Third Collection of Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of June 9, 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. 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 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 an integer repeats itself, at which point the sequence admits defeat and stops. Your task 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.

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, not in 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))`

Lehmer code: encoding

```
def lehmer_code(perm):
```

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 integer 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 collection), this allows the permutations of 0, ..., $n - 1$ to be encoded **bijectively** ("one-to-one") to the integers 0, ..., $n! - 1$. The inverse versions of these two functions applied in reverse order will then encode 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 boss 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 into a specific complexity class is still an open question.

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. 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 opened herp derp 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 partnership 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`

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...)

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 do 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, each barber `i` always takes 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.

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.

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 each woman *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 number 7 for both of her daughters 14 and 15.

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 sufficiently distinct from each other genetically 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 **lowest 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.

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'`

Poker test of randomness

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

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.

One such test looks at all the n – 4 sublists of five elements of the `sequence` of *n* elements, and classifies each sublist according to which hand in dice poker they correspond 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]`

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.

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 under these constraints, the minimum possible value of the largest integer that appears in either list.

`first`	`second`	Expected result
`'11'`	`'1111'`	`11`
`'10'`	`'01'`	`4`
`'000111'`	`'111000'`	`12`
`'01101'`	`'0000'`	`12`
`'0001'`	`'1110110'`	`11`
`'0111000011'`	`'001000001'`	`24`
`'1100010110000100'`	`'01011110'`	`26`

This cute little problem is the problem "Zrinka" in the DMOJ online problem collection.

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. 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 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 `101001001`1`1001`1`000100101`1`01`1`0001`1.

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]`
`'1001000001110001001011100100110101011111'`	`[95, 132, 27, 20, 1]`
`'00101011'`	`[32]`
`'0010010101000110100000001011011'`	`[749, 523, 2]`

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

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*.

Optimal text filling

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

You are given a `text` string guaranteed to consist 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, but 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`

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`

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 use 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.

`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`

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** number 0 and **holes** numbered from 1 to *n* – 1, each hole initially containing some number of pebbles. Your goal is to make moves to get as many pebbles as you can in the store before you inevitably paint yourself in a corner so that there are no more possible moves. This function should return this maximal number of pebbles.

An individual move consists of picking up all pebbles of some nonempty non-store hole, and sowing these pebbles into holes starting from the hole immediately to the left of 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 in a cyclic fashion from 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.

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.

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 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`

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 in a world that was weary of war and destruction and desperately needed something more interesting and productive to do. 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.

The way of a man with array

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

Jack, an outwardly respectable Victorian gentleman who is secretly a malignant narcissist, knows some ladies numbered 0, ..., $n - 1$. The social hierarchy of these women 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 validation of the lady number 0 in his harem.

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 remove any one of the ladies from his harem. Thanks to Jack's well honed skills of psychological manipulation, any lady who has been thus removed can later be successfully courted to rejoin the harem under the same condition of all her immediate predecessors being in Jack's harem at that moment, which makes her powerless to resist Jack's incessant hoovering and love bombing 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. The total number of moves required to reach this goal 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 total number of ladies simultaneously in his harem at any given time. 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 must surely know, the direct approach is rarely optimal.

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