Third Collection of Python Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of November 2, 2024

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 cute problem comes from the post "Double (or triple) and sort" in Michael Lugo's blog "God Plays Dice". Starting from the given positive integer `n`, multiply the current number by the given `multiplier`, and sort the resulting digits in ascending order to get the next number in the sequence. Note that this operation can make the number smaller, since all zero digits will effectively disappear from the front. For example, given `n` equal to 513 and `multiplier` equal to 2, the product 1026 with its digits sorted becomes 0126, which equals 126. Rinse and repeat until you come to an integer that you have seen before, and return the largest integer that you encountered along the way in this sequence.

`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 terminates 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 so that `friends[i]` gives the list of friends of the person *i*. Friendship or its absence is always symmetric between any two people. This problem investigates the famous graph theoretical phenomenon known as the friendship paradox; on average, an individual's friends have more friends than that individual. As explained on the linked Wikipedia page, this phenomenon has a solid mathematical basis in that more popular people will appear as someone's friends more often than unpopular people, which then distorts the average number of friends that your friends have compared to the size of your own local friendship neighbourhood. Same as how the grass is always greener on the other side of the fence, your friends usually seem to be more successful than you are.

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 first quantity is the average number of friends per person, which should be easy enough to compute. The second quantity is the average number of friends among the friends per the average person. Both of these quantities should, of course, be computed and returned as exact `Fraction` objects.

`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 course 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. In fact, this decision problem turns out to be PSPACE-complete, making even the notorious NP-complete decision problems seem like a walk in the park in comparison. This is actually the only PSPACE-complete problem to appear anywhere in these three collections of Python problems!

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'`

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 they say.

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 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 a 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]`

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 system every day in expressing time with seconds and minutes both given in base 60, whereas hours are expressed in 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]`

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 as a function of r. Of course, in this course we can use this classic as an exercise of writing loops to solve the problem efficiently 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 $2x + 1$ 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 that are directly visible from the origin without any other points standing directly in between.

Maximal palindromic integer

```
def maximal_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 the question. 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.

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

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 makes 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 in 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.