Practice 3

COMP9021, Term 3, 2019

1 Finding particular sequences of prime numbers

Write a program consecutive_primes.py that finds all sequences of 6 consecutive prime 5-digit numbers, say (a, b, c, d, e, f), with $b = a + 2$, $c = b + 4$, $d = c + 6$, $e = d + 8$, and $f = e + 10$. So a, b, c, d, e and f are all 5-digit prime numbers and no number between a and b, between b and c, between c and d, between d and e, and between e and f is prime.

The expected output is:

```
The solutions are:

    13901   13903   13907   13913   13921   13931
    21557   21559   21563   21569   21577   21587
    28277   28279   28283   28289   28297   28307
    55661   55663   55667   55673   55681   55691
    68897   68899   68903   68909   68917   68927
```

2 Finding particular sequences of triples

Write a program triples_1.py that finds all triples of positive integers (i, j, k) such that i, j and k are two digit numbers, no digit occurs more than once in i, j and k, and the set of digits that occur in i, j or k is equal to the set of digits that occur in the product of i, j and k.

The expected output is:

```
20 x 79 x 81 = 127980
21 x 76 x 80 = 127680
28 x 71 x 90 = 178920
31 x 60 x 74 = 137640
40 x 72 x 86 = 247680
46 x 72 x 89 = 294768
49 x 50 x 81 = 198450
56 x 87 x 94 = 457968
```

3 Finding special triples of the form $(n, n+1, n+2)$

Write a program triples_2.py that finds all triples of consecutive three-digit integers each of which is the sum of two squares, that is, all triples of the form $(n, n+1, n+2)$ such that:

- n, $n+1$ and $n+2$ are integers at least equal to 100 and at most equal to 999;
- each of n, $n+1$ and $n+2$ is of the form $a^2 + b^2$.

Hint: As we are not constrained by memory space for this problem, we might use a list that stores an integer for all indexes n in $[100, 999]$, equal to 1 in case n is the sum of two squares, and to 0 otherwise. Then it is just a matter of finding three consecutive 1's in the list. This idea can be refined (by not storing 1s, but suitable nonzero values) to not only know that some number is of the form $a^2 + b^2$, but also know such a pair $(a, b)\ldots$

If an integer n is of the form $a^2 + b^2$, then the decomposition is not necessarily unique. We want each decomposition that is output to be the minimal one w.r.t. the natural ordering of pairs of integers (that is, $(a, b) < (a', b')$ iff either $a < a'$ or $a = a'$ and $b < b'$).

The expected output is:

```
(144, 145, 146) (equal to (0^2+12^2, 1^2+12^2, 5^2+11^2)) is a solution.
(232, 233, 234) (equal to (6^2+14^2, 8^2+13^2, 3^2+15^2)) is a solution.
(288, 289, 290) (equal to (12^2+12^2, 0^2+17^2, 1^2+17^2)) is a solution.
(360, 361, 362) (equal to (6^2+18^2, 0^2+19^2, 1^2+19^2)) is a solution.
(520, 521, 522) (equal to (6^2+22^2, 11^2+20^2, 9^2+21^2)) is a solution.
(576, 577, 578) (equal to (0^2+24^2, 1^2+24^2, 7^2+23^2)) is a solution.
(584, 585, 586) (equal to (10^2+22^2, 3^2+24^2, 15^2+19^2)) is a solution.
(800, 801, 802) (equal to (4^2+28^2, 15^2+24^2, 19^2+21^2)) is a solution.
(808, 809, 810) (equal to (18^2+22^2, 5^2+28^2, 9^2+27^2)) is a solution.
```

4 Miller Rabin primality test (optional)

Let p be a strictly positive natural number.

4.1 Fermat's little theorem and Pingala's algorithm

Let a be a strictly positive natural numbers with $a < p$. By Fermat's little theorem, if p is prime then $a^{p-1} \bmod p = 1$; so if $a^{p-1} \bmod p \neq 1$ then a "witnesses" that p is not prime. Pingala's algorithm offers a fast computation of a^{p-1} modulo p. We illustrate it with $p = 101$ and $a = 3$. In base 2, 100 reads as 1100100. So 100 is equal to

$$(((((1 \times 2 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 0$$

(starting with the leftmost 1 in 1100100, "inserting to the right" each of the remaining bits in 1100100, from left to right, requires to "shift" the previous bits to the left, corresponding to a multiplication by 2, and adding that bit to the result), that is,

$$(((((2 + 1) \times 2) \times 2) \times 2 + 1) \times 2) \times 2. \tag{1}$$

Hence 3^{100} is equal to

$$(((((3^2 \times 3)^2)^2)^2 \times 3)^2)^2$$

(starting with 3, the initial 2 and a multiplication by 2 in (**??**) necessitate to raise to the power 2, and an addition of 1 in (**??**) necessitates to multiply by 3). Hence $3^{100} \bmod 101$ is equal to

$$(((((3^2 \bmod 101 \times 3 \bmod 101)^2 \bmod 101)^2 \bmod 101)^2 \bmod 101 \times 3 \bmod 101)^2 \bmod 101)^2 \bmod 101,$$

which evaluates to 1. So using 3 as a possible witness, the conclusion is that 101 might be prime, which is the case indeed.

For another example, take $p = 91$, and a equal to either 3 or 5. In base 2, 90 reads as 1011010. Hence $3^{90} \bmod 91$ is equal to

$$(((((3^2 \bmod 91)^2 \bmod 91 \times 3 \bmod 91)^2 \bmod 91 \times 3 \bmod 91)^2 \bmod 91)^2 \bmod 91 \times 3 \bmod 91)^2 \bmod 91,$$

which evaluates to 1. So using 3 as a possible witness, the conclusion is that 91 might be prime, which is wrong. On the other hand, $5^{90} \bmod 91$ is equal to

$$(((((5^2 \bmod 91)^2 \bmod 91 \times 5 \bmod 91)^2 \bmod 91 \times 5 \bmod 91)^2 \bmod 91)^2 \bmod 91 \times 5 \bmod 91)^2 \bmod 91,$$

which evaluates to 64. So using 5 as a possible witness, the conclusion is that 91 is definitely not prime.

4.2 ROO (Roots One One) property of primes

ROO states: if p is prime then for all natural numbers x, if $x^2 \bmod p = 1$ then $x \bmod p = \pm 1$. Indeed, if $x^2 \bmod p = 1$ then $x^2 - 1 \bmod p = 0$, that is, $(x-1)(x+1) \bmod p = 0$. If p is prime, this implies that $x - 1 \bmod p = 0$ or $x + 1 \bmod p = 0$, hence $x \bmod p = \pm 1$.

By the above, $3^{90} \bmod 91$, which recall evaluates to $1 \bmod 91$, is equal to $x^2 \bmod 91$ with x equal to

$$((((3^2 \bmod 91)^2 \bmod 91 \times 3 \bmod 91)^2 \bmod 91 \times 3 \bmod 91)^2 \bmod 91)^2 \bmod 91 \times 3 \bmod 91,$$

which evaluates to 27. The conclusion is that, making additional use of ROO, 3 can after all witness that 91 is not prime.

More generally, say that a number a in $\{2, \dots p-1\}$ *strongly witnesses* that p is not prime if

- either $a^{p-1} \bmod p \neq 1$,
- or $a^{p-1} \bmod p = 1$ but the computation of $a^{p-1} \bmod p = 1$ by Pingala's algorithm ends in raising some number x_1 to the power 2 modulo p (because the binary representation of $p-1$ ends in 0, which is obviously the case for those considerations to be of interest, as even numbers are directly known not to be prime) and $x_1^{p-1} \bmod p \neq \pm 1$,
- or $x_1^{p-1} \bmod p = 1$ but the computation of $x_1^{p-1} \bmod p = 1$ by Pingala's algorithm ends in raising some number x_2 to the power 2 modulo p (because the second rightmost bit in the binary representation of $p-1$ ends in 0) and $x_2^{p-1} \bmod p \neq \pm 1$,
- $\dots$

4.3 Questions

Write a Python program `miller_rabin_primality_test.py` that implements 3 functions:

- A function `claims_is_not_prime()` that takes two integers a and p with $1 < a < p$ as arguments, and returns `True` if a is found out not to strongly witness that p is prime (in which case p is necessarily not prime), and `False` otherwise (in which case p is likely to be prime but might still not be). Using the `doctest` module to test `claims_is_not_prime()`, the following behaviour would then be observed:

```
>>> claims_is_not_prime(2, 41041)
True
>>> claims_is_not_prime(3, 667)
True
>>> claims_is_not_prime(2, 991)   # Prime indeed
False
>>> claims_is_not_prime(3, 61609) # Prime indeed
False
>>> claims_is_not_prime(2, 2047)  # Actually not prime
False
>>> claims_is_not_prime(3, 121)   # Actually not prime
False
>>> claims_is_not_prime(5, 781)   # Actually not prime
False
>>> claims_is_not_prime(7, 25)    # Actually not prime
False
```

- A function `miller_rabin_primality_test()` that takes a sequence of integers greater than 1 as first argument and a larger integer as second argument, and returns `True` if all elements of the former and found out to strongly witness that the latter is prime, and `False` otherwise. Using the `doctest` module to test `claims_is_not_prime()`, the following behaviour would then be observed:

  ```
  >>> miller_rabin_primality_test([8, 13, 15], 103565)
  False
  >>> miller_rabin_primality_test([20, 21], 31327)     # Prime indeed
  True
  >>> miller_rabin_primality_test([20, 25, 30], 42127) # Actually not prime
  True
  ```

- A function `smallest_miller_rabin_primality_test_failure()` that takes a sequence S of integers greater than 1 as first argument and an integer as second argument, and returns the smallest integer p greater than all members of S and at most equal to n such that all members of S strongly witness that p is prime whereas p is not prime, in case there is indeed such a number p (otherwise, the function returns `None`).

  ```
  >>> smallest_miller_rabin_primality_test_failure([2, 3], 10_000_000)
  1373653
  >>> smallest_miller_rabin_primality_test_failure([2, 3, 5], 30_000_000)
  25326001
  ```

  The previous two outputs testify the quality of the Miller Rabin primality test. If it was efficient enough, `smallest_miller_rabin_primality_test_failure()`, provided with the sequence `[2, 3, 5, 7, 11]` as first argument, would reveal that the test gives a correct answer for numbers up to 2 trillions.

  It can be proved that when p is a large number, randomly choosing k numbers and applying the Miller Rabin primality test with those k numbers incorrectly returns `True` with a probability of $\frac{1}{4^k}$, which very quickly becomes extremely small.

5 Dice rolls (optional, needs a module not installed on CSE computers)

Write a program dice_rolls.py that prompts the user twice, for strictly positive integers s_1, ..., s_k intended to represent the number of sides of some dice, and for an integer N meant to represent the number of times these dice should be cast. If the first input is empty, then a single six-sided die will be used. If the first input is not empty, then any part of it which is not a strictly positive integer will be replaced by 6 (so for instance, inputting 12 0 3 -1 python 4 5A is equivalent to inputting 12 6 3 6 6 4 6). If the second input is empty or is not a strictly positive integer, then the number of rolls will be set to 1,000.

Here are possible interactions:

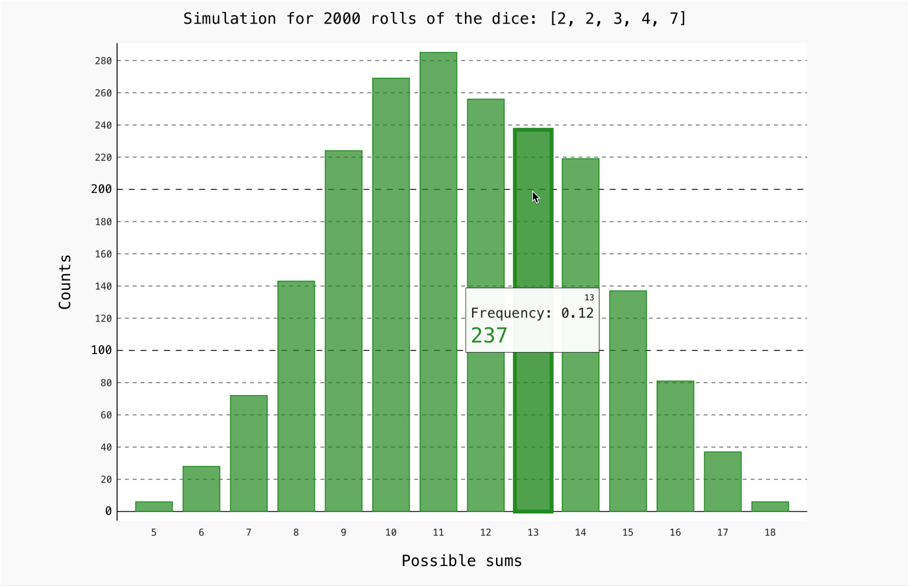
```
$ python3 dice_rolls.py
Enter N strictly positive integers (number of sides of N dice):
You did not enter any value, a single standard six-sided die will be rolled.

Enter the desired number of rolls:
Input was not provided or invalid, so the default value of 1,000 will be used.
$ python3 dice_rolls.py
Enter N strictly positive integers (number of sides of N dice): 2 0 3 python
Some of the values, incorrect, have been replaced with the default value of 6.

Enter the desired number of rolls: 0
Input was not provided or invalid, so the default value of 1,000 will be used.
$ python3 dice_rolls.py
Enter N strictly positive integers (number of sides of N dice): 2 4 2 7 3

Enter the desired number of rolls: 2000
```

The program should generate N times k random numbers between 1 and s_1, ..., s_k, respectively, sum them up, and display the N sums in the form of a histogram, created as an object of class Bar of the pygal module, that can be displayed in a browser by opening a file named dice_rolls.svg—check out render_to_file(). To create the histogram from the N sums, check out add(). The histogram should have—check out the Style class from the pygal.style module:

- as title for the histogram, Simulation for N rolls of the dice: L where L is the ordered list of the number of sides of the dice;
- as labels on the x-axis, all possible sums;
- as title for the x-axis, Possible sums;
- the major labels of the y-axis having a font size of 12 pt;
- as title for the y-axis, Counts;

- tooltips displaying, besides the count, `Frequency:` *f* where *f* is the count divided by *N*, displayed with 2 digits after the decimal point;
- bars having the colour whose rgb code is `#228B22`;
- no legend.

Here is one possible such histogram obtained with `2 4 2 7 3` as first input and `2000` as second input, with the cursor hovering over the bar for the sum of 13.


Simulation for 2000 rolls of the dice: [2, 2, 3, 4, 7]
Counts
Frequency: 0.12
Possible sums