – 1, the probability of one interval overlapping all other *n* – 1 intervals is the exact same 2/3, for every $n > 1$. Yes, that highly counterintuitive probability really is exactly 2/3, even when *n* is up in the gazillions!

Split at None

```
def split_at_none(items):
```

The original Fizzbuzz problem to quickly screen out the utterly hopeless candidates for programming jobs will soon be almost two decades old and too well known, so our present time requires more sophisticated tests for this purpose. Modern problems require modern solutions, as the famous expression goes.

A recent tweet by Hasen Judi offers another interview screening problem that is more fitting the power of modern scripting languages, yet requires no special techniques past the level of basic imperative programming taught in any introductory programming course. Given a list of `items`, some of which equal `None`, return a list that contains as its elements the sublists of consecutive `items` split around the `None` elements serving as separator marks, each consecutive block of elements a separate list in the result. Note the expected correct behaviour whenever these `None` separators are located at the beginning or at the end of the list, and the expected correct behaviour whenever multiple `None` values are consecutive `items`.

`items`	Expected result
`[-4, 8, None, 5]`	`[[-4, 8], [5]]`
`[None, None, 12, None, 3]`	`[[], [], [12], [3]]`
`[None, None, None, None, 5, None]`	`[[], [], [], [], [5], []]`
`[None, -36, 25, 28, None, -27, -24, 34, -28, 24, 44, 6, 33, 20, None]`	`[[], [-36, 25, 28], [-27, -24, 34, -28, 24, 44, 6, 33, 20], []]`
`[53, 20, -49, 34, None, 13, -43, -14, 53, -6, None, None, None, -26, None, 14, None, None, -11]`	`[[53, 20, -49, 34], [13, -43, -14, 53, -6], [], [], [-26], [14], [], [-11]]`

(Cue the haughty reply tweets of “I have no idea how to solve this totes stupid and meaningless problem that has nothing to do with real work! It's just that those stupid employers don't see what a smart and productive programmer I am in practice, but I just always panic and freeze when I am given simple tests” in three, two, one...)

Chunk sorting

```
def chunk_sorting(permutation):
```

Sorting the given random access sequence of elements in ascending order has pretty much been a solved problem during the entire lifetime of most students reading this, and the best algorithms have long been implemented in the standard libraries of all serious programming languages.

The sorting problem is still fruitful in teaching algorithm analysis and related design techniques. Consider the following interesting variation of sorting, given as the problem "Max Chunks To Make Sorted" at LeetCode. Your task is to split the given `permutation` of integers 0, ..., *n* – 1 into non-empty contiguous chunks so that when each chunk is sorted separately in place (using any one of the standard sorting algorithms, doesn't matter which one) independent of the other chunks, the entire list becomes sorted. Of course one chunk spanning the entire `permutation` would always work, but this function should determine the largest number of chunks that can work.

Our automated tester will try out your function with 20,000 test cases, most of which contain thousands of elements, so your algorithm needs to operate **in place** in the given `permutation` list and not extract any slices, nor use other inefficient list operations or keep looping back and forth the list like some Shlemiel. To develop the swift and sure **greedy algorithm** for this problem that will operate in a single for-loop through the entire list never straying or turning back, you should start by noticing that of course the smallest element 0 must be part of the first chunk, but where will that first chunk end? Nope, not necessarily at that element 0, as the third and fifth test cases in the table below illustrate. Then after that first chunk, where does the second chunk end?

`permutation`	Expected result
`[1, 0, 2]`	2
`[0, 1, 4, 5, 3, 2]`	3
`[3, 0, 2, 4, 1, 6, 5, 7]`	2
`[0, 4, 3, 6, 5, 1, 7, 8, 2]`	2
`[2, 0, 1, 6, 8, 7, 4, 3, 5, 9]`	3
`[1, 2, 0, 3, 8, 5, 10, 4, 7, 6, 11, 9]`	3

Merge the two biggest

```
def merge_biggest(items):
```

Your task is to repeatedly perform the following operation to the given list of `items`; remove its two largest elements *a* and *b*, and if their absolute difference *c* = abs(*a* – *b*) is nonzero, insert *c* to the `items`. Rinse and repeat until only one element remains, and return that element as the answer. Should the list become empty after removing the last two elements that were equal, return 0.

For example, if the `items` list initially contains the elements `[7, 1, 5]`, removing the two largest elements 7 and 5 and inserting their absolute difference 2 makes the list to be `[2, 1]`. Repeating the same operation makes the list to be `[1]` whereupon the function terminates, returning 1.

The automated tester will give your function ten thousand test cases, most of them containing thousands of elements. To allow your function to perform this operation efficiently so that it will pass the automated tester within the time limit, this would be a good time to turn the `items` list into a **binary heap priority queue** using the functions of the `heapq` module of the Python standard library. The powerful operations `heappop` and `heappush` will then perform the required operations efficiently even for humongous lists... but how to finagle these operations to extract the largest element from the queue instead of the smallest one? You need to think up some way to, uh, *negate* the direction of the ordering that is used to organize the elements inside the priority queue.

`items`	Expected result
`[2, 10, 3]`	5
`[7, 1, 5]`	1
`[7, 11, 11, 1, 15, 11]`	2
`[57, 18, 47, 45, 52, 59, 66, 1, 49, 13, 66, 51, 39, 50, 13, 49, 9, 43, 37, 39, 19, 59]`	1

Lehmer code: decoding

```
def lehmer_decode(lehmer):
```

This problem is the inverse of the earlier problem of calculating the Lehmer code encoding for the given permutation. This function should reconstruct and return the original permutation of the nonnegative integers 0, ..., *n* – 1 that produces the given Lehmer code.

This problem is not quite as straightforward as the mechanistic Lehmer encoding seen as the earlier problem, but requires a bit of combinatorial thinking to get started. Note how the first element of the reconstructed permutation must necessarily equal the first element of the Lehmer code. But how would you then construct the rest of the required permutation?

`lehmer`	Expected result
`[1]`	`[1, 0]`
`[1, 0]`	`[1, 0, 2]`
`[1, 1]`	`[1, 2, 0]`
`[2, 0, 1, 0, 1]`	`[2, 0, 3, 1, 5, 4]`
`[2, 3, 0, 2, 2, 0]`	`[2, 4, 0, 5, 6, 1, 3]`
`[7, 1, 1, 4, 1, 0, 1]`	`[7, 1, 2, 6, 3, 0, 5, 4]`

Bayesian updating of dice

```
def bayes_dice_update(dice, rolls):
```

You have a box of `dice`, each with a different number of sides as in the game of Dungeons and Dragons. Your friend randomly takes out one die from the box and makes a series of `rolls` with that same one die out of your view, and then tells you the results of these rolls. Your task is to calculate, for each die in the box, the exact probability that your friend used that particular die.

Initially, each of the *n* dice has the same **prior** probability 1/ *n* of being the chosen die. However, the **posterior** probabilities of these die will usually be very different after seeing the resulting rolls. According to the famous Bayes' theorem, the conditional probability *P*(*d* | `rolls`) can be written equivalently in the form *P*(`rolls` | *d*) *P*(*d*) / *P*(`rolls`). The term *P*(*d*) is known to equal 1 / *n*, and since the individual rolls are **independent** of each other, the term *P*(`rolls` | *d*) can be computed simply as the product of the probabilities of the individual rolls.

To get rid of the problematic term *P*(`rolls`), we only need to note that this term is the same unknown constant factor for each die, and can therefore be completely ignored! Therefore all we need to do is to calculate *P*(`rolls` | *d*) *P*(*d*) for each die, and then **normalize** the resulting probabilities so that they add up to one, seeing that precisely one of these dice must be the one that your friend drew from the box. Of course, you should perform all calculations using exact `Fraction` objects.

`dice`	`rolls`	Expected result
`[5, 6]`	`[2]`	`[Fraction(6, 11), Fraction(5, 11)]`
`[3, 4, 5]`	`[2, 2]`	`[Fraction(400, 769), Fraction(225, 769), Fraction(144, 769)]`
`[2, 7, 10, 11]`	`[2, 7, 3]`	`[Fraction(1331000, 2130533)], Fraction(456533, 2130533), Fraction(343000, 2130533)`

Haircut

```
def haircut(speed, n):
```

You need a haircut and go to a popular barbershop that employs barbers numbered 0, ..., *M*, each barber renting a chair. In this shop, deeply spirited in Taylorist principles of standardization and efficiency, the barber number `i` will always take the same number of minutes `speed[i]` to complete a haircut, regardless of the customer. You arrive at the barbershop in the morning just before it is about to open, and arrive in position `n` in the queue of customers already waiting ahead of you. Each customer always walks into the first available chair, regardless of the speed of the barber who operates at that chair. Which barber will be the one to eventually service you?

`speed`	`n`	Expected result
`[36, 36, 10]`	`7`	`0`
`[28, 16, 51, 36]`	`8`	`3`
`[15, 41, 20, 31, 24]`	`11`	`2`
`[10, 11, 19, 46, 47, 60, 60]`	`22`	`2`

This is the problem "Haircut" from Google Code Jam 2015.

Kadane in triplicate

```
def max_three_disjoint_sublists(items):
```

The maximum subarray problem is a classic exercise in algorithmic thinking that asks you to find the contiguous sublist of the given list of integers that has the largest sum. (Some of the original list elements are negative so that the answer isn't trivially "duh, just take that entire list, bro".) A bit surprisingly, the maximum subarray can be found with one simple linear loop through the original list with Kadane's algorithm, instead of iterating through a quadratic number of the possible pairs of start and end positions of all possible sublists.

Initialize a variable *m* to zero, and loop through all the elements *e* in the list. For each list element e, simply update *m* = max(*m* + *e*, *e*). This way, the variable *m* always contains the largest possible sum of any sublist that ends with that element *e* in that position of the entire list. Note how we don't care about which preceding elements added up that maximum sum *m*, only of the value of that maximum sum itself. Then on the side, keep track of the largest sum that you have seen so far along in the entire list with the position of that largest sum. Once the loop has terminated, you can reconstruct the sublist using another linear loop going through that sublist in reverse from its stored position.

Now that you know all that, you are hopefully ready to tackle the following problem; find the largest sum that can be formed by adding up the elements of **three** contiguous sublists of your choice from the given list of `items`. All three sublists must contain at least one element, and their positions must be fully **disjoint** so that no position is included in any two of your chosen lists. Your function should be fast enough to handle thousands of pseudorandom lists with thousands of elements each within the time limit of the automated fuzz tester.

`items`	Expected result
`[-2, -2, -2]`	`-6`
`[2, 3, 4, -5, -5, -5]`	`9`
`[-1, -2, 7, -7, 5, -1, -3, -5]`	`11`
`[-17, -15, 21, 44, -20, 40, -21, -11, -15, 31, -17, -21, -11, 25, -4, -4, 47, -11, 44, -17, -9, -8, -19, -22]`	`213`

Knaves of the round table

```
def knaves_of_round_table(answers):
```

Some knights and knaves are gathered cyclically around King Arthur's round table, these men numbered 0, ..., *n* – 1 based on their positions. As we have learned from the works of Raymond Smullyan, knights always tell the truth and knaves always lie whenever they open their mouths to speak. Unfortunately, we cannot tell on sight which men are knights and which are knaves. (If asked directly which kind of a man he is, each man would claim to be a knight; not very helpful to us.)

To start resolving this mess, we ask each man how many of the two men sitting immediately to his left and right are knights, collected to the list of `answers`. These `answers` do not always uniquely pinpoint the positions of the knights and knaves, but at least we can identify all possible assignments that are consistent with these `answers`. For example, there are two possible solutions consistent with three men's answers `[2, 2, 2]`; either all three men are knights telling the truth, or all three men are knaves lying their asses off.

This function should return the list of possible counts of how many knights are seated at the round table. (Again, the function does not return the possible positions of these knights, but the possible total number of knights.) The automated tester will only give your function `answers` for which there exists at least one assignment of knights and knaves consistent with these `answers`.

This function can recursively fill out all possible assignments of these men to knights and knaves from left to right, backtracking from the partial assignment built so far as soon as it hits a local contradiction. The vast majority of possible branches in this search will be immediately pruned away with some local contradiction, making this recursive search fast even if hundreds of men were seated around King Arthur's table.

`answers`	Expected result
`[1, 2, 2]`	`[0]`
`[0, 2, 0]`	`[1]`
`[2, 1, 1, 1]`	`[0, 3]`
`[2, 2, 2, 2, 2, 2]`	`[0, 6]`
`[1, 2, 2, 1, 2, 1, 2]`	`[0, 3, 4]`
`[0, 1, 2, 2, 2, 2, 2, 2, 1]`	`[8]`

Six of this, half a dozen of the other

```
def assign_sides(costs):
```

You are given 2*n* items, and you need to assign exactly *n* of these items of your choice on this side, and the remaining *n* items on the other side. Each item is given as a two-tuple `(this, other)` that gives the cost of assigning that item on this and on the other side, respectively. This function should compute the optimal assignment of these 2*n* items to this and the other sides to minimize the total overall cost of the assignment.

This problem may initially seem like a backtracking recursion where each item can be assigned either way, terminating at the base case where either side has *n* items assigned there. However, this would be serious overpaying for overkill; some clever preprocessing of these `costs` actually allows the optimal assignment of these items to be computed in a single linear pass over the costs. The key insight starts from the realization that you must assign every item somewhere, and that one dollar saved equals that same one dollar, no matter where it comes from.

`costs`	Expected result
`[(4, 6), (1, 3), (3, 2), (3, 5), (4, 2), (3, 1)]`	13
`[(1, 1), (8, 6), (7, 9), (4, 6), (2, 9), (3, 2), (8, 3), (6, 4), (4, 3), (3, 5)]`	35
`[(5, 9), (5, 1), (4, 7), (7, 11), (10, 7), (11, 11), (11, 8), (10, 9), (5, 9), (5, 9), (2, 11), (8, 8), (11, 3), (7, 1)]`	76

Factoradical dudes

```
def factoradic_base(n):
```

Several problems in our original two Python problem collections have showcased problems of representing integers in positional number systems in all sorts of weird bases instead of the familiar positive integer bases ten and two, these bases could be negative, rational or even complex numbers whose powers multiplied by the coefficient of that position then add up to that integer.

In this problem we look at an interesting positional number system that uses a **mixed base** where the base is not the same for every position, but depends on the position. The idea of a variable base might initially seem weird in general. However, you already use such a system every day in expressing time with seconds and minutes both given in base 60, whereas hours are expressed in either base 12 or 24 depending on which side of the pond you live in. Imperial measurements for weights and lengths also use mixed bases to express each unit as a multiple of the next lower unit.

The factoradic number system uses the factorials 1, 2, 6, 24, 120, ... as variable bases. If the positions are numbered starting from 1, the base in position k equals $k!$, the product of first k positive integers. The possible coefficients for the position k are then 0, ..., k. If the coefficient of the position k were equal to $k + 1$, the resulting represented term $(k + 1)k!$ would equal $(k + 1)!$ and could therefore be carried over to the next position, analogous to any other positional number system.

This function should return the factoradic representation of the given positive integer n as the list of its factoradic coefficients so that the coefficient in the position $k - 1$ of the result list gives the coefficient of the factorial $k!$ in the representation. The automated tester will give your function some pretty humongous numbers to convert, but since factorials grow exponentially fast, your function should return the list of coefficients rapidly even for numbers that have hundreds of digits.

n	Expected result
1	`[1]`
11	`[1, 2, 1]`
96	`[0, 0, 0, 4]`
1781	`[1, 2, 0, 4, 2, 2]`
4146434844171	`[1, 1, 0, 2, 3, 1, 6, 7, 2, 10, 4, 11, 7, 2, 3]`

Factoradic numbers can be used to encode arbitrary permutations of distinct elements one-to-one into positive integers with the **Lehmer coding** seen in earlier problems.

Tarjan's strongly connected components

```
def tarjans_scc(edges):
```

The basic graph search algorithms of breadth-first search and depth-first search can be augmented to compute other graph properties. Consider the problem of finding the strongly connected components of the given directed graph, that is, maximal subsets of nodes so that you can get from any node in that component to any other node in that same component, but once you leave that component, there is no coming back. (If there were such a way back, the node that you exited to would, by definition, be part of that same strongly connected component.) The strongly connected components of the graph are a **partition** of its nodes so that each node belongs to exactly one strongly connected component. If nothing else, each node by itself is a singleton component.

Tarjan's strongly connected components algorithm is an ingenious application of depth-first search to enumerate all strongly connected components of the given graph as a byproduct of a single depth-first search through that graph. Your task here is to translate the pseudocode given on the Wikipedia page (or any other source where fine algorithms are sold) to Python. The basic list data type of Python will work nicely as the `index`, `lowlink` and `on_stack` bookkeeper, but also as the **stack** from where to pop the nodes of the strongly connected component represented by the current node whose `index` and `lowlink` ended up being equals.

Your function is given a directed graph whose nodes are integers 0, ..., *n* – 1 as the list of `edges` emanating from each node. The function should return the list of strongly connected components so that the nodes of each component are listed in ascending sorted order, and the components themselves are listed in ascending order of their lowest-numbered nodes.

`edges`	Expected result
`[[1, 4, 3, 2], [3, 4, 0], [4, 1, 0, 3], [2], [0, 3, 2, 1]]`	`[[0, 1, 2, 3, 4]]`
`[[4, 3], [2, 5, 4], [4, 1, 0, 3, 5], [4, 0], [], [1, 2]]`	`[[0, 3], [1, 2, 5], [4]]`
`[[2, 11, 10, 3, 12], [0, 3, 12, 2, 11, 4], [1, 0, 12], [4, 1, 6], [7], [4, 8], [8, 4], [4, 5], [6], [8, 10, 6, 12], [9, 12, 11, 7], [10], [1, 10, 0, 11]]`	`[[0, 1, 2, 3, 9, 10, 11, 12], [4, 5, 6, 7, 8]]`

As a nostalgic trip down the memory lane, your author took two days to understand and implement this algorithm in C needed for his work in his Master's thesis back in 1995, whereas now that same deal took about fifteen minutes.

Copeland harder

```
def copeland_method(ballots):
```

Consider an election where multiple candidates compete to be the single winner of that district. The first-past-the-post plurality voting used in most Anglosphere elections at least has simplicity and tradition going for it, but this method lacks many desirable properties of fair election systems.

In Copeland's method, instead of voting just for the one preferred candidate, each voter ranks all candidates in descending preference order in the ballot. To determine the winner, each candidate has a one-on-one match against each other candidate. In this one-on-one match, all other candidates are ignored, and the match is won by the candidate that most voters prefer against his opponent looking at only those two candidates on each ballot. The candidate who wins the one-on-one match gets one matchpoint for this win. Should the candidates happen to tie, both get half a matchpoint.

Note that the magnitude of winning a single match does not affect the matchpoint gain; beating the opponent by one vote gives that candidate the exact same one matchpoint as winning by a million votes. It is also irrelevant how much each voter prefers one candidate over the other one, as only the ordinal preference matters regardless of the cardinal magnitude of this preference.

The winner of the overall election is the candidate with the most matchpoints. However, unless the voter preferences determine a unique Condorcet winner, the mechanism can result in a tie. This function should return the sorted list of winners in the election between candidates 0, ..., $n - 1$, where each ballot is some permutation of these candidates indicating the preferences of that voter.

`ballots`	Expected result
`[[1, 0, 2], [2, 0, 1], [2, 1, 0], [0, 2, 1], [1, 2, 0]]`	`[2]`
`[[1, 2, 0, 3], [3, 2, 0, 1], [2, 3, 0, 1], [1, 3, 0, 2], [3, 1, 0, 2], [2, 1, 0, 3]]`	`[1, 2, 3]`
`[[1, 0, 2, 3], [3, 0, 2, 1], [0, 3, 2, 1], [3, 0, 2, 1], [0, 3, 2, 1], [1, 3, 2, 0]]`	`[0, 3]`

Instant-runoff voting

```
def instant_runoff(ballots):
```

Following the previous Copeland method problem, consider again an election where multiple candidates compete to be the single winner of that district. Same as with the Copeland method, instant runoff voting makes the voters rank all candidates in that voter's preference order in the ballot. The difference is that the winner is not computed with pairwise matches, but by repeatedly eliminating the least favoured candidate in a plurality vote until a strict majority winner emerges.

Initially, each ballot goes for the candidate who was that voter's first preference. If none of the candidates gets a majority of votes, the candidate with the least ballots is eliminated from the race, and his ballots are distributed to the remaining candidates, each ballot to the candidate who was that voter's second preference. Elimination of candidates in this manner continues until one remaining candidate has the strict majority of votes (that is, *more* than half) and is declared the winner.

This function should determine the winner of candidates numbered 0, ..., n – 1 in the instant runoff election, when each ballot is a permutation of these candidates listed in the preference order of the vote casting that ballot. To make the results unambiguous for the automated fuzz tester, if several remaining candidates have the same smallest number of ballots in the current round, eliminate the highest-numbered such candidate.

To implement this function correctly, note that a candidate who has been eliminated in any previous round cannot come back to life by receiving ballots from candidates eliminated in the current round, but each one of their ballots is passed on to that voter's next preferred candidate who is still alive and kicking in the election.

`ballots`	Expected result
`[[1, 0, 2], [2, 0, 1], [2, 1, 0], [0, 2, 1], [1, 2, 0]]`	2
`[[1, 0, 2], [0, 1, 2], [1, 0, 2], [2, 0, 1], [0, 2, 1]]`	1
`[[1, 4, 2, 3, 0], [3, 4, 2, 1, 0], [2, 4, 3, 1, 0], [3, 4, 2, 1, 0], [2, 4, 3, 1, 0], [4, 2, 3, 1, 0], [3, 2, 4, 1, 0], [4, 2, 3, 1, 0]]`	4
`[[1, 0, 2, 3, 4, 5], [5, 0, 2, 3, 4, 1], [0, 5, 2, 3, 4, 1], [1, 2, 5, 0, 4, 3], [0, 2, 5, 1, 4, 3], [3, 5, 4, 0, 2, 1], [4, 5, 3, 0, 2, 1], [5, 4, 3, 0, 2, 1], [5, 2, 0, 3, 4, 1]]`	5

Cousin explainer

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

Women in a matrilineal family tree have been numbered so that Eve at the root has the number 1, and the woman whose number is *i* has exactly two daughters who are numbered 2*i* and 2*i* + 1. For example, the two daughters of the woman number 7 would be the two women numbered 14 and 15. In this scheme, the mother of woman number *i* is given by the formula *i* / 2, rounding down to an integer for odd *i*. As you can see, this formula gives the mother's number 7 for both of her daughters 14 and 15. (This is also exactly how a binary tree structure is imposed on a list when treating it as a **binary heap**, as implemented in the `heapq` standard library module.)

Given two women numbered `a` and `b`, this function should return in English what the woman `b` is to the woman `a`. For simplicity, we assume that all the men that these women procreated with were genetically sufficiently distinct from each other so that the familial relationship between any two women in this problem is solely determined by this matrilineal tree.

To determine the relationship between women `a` and `b`, follow the parent lineage from both women to their **nearest common ancestor**, keeping count of the number of steps from both women needed to reach that ancestor. These two counts determine the relationship, as illustrated in the tweet "The cousin explainer". For example, if one of these step counts is zero, that woman is a direct ancestor of the other one. If one of the step counts is one, that woman is the (some number of `great`) aunt of her (some number of `great`) niece. Otherwise these two women are cousins to the degree determined by the smaller of the two counts of steps to their common ancestor. The count of **removals** is determined by the difference of these two step counts. For this count of removals, use `'once'`, `'twice'` and `'thrice'` for the first three, after which use `'four times'`, and so on. Also do not use hyphens, but say "great grandniece" instead of "great-grandniece".

a	b	Expected result
1	3	`'daughter'`
3	5	`'niece'`
5	18	`'grandmother'`
9	7	`'first cousin once removed'`
3	18	`'great great grandniece'`
344	28	`'third cousin four times removed'`
612	5	`'great great great great great grandaunt'`
1133	3668	`'ninth cousin once removed'`

Maximize exam confusion

```
def maximal_confusion(answer_key, k):
```

This deceptively simple problem is the problem 2024 in LeetCode, "Maximize the Confusion of an Exam". Too dumb to create new original exams himself, this one teacher has inherited a multiple choice exam whose `answer_key` consists of characters `'T'` and `'F'` to indicate whether the correct answer to each particular question is `True` or `False`. Being generally unfit for his teaching position and an insecure impostor who knows that his students who paid hard earned money to take the course will soon notice his incompetence unless their minds are directed elsewhere, he decides to play sadistic mind games with his students and change at most *k* questions to opposite answers to maximize the longest run of consecutive questions with the same answer, either all `True` or all `False`, to gaslight the exam takers to question their sanity.

Of course, even this impostor will soon realize that all *k* changes should be to the same direction (either all change some `True` answer to `False`, or all change some `False` answer to `True`), and soon after that, realize that there is no point leaving gaps between these changes, but all changes should take place in questions that are consecutive in the subsequence of questions with the same answer. After these flashes of insight, perhaps you can hold your nose and help him to achieve this maximal confusion.

Your function should return the length of the longest run achievable by at most *k* changes. Despite the despicable premise, it is possible to polish this turd enough to reveal the diamond underneath; try to solve this problem in a single for-loop with O(1) extra memory, for a solution so ravishingly beautiful in its simplicity that any student working through it and proving it correct will instantly level up in his or her algorithmic and coding problem skills.

`answer_key`	`k`	Expected result
`'FTFFFF'`	3	6
`'TFFTTFFTT'`	2	6
`'TTFFFTFFTFT'`	3	9
`'TFFFFTTTTTFFFF'`	3	8

Burrows–Wheeler transform

```
def burrows_wheeler_encode(text, positions):
```

Burrows–Wheeler transform is a famous algorithm to convert any `text` string into a string of equal length with the original characters permuted in a way that allows various compression algorithms to compress that string much better than how they would compress the original. Of course, after decompressing to get back the transformed string, this transform can be effectively and efficiently reversed to restore the original `text`, as a topic for a followup question after this one.

This function should compute the Burrows–Wheeler transform of the given `text` string. However, instead of returning the entire transformed string, you should only return the characters in the given `positions` of the result as **proof of work** of having computed this transformation while keeping the result small for our automated fuzz tester and the `expected_answers` file.

The educational point of this exercise is to take the inefficient Python implementation that explicitly constructs all cyclic rotations explicitly as the starting point, and rewrite it to operate **in place** so that only one copy of the `text` is kept in memory at all times. The fuzz tester will give your function long strings composed of random words from the `words_sorted.txt` file, so efficient operation of your function here is a must. Each `text` is also guaranteed to end with the terminator `'$'`.

Your function should create the list of all position indices to the text and sort this list with a **custom comparator** function that, given two positions `i` and `j`, compares the two cyclic rotations of `text` starting from those positions in lexicographical order, traversing the characters in lockstep from both positions and comparing them pairwise until a mismatch is found. Use the `cmp_to_key` function of the Python `functools` standard library module to pass your custom comparator function to the `sort` method, and then just extract the last characters of the cyclic substrings starting from the indices in the `positions` given.

`text`	`positions`	Expected result
`'a$'`	`range(2)`	`'a$'`
`'abc$'`	`range(4)`	`'c$ab'`
`'abab$'`	`range(5)`	`'bb$aa'`
`'baab$'`	`range(5)`	`'bbaa$'`
`'pilloriesafterpartlepidopterous$'`	`[10, 12, 17, 23, 27]`	`'rtree'`

Bron–Kerbosch maximal clique enumeration

```
def maximal_cliques(edges):
```

The pairwise friendship relation of a group of *n* people is expressed as a graph so that `edges` is an **adjacency list** where each term `edges[u]` gives the list of friends of the person *u*. Friendship is assumed to be symmetric so that *v* is a friend of *u* whenever *u* is a friend of *v*, so this graph is effectively undirected. Some people are loners with no friends, so their friend list is empty.

In graph theory, a **clique** is a set of nodes who are all pairwise friends with each other. A clique is said to be **maximal** if no node can be added to it without breaking the defining clique property of the universal pairwise friendship. This function should return the list of all maximal cliques of three or more nodes in the graph with the given `edges`. The nodes of each maximal clique should be listed in ascending order, and the cliques themselves should be listed in descending order of size, cliques of equal size listed in ascending lexicographic order. Note that unlike in the earlier Tarjan's strongly connected components problem, maximal cliques are not an equivalence partitioning of the nodes in that graph, but it is perfectly possible for some social person to simultaneously belong to multiple separate maximal cliques.

You should implement the Bron–Kerbosch maximal clique enumeration algorithm as a nested function inside this function. Maintain a list of maximal cliques outside the recursion, and append the maximal cliques to this list as your recursion finds them. The pseudocode of the Bron–Kerbosch algorithm should translate neatly to Python and its operations on sets in the language itself. To ensure that your function will pass our pseudorandom fuzz test suite in a reasonable time, you should implement the version of the Bron–Kerbosch algorithm that uses pivot nodes, making a good choice of a pivot node each time to ruthlessly prune the redundant branches of the search.

`edges`	Expected result
`[[1, 3, 2, 5], [0, 2, 4, 3], [1, 0], [0, 1, 5], [1], [0, 3]]`	`[[0, 1, 2], [0, 1, 3], [0, 3, 5]]`
`[[1, 4], [0, 2, 5, 4, 7, 3], [1, 7, 3], [2, 1, 5], [0, 1, 7], [1, 7, 3], [], [2, 5, 1, 4]]`	`[[0, 1, 4], [1, 2, 3], [1, 2, 7], [1, 3, 5], [1, 4, 7], [1, 5, 7]]`
`[[1, 3, 6, 2, 5], [0, 2, 5, 3], [1, 3, 0, 5], [0, 2, 6, 1, 5], [], [1, 0, 2, 3], [0, 3]]`	`[[0, 1, 2, 3, 5], [0, 3, 6]]`

Queue jockeys

```
def queue_jockeys(permutation, moves):
```

A simulated system consists of two queues. The first queue initially contains the integers 0, ..., $n - 1$ as listed from front to back in the given `permutation`, whereas the second queue is initially empty. A series of `moves`, each move naming one of the stored elements in this system, is executed in order so that each move removes that named element from whichever queue it is currently in, and places that element in the front of the other queue. This function should return a two-tuple giving the contents of the first and second queues as lists after all these `moves` have been executed.

Of course at this point of your studies, this function should not be hard to write using two Python lists to store the contents of these queues, and using simple Python list slicing and dicing operations to perform each individual move. Alas, after the smooth start of small inputs suitable for debugging, the automated tester will give your function inputs where n is in the thousands and the number of `moves` is in tens of thousands, making sure that any solutions based on list slicing and dicing will not pass the automated tests within the set time limit. You need to design a more efficient representation of the system state that allows you to perform each move in a small constant time, regardless of the number of elements in both queues.

Note also that even though in the examples below `moves` is always a Python list, for the later inputs the automated tester will make `moves` to be a lazy **generator** object that allows you to iterate through the `moves` with a for-loop, but does not allow random access to jump back and forth or rewinding the moves. This should not really affect any reasonable implementation of this function, though. (Any generator object turns into a `list` with a single function call anyway, if needed.)

`permutation`	`moves`	Expected result
`[0, 1]`	`[0, 1, 1, 0]`	`([0, 1], [])`
`[3, 1, 0, 2]`	`[0, 0, 0, 1, 3, 0, 1, 2]`	`([1, 0], [2, 3])`
`[4, 3, 0, 1, 2]`	`[4, 2, 0, 1, 3, 3, 4, 3]`	`([4], [3, 1, 0, 2])`

Smallie Bigs

```
def partition_point(items):
```

The famous quicksort algorithm to, well, *quickly sort* the elements of the given list starts by **partitioning** the elements of the list into two lists of "small" and "big" elements so that every element in the list of small elements is less than or equal to every element in the list of big elements. Various clever ways to perform this partition to make both sides roughly equal in size have been invented, thus making the "divide and conquer" recursion of the quicksort algorithm balanced, as all things should be.

However, in this problem we want to determine whether a good partition point already exists for the given list of `items`. This function should determine whether the `items` list can already be split at some position into two sublists so that every element in the left sublist is less than or equal to every element in the right sublist. This function should return the number of elements in the left sublist of the resulting partition. If multiple solutions exist, you should return the one that makes the number of elements in the left sublist as close to the half of the length of the original `items` list.

The automated fuzz tester will give this function only lists where such a partition exists with both sides ending up having at least one element. Furthermore, the tester will give your function 10,000 test cases that contain up to thousands of elements, so your function needs to operate in linear time with respect to the length of the list, instead of sawing back and forth in an inner loop for each individual element like some kind of Shlemiel.

`items`	Expected result
`[0, 9, 15, 13, 14]`	2
`[5, 11, 3, 13, 18, 22]`	3
`[53, 23, 31, 23, 3, 47, 42, 22, 79, 65]`	8
`[108, 148, 107, 6, 44, 39, 130, 281, 185, 274, 175, 240, 168, 188]`	7

Gauss circle

```
def gauss_circle(r):
```

The Gauss circle problem of **combinatorial geometry** asks how many points (*x*, *y*) where both coordinates *x* and *y* are integers lie within the disk of radius *r* centered at the origin. (Points at the edge of the disk are counted as being within the disk.) This famous problem is still open in that nobody has discovered a closed form solution to this problem as a function of the radius *r*. Of course, in this course we can still use this classic problem as an exercise of writing loops to solve the problem efficiently at least for reasonably small *r*.

One simple way to count up the points is to add them up one row of points at the time. Initialize the total count to zero and start traversing the boundary of the disk at its topmost point (0, *r*), surely part of the disk. Whenever you are standing at the point (*x*, *y*), add the 2*x* + 1 points on the row *y* to the total count, and do the same for the points in the row –*y* whenever *y* > 0. Decrement the *y*-coordinate, and increment the *x*-coordinate as long as the point (*x*, *y*) remains inside the disk. Once you have added up the points in all rows in this manner, return the final total.

`r`	Expected result
`1`	`5`
`2`	`13`
`7`	`149`
`58`	`10557`
`1809`	`10280737`
`1000000`	`3141592649625`

The related Euclid's orchard problem is otherwise the same, but counts only the points (*x*, *y*) that are directly visible from the origin without any other points standing directly in between the origin and that point, which happens precisely when *x* and *y* are **relatively prime** with no factors higher than one in common.

Zeckendorf decoding

```
def zeckendorf_decode(fits):
```

Fibonacci numbers are often used to teach recursion in computer science. However, they can also be used as building blocks of all positive integers due to the famous Zeckendorf's theorem that says that any positive integer can be expressed as a sum of distinct non-consecutive Fibonacci numbers in exactly one way, and that representation can be easily computed with the greedy algorithm that always uses the largest Fibonacci number that still fits in the remaining number. For example, the unique representation of 72 as such a sum is 1 + 3 + 13 + 55.

As an alternative in expressing a positive integer in binary "bits" as sum of distinct powers of two, **Zeckendorf encoding** turns this sum of distinct Fibonacci numbers as binary "fits" where the ones indicate the Fibonacci numbers included and the zeros indicate the numbers excluded in the sum. Unlike in ordinary **positional number** systems where the digits are written in order of decreasing powers of the base, Zeckendorf encoding writes the fits in order of increasing Fibonacci numbers. For example, the Zeckendorf encoding for 72 would be `101001001`. This order helps greatly when encoding an arbitrary long list of integers, each of an arbitrarily large magnitude, where the Zeckendorf encoding shines since the impossible fit pattern 11 can be used to denote the "comma" that separates two consecutive numbers in the list, the first of these two fits serving a double duty as the highest order fit of the integer being encoded! For example, the list `[72, 6, 81, 2, 5]` would be encoded as a string of fits `101001001`1`1001`1`000100101`1`01`1`00011`.

This function should decode the given sequence of `fits` back to the original list of integers. Your function can assume that the last two `fits` are always `11`.

`fits`	Expected result
`'01001110001101011'`	`[10, 9, 7]`
`'011'`	`[2]`
`'100100000111000100101110010011010101111'`	`[95, 132, 27, 20, 1]`
`'00101011'`	`[32]`
`'00100101010001101000000001011011'`	`[749, 523, 2]`

When integers are expressed in positional binary, encoding the separator between the integers of arbitrary magnitude can get quite tricky. Elias gamma coding is one clever way to achieve this.

Sultan's daughter

```
def sultans_daughter(expression):
```

Similar in spirit and style to the "McCulloch's Second Machine" problem in the original collection of *109 Python Problems*, **Sultan's daughter** is an esoteric programming language based on the string rewriting paradigm, this language also devised by none other than Raymond Smullyan himself.

Expressions written in Sultan's daughter are strings that consist of digit characters `0` to `9` only, with no other characters allowed. To evaluate an expression of the form `dX`, where `d` is some leading digit character and `X` some string of digit characters (possibly empty), the first digit `d` determines which operation is applied to the recursively computed value of the rest of the expression `X`.

Evaluation rules for the possible values of `d` are listed on the Wiki page for Sultan's daughter at the site `esolangs.org` for the reader to consult. For example, since the expression `19872` evaluates to `987`, the expression `319872` evaluates to `987987`, and therefore the expression `4319872` evaluates to `789789`. This function should iterate the given `expression` of digit characters `0` to `9` until the evaluation either terminates when none of the rewriting rules apply, or is nonterminating due to reaching some expression that has already been encountered during that evaluation. This function should return the number of iteration steps required to reach termination. In case of a non-terminating `expression`, return `-1` instead.

`expression`	Expected result
`'663653'`	`1`
`'1171112222'`	`4`
`'13115616633131152222222'`	`8`
`'11313116113114722222222'`	`10`
`'47431474312'`	`-1`

That last expression `47431474312` is actually a quine in Sultan's daughter, evaluating to itself.

Limited swap sorting

```
def limited_swaps(perm, swaps):
```

All **comparison sort** algorithms operate by swapping two elements according to some discipline that guarantees that after these comparisons and conditional swaps, the resulting list will be in sorted order. This problem looks at an interesting restriction of sorting problem in which the function is given the list of `swaps` as tuples `(i, j)` that the algorithm is allowed to perform, where `i` and `j` are position indices.

Given the list `perm` that is guaranteed to be some **permutation** of integers 0, ..., $n - 1$, this function should determine how close to sorted order that permutation can be brought when limited to using only the given `swaps`. This function should return the **lexicographically smallest** list that is reachable from the given `perm` using only the allowed `swaps`. Note that the same individual swap may be used multiple times during the sorting process to marshal the elements towards their final resting positions.

`perm`	`swaps`	Expected result
`[2, 0, 1]`	`[(0, 1], (0, 2)]`	`[0, 1, 2]`
`[2, 3, 1, 0]`	`[(1, 2), (2, 3)]`	`[2, 0, 1, 3]`
`[0, 3, 2, 4, 1]`	`[(0, 1), (0, 2), (1, 2)]`	`[0, 2, 3, 4, 1]`
`[3, 4, 0, 6, 1, 2, 5]`	`[(1, 4), (3, 4), (5, 6)]`	`[3, 1, 0, 4, 6, 2, 5]`

This problem is "Mosey's Birthday" in the DMOJ online problem collection. (Why that problem is called that over there, well, your guess is basically as good as mine.)

Average run length test of randomness

```
def average_run_length(sequence):
```

As in the earlier "Poker test" problem, statistical tests to evaluate the quality of **pseudorandom number generators** first produce a long sequence of pseudorandom numbers from that generator, and then calculate various statistical quantities for that pseudorandom sequence and compare those quantities to the quantities that a truly random sequence would be expected to have.

Another such test computes the average length of the non-ascending and non-descending runs of the given sequence. For example, the sequence `[2, 2, 0, 1, 2, 6]` starts with a non-ascending run `[2, 2, 0]` of length 3, followed by a non-descending run `[0, 1, 2, 6]` of length 4. Note how the turning point element where direction of the run turns is included in both runs. The average run length of this example sequence is therefore (3 + 4) / 2 = 7/2.

This function should compute the average run length of the given `sequence` of integers, and return the answer as an exact `Fraction` object. Be careful to ensure that your function also correctly handles the sequences that start with a run of equal values, forcing you to wait and see whether that first run is non-ascending or non-descending.

`sequence`	Expected result (as `Fraction`)
`[3, 1]`	`2`
`[1, 1, 1, 3, 4, 2, 5]`	`3`
`[1, 1, 1, 5, 5, 2]`	`7/2`
`[1, 3, 8, 8, 11, 9, 10, 11]`	`10/3`
`[4, 4, 8, 14, 15, 11, 11, 7, 0, 3, 1, 1, 9]`	`17/5`
`[42, 42, 42, 42]`	`4`

Students interested on this stuff can also find out what the expected average run length would be for a truly random sequence of integers from 1, ..., *n*.

Count increasing paths

```
def count_increasing_paths(matrix):
```

You are given a *n*–by–*n* element square `matrix` that contains the integers 0, ..., $n^2 - 1$ in some order. The two-dimensional square `matrix` is given as the list of its individual rows, which allows you to index the element in the row `i` and the column `j` with the expression `matrix[i][j]`, just like you did in your linear algebra courses.

Your task is to count how many separate increasing paths there exist inside the entire `matrix` when you are allowed to start the path from any cell and keep moving to any of the four axis-aligned compass directions, as long as you always take each step into an adjacent cell that contains a higher value than the cell that you are moving away from. The edges of the `matrix` do not wrap around to the other side of the matrix in toroidal fashion, but are solid walls reaching to infinity with nothing existing outside these numbers in the finest sadistic fashion.

The increasing paths in the matrix have a nice self-similar structure that allows you to solve this problem recursively. The number of increasing paths of length k that terminate in the cell (i, j) can be computed recursively by adding up the number of paths of length $k - 1$ that lead to those neighbouring cells that contain a lower value than the cell (i, j). The base case of this recursion is the count of singleton paths starting and ending at any one cell being equal to 1. You can either write a recursive function to solve this and use the `@lru_cache` decorator to **memoize** it, or you can use **dynamic programming** to fill in the table of path counts for the length k from the previous table that contains the path counts for the length $k - 1$. If you choose to take the latter route, note that you only need to maintain the **sliding window** of two such tables assuming that you add up path counts in ascending order, since each count of paths of the length k depends only on the counts of paths of the length $k - 1$.

`matrix`	Expected result
`[[1, 0], [2, 3]]`	`11`
`[[3, 0], [1, 2]]`	`8`
`[[1, 2, 8], [6, 3, 4], [0, 5, 7]]`	`39`
`[[10, 15, 1, 12], [14, 9, 7, 0], [11, 6, 2, 13], [8, 5, 3, 4]]`	`89`
`[[36, 31, 39, 5, 17, 1, 23], [13, 35, 48, 4, 38, 18, 43], [21, 28, 10, 12, 25, 34, 6], [16, 2, 8, 19, 3, 11, 44], [41, 32, 47, 7, 37, 29, 0], [46, 14, 22, 27, 45, 26, 15], [24, 33, 40, 9, 42, 30, 20]]`	`258`

Cocke–Younger–Kasami parsing

```
def cyk_parser(text, rules):
```

A context-free grammar consists of **nonterminal symbols** (here, uppercase letters) of which `'A'` is conventionally the **start symbol**, and **terminal symbols** (here, lowercase letters) that the `text` string is made of. The grammar itself is defined by its rewrite `rules` of the form $S \to \alpha$ that determine the sequences of terminal and nonterminal characters that the left hand side S can be replaced with. The `rules` are given as a Python dictionary whose keys are the left hand side nonterminals and the values are the lists of possible right hand sides for those nonterminals. The grammar **produces** the strings of terminal characters that can be somehow produced by application of rewrite rules in some order from the start symbol, each rule applied one at the time to one occurrence of its left hand side in the current string of symbols.

The grammar is furthermore said to be in Chomsky normal form if the right hand side of each rewrite rule is either a single terminal character, or two nonterminal characters. Whether the given terminal string α can be produced from a grammar whose `rules` are in Chomsky normal form can always be efficiently determined with the Cocke–Younger–Kasami algorithm.

The CYK algorithm is a simple recursion on the current nonterminal symbol S and terminal string α, equaling the start symbol and the entire `text` in the top level call. The base case of recursion is when α is a single terminal character, easily answered by checking if there is a rewrite rule $S \to \alpha$. Otherwise, check whether α can be split into two nonempty pieces $\alpha = \beta\gamma$, and the grammar has some rewrite rule $S \to BC$ for which B produces the left piece β and C produces the right piece γ. You can write this parsing algorithm as straightforward recursion using the `@lru_cache` decorator to tame the exponential branching of overlapping possibilities, or as a bottom-up **dynamic programming** systematic tabulation of subproblem results by translating the algorithm pseudocode from the linked Wikipedia page to Python.

`text`	`rules`	Expected result
`'ccb'`	`{'A': ['a', 'b', 'BA'], 'B': ['c', 'BC', 'b', 'CA'], 'C': ['CA', 'a', 'c', 'BC']}`	`True`
`'bccb'`	`{'A': ['BC', 'DB', 'b'], 'B': ['d', 'BA', 'c', 'b'], 'C': ['AC', 'DC', 'AB', 'a'], 'D': ['DB']}`	`False`

For some reason, it's practically a law that whenever you are talking about grammars and theory of automata and parsing like you were some fifties mathematician with a crew cut and horn rimmed glasses, it's compulsory to use Greek letters to denote various strings.

Avoidant subset of points on a circular line

```
def farthest_points(points, p, k):
```

A number of `points` have been placed along a line segment of length `p` that has been wrapped around **cyclically** so that its endpoints are connected to each other, making that line segment a weird little thing a bit like a circle, but not quite so. The `points` are given in sorted order in the list, and each point lies in the range [0, *p*) falling on the line.

Your task is to choose a subset of exactly `k` of these `points` to maximize the minimum pairwise distance among your chosen subset of points. The distances of the other pairs of points in your chosen subset do not matter, but the two points that lie closest to each other determine the solution cost. Since this distance-maximizing subset of points is not necessarily unique, your function should return that maximum distance of the two closest points in your chosen subset.

`points`	`p`	`k`	Expected result
`[3, 5, 8, 11, 13, 20]`	`21`	`3`	`6`
`[3, 5, 7, 12, 13, 20, 21]`	`22`	`4`	`4`
`[0, 1, 3, 6, 11, 15, 16, 22]`	`24`	`4`	`5`
`[0, 1, 7, 10, 11, 13, 14, 19, 20, 27]`	`29`	`4`	`6`
`[3, 4, 9, 10, 12, 14, 18, 20, 21, 23, 24, 25, 28, 31, 34, 40, 42]`	`43`	`4`	`10`

Diamond sequence

```
def diamond_sequence(k):
```

One of the examples in the book "Mathematical Diamonds" by the late great Ross Honsberger features a beautiful infinitely intricate sequence defined by a simple self-referential rule. Being a mathematician instead of a computer scientist, the numbering of sequence positions starts at 1, at which position the first element of the sequence equals one. After that, the element in each position *k* equals the smallest previously unseen positive integer that makes the sum of all elements up to and including that position to be divisible by *k*. The reader can verify that the sequence starts as

1, 3, 2, 6, 8, 4, 11, 5, 14, 16, 7, 19, 21, 9, 24, 10, 27, 29, 12, 32, 13, 35, 37, 15, 40, ...

This function should return the element in position k of this sequence. For simplicity, the automated tester will call this function for values of *k* from 2 to 999999 in this order, so your function does not need to remember the entire sequence so far, but can store the previous element and the set of values seen so far in `global` variables.

Furthermore, since this set of values seen so far consists of positive integers and is **monotonic** (that is, an element once seen will never be unseen), you should also maintain in a global variable the lowest positive integer that has not appeared in the sequence so far. This way, you can save memory by always removing the elements from the set that are lower than this lowest unseen element.

k	Expected result
2	3
7	11
11	7
1000	1618
1618	1000

This sequence has an amazing property illustrated in the above table; if the element in the position *k* equals *m*, then the element in the position *m* equals *k*. In other words, this sequence is a **permutation** of all positive integers that, after the position 1 that gets mapped to itself, consists of nothing but two-cycles. This fact is all the more astonishing in that the process of generating this sequence seemingly has nothing to do with this property, and yet it chugs out the elements in seemingly random order, producing these neat two-cycles as certainly as a train travelling on its tracks, never slipping or missing a beat.

Minimize largest triangle

```
def min_max_triangle(points):
```

You are given a list of 3*n* points on two-dimensional plane. Your task is to divide these points into *n* groups of three points each so that when each group of points is considered a triangle, the area of the largest triangle is as small as possible. This function should return the area of the largest triangle multiplied by two, to ensure that the result is always an integer. For example, if the area of the largest triangle is 5 1/2, your function should return 11.

This problem can be solved recursively by realizing that the first point has to be part of some triangle anyway, so loop through the possibilities of choosing the other two points for that triangle. For each such triplet, compute the best grouping for the remaining points. Once the area of some triangle created in the recursion is not smaller than that of the best solution found so far, you can terminate that branch of the search immediately and return to the previous level.

The example program `geometry.py` in the `ikokkari/PythonExamples` repository contains a utility function to compute the **cross product** of three points that gives twice the signed area of that triangle. Note that depending on the orientation of the three points, the signed area can be negative, so remember to take the absolute value of this quantity to use as the area of the triangle.

```
def cross(p1, p2, p3):
    (x1, y1), (x2, y2), (x3, y3) = p1, p2, p3
    return (x2 - x1) * (y3 - y1) - (y2 - y1) * (x3 - x1)
```

points	Expected result
`[(2, 4), (1, 2), (4, 3), (0, 3), (3, 0), (2, 2)]`	4
`[(10, 15), (13, 14), (11, 0), (14, 1), (4, 12), (12, 9), (0, 12), (10, 10), (15, 3), (1, 3), (3, 5), (0, 11)]`	6
`[(6, 12), (25, 16), (12, 17), (5, 14), (20, 17), (23, 19), (11, 22), (8, 9), (3, 3), (8, 22), (22, 0), (16, 18), (21, 18), (4, 17), (5, 2)]`	13

Optimal text filling

```
def optimal_ab_filling(text, ab, ba):
```

You are given a `text` string guaranteed to consist only of characters `'a'` and `'b'`, and the period character that acts as a placeholder for an empty location. Your task is to fill in these empty locations with characters `'a'` and `'b'` of your choice to minimize the cost over the entire string, and return the minimal cost that you achieved. Two adjacent characters that are equal to each other incur no cost, whereas any two adjacent characters `'ab'` incur the cost `ab`, and any two adjacent characters `'ba'` incur the cost `ba`.

This problem can be solved recursively with a nested function that takes two parameters; the position `i` to the `text` string, and the character that was placed at the previous position `i-1`. This function should try the possible characters to the position `i` (whenever the `text` already contains a non-placeholder character in that position, the choice is forced without branching) and choose the character that minimizes the total overall cost continuing from that position. Since this recursion visits the same subproblems exponentially many times, use the `lru_cache` magic to prune this into a linear number of branches.

`text`	`ab`	`ba`	Expected result
`'b.a'`	`3`	`4`	`3`
`'b.b.a'`	`1`	`2`	`2`
`'....ab'`	`4`	`1`	`4`
`'b.aaa..a'`	`3`	`2`	`2`
`'..a.baa.'`	`5`	`5`	`10`

This problem is adapted from the Google Code Jam 2021 problem "Moons and Umbrellas".

S-expression evaluator

```
def s_eval(expression):
```

Lisp, the great granddaddy of all functional programming languages, is known for its **homoiconic** syntax that makes no distinction between code and data, since both code and data are presented in the exact same syntactic form of **s-expressions** of fully parenthesized expressions written in **prefix notation**. An s-expression is either an **atom** (such as 42) or a structure `(f a1 ... an)` where `f` is a function symbol and each `ai` is an s-expression whose value becomes that argument to the function `f`. S-expressions are evaluated in order of **inside out** so that, for example, the expression `(* (+ 1 1) (– 7 3))` first simplifies to `(* 2 4)`, which then simplifies to `8`.

This problem asks you to write a function that evaluates the given s-`expression` limited to the situation where the only functions are the three basic arithmetic operators `'+'`, `'–'` and `'*'` that always receive exactly two arguments each, and the atoms are nonnegative integers. Your function does not need to be able to deal with syntactically incorrect or invalid s-expressions.

In a more advanced course, this problem could be solved with a classic **recursive descent parser**, but can actually be solved even more simply with one linear loop aided by an initially empty `list` used as a **stack**. Loop through the characters of the `expression` and skip all whitespace and left parenthesis characters, but `append` each arithmetic operator and integer atom that you encounter to the stack. Whenever you encounter a closing parenthesis, `pop` the preceding two arguments and the operator from the stack, perform that corresponding arithmetic operation to those two arguments, and `append` the result back to the stack. Once you reach the end of a legal S-expression, the stack will contain exactly one integer atom to return as the result. Make sure to also correctly handle the special case where the `expression` is a single integer atom.

`expression`	Expected result
`42`	`42`
`(+ 2 2)`	`4`
`(* (* 0 4) (+ 2 3))`	`0`
`(* (* 58 (– 47 49)) (* 40 (– 40 49)))`	`41760`

Expand recursively run-length encoded string

```
def expand_string(text):
```

Text strings that contain repeated patterns can sometimes be efficiently compressed with **run-length encoding**, where the repeated pattern is stored only once with the repetition count. If the pattern itself contains repeated subpatterns, these subpatterns can themselves be run-length encoded, this recursive structure potentially nesting arbitrarily far.

Let us use the syntax *k*`[pattern]` to indicate that the `pattern` is to be repeated *k* times. For example, `2[abc]` would expand to `abcabc`. If the pattern itself contains such recursive repetitions, they are expanded inside out before the outer expansion is performed. For example, `2[a3[bc]]` would first expand to `2[abcbcbc]`, which then expands to `abcbcbcabcbcbc`.

This function should expand the given `text` in this manner. Your function may assume that the `text` will follow the syntax rules of this encoding, and does not need to be able to deal with any malformed input strings. Same as in the previous problem of evaluating recursively nested S-expressions, of course you could but you don't need to write a full-on recursive descent parser, since a stack-based approach that executes one linear for-loop through the `text` without need to backtrack is able to handle this job.

`text`	Expected result
`'4[x]'`	`'xxxx'`
`'4[2[u]]xqk4[r]2[i]'`	`'uuuuuuuuxqkrrrrii'`
`'t5[4[n]]'`	`'tnnnnnnnnnnnnnnnnnnn'`
`'l3[io]bs2[d]iiyg2[2[kw]i]'`	`'lioioiobsddiiygkwkwikwkwi'`

This cute little problem is the Leetcode problem 394, “Decode string”.

Maximum palindromic integer

```
def maximum_palindrome(digits):
```

Here is a cute little puzzle from a technical interview problem collection for which writing the actual code is not too difficult, but the difficulty is in coming up with the algorithm to solve the problem under the time limit of solving the question under pressure of getting the job and securing your future on the good side of the barbed wire fence in the diminishing race to the bottom that the entire computing industry has suddenly turned into since the golden years of the youth of this author. Yes, children, gather around to hear how there once was a peaceful time when you could get an $80K entry level job (equivalent to about $150K compensation in today's money) for merely knowing how to write loops in Visual Basic.

Given a string of `digits` that you are allowed to use, your task is to construct the largest possible integer that is a palindrome, that is, reads the same from left to right as from right to left. The resulting number may not contain any leading zero digits, but may contain zeros inside the number itself. Each digit can be used as many times as it appears inside the `digits` string.

`digits`	Expected result
`'0'`	`0`
`'011'`	`101`
`'8698'`	`898`
`'123123123'`	`3213123`
`'225588770099'`	`987520025789`

Optimal palindromic split

```
def palindrome_split(text):
```

Any `text` string can always be split into disjoint contiguous pieces so that each individual piece is a palindrome, that is, reads the same from left to right and right to left. Of course, making each piece a singleton character trivially satisfies the definition of a palindrome, but some strings allow for a more elegant partition that consists of significantly longer pieces.

Following the general principle of how you get what you reward, let us define that each palindromic piece whose length is k characters is worth k^2 points. For example, a piece with five characters would give you 25 points, much better than using those same five characters as five separate pieces worth a measly one point each. This function should determine the optimal way to split the given `text` string into palindrome pieces to maximize the overall score, and return that maximal score.

This problem cannot be solved with a **greedy** approach where you make the first palindromic piece to be as long as possible, and then try to maximize the score over the remaining `text` in the same greedy vein. The greedy choice for the first palindrome might consume some characters that were necessary for the next piece to be maximally long, such as in `'SOSAIPPUAKAUPPIAS'`. You should instead use **dynamic programming** to compute the maximum score M(i) as the maximum over the immediate scores of each possible palindrome that starts from position i, plus the maximum score achievable from the position after that initial palindrome. You can solve this as a recursive function using the `@lru_cache` decorator to **memoize** it to avoid iterating through the same subproblems an exponential number of times, or you can go hardcore and tabulate the maximum scores backwards from the end of the `text` to the beginning, and then return `table[0]` as your final answer.

`text`	Expected result
`'AXA'`	`8`
`'ABAB'`	`10`
`'())('`	`16`
`'CABCCBA'`	`37`
`'CACAACCAABBAACCAA'`	`205`
`'AACBAACBCAACCA'`	`60`

Also, do not fail to appreciate afterwards how your algorithm is not actually dependent on the fact that the acceptability of piece is being a palindrome and that the scoring function is the square of piece length, but the dynamic programming algorithm itself is exactly the same for literally any acceptability criterion and scoring function that can be freely plugged in.

Semiconnected guys

```
def is_semiconnected(edges):
```

A **directed graph** consists of nodes numbered 0, ..., *n* – 1. The `edges` of this graph are given as a list where each element `edges[u]` is the list of the edges emanating from the node *u* to its immediate neighbours. This function should determine whether this graph is **semiconnected**, that is, for each two nodes *u* and *v* there exists a directed path from *u* to *v*, or a directed path from *v* to *u*. Note that this question is not the same as testing whether the graph is fully connected when each edge is considered to be undirected, since the edges in each directed path still have to be consistently pointing in the same direction. Each individual node is considered to be automatically connected to itself with the empty path.

The simplest way to solve this problem would be to implement the classic Floyd–Warshall all-pairs reachability algorithm to determine the reachability between all pairs of nodes simultaneously, and then verify from the resulting reachability matrix that the nodes of each pair of nodes in the graph are reachable from each other in at least one direction. Since the paths connecting nodes to each other inevitably have many subpaths in common, the Floyd–Warshall algorithm is superior to performing a breadth-first search separately from each node and collating the final results of these searches.

`edges`	Expected result
`[[1, 3], [], [0], [1, 0]]`	`True`
`[[3, 2], [4], [], [1], [0]]`	`False`
`[[5], [6, 2], [5, 4, 7, 0, 6], [6], [3], [4, 7, 6, 3], [5, 0], [0, 3, 2, 5]]`	`True`
`[[1, 7, 4, 3], [4, 6], [5], [1, 7, 6, 5], [7], [1, 6], [2, 3], [2, 5]]`	`True`
`[[1], [5], [1], [7], [6, 5], [1], [2, 4], [1, 0, 8], [4, 6]]`	`False`

Falling squares

```
def falling_squares(squares):
```

Starting with an initially flat surface with uniform height of zero, some `squares` fall straight down from a large height one square at the time, in the style of Tetris. Each square is represented as a two-tuple `(x, w)`, where `x` is the *x*-coordinate of the left edge of the square, and `w` is the width (and thus also the height) of that square. Each square falls straight down until some non-zero part of it lands on a flat surface, either the floor or some previous square. The surfaces between squares are assumed to be infinitely sticky, so that the square will stick to where it lands, again in Tetris style, without tipping over, regardless of any dramatic overhang of the column squares set up so far.

Given the list of `squares` in the order in which they fall one by one, this function should return a list of the same length whose each element equals the largest height that the entire resulting structure reaches after dropping that square. Your function should somehow keep track of the shape of the current surface, and efficiently update that surface after each individual square.

This problem is a hardened version of problem 699 of LeetCode, "Falling squares". Their version keeps the coordinates and square sizes small enough for you to simply maintain a **heightfield**, a list of current height values for the range of x-coordinates that the squares fall on, and update it with a trivial for-loop for each new square. In our version, after the smooth start with test cases small enough to help your debugging, the automated tester will start shooting up these numbers exponentially. Instead of using a heightfield, you therefore need to set up some kind of higher level representation for the outline of the shape created by the squares so far that remains efficient in both time and space even for square positions and widths up in the gazillions.

`squares`	Expected result
`[(4, 3), (6, 1), (5, 1)]`	`[3, 4, 4]`
`[(8, 1), (2, 1), (11, 1), (3, 2)]`	`[1, 1, 1, 2]`
`[(4, 3), (6, 3), (2, 1), (2, 2), (13, 1)]`	`[3, 6, 6, 6, 6]`
`[(7, 2), (3, 2), (7, 1), (9, 3), (5, 2), (9, 3), (7, 4), (10, 1)]`	`[2, 2, 3, 3, 3, 6, 10, 11]`

Tchuka Ruma

```
def tchuka_ruma(board):
```

Tchuka Ruma is an interesting solitaire variation of Mancala, the family of **sowing games**. The `board` consists of a **store** in position 0 and **holes** in positions 1, ..., *n* – 1. The store is initially empty, and each hole initially contains some number of pebbles. Your goal is to get as many pebbles as you can in the store before you inevitably paint yourself in a corner with no more possible moves. This function should return this maximal number of pebbles that end up in the store.

An individual move consists of picking up all pebbles of some nonempty non-store hole, and sowing these pebbles into holes starting from the position immediately preceding the chosen hole and continuing leftward one pebble at the time. If this sowing process goes past the left edge of the board after placing a pebble in the store, the sowing continues cyclically in the holes at the right end of the `board`. This sowing move can then end in three possible ways:

- If the last pebble ends up in the store, that move is successfully completed. The player gets to freely choose his next move from the available nonempty holes.
- If the last pebble ends up in an empty non-store hole, the game is over with the final score equal to the number of pebbles that ended up in the store.
- Otherwise, the move is not over, but continues by the player picking up all the pebbles in that last hole and sowing them leftward in the exact same fashion, with the same end conditions.

`board`	Expected result
`[0, 3, 0]`	1
`[0, 3, 1]`	3
`[0, 3, 2, 4]`	7
`[0, 1, 4, 4, 5]`	13
`[0, 5, 4, 7, 7, 0, 0, 8, 1]`	2
`[0, 5, 9, 3, 0, 1, 9, 2, 3]`	32

Since the sowing process can only move these pebbles leftward and cannot skip over the store from where no pebbles ever get picked up from, we see that even though a single move can consist of several stages of picking up and sowing pebbles, each move must eventually terminate after a finite number of steps, since during each round trip around the board, the number of pebbles available for seeding will decrease by one.

Slater–Vélez sequence

```
def slater_velez(k):
```

This problem greatly resembles the earlier “Diamond sequence” problem and can be solved with the same technique, even though the sequence of positive integers that the self-referential process chugs out is even more chaotically whimsical than in the diamond sequence. The numbering of sequence positions again starts at 1 instead of 0, and the first element in that first position also equals 1. After that, each element is always the smallest positive integer *n* that has not appeared so far in the sequence, under the constraint that the absolute difference of *n* and its immediate predecessor in the previous position of the sequence also has not previously appeared as the absolute difference of any two consecutive elements earlier in the sequence. With some quick arithmetic, the reader can verify that this sequence starts its journey as

1, 2, 4, 7, 3, 8, 14, 5, 12, 20, 6, 16, 27, 9, 21, 34, 10, 25, …

since the sequence of absolute differences between the consecutive elements of the previous sequence begins as

1, 2, 3, 4, 5, 6, 9, 7, 8, 14, 10, 11, 18, 12, 13, 24, 15, …

This function should return the value in position *k* of this sequence. The automated tester will again give your function consecutive values of *k* starting from 2 upwards, so your function can store the previous element and the sets of previously seen values and previously seen absolute differences as global variables. Since both sets involved in this problem are monotonic, you can also apply the same technique as in the diamond sequence problem of keeping track of the lowest unseen value in both sets, which eliminates the need to explicitly store the elements lower than that value and massively alleviates the memory use of this function.

Odds and evens

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

This function is given two strings `first` and `second`, both guaranteed to consist of digit characters `'0'` and `'1'` only. These strings are to be converted to lists of positive integers so that each `'0'` becomes an even integer and each `'1'` becomes an odd integer, under the two constraints that (a) both resulting lists must be in ascending sorted order and (b) each integer may appear at most once in these two lists. Your task is to determine, over all possible ways to convert these strings into lists of integers under these constraints, the minimum possible value of the largest integer to appear in either list.

`first`	`second`	Expected result
`'1111'`	`'1'`	`9`
`'000111'`	`'000111'`	`12`
`'0001'`	`'0001'`	`9`
`'10111011'`	`'10111011'`	`21`

This cute little problem is the problem "Zrinka" in the DMOJ online problem collection. It is actually a bit harder than it initially seems in that this author misunderstood his own problem and made an error in the original version that one tyro student pointed out. However, when this specification was given to Claude Sonnet 4, it made the same error as the original author!

The prodigal sequence

```
def front_back_sort(perm):
```

When you are allowed to repeatedly swap any two elements in the list, many efficient **comparison sort** algorithms are known to sort any list of items in the sorted order. However, suppose that your moves are restricted to picking any one element from the list, and moving that element to either to the front or to the back of the list. Given the `perm`utation of integers 0, ..., *n* – 1, how many such moves would be needed to rearrange the elements of that permutation in sorted order?

Of course, *n* – 1 moves will always trivially suffice by repeatedly moving each element to the back starting from one and going in sorted order. But how do you recognize if some smaller number of moves would also work?

`perm`	Expected result
`[0, 1]`	0
`[1, 0, 2]`	1
`[0, 2, 3, 1]`	2
`[2, 1, 3, 4, 0]`	2
`[2, 1, 4, 0, 3]`	4
`[0, 1, 5, 2, 4, 3]`	2

Solution to this problem can also benefit a lot from thinking instead of just brute forcing through all possible move permutations.

Descending suffix game

```
def descending_suffix_game(board, n):
```

This cute little problem was apparently used as a job interview question for a quant position at Jane Street, the trading firm known for their love of puzzles (solutions). You have a shuffled deck made of *n* cards marked with integers 1, …, *n*, and are invited to build a board from the cards that you draw randomly from the deck, one card at the time. Each round, you get to choose whether to **stand** with the board that you have built so far, or if the deck still has cards remaining, **hit** to draw the next random card and append it at the end of your current board. Once you choose to stand, your final score equals the sum of the cards in the longest descending suffix of the board. For example, if you choose to stand with the board `[9, 3, 8, 4, 2]`, your final score would be 8 + 4 + 2 = 14.

This function should determine the expected value of the given `board` assuming that the player chooses his moves optimally, analogous to a blackjack player mechanistically following the basic strategy card to maximize his expected result. Under the basic assumptions of sufficiently small stakes in a repeated game so that your utility curve of points is linear, the expected value and thus the expected utility of the given `board` equals max(*s*, *h*), where *s* is the value of standing with the descending suffix that you have so far (easy enough to compute), and *h* is the expected value of hitting, averaged over all the possible cards that can come out next under the premise that you will continue making optimal decisions of standing or hitting regardless of whatever card the gods of randomness happen to give you. You should do all calculations with exact `Fraction` objects.

board	n	Expected result
`[4, 1, 2]`	5	13/2
`[1, 4, 3, 7]`	7	77/6
`[11, 1, 9, 5, 12]`	12	100339/5040
`[12, 10, 7, 5, 3]`	12	37

The original question asked to calculate the expected value of the game for *n* = 9 starting from the empty board. This turns out to equal 2749727/181440, approximately 15.155, agreeing with the commenters in the Math Overflow thread linked above.

The Borgesian dictionary

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

A popular phone and tablet game that this author has seen ads for and people playing on the transit gives the player a bunch of `letters` to be used to create as many words as possible. Since this mindless task of patience and memory of iterating through all possibilities seems better suited for a machine than a man, it's time to automate this task and relieve the humans from at least this one burden imposed on us by these modern times. This function should return the alphabetical list of all legal `words` that can be created from these `letters`. You can use each letter in the word up to as many times as it appears in the `letters`. Since the wordlist that we use in these problems is pretty huge, this function should return only the words that have at least seven letters. For simplicity, your function may also assume that both `letters` and `words` are given in sorted order.

This function should be a relatively simple exercise on backtracking search, building the result word one character at the time by looping over all the remaining letters, and recursively trying to extend the new word with each such letter until a dead end is reached. Make sure also not to scrub through redundant search paths of duplicated letters in the same level of recursion, and use **binary search** from the Python standard library `bisect` module to quickly determine whether the word that you have built up so far it itself a legal word, and whether it could in principle be extended into a longer word. The efficiency of all backtracking algorithms depends on their ability to turn back as soon as possible once they have painted themselves into a dead end corner. Will you align yourself with the flow of the universe and reach your goal in harmony, or will you keep pushing against infinity only to perish like a dog?

`letters`	Expected result (using the wordlist `words_sorted.txt`)
`'aghimoopss'`	`['agomphosis', 'amishgo', 'gaposis', 'gomphosis', 'goosish', 'mishaps', 'samshoo', 'shamois', 'shampoo', 'shampoos', 'sophism']`
`'aeekrstuy'`	`['aesture', 'austere', 'estuary', 'euaster', 'euskera', 'keyseat', 'keyster', 'restake', 'retakes', 'sakeret', 'sautree', 'streaky', 'turkeys', 'tuyeres']`
`'aabcgkostt'`	`['attacks', 'catboat', 'catboats', 'costata', 'gasboat', 'sackbag', 'tabasco']`

The Dictorian borgesitory

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

Kind of like an upside-down version of the previous “Borgesian dictionary” problem, this function is given a list of `words` and some `letters` that you are allowed to use. Your task is to choose a subset of `words` so that all these chosen words can be made from the given `letters`, when each individual letter can be used only at most as many times as it appears in `letters`. The score for the subset of words is the sum of the lengths of those words, so longer words are better. This function should return the highest total score that can be achieved with the given `words` and `letters`.

This problem is a pretty basic backtracking recursion (you either take in each word or leave it), but this problem is also an opportunity for you to try out the excellent `Counter` data structure in the Python standard library `collections` module to keep track of the remaining letters and quickly determine if the current word can still be constructed from your remaining letters.

words	letters	Expected result
`['hymnody', 'lycodes', 'phrasal', 'chams']`	`'aaabccddehhjllmnoopqrsssyy'`	`19`
`['dicht', 'dangler', 'rosal', 'ostend']`	`'aabbddddeeghillnoorstyy'`	`12`
`['nonfluids', 'ululate', 'tentability', 'shook', 'ticktacker', 'spectacularism', 'bourre', 'inexposure', 'sassan']`	`'aaaaaabbccccddeeeeeeeffiiiiiikkkllllmmnnnnoooooppqrrrrrrssssssstttttttuuuuuwxxxy'`	`60`

The art of strategic sacrifice

```
def pick_it(items):
```

You are given a list of `items`, each item an integer. The list contains a total of *n* items, and you need to perform the following step exactly *n* – 2 times; choose any one of the `items` that is neither the first or the last item remaining, and remove that item from the list. At each such removal, your score increases by the square of the current predecessor of that item plus the item itself, and decreases by the square of the current successor of that item. This function should compute the highest score achievable by performing these *n* – 2 item removals in the optimal order.

This problem is adapted from the problem "Pick it" in the DMOJ online problem collection by adding the squaring and subtraction of the neighbouring elements to make the whole thing more nonlinear.

Similar to the previous problem "The Borgesian Dictionary", this problem is a pretty basic exercise in backtracking, so that each level of this recursion loops through the remaining elements and for each such element, recursively tries to build up the rest of the score from the remaining elements. To allow the code to find the current predecessor and successor of each element in constant time, you should use the **dancing links technique** with two lists `prev` and `succ`, both of length *n*. The elements `prev[i]` and `succ[i]` give the current predecessor and successor of the element in the position *i*, respectively. To remove the element in position *i*, update the predecessor and successor information of its neighbours, and after returning from the backtracking recursion, downdate the same information of its neighbours back to how they were before the removal.

`items`	Expected result
`[1, 4, 3, 5]`	`-25`
`[1, 5, 7, 1, 6, 2, 2]`	`142`
`[3, 7, 1, 2, 2, 5, 6, 7, 7]`	`117`
`[10, 8, 1, 6, 7, 6, 6, 7, 9, 9]`	`481`

Colonel Blotto and the spymaster

```
def colonel_blotto(friend, enemy, prize):
```

"Colonel Blotto game" is an umbrella term for a family of games where the players have to blindly commit their resources in separate battlefields before knowing how the enemy will allocate their resources. Once both sides have made this commitment, the `prize` for each separate battlefield goes to the player who allocated more resources in that battlefield in a winner-takes-all fashion.

In our version of such a game, the `friend` list gives the strengths of *n* units to be distributed over *n* battlefields, listed in descending sorted order. However, instead of us having to allocate these units under fog of war, our valiant spymaster has bravely ran through the eye of the needle to acquire the complete allocation of `enemy` forces over these battlefields! This function should compute the maximum total prize available by allocating our units wisely with this knowledge. To ensure that there are no ties in any battlefield, our heroic forces all have an even-valued strength, whereas those dastardly invaders all have an odd-valued strength. You must allocate exactly one unit in each individual battlefield, and are not allowed to combine multiple units into one for larger strength. All three lists, `friend`, `enemy` and `prize` are guaranteed to have the same number of elements.

A common principle in all Colonel Blotto games (for example, electoral **gerrymandering**) is that if you are going to lose some battlefield, you really want to lose it by as much as possible, since this doesn't make enemy's gain there any larger while saving our precious resources for better use while the enemy allocates its power to fight air. In the same vein, if you are planning to win some battlefield, you should try to eke out as small a victory as possible. This idea gives you a simple two-way choice for each battlefield; either give it up and assign your weakest remaining unit there, or choose to win and assign your weakest remaining unit that is able to beat the enemy force there.

`friend`	`enemy`	`prize`	Expected result
`[14, 10, 6]`	`[9, 13, 21]`	`[7, 1, 5]`	`8`
`[18, 12, 4, 2]`	`[9, 3, 21, 17]`	`[7, 3, 1, 12]`	`22`
`[18, 16, 16, 14, 12, 8, 6, 4]`	`[11, 9, 13, 5, 17, 19, 17, 21]`	`[1, 3, 15, 11, 7, 2, 4, 7]`	`37`

Unamerican gladiators

```
def gladiators(our, their):
```

Here is another banger from Peter Winkler's "Mathematical Puzzles: A Connoisseur's Collection". Each gladiator has a strength that is a positive integer. When two gladiators with strengths x and y duel, they each win with probabilities of $x / (x + y)$ and $y / (x + y)$ proportional to their respective strengths, so that with some good fortune, even David can slay Goliath. The winner of the duel absorbs his fallen opponent's strength, *Highlander* style, so that his new strength becomes $x + y$.

Two teams of gladiators, `our` valiant heroes versus `their` dastardly villains, take turns to choose any one of their gladiators and any one of their opponents gladiators to a duel, and continue in this manner until one team has been utterly vanquished. If `our` boys go first, and both teams always choose the gladiators to maximize their team's chance of winning, what is the probability that our team of heroes emerges victorious in the end? You should, of course, use exact `Fraction` objects for all calculations and the final result. The fate of any individual gladiator doesn't matter, but in the best samurai spirit, each gladiator will gladly face even hopeless odds if it helps maximize his team's chances of triumph.

This problem can be solved with **expectimax** search, a variation of minimax search where at each level of recursion, the consequences of the duels between each possible choice of gladiators for this round, win and lose, are evaluated recursively from the opposing team's point of view, and the possible results are added together weighted by the win/lose probabilities. Then just return the expected value of the best possible choice for the duelling pair at that level. However, as it always seems to happen with expectimax, the size of the search tree branches massively and soon becomes prohibitive. But should the reader add some debugging outputs for the expected values of each move, they should soon realize something very interesting that ought to help speed up this search by several orders of magnitude.

`our`	`their`	Expected result
`[4, 1]`	`[3, 3]`	`5/11`
`[9, 6, 1]`	`[4, 2, 8]`	`8/15`
`[5, 11, 7, 5]`	`[10, 10, 1, 7]`	`1/2`
`[15, 14, 15, 15, 24]`	`[23, 19, 7, 14, 8]`	`83/154`

Fox and hounds

```
def fox_and_hounds(fox, hounds):
```

Fox and hounds is an abstract strategy game whose complexity and state space are small enough for us to here calculate the optimal moves for both players in any possible game situation. As explained on the linked Wikipedia page, the game is played on black squares of an 8-by-8 checkerboard where the four hounds start from row zero, aiming to capture one fox who starts from row seven. On its turn, a hound can only move forward towards the last row, whereas the fox can move one step to any one of the four diagonal directions. There are no captures, promotions or jumps, but the fox wins if it reaches the zero row, and the hounds win if they block the fox and make it unable to move.

From the starting position, this game is known to be a win for the hounds, but this function should compute whether the `fox` has a winning strategy against the four `hounds` in the given position on his turn to move. The **minimax search algorithm** that can solve not just this but any **complete information two player game of alternating moves** consists of two **mutually recursive** functions `fox_move` and `hounds_move` that each loop through the possible moves for that player in the given situation, and for each move, call the other function to determine if the opponent has a winning strategy in the state resulting from that move. Both functions return `True` if there exists a move for which the opponent does not have a winning response, and `False` otherwise.

Despite the small state space of this game, the space of possible move sequences is still exponentially large, especially since many possible sequences of moves lead to exact same game states. This necessitates the use of **transposition tables** to remember the states that have already been encountered during the recursive search. Should the search reach the same game state again, just look up the result of that state from the table. The easiest might be to use four sets `fox_win`, `fox_lose`, `hounds_win` and `hounds_lose` to store each state after first evaluation. The reader should still think up additional ways to speed up this algorithm in various places.

fox	hounds	Expected result
`(7, 0)`	`[(0, 1), (0, 3), (0, 5), (0, 7)]`	`False`
`(6, 1)`	`[(1, 2), (3, 2), (4, 3), (4, 5)]`	`False`
`(6, 5)`	`[(0, 5), (2, 1), (3, 0), (5, 6)]`	`True`
`(5, 0)`	`[(1, 2), (2, 7), (3, 0), (3, 4)]`	`False`
`(7, 0)`	`[(1, 0), (3, 6), (4, 7), (5, 4)]`	`True`

Toads and frogs

```
def toads_and_frogs(board):
```

Continuing on the theme of complete information abstract strategy games with alternating moves made by cute animals, toads and frogs produces surprisingly complex situations in combinatorial game theory despite its bare and simple rules and setup. Some toads (`'T'`) and frogs (`'F'`), at least one of each type of animal, have been placed on a one-dimensional `board` of squares. On the toads turn to move, one toad can either move one step to the right to an adjacent empty square (`'.'`), or hop over a frog immediately to its right to an empty square immediately behind that frog. (Unlike in checkers, hopping over a frog does not capture it.) Frogs move the exact same way, except to the left. The blocked side, unable to make any move on its turn, loses the game. The board does not wrap around cyclically, but the animals that reach the end of the board either way are blocked.

This function should calculate whether there exists a winning strategy for the toads on the given `board` starting from the toads turn to move. The two mutually recursive functions for finding the best moves for toads and frogs (or just one suitably parameterized function for this symmetric game) in the given situation are similar as in the previous game of fox and hounds. However, the use of **transposition tables** to memoize game states that have been previously encountered during the recursive search is even more essential in this game, since this game is basically nothing but transposed move sequences leading to exact same game states. In addition, you still need to think up ways to speed up your code to pass the automated fuzz tester within the set time limit.

`board`	Expected result
`'T..F'`	`False`
`'T...T.F..F'`	`False`
`'TTT...FF.F..'`	`True`
`'..T.TT.TTFFFF.F.'`	`True`
`'T.TFTTT.FT..F.FF.F'`	`False`

Finding the winning strategy in toads and frogs is a PSPACE-complete problem, so every algorithm will fall into exponential time occasionally, but most instances can be solved in polynomial time.

Post correspondence problem

```
def post_correspondence_problem(first, second, lo, hi):
```

Post Correspondence Problem is a famous problem on string matching, first discovered way back in the year 1946 after all the research silently done to explore the theoretical limits of computability even during the massive world war.. Given two lists `first` and `second` of equal length of strings made of letters `a` and `b`, such as `['a', 'ab', 'bba']` and `['baa', 'aa', 'bb']`, the problem is to find some nonempty sequence of position indices so that concatenating the elements of both of these lists separately as listed in this sequence produces the exact same result for both lists. For example, as the reader can quickly verify, the position sequence `(2, 1, 2, 0)` produces the same result string `'bbaabbbaa'` and `'bbaabbbaa'` for both of these lists, here colour coded for your convenience.

This function should determine whether there exists some position sequence that produces the same result for `first` and `second` lists so that the length of the result is at least `lo` characters and at most `hi` characters. This problem should be a pretty straightforward exercise on backtracking, but you can look for ways to prune the search and alternatives at each level of recursion.

first	second	lo	hi	Expected result
`['a', 'ab', 'bba']`	`['baa', 'aa', 'bb']`	5	10	`True`
`['ab', 'bbab', 'bbb']`	`['aba', 'aabb', 'abba']`	5	5	`False`
`['baa', 'bb', 'bba', 'bbbb', 'a']`	`['b', 'baba', 'baaa', 'aaaa', 'aa']`	6	12	`True`

In absence of `lo` and `hi` parameters, this famous problem is **algorithmically undecidable** in that there simply does not exist any finite algorithm that is guaranteed to determine the existence or nonexistence of a nonempty solution for the two given parameter lists in finite time in all possible cases. The problem is **semidecidable** in that if a solution exists, systematic generation of all solution paths in breadth first manner will eventually find one, but there is no general way to determine in finite time whether the current partial solution cannot be extended to a solution.

Optimal bridge placement

```
def optimal_bridges(perm, k):
```

A straight river, uniformly one unit wide, flows through the peaceful land from east to west. On the north side of that river are n houses, placed at uniform distances at positions 0, ..., n – 1, with an idyllic linear street connecting all these houses along the riverbank. On the south side of the river, n workplaces have been similarly positioned at 0, ..., n – 1 along another linear riverbank street. The homeowner `i` works in the workplace `perm[i]` and is the only homeowner working in that place, so `perm` is guaranteed to be a permutation of integers 0, ..., n – 1.

Your task is to build exactly k bridges across the river. Each unit length bridge must be placed at an exact integer position, and cross the river directly to the same position on the other side. Your task is to place these bridges optimally to minimize the sum of distances from the homeowners to their workplace, each homeowner using the bridge that gives him the shortest path to where he is going. The same bridge can be shared by any number of homeowners on their way to work. The distance travelled by each individual homeowner `i` to his workplace `perm[i]` is therefore given by the formula `abs(i-b) + 1 + abs(perm[i]-b)`, using the best bridge position `b` for that homeowner. This function should return the smallest possible total distance over all homeowners.

The automated fuzz tester will give your function enough test cases large enough so that brute forcing through the `combinations(range(n), k)` possible bridge positions will not pass these tests within the set time limit. Instead, you need to loop through all possible groupings of which block of homeowners will be using which bridge, after which you determine where to place that particular bridge to give the shortest total trip within only that group of homeowners.

`perm`	`k`	Expected result
`[0, 1]`	2	2
`[4, 2, 1, 0, 3]`	2	15
`[1, 4, 2, 3, 0, 6, 5]`	2	21
`[5, 7, 10, 4, 0, 6, 11, 2, 13, 8, 9, 1, 12, 14, 3]`	3	87

This problem is "Palembang bridges" in the DMOJ problem collection and online judge.

The way of a man with array

```
def pebblery(predecessors):
```

Jack, an outwardly respectable Victorian gentleman who is secretly a malignant narcissist, knows *n* ladies numbered 0, ..., *n* – 1. The social hierarchy of these ladies forms a **directed acyclic graph** so that each lady number `i` has some immediate `predecessors[i]` who are all numbered higher than her. Jack's ultimate goal is to claim the lady number 0 in his harem for his ultimate validation.

Jack's demented game of courtship proceeds as follows. Each round, Jack can do one of his two possible moves. He can successfully add in his harem any lady whose all immediate predecessors are currently in his harem. (Thus if some lady has no predecessors, Jack can claim her in his harem at any time.) Alternatively, he can coldly discard any one of the ladies from his harem. Thanks to Jack's well honed skills of psychological manipulation, any lady so discarded can later be courted to rejoin the harem under the same precondition of all her immediate predecessors being in Jack's harem at that moment, making her powerless to resist Jack's incessant hoovering and love bombing with promises of how it will all be totally different this time.

However, being paranoid of the women of his harem secretly scheming together against him, Jack wants to minimize the size of his harem at any one time on the path of acquiring the favour of the lady number 0, not including the lady zero herself in this count since Jack doesn't give a jack about anything after that final conquest and will be dumping the rest of his harem anyway. The total number of moves required to reach the ultimate lady is irrelevant to him; the mastery of delicate placement of favours along the way is reward enough anyway. Given the social ranking `predecessors` structure of these ladies, compute Jack's optimal courtship strategy that minimizes the maximum number of ladies in his harem at any given time, and return that minimum. The strategic removal and replacement of ladies along the way is the key to success in this work, but the complexity of this search increases dramatically when multiple paths lead to the intermediate goal. As any skilled practitioner surely knows, direct approach is rarely optimal.

`predecessors`	Expected result
`[[1, 3], [2], [3], []]`	2
`[[1, 2], [2], [3, 4, 5], [4, 5], [], []]`	4
`[[1, 2, 4, 5], [4, 5, 6], [3, 4, 6], [], [5], [6], []]`	5
`[[1, 2, 6], [3, 5, 6, 8, 9], [], [4, 10], [8], [6, 7, 8], [9], [11], [11], [], [], []]`	6