Selected Project Euler Exercises

Oleg Sivokon

<2015-10-15 Thu>

Contents

1 Foreword

2 Exercises

2.1 1000-digits Fibonacci number PE-25

2.1.1 Problem Statement

The Fibonacci sequence is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},\ \textit{where}\ F_1 = 1\ \textit{and}\ F_2 = 1\ .$$

Hence the first 12 terms will be:

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_3 &= 2 \\ F_4 &= 3 \\ F_5 &= 5 \\ F_6 &= 8 \\ F_7 &= 13 \\ F_8 &= 21 \\ F_9 &= 34 \\ F_{10} &= 55 \\ F_{11} &= 89 \\ F_{12} &= 144 \end{aligned}$$

The 12^{th} term, F_{12}, is the first term to contain three digits.

What is the index of the first term in the Fibonacci sequence to contain 1000 digits?

2.1.2 Discussion

Start with some initial guess: $fib(1000)$ and double the size until we find a number with more than 1000 digits once done, subtract the half of the last increment, depending on whether we overshoot or undershoot subtract or add the quarter of the last increment and so on, until we find two adjacent Fibonacci numbers such that the first has at most 999 digits and the second has 1000 or more digits. This will ensure that we calculate at most $log(1000) \approx 10$ operations. Afterwards, because there could be several Fibonacci numbers of the same length, we'll find the smallest one using linear search (this won't take more than $log(2^4) = 4$ operations.)

2.1.3 Solution

```
from combinatorics import fib, digits

n = 1000
guess = fib(n)
while len(digits(guess)) < 1000:
    n *= 2
    guess = fib(n)
m = n // 4
n -= m
while m > 1:
    guess = fib(n)
    m //= 2
    if len(digits(guess)) > 1000:
        n -= m
    else:
        n += m
while len(digits(guess)) >= 1000:
    n -= 1
    guess = fib(n)
return n + 1
```

```
4782
```

2.2 Goldbach's other conjecture PE-46

2.2.1 Problem statement

It was proposed by Christian Goldbach that every odd composite number can be written as the sum of a prime and twice a square.

$$\begin{aligned}
9 &= 7 + 2 \times 12\\
15 &= 7 + 2 \times 22\\
21 &= 3 + 2 \times 32\\
25 &= 7 + 2 \times 32\\
27 &= 19 + 2 \times 22\\
33 &= 31 + 2 \times 12
\end{aligned}$$

It turns out that the conjecture was false.

What is the smallest odd composite that cannot be written as the sum of a prime and twice a square?

2.2.2 Discussion

There doesn't seem to be an easier way than just a brute-force search. In other words, the code simply generates subsequent odd composite numbers and tests whether the number complies with the contstraints of conjecture.

2.2.3 Solution

```python
import math
from primes import odd_composite, primes

def decomposes_goldbach(n):
    """
    Returns True iff n can be decomposed into p + 2 * i^2, where
    p is a prime and i is an integer.
    """
    for p in primes():
        if p > n:
            return False
        rest = n - p
        if rest % 2 == 0 and int(math.sqrt(rest // 2)) ** 2 == rest // 2:
            return True

for c in odd_composite():
    if not decomposes_goldbach(c):
        return c
```

```
5777
```

2.3 Largest sum of consequtive primes PE-50

2.3.1 Problem statement

The prime 41, can be written as the sum of six consecutive primes:

$$41 = 2 + 3 + 5 + 7 + 11 + 13$$

This is the longest sum of consecutive primes that adds to a prime below one-hundred.

The longest sum of consecutive primes below one-thousand that adds to a prime, contains 21 terms, and is equal to 953.

Which prime, below one-million, can be written as the sum of the most consecutive primes?

2.3.2 Discussion

In the preparation phase the code generates all primes which add up to less than a million, while at the same time calculating sums of the primes generated so far. This acts as a cache for the sums we would have to calculate repeatedly whenever we'd verify every other sequence of primes.

During the next step the code choses sums in order of decreasing number of summands, iteratively updating the maximum prime for the longest sequence found so far. Once no sums of the longest-so-far sequence can be found, the lagorithm terminates.

2.3.3 Solution

```python
from primes import primes, is_prime

sieve, prefixes, max_prime, longest_seq = [], [0], 0, 0
for p in primes():
    if prefixes[-1] + p > 1000000:
        break
    sieve.append(p)
    prefixes.append(prefixes[-1] + p)
terms = 1
for i in range(len(prefixes)):
    for j in range(i + terms, len(prefixes)):
        n = prefixes[j] - prefixes[i]
        if j - i > terms:
            is_p, sieve = is_prime(n, sieve)
            if is_p:
                terms, max_prime = j - i, n
return max_prime
```

```
997651
```

2.4 Poker game PE-54

2.4.1 Problem statement

In the card game poker, a hand consists of five cards and are ranked, from lowest to highest, in the following way:

High Card Highest value card.

One Pair Two cards of the same value.

Two Pairs Two different pairs.

Three of a Kind Three cards of the same value.

Straight All cards are consecutive values.

Flush All cards of the same suit.

Full House Three of a kind and a pair.

Four of a Kind Four cards of the same value.

Straight Flush All cards are consecutive values of same suit.

Royal Flush Ten, Jack, Queen, King, Ace, in same suit.

The cards are valued in the order:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace.$$

If two players have the same ranked hands then the rank made up of the highest value wins; for example, a pair of eights beats a pair of fives (see example 1 below). But if two ranks tie, for example, both players have a pair of queens, then highest cards in each hand are compared (see example 4 below); if the highest cards tie then the next highest cards are compared, and so on.

Consider the following five hands dealt to two players:

Player 1	Player 2	Winner
5H 5C 6S 7S KD Pair of Fives	2C 3S 8S 8D TD Pair of Eights	Player 2
5D 8C 9S JS AC Highest card Ace	2C 5C 7D 8S QH Highest card Queen	Player 1
2D 9C AS AH AC Three Aces	3D 6D 7D TD QD Flush with Diamonds	Player 2
4D 6S 9H QH QC Pair of Queens Highest card Nine	3D 6D 7H QD QS Pair of Queens Highest card Seven	Player 1
2H 2D 4C 4D 4S Full House With Three Fours	3C 3D 3S 9S 9D Full House with Three Threes	Player 1

The file, poker.txt, contains one-thousand random hands dealt to two players. Each line of the file contains ten cards (separated by a single space): the first five are Player 1's cards and the last five are Player 2's cards. You can assume that all hands are valid (no invalid characters or repeated cards), each player's hand is in no specific order, and in each hand there is a clear winner.

How many hands does Player 1 win?

2.4.2 Discussion

Player's hand will be represented by a class `Hand`. This class is initalized with the raw card data. During initialization the cards are sorted in order from least to highest denomination. `Hand` class will aslo implement operators required for comparison. Once such operator is called, the code will determine (and cache):

1. The value of the hand. Hands are given values using the following scheme:
    - If no special combination is found (s.a. *flush*), then the hand is worth as much as its highest denomination card.
    - Special denominations recieve one point more than the highest card, plus their relative rank, i.e. *one pair* receives 15, *two pairs* recieves 16 and so on.
    - Ties aren't broken at this time.
2. The stretches of cards of the same denomination.
3. If the code encounters a tie, it tries to break it using the rules given below.
    - Given the `stretches` information the code will select only the cards that will influence the decision.
    - Compare selected cards in order from highest to lowest denomination.

The `cards_txt` file can be found in `../etc/p054_poker.txt`.

2.4.3 Solution

```
from cards import Hand

with open(cards_txt, 'r') as f:
    wins = 0
    for line in f:
        cards = line.strip().split(" ")
        if Hand(cards[:5]) > Hand(cards[5:]):
            wins += 1
    return wins
```

376

3 Appendix

3.1 Namespace Documentation

4 cards Namespace Reference

Classes

- class Hand

Functions

- def cards_cmp (a, b)
- def denom (card)

4.1 Function Documentation

4.1.1 def cards.cards_cmp (*a*, *b*)

4.1.2 def cards.denom (*card*)

5 combinatorics Namespace Reference

Functions

- def fib (n)
- def digits (n)
- def nub (nums)
- def has_square (n)

5.1 Function Documentation

5.1.1 def combinatorics.digits (*n*)

```
The digits of the argument, not counting the sign.
```

5.1.2 def combinatorics.fib (*n*)

```
It is possible to compute N'th Fibonacci number analytically,
but I was asked not to do this, and to keep it simple. So, here
you have the iterative version.
```

5.1.3 def combinatorics.has_square (*n*)

```
Returns True iff n contains a perfect square as a factor.
```

5.1.4 def combinatorics.nub (*nums*)

```
Returns a copy of nums with duplicates removed.  The numbers
in nums will be solrted in increasing order.
```

6 primes Namespace Reference

Functions

- def primes_seq (n)
- def is_next_prime (n, sieve)
- def is_prime (n, sieve)
- def next_prime (sieve)
- def primes
- def odd_composite ()
- def factors (n)

6.1 Function Documentation

6.1.1 def primes.factors (*n*)

```
Generates all prime factors of n.
```

6.1.2 def primes.is_next_prime (*n, sieve*)

```
Tests whether n is prime.  Sieve should contain all primes less than n.
```

6.1.3 def primes.is_prime (*n, sieve*)

6.1.4 def primes.next_prime (*sieve*)

```
Given the sieve containing primes searches for the prime greater than the
last element of the sieve, such that there are no primes less than this
one which are not in the sieve.
```

6.1.5 def primes.odd_composite ( )

```
Iterator.  Generates odd composite numbers.
```

6.1.6 def primes.primes (*n = None*)

```
Iterator.  Generates primes.
```

6.1.7 def primes.primes_seq (*n*)

```
Generates a sequence of primes up to n.
```

6.2 Class Documentation

7 cards.Hand Class Reference

Inheritance diagram for cards.Hand:

Public Member Functions

- def __init__ (self, cards)
- def n_of_a_kind (self, n)
- def royal_flush (self)
- def straight_flush (self)
- def four_of_a_kind (self)
- def full_house (self)
- def flush (self)
- def straight (self)
- def three_of_a_kind (self)
- def two_pairs (self)
- def one_pair (self)
- def value (self)
- def scoring_cards (self, score)
- def break_tie (self, other)
- def __lt__ (self, other)
- def __le__ (self, other)
- def __eq__ (self, other)
- def __ne__ (self, other)
- def __gt__ (self, other)
- def __ge__ (self, other)

7.1 Constructor & Destructor Documentation

7.1.1 def cards.Hand.__init__ (*self*, *cards*)

7.2 Member Function Documentation

7.2.1 def cards.Hand.__eq__ (*self*, *other*)

7.2.2 def cards.Hand.__ge__ (*self*, *other*)

7.2.3 def cards.Hand.__gt__ (*self*, *other*)

7.2.4 def cards.Hand.__le__ (*self*, *other*)

7.2.5 def cards.Hand.__lt__ (*self*, *other*)

7.2.6 def cards.Hand.__ne__ (*self*, *other*)

7.2.7 def cards.Hand.break_tie (*self*, *other*)

7.2.8 def cards.Hand.flush (*self*)

7.2.9 def cards.Hand.four_of_a_kind (*self*)

7.2.10 def cards.Hand.full_house (*self*)

7.2.11 def cards.Hand.n_of_a_kind (*self*, *n*)

7.2.12 def cards.Hand.one_pair (*self*)

7.2.13 def cards.Hand.royal_flush (*self*)

7.2.14 def cards.Hand.scoring_cards (*self*, *score*)

7.2.15 def cards.Hand.straight (*self*)

7.2.16 def cards.Hand.straight_flush (*self*)

7.2.17 def cards.Hand.three_of_a_kind (*self*)

- cards.py

7.3 File Documentation

8 cards.py File Reference

Classes

- class cards.Hand

Namespaces

- cards

Functions

- def cards.cards_cmp (a, b)
- def cards.denom (card)

9 combinatorics.py File Reference

Namespaces

- combinatorics

Functions

- def combinatorics.fib (n)
- def combinatorics.digits (n)
- def combinatorics.nub (nums)
- def combinatorics.has_square (n)

10 primes.py File Reference

Namespaces

- primes

Functions

- def primes.primes_seq (n)
- def primes.is_next_prime (n, sieve)
- def primes.is_prime (n, sieve)
- def primes.next_prime (sieve)
- def primes.primes
- def primes.odd_composite ()
- def primes.factors (n)