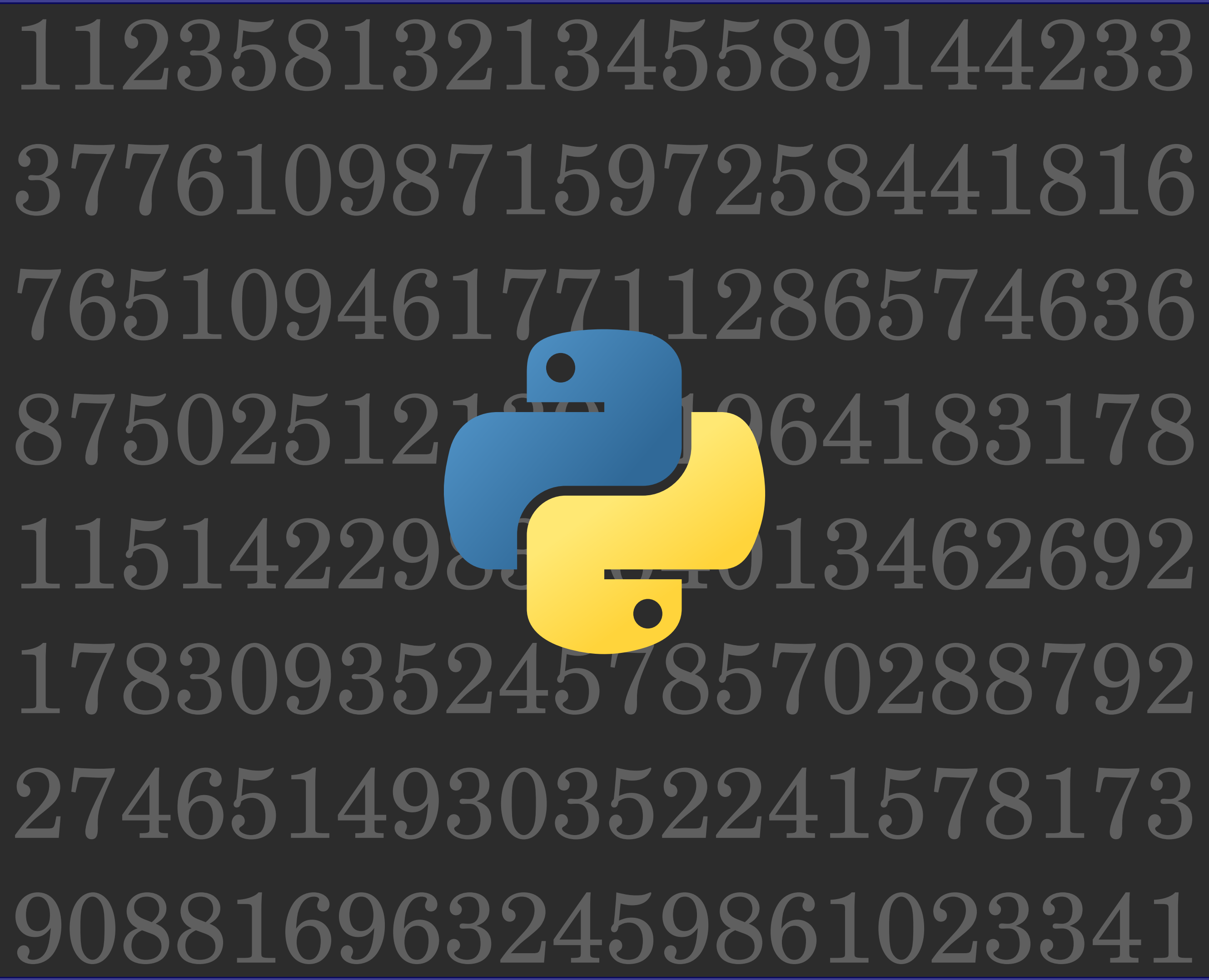
PROJECT EULER

The First 100 Problems Solved with Python



MIGUEL CASAÑAS

Introduction

This book contains solutions to the first 100 problems from Project Euler, each written in Python and accompanied by a brief explanation of the approach. Project Euler is a series of challenging mathematical and computational problems that require more than just mathematical insight to solve. This book is designed to support learners and enthusiasts in understanding efficient programming techniques and mathematical reasoning through problem-solving.

How to Use This Book

The purpose of this book is to aid in your learning journey, and we encourage you to approach each problem with a mindset for growth. Here's how to best utilize this book:

▷ **Attempt Each Problem First:** For each problem, try to solve it independently before consulting the solutions provided. The problems are meant to challenge and improve your problem-solving skills.

▷ **Use the Explanations to Enhance Understanding:** Each solution gives a brief explanation, highlighting key concepts and strategies. Use these explanations to understand the thought process behind each solution.

▷ **Review Python Code Structure:** Observe how each solution is structured in Python. Coding solutions cleanly and efficiently is a valuable skill, and most problems provide an example of good programming practices.

Contact Information

For questions, comments, or feedback, feel free to reach out:

▷ **Telegram:** Connect with me on Telegram at `@miguelcasanas`.

▷ **LinkedIn:** Find me on LinkedIn: https://www.linkedin.com/in/miguelcasanas

▷ **Support:** If you find this book helpful and wish to support future projects, donations are appreciated.

Note on Project Euler's Permissions

Project Euler has provided guidance on sharing solutions for its problems. Here's an excerpt from their policy:

> *"I learned so much solving problem XXX, so is it okay to publish my solution elsewhere?*
>
> *It appears that you have answered your own question. There is nothing quite like that 'Aha!' moment when you finally beat a problem which you have been working on for some time. It is often through the best of intentions in wishing to share our insights so that others can enjoy that moment too. Sadly, that will rarely be the case for your readers. Real learning is an active process and seeing how it is done is a long way from experiencing that epiphany of discovery. Please do not deny others what you have so richly valued yourself.*
>
> *However,* ***the rule about sharing solutions outside of Project Euler does not apply to the first one-hundred problems****, as long as any discussion clearly aims to instruct methods, not just provide answers, and does not directly threaten to undermine the enjoyment of solving later problems. Problems 1 to 100 provide a wealth of helpful introductory teaching material and if you are able to respect our requirements, then we give permission for those problems and their solutions to be discussed elsewhere."*

Joining Project Euler

If you have not yet joined Project Euler, consider signing up and contributing to this vibrant community of problem solvers. You can add me to your list of friends using this key:

```
2152595_2ijDAbuxbUQikEeHElW44nRzIqNbLSnn
```

List of Problems

1 Multiples of 3 or 5

Problem [Project Euler]

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get $3, 5, 6$, and 9. The sum of these multiples is 23.
Find the sum of all the multiples of 3 or 5 below 1000.

Solution [Buy me a beer]

In this problem, we're asked to find the sum of all natural numbers below 1000 that are divisible by 3 or 5. While a straightforward solution might involve iterating through each number less than 1000 and summing those divisible by 3 or 5, this approach becomes inefficient for larger limits, like 10^9. Instead, we use a more mathematical approach, leveraging the properties of arithmetic sequences to calculate the sum efficiently.
First, we recognize that the numbers divisible by 3 below 1000 form an arithmetic sequence: $3, 6, 9, \ldots$, where each term increases by 3. Similarly, the numbers divisible by 5 are $5, 10, 15, \ldots$. We want to find the sum of all terms in these sequences below 1000 without explicitly iterating through each multiple. For this, we turn to a formula for the sum of an arithmetic sequence, which Euclid's division lemma helps us express compactly.
According to Euclid's lemma, the largest multiple of a number k below a limit n is kq, where $q = \left\lfloor \frac{n-1}{k} \right\rfloor$. The sum of all multiples of k below n can then be written as $S_k = kT(q)$, where $T(q)$ represents the q-th triangular number, defined as $T(q) = \frac{q(q+1)}{2}$. This triangular number formula allows us to sum the sequence without needing to add each individual term, as it gives the cumulative sum directly.
Using this approach, we calculate $S(3)$, the sum of all multiples of 3, and $S(5)$, the sum of all multiples of 5. However, numbers that are multiples of both 3 and 5 (i.e., multiples of 15) are included twice in $S(3) + S(5)$. To correct this, we subtract $S(15)$, the sum of all multiples of 15 below 1000. This adjustment is an application of the inclusion-exclusion principle, which ensures each number is counted only once.
Thus, the final result is computed as $S(3) + S(5) - S(15)$, an efficient and elegant solution that avoids unnecessary iteration and scales well for much larger limits.

Code [GitHub]

```python
LIMIT = 1000

# condition = lambda x: x % 3 == 0 or x % 5 == 0
# print(sum(filter(condition, range(LIMIT))))

def S(k):
    T = lambda n: n * (n + 1) // 2
    return k * T((LIMIT - 1) // k)

print(S(3) + S(5) - S(15))
```

2 Even Fibonacci Numbers

Problem [Project Euler]

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

Solution [Buy me a beer]

In this problem, we're tasked with finding the sum of even-valued terms in the Fibonacci sequence, up to a maximum term of four million. The Fibonacci sequence itself is defined recursively, with each term being the sum of the two preceding terms, and it starts with $F_0 = 0$ and $F_1 = 1$. This recurrence relation, $F_n = F_{n-1} + F_{n-2}$, generates a sequence of numbers that grows quickly.

Our approach leverages Python's generator function to efficiently produce Fibonacci terms up to the limit of four million. Instead of storing all Fibonacci numbers, we use variables to hold only the last two terms, updating them to calculate the next term in the sequence. This keeps memory usage low and makes the code more efficient.

We're specifically interested in the even terms of the sequence. To achieve this, we filter out the odd terms as we generate each Fibonacci number. By summing the results of this filtered sequence, we obtain the desired total.

Code [GitHub]

```python
def fib(limit):
    a, b = 0, 1
    while (c := a + b) <= limit:
        yield c
        a, b = b, c

is_even = lambda n: n % 2 == 0
print(sum(filter(is_even, fib(4_000_000))))
```

3 Largest Prime Factor

Problem [Project Euler]

The prime factors of 13195 are $5, 7, 13$, and 29.
What is the largest prime factor of the number 600851475143?

Solution [Buy me a beer]

In this problem, we aim to find the largest prime factor of the number 600,851,475,143. The approach we take involves iterating over potential divisors, starting from the smallest prime, 2, and dividing N by each divisor as long as it divides N evenly. Each time we divide, we effectively reduce N by a factor, simplifying our work as we progress.
By continuously dividing N by each divisor k while N is divisible by k, we ensure that we strip away all factors of k from N. When we reach a divisor that no longer divides N, we move on to the next one. This process continues until N becomes 1, indicating that we've fully factored the original number. At this point, the last divisor k we used is the largest prime factor of the original N.
This approach is efficient because it eliminates smaller factors early and progressively reduces N as we find factors, minimizing the number of necessary divisions. By the time we complete the loop, we've found the largest prime factor without needing to check divisors beyond what's necessary. This method is computationally feasible even for large numbers like 600,851,475,143, making it a practical solution for finding the largest prime factor.

Code [GitHub]

```python
N = 600_851_475_143

for k in range(2, N + 1):
    while N % k == 0: N //= k
    if N == 1: break

print(k)
```

4 Largest Palindrome Product

Problem [Project Euler]

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.
Find the largest palindrome made from the product of two 3-digit numbers.

Solution [Buy me a beer]

In this problem, we aim to find the largest palindromic number that can be formed as the product of two 3-digit numbers. A palindromic number reads the same forwards and backwards, which we can check by converting the number to a string and comparing it with its reverse.
Our approach involves iterating through all pairs of 3-digit numbers, starting from 100 up to 999. By setting up two nested loops, we consider every possible product $x \times y$, where x and y are in the range of 3-digit numbers. To avoid redundant calculations, we start the inner loop at $y = x$, ensuring that we don't repeat products we've already checked in previous iterations. For each product, we check if it is palindromic using our helper function `is_palindromic`, which converts the number to a string and compares it to its reverse. If the product is palindromic and larger than any previously found palindrome, we update our `largest` variable to store this new maximum.
By the end of the nested loops, `largest` holds the maximum palindromic product found, which is the solution to the problem.

Code [GitHub]

```python
def is_palindromic(n):
    n = str(n)
    return n == n[::-1]

largest = 0
for x in range(100, 1000):
    for y in range(x, 1000):
        if is_palindromic(product := x * y):
            largest = max(largest, product)

print(largest)
```

5 Smallest Multiple

Problem [Project Euler]

2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.
What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

Solution [Buy me a beer]

In this problem, we need to find the smallest positive number that is evenly divisible by all numbers from 1 to 20. This means we need to calculate the least common multiple (LCM) of the numbers in this range. By definition, the LCM of a set of numbers is the smallest number that each of them divides without a remainder. To compute this efficiently, we use the fundamental theorem of arithmetic, which states that every integer greater than 1 can be uniquely factored into prime numbers.
Our approach is to determine the maximum power of each prime factor needed for any number from 1 to 20. For instance, for divisibility by 8, our final number must include 2^3 since 8 is 2^3. We factorize each integer from 2 to 20 and track the highest power of each prime factor required. This way, we ensure that the resulting product is divisible by every integer in the range.
To accomplish this, we define a helper function `factorint` that factorizes a given integer n using a dictionary-like structure (a `Counter`). This function counts the occurrences of each prime factor in n, and by iterating through

numbers 2 to 20, we build up a cumulative record, `lcm_factors`, of the highest powers of each prime needed.
Once we have the maximum power of each prime factor, we use the `prod` function to calculate the product of each prime raised to its maximum power, yielding the LCM. This product is our solution, as it is the smallest number divisible by all integers from 1 to 20.
While Python's `lcm` function can compute this in a single line, our approach provides insight into the factorization process and the reasoning behind finding an LCM through prime factor powers.

Code [GitHub]

```python
# from math import lcm
# print(lcm(*range(1, 21)))

from collections import Counter
from math import prod

def factorint(n):
    factors = Counter()
    for k in range(2, n + 1):
        while n % k == 0:
            factors[k] += 1
            n //= k
        if n == 1: return factors

lcm_factors = Counter()
for n in range(2, 21):
    for p, m in factorint(n).items():
        lcm_factors[p] = max(lcm_factors[p], m)

print(prod(p**m for p, m in lcm_factors.items()))
```

6 Sum Square Difference

Problem [Project Euler]

The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \cdots + 10^2 = 385.$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \cdots + 10)^2 = 55^2 = 3025.$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.
Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

Solution [Buy me a beer]

In this problem, we're asked to find the difference between the square of the sum and the sum of the squares of the first 100 natural numbers. We can approach this problem efficiently by using mathematical formulas rather than summing each term individually.
The sum of the first n natural numbers is given by the formula for the n-th triangular number:

$$T(n) = \frac{n(n+1)}{2}.$$

We used triangular numbers in Problem 1 to find sums of multiples, and here $T(100)$ gives the sum of the first 100 natural numbers. We then square this value to obtain $T(100)^2$, representing the square of the sum.
To find the sum of the squares of the first n natural numbers, we use the formula for the n-th square pyramidal number:

$$P(n) = \frac{n(n+1)(2n+1)}{6}.$$

In this case, $P(100)$ gives the sum of the squares of the first 100 natural numbers.
With these two quantities, we calculate the difference by subtracting $P(100)$ from $T(100)^2$. This difference gives us the answer to the problem: the discrepancy between the square of the sum and the sum of the squares for the first 100 natural numbers.
This method is efficient, as it leverages formulas for triangular and square pyramidal numbers rather than explicit summation. The solution also illustrates an application of Faulhaber's formula for sums of powers, which generalizes formulas like those for triangular and square pyramidal numbers.

Code [GitHub]

```python
T = lambda n: n * (n + 1) // 2
P = lambda n: T(n) * (2 * n + 1) // 3

print(T(100)**2 - P(100))
```

7 10 001st Prime

Problem [Project Euler]

By listing the first six prime numbers: $2, 3, 5, 7, 11$, and 13, we can see that the 6th prime is 13.
What is the 10 001st prime number?

Solution [Buy me a beer]

In this problem, we're asked to find the 10,001st prime number. The straightforward approach would be to check each number individually for primality until reaching the 10,001st prime, but this would be computationally inefficient. Instead, we use a more efficient algorithm called the Sieve of Eratosthenes.
The Sieve of Eratosthenes is a classical algorithm for finding all prime numbers up to a specified limit. It works by iteratively marking the multiples of each prime starting from 2. Initially, we assume that all numbers are prime, setting up a list where each entry is `True`. We then proceed to mark multiples of each prime as `False` (indicating they are not prime) starting from p^2, because any smaller multiple of p would already have been marked by an earlier prime factor.
For this problem, we set a reasonable limit (e.g., 10^6) and apply the sieve. Once the sieve has processed all numbers up to this limit, we extract the list of primes by selecting all indices that remain `True`. By doing this, we ensure that `primes` contains all prime numbers up to our chosen limit in ascending order.
Finally, to find the 10,001st prime, we simply access the 10,000th index in this list (since indexing starts at zero). While Python's `sympy` library offers a direct `prime` function to retrieve the n-th prime, implementing the sieve ourselves provides deeper insight into the underlying algorithm and its efficiency.

Code [GitHub]

```python
# from sympy import prime
# print(prime(10_001))

LIMIT = 10**6

is_prime = [True] * (LIMIT + 1)
for p in range(2, LIMIT + 1):
    if is_prime[p]:
        for i in range(p**2, LIMIT + 1, p):
            is_prime[i] = False

primes = [p for p in range(2, LIMIT + 1) if is_prime[p]]

print(primes[10_000])
```

8 Largest Product in a Series

Problem [Project Euler]

The four adjacent digits in the 1000-digit number that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$.

```
7316717653133062491922511967442657474235534919493496983520312774506326239
5783180169848018694788518438586156078911294949545950173795833195285320880
5511125406987471585238630507156932909632952274430435576689664895044524452
3161731856403098711121722383113622298934233803081353362766142828064444866
4523874930358907296290491560440772390713810515859307960866701724271218839
9879790879227492190169972088809377665727333001053367881220235421809751254
5405947522435258490771167055601360483958644670632441572215539753697817977
8461740649551492908625693219784686224828397224137565705605749026140797296
8652414535100474821663704844031998900088952434506585412275886668811642717
1479924442928230863465674813919123162824586178664583591245665294765456828
4891288314260769004224219022671055626321111109370544217506941658960408071
9840385096245544436298123098787992724428490918884580156166097919133875499
2005240636899125607176060588611646710940507754100225698315520005593572972
571636269561882670428252483600823257530420752963450
```

Find the thirteen adjacent digits in the 1000-digit number that have the greatest product. What is the value of this product?

Solution [Buy me a beer]

In this problem, we need to find the thirteen adjacent digits within a 1000-digit number that yield the greatest product. Given the length of the number, we aim to find the most efficient way to calculate the product of every possible set of thirteen consecutive digits, comparing each product to identify the maximum.

Our approach involves iterating through each possible position in the 1000-digit number where a sequence of thirteen digits can start. At each position i, we extract a substring of thirteen digits, convert each character in this substring to an integer, and then calculate the product of these integers using Python's `prod` function. This product is then compared with the current maximum stored in `greatest`, updating it whenever a new, larger product is found.

By sliding this 13-digit window one position at a time across the entire number, we ensure that every possible combination of thirteen adjacent digits is considered. Once we've iterated through all potential starting positions, the variable `greatest` holds the largest product found, which is our solution to the problem.

Code [GitHub]

```python
from math import prod

large_number = "NUMBER HERE"

greatest = 0
for i in range(len(large_number) - 12):
    product = prod(map(int, large_number[i:i + 13]))
    greatest = max(greatest, product)

print(greatest)
```

9 Special Pythagorean Triplet

Problem [Project Euler]

A Pythagorean triplet is a set of three natural numbers, $a < b < c$, for which

$$a^2 + b^2 = c^2.$$

For example, $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.
There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find the product abc.

Solution [Buy me a beer]

In this problem, we need to find a Pythagorean triplet (a, b, c) for which $a + b + c = 1000$ and calculate the product abc.
To solve this, we iterate through possible values of a and b, since c can be determined from the condition $a + b + c = 1000$, which simplifies to $c = 1000 - a - b$. This allows us to compute c directly based on any pair (a, b), significantly reducing the search space.
To further reduce the search space, we start b from $a + 1$ (ensuring $a < b$) and break the inner loop as soon as c becomes less than or equal to b. This approach avoids invalid triplets where c would not be the largest value in (a, b, c) and prevents unnecessary calculations.
For each valid triplet (a, b, c), we then check if the Pythagorean condition $a^2 + b^2 = c^2$ holds. When it does, we have found the unique triplet that meets both conditions specified in the problem.
Finally, we calculate and return the product abc, which gives us the desired result.

Code [GitHub]

```python
def find_triplet():
    for a in range(1, 1000):
        for b in range(a + 1, 1000):
            c = 1000 - a - b
            if c <= b: break
            if a**2 + b**2 == c**2:
                return a, b, c

a, b, c = find_triplet()
print(a * b * c)
```

10 Summation of Primes

Problem [Project Euler]

The sum of the primes below 10 is $2 + 3 + 5 + 7 = 17$.
Find the sum of all the primes below two million.

Solution [Buy me a beer]

In this problem, we need to find the sum of all prime numbers below two million. Since the Sieve of Eratosthenes was introduced in Problem 7, we'll reuse it here without much explanation, as it provides an efficient way to generate a list of prime numbers up to a specified limit.
We set the limit to 2×10^6 and use the Sieve of Eratosthenes to mark non-prime numbers in a Boolean array. Starting with the smallest prime, 2, we mark all multiples of each prime p (starting from p^2) as non-prime. This continues until all numbers up to the limit have been processed.
After building the sieve, we generate a list of primes by selecting all indices in the array that remain marked as `True`. Finally, we compute the sum of these primes, which gives us the desired result: the sum of all primes below two million.
Alternatively, Python's `sympy` library offers a one-line solution using `primerange` to generate primes in a given range, but implementing the sieve directly provides a deeper understanding of the process and allows for optimized control over the calculation.

Code [GitHub]

```python
# from sympy import primerange
# print(sum(primerange(2_000_000)))

LIMIT = 2*10**6

is_prime = [True] * (LIMIT + 1)
for p in range(2, LIMIT + 1):
    if is_prime[p]:
        for i in range(p**2, LIMIT + 1, p):
            is_prime[i] = False

primes = [p for p in range(2, LIMIT + 1) if is_prime[p]]
print(sum(primes))
```

11 Largest Product in a Grid

Problem [Project Euler]

In the 20×20 grid below, four numbers along a diagonal line have been marked in red.

```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

The product of these numbers is $26 \times 63 \times 78 \times 14 = 1788696$.
What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20×20 grid?

Solution [Buy me a beer]

In this problem, we aim to find the greatest product of four adjacent numbers in any direction (up, down, left, right, or diagonally) in a given 20×20 grid. This problem is an extension of Problem 8, which involved finding the largest product of consecutive numbers in a single line. Here, we must consider two-dimensional adjacency, which increases the complexity.
To solve this, we first represent the grid as a list of lists, where each inner list corresponds to a row of integers. With this structure, we can access

any element by its row and column indices. We then define a function that systematically generates all possible groups of four adjacent numbers in each relevant direction.
In this function, we iterate through each possible starting position (i, j) in the grid. At each position, we extract groups of four numbers horizontally, vertically, and, when space allows, in both diagonal directions. This approach ensures that we cover all potential sets of four adjacent numbers within the grid, without duplicating calculations.
After generating all groups of four numbers, we calculate the product of each group and identify the maximum product among them. This approach efficiently handles the grid's two-dimensional structure by leveraging the systematic generation of groups and calculating each product only once, yielding the desired result: the largest product of four adjacent numbers in any direction.

Code [GitHub]

```python
from itertools import product
from math import prod

numbers = """GRID HERE"""

grid = [
    list(map(int, row.split()))
    for row in numbers.splitlines()
]

SIZE = len(grid)

def groups():
    for i, j in product(range(SIZE), range(SIZE - 3)):
        yield grid[i][j:j + 4]
        yield [grid[j + k][i] for k in range(4)]
        if i < SIZE - 3:
            yield [grid[i + k][j + k] for k in range(4)]
            yield [grid[i + 3 - k][j + k] for k in range(4)]

print(max(map(prod, groups())))
```

12 Highly Divisible Triangular Number

Problem [Project Euler]

The sequence of triangle numbers is generated by adding the natural numbers. So the 7th triangle number would be $1 + 2 + 3 + 4 + 5 + 6 + 7 = 28$. The first ten terms would be:

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \dots$$

Let us list the factors of the first seven triangle numbers:

$$\begin{aligned} \mathbf{1}&: 1\\ \mathbf{3}&: 1, 3\\ \mathbf{6}&: 1, 2, 3, 6\\ \mathbf{10}&: 1, 2, 5, 10\\ \mathbf{15}&: 1, 3, 5, 15\\ \mathbf{21}&: 1, 3, 7, 21\\ \mathbf{28}&: 1, 2, 4, 7, 14, 28 \end{aligned}$$

We can see that 28 is the first triangle number to have over five divisors. What is the value of the first triangle number to have over five hundred divisors?

Solution [Buy me a beer]

In this problem, we need to find the first triangle number with over 500 divisors. A triangle number is defined as $T_n = \frac{n(n+1)}{2}$, where each term in this sequence is generated by summing consecutive natural numbers. Our solution calculates these triangle numbers one by one until it finds one with the desired divisor count.
To efficiently factorize numbers and count their divisors, we precompute a smallest prime factor array (using a modified sieve) for all numbers up to a limit, 10^5. This array, called `smallest`, stores the smallest prime factor for each integer up to the limit, allowing quick access to prime factors for numbers within this range.
The `factorint` function uses this `smallest` array to factorize a given number n:

- ▷ If n is within the `smallest` array's range, we directly look up the smallest factor from the array, which speeds up factorization.

▷ If n exceeds the limit of `smallest`, we find its smallest factor by iterating through primes up to the limit. This ensures that we can handle numbers larger than the precomputed array range.

With the prime factor counts from `factorint`, the `divisor_count` function calculates the number of divisors of n using the formula $\prod_i(m_i + 1)$, where m_i is the exponent of each prime factor.
We generate triangle numbers using a generator expression, and for each triangle number, we check its divisor count using `divisor_count`. As soon as we find a triangle number with more than 500 divisors, we stop the search and return this number as our result.
For comparison, a one-line solution using Python's `sympy` library could have calculated divisors more directly with `divisor_count`. However, our approach with the precomputed smallest factor array is designed for efficiency and offers greater insight into prime factorization.

Code [GitHub]

```python
# from itertools import count
# from more_itertools import first
# from sympy import divisor_count
# T = (n * (n + 1) // 2 for n in count(1))
# print(first(filter(lambda x: divisor_count(x) > 500, T)))

from collections import Counter
from math import prod
from itertools import count

LIMIT = 10**5

smallest = list(range(LIMIT + 1))
for p in range(2, LIMIT + 1):
    if smallest[p] == p:
        for i in range(p**2, LIMIT + 1, p):
            smallest[i] = p

def factorint(n):
    factors = Counter()
    while n > 1:
        if n > LIMIT:
            for p in range(2, LIMIT + 1):
                if smallest[p] != p: continue
                if n % p == 0: break
        else: p = smallest[n]
        factors[p] += 1
        n //= p
    return factors
```

```python

def divisor_count(n):
    return prod(m + 1 for m in factorint(n).values())

triangle_numbers = (n * (n + 1) // 2 for n in count(1))

for t in triangle_numbers:
    if divisor_count(t) > 500: break

print(t)
```

13 Large Sum

Problem [Project Euler]

Work out the first ten digits of the sum of the following one-hundred 50-digit numbers.

```
37107287533902102798797998220837590246510135740250
46376937677490009712648124896970078050417018260538
74324986199524741059474233309513058123726617309629
91942213363574161572522430563301811072406154908250
23067588207539346171171980310421047513778063246676
89261670696623633820136378418383684178734361726757
28112879812849979408065481931592621691275889832...
```

Rest of the number here.

Solution [Buy me a beer]

In this problem, we are asked to find the first ten digits of the sum of one hundred 50-digit numbers. Given the size of each number, calculating the sum directly may seem challenging, but Python's arbitrary-precision integers make it straightforward to handle large numbers accurately.
We first store the 100 numbers as a multi-line string, then split this string by lines to extract each 50-digit number. Using `map`, we convert each line to an integer, enabling us to add them together with Python's built-in `sum` function.
After calculating the total sum, we convert it to a string and slice out the first ten digits. This approach efficiently retrieves the desired result by focusing only on the leading digits, which is possible because the sum itself

has many more than ten digits.
This solution leverages Python's ability to handle large integers without overflow and avoids unnecessary complexity by working directly with the sum of the numbers.

Code [GitHub]

```python
numbers = """NUMBERS HERE"""

S = sum(map(int, numbers.splitlines()))
print(str(S)[:10])
```

14 Longest Collatz Sequence

Problem [Project Euler]

The following iterative sequence is defined for the set of positive integers:

$$\begin{aligned} n &\to n/2 \quad (n \text{ is even}) \\ n &\to 3n+1 \quad (n \text{ is odd}) \end{aligned}$$

Using the rule above and starting with 13, we generate the following sequence:

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers eventually reach 1.
Which starting number, under one million, produces the longest chain?
NOTE: Once the chain starts, the terms are allowed to exceed one million.

Solution [Buy me a beer]

In this problem, we need to find the starting number under one million that produces the longest Collatz sequence. Each sequence follows the Collatz rule iteratively until it reaches 1, and it is conjectured that all starting numbers eventually reach 1 (the Collatz conjecture).
To solve this problem efficiently, we use memoization to avoid recalculating sequences for numbers we've already encountered, as many terms repeat across different starting values. We define a recursive function,

`chain_length`, which computes the length of the sequence for a given starting number n. By applying Python's `@cache` decorator, we implement memoization, storing previously computed sequence lengths so we can reuse these values instead of recalculating them. This significantly reduces the number of recursive calls, as each term only needs to be computed once.
To find the starting number that produces the longest sequence, we use Python's `max` function with a `key` argument that applies `chain_length` to each number in the range from 1 to one million. This approach efficiently identifies the number under one million that generates the longest Collatz chain.
Using memoization optimizes performance by storing sequence lengths, making the solution computationally feasible even with the large input range.

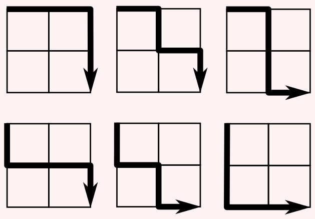
Code [GitHub]

```python
from functools import cache

@cache
def chain_length(n):
    if n == 1: return 1
    next_term = 3*n + 1 if n % 2 else n >> 1
    return chain_length(next_term) + 1

print(max(range(1, 10**6), key=lambda n: chain_length(n)))
```

15 Lattice Paths

Problem [Project Euler]

Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.



How many such routes are there through a 20×20 grid?

Solution [Buy me a beer]

In this problem, we need to find the number of unique routes through a 20×20 grid, moving only right or down, starting from the top-left corner and ending at the bottom-right corner.

Combinatorial Approach

This problem can be solved combinatorially by recognizing that any path through an $n \times n$ grid consists of exactly n right moves and n down moves, for a total of $2n$ moves. To count the distinct paths, we need to determine how many ways we can arrange n right moves and n down moves in a sequence of $2n$ moves.
This is a classic combinatorial problem, where the number of ways to choose n positions for either right or down moves out of $2n$ total positions is given by the binomial coefficient:

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2}$$

For a 20×20 grid, this becomes $\binom{40}{20}$, which gives the total number of unique routes.

Recursive Solution with Memoization

Alternatively, we can approach this problem using recursion with memoization. We define a function `count_paths(i, j)` that calculates the number of paths from the top-left corner $(0,0)$ to a given cell (i,j) in the grid. For each cell (i,j):

- ▷ If we are at the starting cell $(0,0)$, there is only one path.
- ▷ For other cells, we can only arrive from the cell above $(i-1,j)$ or from the cell to the left $(i,j-1)$, so the number of paths to (i,j) is the sum of paths to these two cells.

To optimize, we use Python's `@cache` decorator, which memoizes the results of each cell calculation. This way, each subproblem is computed only once, making the solution efficient.
The combinatorial solution is more direct, but the recursive approach with memoization demonstrates a dynamic programming perspective on the problem, building the solution from subproblems.

Code [GitHub]

```python
# from math import comb
# print(comb(40, 20))

from functools import cache

@cache
def count_paths(i, j):
    if (i, j) == (0, 0): return 1
    paths = count_paths(i - 1, j) if i else 0
    paths += count_paths(i, j - 1) if j else 0
    return paths

print(count_paths(20, 20))
```

16 Power Digit Sum

Problem [Project Euler]

$2^{15} = 32768$ and the sum of its digits is $3 + 2 + 7 + 6 + 8 = 26$.
What is the sum of the digits of the number 2^{1000}?

Solution [Buy me a beer]

Python's built-in arbitrary-precision integers allow us to handle large numbers like 2^{1000} directly. First, we calculate 2^{1000} and convert it to a string to access each digit individually. Then, by mapping each character in the string representation to an integer and summing these values, we obtain the sum of the digits.
This approach leverages Python's ability to seamlessly manage large numbers, providing a straightforward method to compute the digit sum without requiring complex algorithms.

Code [GitHub]

```python
print(sum(map(int, str(2**1000))))
```

17 Number Letter Counts

Problem [Project Euler]

If the numbers 1 to 5 are written out in words: one, two, three, four, five, then there are $3 + 3 + 5 + 4 + 4 = 19$ letters used in total.
If all the numbers from 1 to 1000 (one thousand) inclusive were written out in words, how many letters would be used?
NOTE: Do not count spaces or hyphens. For example, 342 (three hundred and forty-two) contains 23 letters and 115 (one hundred and fifteen) contains 20 letters. The use of 'and' when writing out numbers is in compliance with British usage.

Solution [Buy me a beer]

In this problem, we need to determine the total number of letters used if all numbers from 1 to 1000 are written out in words, following British English conventions. Specifically, we do not count spaces or hyphens, and we include 'and' in numbers such as 'three hundred and forty-two.'
To solve this, we define a dictionary, `words`, which maps individual numbers and key words (e.g., 'hundred' and 'thousand') to their word forms. This allows us to look up the word length for any part of a number.
The function `letters` calculates the number of letters in the word form of a given number n by:

- ▷ Returning the word length directly if n is in `words`.
- ▷ If n is a multiple of 1000 (such as 1000 itself), adding the letters for 'thousand' to the letters in the word representation of $n/1000$.
- ▷ If n is a multiple of 100 (like 300), adding the letters for 'hundred' to the letters in the word representation of $n/100$.
- ▷ If n is over 100 but not a multiple of 100, adding the letters for 'hundred,' the connector 'and' (3 letters), and recursively calculating the letters for the remaining part (e.g., 'forty-two' in 'three hundred and forty-two').
- ▷ Otherwise, breaking down the number by tens and units to find the sum of letters (e.g., calculating 'forty' and 'two' separately for 42).

Finally, we calculate the total by summing the letter counts for each number from 1 to 1000.

Code [GitHub]

```python
words = {
    1: "one", 2: "two", 3: "three", 4: "four", 5: "five", 6:
       ↪ "six", 7: "seven", 8: "eight", 9: "nine",
    11: "eleven", 12: "twelve", 13: "thirteen", 14: "
       ↪fourteen", 15: "fifteen", 16: "sixteen", 17: "
       ↪seventeen", 18: "eighteen", 19: "nineteen",
    10: "ten", 20: "twenty", 30: "thirty", 40: "forty", 50:
       ↪"fifty", 60: "sixty", 70: "seventy", 80: "eighty",
       ↪90: "ninety",
    100: "hundred", 1000: "thousand"
}

def letters(n):
    if n % 1000 == 0:
        return letters(n // 1000) + len(words[1000])
    if n % 100 == 0:
        return letters(n // 100) + len(words[100])
    if n in words: return len(words[n])
    if n > 100: return letters(n // 100) + len(words[100]) + 3 + letters(n % 100)
    return letters((n // 10) * 10) + letters(n % 10)

print(sum(map(letters, range(1, 1001))))
```

18 Maximum Path Sum I

Problem [Project Euler]

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

3
7 4
2 **4** 6
8 5 **9** 3

That is, $3 + 7 + 4 + 9 = 23$.
Find the maximum total from top to bottom of the triangle below:

```
                            75
                          95  64
                        17  47  82
                      18  35  87  10
                    20  04  82  47  65
                  19  01  23  75  03  34
                88  02  77  73  07  63  67
              99  65  04  28  06  16  70  92
            41  41  26  56  83  40  80  70  33
          41  48  72  33  47  32  37  16  94  29
        53  71  44  65  25  43  91  52  97  51  14
      70  11  33  28  77  73  17  78  39  68  17  57
    91  71  52  38  17  14  91  43  58  50  27  29  48
  63  66  04  68  89  53  67  30  73  16  69  87  40  31
04  62  98  27  23  09  70  98  73  93  38  53  60  04  23
```

NOTE: As there are only 16384 routes, it is possible to solve this problem by trying every route. However, Problem 67, is the same challenge with a triangle containing one-hundred rows; it cannot be solved by brute force, and requires a clever method! ;o)

Solution [Buy me a beer]

In this problem, we are asked to find the maximum total from top to bottom in a triangle of numbers by moving to adjacent numbers on each row. While there are 16384 possible routes from the top to the bottom, a recursive approach can help generate each possible path without explicitly calculating each one.
We begin by representing the triangle as a list of lists, where each inner list contains the numbers for a single row. This makes it easy to access any element by its row and column indices.
The function `path_gen` generates all possible paths from the top of the triangle to the bottom by recursively exploring each possible path:

- ▷ Starting at the top (position $i = 0$, $j = 0$), the function appends the current element to the path.
- ▷ If the bottom of the triangle is reached, it yields the completed path.
- ▷ Otherwise, it recursively generates paths by moving to the next row, either staying in the same column or moving to the next column.

Once all paths have been generated, we use `map(sum, ...)` to calculate the sum of each path and then apply `max` to find the path with the highest sum. This approach provides an exhaustive search over all paths while avoiding explicit brute-force enumeration.
This solution is effective for the given triangle size. However, for larger triangles (like in Problem 67), a more efficient approach, such as dynamic programming, would be necessary to handle the exponential growth in possible paths.

Code [GitHub]

```python
triangle = """TRIANGLE HERE"""

T = [
    list(map(int, row.split()))
    for row in triangle.splitlines()
]

def path_gen(i, j, path=[]):
    path = path + [T[i][j]]
    if i == len(T) - 1: yield path
    else:
        yield from path_gen(i + 1, j, path)
        yield from path_gen(i + 1, j + 1, path)

print(max(map(sum, path_gen(0, 0))))
```

19 Counting Sundays

Problem [Project Euler]

You are given the following information, but you may prefer to do some research for yourself.

▷ 1 Jan 1900 was a Monday.

▷ Thirty days has September,
April, June and November.
All the rest have thirty-one,
Saving February alone,
Which has twenty-eight, rain or shine.
And on leap years, twenty-nine.

▷ A leap year occurs on any year evenly divisible by 4, but not on a century unless it is divisible by 400.

How many Sundays fell on the first of the month during the twentieth century (1 Jan 1901 to 31 Dec 2000)?

Solution [Buy me a beer]

We start by defining a function, `is_leap_year(y)`, to check if a given year is a leap year. A year is a leap year if it is divisible by 4, except for years divisible by 100, unless they are also divisible by 400. This function correctly identifies leap years, which is essential for accurately setting February's days.
The function `month_days(year, month)` returns the number of days in a given month for a specified year. February has 28 days in regular years and 29 days in leap years, while other months follow their typical day counts. For months other than February, the number of days alternates between 30 and 31 based on their position in the calendar year.
To count the Sundays that fall on the first of the month, we initialize `day` to 1, representing 1 Jan 1900, which is known to be a Monday. We also initialize `sundays` to zero to store our result. We loop over each year from 1900 to 2000 and, within each year, iterate over each month. For each month, we check if the current `day` is a Sunday (i.e., `day % 7 == 0`) and if the year is within the target range (1901 to 2000). If both conditions are satisfied, we increment the `sundays` counter.
After checking, we advance `day` by the number of days in the current month, which aligns it with the first day of the following month. This method efficiently tracks the days of the week without requiring an iteration through each individual day, making it suitable for counting Sundays over a century-long period.

Code [GitHub]

```python
def is_leap_year(y):
    return (y % 4 == 0) and (y % 100 != 0 or y % 400 == 0)

def month_days(year, month):
    if month == 1: return 29 if is_leap_year(year) else 28
    if month > 6: month += 1
    return 30 if abs(month - 6) % 2 else 31

day, sundays = 1, 0
for year in range(1900, 2001):
    for month in range(12):
        if year >= 1901 and day % 7 == 0: sundays += 1
        day += month_days(year, month)

print(sundays)
```

20 Factorial Digit Sum

Problem [Project Euler]

$n!$ means $n \times (n-1) \times \cdots \times 3 \times 2 \times 1$.
For example, $10! = 10 \times 9 \times \cdots \times 3 \times 2 \times 1 = 3628800$,
and the sum of the digits in the number 10! is $3+6+2+8+8+0+0 = 27$.
Find the sum of the digits in the number 100!.

Solution [Buy me a beer]

To compute 100!, we initialize a list, `factorial`, with a starting value of 1, which represents 0!. We then use a loop to iteratively calculate each successive factorial up to 100! by multiplying the current number n with the last element in the `factorial` list. This iterative approach avoids recalculating smaller factorials each time, making the process efficient.
Once 100! is calculated and stored as the last element in `factorial`, we convert it to a string to access each digit individually. By mapping each character in the string to an integer and then summing these values, we obtain the sum of the digits in 100!.
This problem is indeed similar to Problem 16, where we also needed to compute a large number and sum its digits. Both problems leverage Python's ability to handle arbitrary-precision integers, allowing us to handle large values like 100! without overflow issues.

Code [GitHub]

```python
factorial = [1]
for n in range(1, 101): factorial.append(n * factorial[-1])

print(sum(map(int, str(factorial[-1]))))
```

21 Amicable Numbers

Problem [Project Euler]

Let $d(n)$ be defined as the sum of proper divisors of n (numbers less than n which divide evenly into n).
If $d(a) = b$ and $d(b) = a$, where $a \neq b$, then a and b are an amicable pair and each of a and b are called amicable numbers.
For example, the proper divisors of 220 are $1, 2, 4, 5, 10, 11, 20, 22, 44, 55$ and 110; therefore $d(220) = 284$. The proper divisors of 284 are $1, 2, 4, 71$ and 142; so $d(284) = 220$.
Evaluate the sum of all the amicable numbers under 10000.

Solution [Buy me a beer]

To solve this, we first compute the sum of proper divisors for each number up to a limit higher than 10,000. This limit, set to 25,320 in our solution, ensures that when calculating sums of divisors for values within the target range, we account for any amicable numbers whose divisors might exceed 10,000. We store these sums in a list d, where each index $d[n]$ holds the sum of proper divisors of n.
The initialization of d uses nested loops to populate divisor sums efficiently. For each number n, we iterate over its multiples m (starting from $2n$) and add n to $d[m]$. This approach allows us to accumulate the divisor sums in a single pass through the range, reducing redundant calculations.
To identify amicable numbers, we define the function `is_amicable`, which checks if a given number n forms an amicable pair. The function retrieves the sum of divisors $d[n]$; if $d[n] = n$, the number is not amicable (since $a \neq b$ for amicable pairs). Otherwise, it checks if $d[d[n]] = n$, confirming that n and $d[n]$ form an amicable pair.
Finally, we sum all numbers in the range from 1 to 10,000 that satisfy the `is_amicable` condition.

Code [GitHub]

```python
LIMIT = 25_320

d = [0] * (LIMIT + 1)
for n in range(1, LIMIT + 1):
    for m in range(2*n, LIMIT + 1, n): d[m] += n

def is_amicable(n):
    if (dn := d[n]) == n: return False
    return d[dn] == n

print(sum(filter(is_amicable, range(10_000))))
```

22 Names Scores

Problem [Project Euler]

Using names.txt, a 46K text file containing over five-thousand first names, begin by sorting it into alphabetical order. Then, by working out the alphabetical value for each name, multiply this value by its alphabetical position in the list to obtain a name score.
For example, when the list is sorted into alphabetical order, COLIN, which is worth $3+15+12+9+14 = 53$, is the 938th name in the list. So, COLIN would obtain a score of $938 \times 53 = 49714$.
What is the total of all the name scores in the file?

Solution [Buy me a beer]

We start by reading the file and extracting names using regular expressions. The function `findall(r'\w+', f.read())` captures each word in the file, treating any non-word characters (such as commas or quotation marks) as separators. After loading the names, we sort them alphabetically to assign each one a unique position in the sorted list.
To calculate the alphabetical value of each name, we create a dictionary, `vals`, that maps each letter in the alphabet to its position (e.g., 'A' to 1, 'B' to 2, and so forth). This allows us to compute a name's alphabetical value by summing the values of its letters.

The score of a name in position i (where the first position is $i = 0$) is given by:

$$\text{score} = (\text{position} + 1) \times \text{alphabetical value}$$

We define a lambda function, `alpha_value`, to calculate the alphabetical value of a name and another lambda function, `name_score`, to compute the score for each position. We then use `map` and `sum` to apply `name_score` to each index in the list, giving us the total score for all names in the file.

Code [GitHub]

```python
from re import findall
from string import ascii_uppercase as letters

with open("0022_names.txt") as f:
    names = findall(r'\w+', f.read())
names.sort()

vals = {letters[i]: i + 1 for i in range(len(letters))}

alpha_value = lambda name: sum(vals[char] for char in name)
name_score = lambda i: (i + 1) * alpha_value(names[i])

print(sum(map(name_score, range(len(names)))))
```

23 Non-Abundant Sums

Problem [Project Euler]

A perfect number is a number for which the sum of its proper divisors is exactly equal to the number. For example, the sum of the proper divisors of 28 would be $1 + 2 + 4 + 7 + 14 = 28$, which means that 28 is a perfect number.
A number n is called deficient if the sum of its proper divisors is less than n and it is called abundant if this sum exceeds n.
As 12 is the smallest abundant number, $1+2+3+4+6 = 16$, the smallest number that can be written as the sum of two abundant numbers is 24. By mathematical analysis, it can be shown that all integers greater than 28123 can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of

two abundant numbers is less than this limit.
Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

Solution [Buy me a beer]

We begin by computing the sum of proper divisors for each number up to 28,123. We store these sums in an array, `divsum`, where `divsum[n]` holds the sum of proper divisors of n. This calculation is optimized by only iterating up to LIMIT/2 and updating multiples of each divisor.
Using the `divsum` array, we identify all abundant numbers up to 28,123. A number n is abundant if `divsum[n] > n`. We use a lambda function to filter for such numbers, creating a list called `abundants`.
Next, we determine which numbers up to 28,123 can be written as the sum of two abundant numbers. We use a boolean array, `is_sum`, where each entry `is_sum[ab]` is set to `True` if `ab` is a sum of two elements from the `abundants` list. To avoid redundant calculations, we only iterate over pairs (i, j) where $i \leq j$, and we break out of the inner loop if the sum `ab` exceeds the limit.
Finally, we compute the result by summing all numbers that cannot be written as the sum of two abundant numbers. This is done by summing the indices of `is_sum` that remain `False`. The result gives the sum of all positive integers up to 28,123 that cannot be expressed as the sum of two abundant numbers.

Code [GitHub]

```python
LIMIT = 28_123

divsum = [0] * (LIMIT + 1)
for n in range(1, LIMIT // 2 + 1):
    for m in range(2 * n, LIMIT + 1, n):
        divsum[m] += n

is_abundant = lambda n: divsum[n] > n
abundants = list(filter(is_abundant, range(1, LIMIT + 1)))

is_sum = [False] * (LIMIT + 1)
for i in range(len(abundants)):
    for j in range(i, len(abundants)):
        ab = abundants[i] + abundants[j]
        if ab <= LIMIT: is_sum[ab] = True
        else: break

result = sum(i for i in range(LIMIT + 1) if not is_sum[i])
```

```
print(result)
```

24 Lexicographic Permutations

Problem [Project Euler]

A permutation is an ordered arrangement of objects. For example, 3124 is one possible permutation of the digits 1, 2, 3, and 4. If all of the permutations are listed numerically or alphabetically, we call it lexicographic order. The lexicographic permutations of 0, 1, and 2 are:

012 021 102 120 201 210

What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9?

Solution [Buy me a beer]

To solve this, we use the `nth_permutation` function from the `more_itertools` library, which directly retrieves the n-th permutation without needing to generate all permutations up to that point. The function takes as input the sequence of digits, the length of the permutation (in this case, 10, as we are permuting all 10 digits), and the desired position (999,999 for the millionth permutation, since indexing starts at 0).
Finally, we use `''.join(...)` to convert the tuple returned by `nth_permutation` into a string, producing the required millionth lexicographic permutation as a single output.

Code [GitHub]

```
from more_itertools import nth_permutation
from string import digits

print(''.join(nth_permutation(digits, 10, 999_999)))
```

25 1000-digit Fibonacci Number

Problem [Project Euler]

The Fibonacci sequence is defined by the recurrence relation:

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

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 12th 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?

Solution [Buy me a beer]

First, we create a generator function, `fib()`, which yields each Fibonacci number in sequence. This generator is similar to the one from Problem 2. Then, we add a condition in the generator to stop when the current Fibonacci number reaches 1000 digits. This is done by checking `len(str(a)) >= 1000`; once this condition is met, the generator stops producing terms. Next, we use `enumerate(fib())` to attach an index to each Fibonacci number as it is generated, allowing us to track the position of each term. Finally, we use `last` from the `more_itertools` library to retrieve the last generated pair of index and Fibonacci number before the generator stops.

This gives us the index and value of the first Fibonacci number with 1000 digits. We then extract the index by taking the first element of the pair, `[0]`, as our result.
This approach efficiently generates Fibonacci terms and checks their length without storing the entire sequence, making it well-suited for handling large Fibonacci numbers.

Code [GitHub]

```python
from more_itertools import last

def fib():
    a, b = 0, 1
    while True:
        yield a
        if len(str(a)) >= 1000: break
        a, b = b, a + b

print(last(enumerate(fib()))[0])
```

26 Reciprocal Cycles

Problem [Project Euler]

A unit fraction contains 1 in the numerator. The decimal representation of the unit fractions with denominators 2 to 10 are given:

$$\begin{aligned}
1/2 &= 0.5\\
1/3 &= 0.(3)\\
1/4 &= 0.25\\
1/5 &= 0.2\\
1/6 &= 0.1(6)\\
1/7 &= 0.(142857)\\
1/8 &= 0.125\\
1/9 &= 0.(1)\\
1/10 &= 0.1
\end{aligned}$$

Where 0.1(6) means 0.166666 . . ., and has a 1-digit recurring cycle. It can be seen that $\frac{1}{7}$ has a 6-digit recurring cycle.
Find the value of $d < 1000$ for which $\frac{1}{d}$ contains the longest recurring cycle in its decimal fraction part.

Solution [Buy me a beer]

First, we define a function `cycle_length` to determine the length of the repeating part of $1/d$'s decimal expansion. We perform long division by tracking each remainder's position in `r_pos`, a dictionary that stores the remainder as a key and its position as the value. As we calculate each remainder, we check if it's been seen before. If it has, we've found a cycle, and we get the cycle length by subtracting the first occurrence's position from the current one. If we reach a remainder of zero, there's no repeating cycle, and we return zero.
Then, we loop through all values of d from 2 to 999. For each d, we compute its `cycle_length`. If it's the largest cycle length we've encountered, we update `max_length` and set `longest` to the current d. After the loop, `longest` holds the value of d that produces the longest recurring cycle, and we print this result.

Code [GitHub]

```python
def cycle_length(d):
    r_pos, pos, r = {}, 0, 1

    while r:
        if r in r_pos: return pos - r_pos[r]
        r_pos[r], pos, r = pos, pos + 1, (r * 10) % d

    return 0

max_length = 0
for d in range(2, 1000):
    if (length := cycle_length(d)) > max_length:
        max_length, longest = length, d

print(longest)
```

27 Quadratic Primes

Problem [Project Euler]

Euler discovered the remarkable quadratic formula:

$$n^2 + n + 41$$

It turns out that the formula will produce 40 primes for the consecutive integer values $0 \le n \le 39$. However, when $n = 40$, $40^2 + 40 + 41 = 40(40 + 1) + 41$ is divisible by 41, and certainly when $n = 41$, $41^2 + 41 + 41$ is clearly divisible by 41.
The incredible formula $n^2 - 79n + 1601$ was discovered, which produces 80 primes for the consecutive values $0 \le n \le 79$. The product of the coefficients, -79 and 1601, is -126479.
Considering quadratics of the form:

$$n^2 + an + b, \quad \text{where} \quad |a| < 1000 \quad \text{and} \quad |b| \le 1000$$

where $|n|$ is the modulus/absolute value of n, e.g., $|11| = 11$ and $|-4| = 4$.
Find the product of the coefficients, a and b, for the quadratic expression that produces the maximum number of primes for consecutive values of n, starting with $n = 0$.

Solution [Buy me a beer]

First, we define a formula $f(n) = n^2 + an + b$ and note that both a and b must be odd integers for $f(n)$ to generate primes. This is because:

- ▷ For $f(0) = b$ to be prime, b must be odd.
- ▷ For $f(1) = 1 + a + b$ to remain prime, a must also be odd (since b is odd).

This observation reduces our search space significantly, allowing us to focus only on odd values for a and b.
Next, we define `formula(a, b)`, a generator function that yields values of $f(n) = n^2 + an + b$ for consecutive n starting at $n = 0$. This lets us generate values as needed without calculating them all at once.
Then, we define `count_primes(a, b)`, a helper function to count how many consecutive primes the formula produces for given values of a and b. Using `takewhile`, we keep generating values from `formula(a, b)` as long as they are prime, and `ilen` counts how many primes are generated in this way.

We then iterate over all possible pairs of odd values a and b in the range $[-999, 999]$ using `product(range(-999, 1000, 2), repeat=2)`. For each pair (a, b), we calculate the number of consecutive primes with `count_primes(a, b)`. If this count exceeds `max_primes`, we update `max_primes` and store the product of the coefficients in `prod`.
After the loop, `prod` contains the product of the coefficients a and b that yield the maximum number of consecutive primes for the quadratic formula, and we print it as the answer.

Code [GitHub]

```python
from itertools import count, takewhile, product
from more_itertools import ilen
from sympy import isprime

def formula(a, b):
    for n in count():
        yield n**2 + a*n + b

def count_primes(a, b):
    return ilen(takewhile(isprime, formula(a, b)))

max_primes = prod = 0
for a, b in product(range(-999, 1000, 2), repeat=2):
    if (primes := count_primes(a, b)) > max_primes:
        max_primes, prod = primes, a * b

print(prod)
```

28 Number Spiral Diagonals

Problem [Project Euler]

Starting with the number 1 and moving to the right in a clockwise direction, a 5 by 5 spiral is formed as follows:

21	22	23	24	**25**
20	**7**	8	**9**	10
19	6	**1**	2	11
18	**5**	4	**3**	12
17	16	15	14	**13**

It can be verified that the sum of the numbers on the diagonals is 101. What is the sum of the numbers on the diagonals in a 1001 by 1001 spiral formed in the same way?

Solution [Buy me a beer]

For a general $n \times n$ spiral, where n is odd, we observe that each layer around the center contributes four corners to the diagonal sum. Specifically, the corners of each layer k (where k goes from 1 up to $(n-1)/2$ for an $n \times n$ spiral) follow a predictable pattern in terms of both the number they contain and their positions.
Using formulas for triangular and square pyramidal numbers, we can compute the sum of the diagonals without explicitly generating the entire spiral.

- ▷ The function `T(n)` calculates the sum of the first n integers, using the formula for triangular numbers:

$$T(n) = \frac{n(n+1)}{2}$$

- ▷ The function `P(n)` calculates the sum of the squares of the first n integers, using the formula for square pyramidal numbers:

$$P(n) = \frac{n(n+1)(2n+1)}{6}$$

With these formulas, we can calculate the sum of the diagonals in a 1001×1001 spiral as follows:

$$\text{sum of diagonals} = 16 \times P(500) + 4 \times T(500) + 4 \times 500 + 1$$

This formula accounts for the values at each layer up to 500, which corresponds to a 1001×1001 grid. The final result adds the center value, 1, at the end of the calculation.
This approach avoids explicitly building the spiral and instead uses mathematical sequences to directly compute the sum, making it both efficient and compact.

Code [GitHub]

```python
T = lambda n: n * (n + 1) // 2
P = lambda n: T(n) * (2 * n + 1) // 3

print(16 * P(500) + 4 * T(500) + 4 * 500 + 1)
```

29 Distinct Powers

Problem [Project Euler]

Consider all integer combinations of a^b for $2 \le a \le 5$ and $2 \le b \le 5$:

$$\begin{array}{cccc} 2^2 = 4, & 2^3 = 8, & 2^4 = 16, & 2^5 = 32 \\ 3^2 = 9, & 3^3 = 27, & 3^4 = 81, & 3^5 = 243 \\ 4^2 = 16, & 4^3 = 64, & 4^4 = 256, & 4^5 = 1024 \\ 5^2 = 25, & 5^3 = 125, & 5^4 = 625, & 5^5 = 3125 \end{array}$$

If they are then placed in numerical order, with any repeats removed, we get the following sequence of 15 distinct terms:

$$4, 8, 9, 16, 25, 27, 32, 64, 81, 125, 243, 256, 625, 1024, 3125.$$

How many distinct terms are in the sequence generated by a^b for $2 \le a \le 100$ and $2 \le b \le 100$?

Solution [Buy me a beer]

To solve this, we use Python's `itertools.product` to generate pairs (a, b) for a and b in the range from 2 to 100. This produces all possible combinations of bases and exponents. We then use the function `pow` to compute a^b for each pair, which we apply using `starmap` from the `itertools` module. This function takes each pair as input and computes the corresponding power efficiently.
As we compute each power, we add it directly to a `set`. This ensures that only distinct values are retained, as sets automatically discard duplicates. Finally, we calculate the length of the set to determine the total number of unique terms.
This approach leverages efficient set operations to remove duplicates on-the-fly and the power function to quickly compute each combination, making it a concise and effective solution for generating distinct terms in the sequence.

Code [GitHub]

```python
from itertools import product, starmap

powers = starmap(pow, product(range(2, 101), repeat=2))
print(len(set(powers)))
```

30 Digit Fifth Powers

Problem [Project Euler]

Surprisingly, there are only three numbers that can be written as the sum of fourth powers of their digits:

$$\begin{aligned}1634 &= 1^4 + 6^4 + 3^4 + 4^4,\\ 8208 &= 8^4 + 2^4 + 0^4 + 8^4,\\ 9474 &= 9^4 + 4^4 + 7^4 + 4^4.\end{aligned}$$

As $1 = 1^4$ is not a sum, it is not included.
The sum of these numbers is $1634 + 8208 + 9474 = 19316$.
Find the sum of all the numbers that can be written as the sum of fifth powers of their digits.

Solution [Buy me a beer]

To solve this, we first determine an upper limit for numbers to consider. For a number with d digits, the maximum possible sum of fifth powers of its digits is $d \times 9^5$ (since 9 is the largest single-digit number). As we increase d, there comes a point where $d \times 9^5$ has fewer digits than d, meaning it is impossible for a larger number of digits to satisfy the condition. Using this approach, we compute an upper limit, `max_dpsum`, which is the last value of $d \times 9^5$ before the length constraint is no longer met.
Next, we create a list `powers` that stores the fifth powers of the digits 0 through 9, allowing us to quickly look up the fifth power of each digit in any number.
The function `dpsum(n)` calculates the sum of the fifth powers of the digits of a given number n. It converts n to a string, maps each character to an integer, and uses the `powers` list to retrieve the fifth power for each digit. The sum of these powers is then returned.
We define a lambda function `is_valid` to check if a number n equals the sum of the fifth powers of its digits. Finally, we use `filter` to identify all numbers from 2 to `max_dpsum` that satisfy this condition, and compute the sum of these valid numbers. This excludes 1, as it is not considered a sum in the problem statement.

Code [GitHub]

```python
from itertools import count

for digits in count():
    if len(str(p := digits * 9**5)) < digits: break
    max_dpsum = p

powers = [n**5 for n in range(10)]

def dpsum(n):
    return sum(powers[d] for d in map(int, str(n)))

is_valid = lambda n: dpsum(n) == n
print(sum(filter(is_valid, range(2, max_dpsum + 1))))
```

31 Coin Sums

Problem [Project Euler]

In the United Kingdom, the currency is made up of pound (£) and pence (p). There are eight coins in general circulation:

$$1\text{p}, 2\text{p}, 5\text{p}, 10\text{p}, 20\text{p}, 50\text{p}, £1\ (100\text{p}), \text{and } £2\ (200\text{p}).$$

It is possible to make £2 in the following way:

$$1 \times £1 + 1 \times 50\text{p} + 2 \times 20\text{p} + 1 \times 5\text{p} + 1 \times 2\text{p} + 3 \times 1\text{p}$$

How many different ways can £2 be made using any number of coins?

Solution [Buy me a beer]

We approach this problem by recursively counting the ways to reach the target amount of 200p (or £2) with the coins available.
First, we define `count_ways`, a recursive function that will explore two possibilities for each coin at a given amount: including the coin in the amount or skipping it. For this function:

- ▷ We check if `amount` is zero. If it is, we've successfully reached the target amount in one way, so we return 1.
- ▷ If `amount` goes below zero or we run out of coins to use, we return 0

since that path doesn't reach the target.

The recursion splits into two branches:

- ▷ **With the current coin**: We include the coin by subtracting its value from `amount` and stay at the same `coin_index` to allow reuse of the coin.
- ▷ **Without the current coin**: We exclude the coin and move to the next coin by increasing `coin_index`.

Using `@cache`, we efficiently store and reuse results for each (`amount`, `coin_index`) pair, saving computation time by avoiding recalculations. Finally, calling `count_ways(200)` gives the total number of unique ways to make £2 with the available coins.

Code [GitHub]

```python
from functools import cache

coins = [1, 2, 5, 10, 20, 50, 100, 200]

@cache
def count_ways(amount, i=0):
    if amount == 0: return 1
    if amount < 0 or i == len(coins): return 0

    with_coin = count_ways(amount - coins[i], i)
    without_coin = count_ways(amount, i + 1)

    return with_coin + without_coin

print(count_ways(200))
```

32 Pandigital Products

Problem [Project Euler]

We shall say that an n-digit number is pandigital if it makes use of all the digits 1 to n exactly once; for example, the 5-digit number, 15234, is 1 through 5 pandigital.
The product 7254 is unusual, as the identity, $39 \times 186 = 7254$, containing multiplicand, multiplier, and product is 1 through 9 pandigital.
Find the sum of all products whose multiplicand/multiplier/product identity can be written as a 1 through 9 pandigital.
HINT: Some products can be obtained in more than one way, so be sure to only include it once in your sum.

Solution [Buy me a beer]

To solve this, we generate all possible permutations of the digits 1 through 9, since any valid pandigital identity must use each digit exactly once. We then iterate over these permutations and split each permutation into three parts: the multiplicand, the multiplier, and the product.
Using `combinations`, we select two split points within the permutation to form the multiplicand, multiplier, and product parts. The combination of indices i and j defines the split positions for each part:

$$\texttt{part} = (\texttt{p[:i]}, \texttt{p[i:j]}, \texttt{p[j:]})$$

This split gives us the integers a, b, and c, which represent the multiplicand, multiplier, and product, respectively. We then check if $a \times b = c$; if this condition holds, we yield the product c.
To ensure each valid product is counted only once, we use `set(products())` to store unique values and eliminate duplicates. Finally, we calculate the sum of the unique products and print the result.

Code [GitHub]

```python
from string import digits
from itertools import combinations, permutations

def products():
    for p in permutations(digits[1:]):
        p = ''.join(p)
        for i, j in combinations(range(3, 6), 2):
            part = p[:i], p[i:j], p[j:]
            a, b, c = map(int, part)
            if a * b == c: yield c

print(sum(set(products())))
```

33 Digit Cancelling Fractions

Problem [Project Euler]

The fraction $\frac{49}{98}$ is a curious fraction, as an inexperienced mathematician attempting to simplify it may incorrectly believe that $\frac{49}{98} = \frac{4}{8}$, which is correct, is obtained by cancelling the 9s.
We shall consider fractions like $\frac{30}{50} = \frac{3}{5}$ to be trivial examples.
There are exactly four non-trivial examples of this type of fraction, less than one in value, and containing two digits in the numerator and denominator.
If the product of these four fractions is given in its lowest common terms, find the value of the denominator.

Solution [Buy me a beer]

To solve this problem, we identify non-trivial two-digit fractions that can be simplified by "digit cancellation" in a way that's mathematically incorrect but leads to the correct reduced fraction. We iterate through possible two-digit values for the numerator and denominator, identifying fractions that are less than one in value.
For each fraction $\frac{n}{d}$, where n is the numerator and d is the denominator, we split the digits of n and d into tens and units places (i.e., `n_1`, `n_0` for the numerator and `d_1`, `d_0` for the denominator). We then examine two conditions where an incorrect cancellation might still yield the correct result:

▷ First, we check if the units digit of n is equal to the tens digit of d (i.e., `n_0 == d_1`), and if the fraction formed by canceling these digits, $\frac{\texttt{n\_1}}{\texttt{d\_0}}$, equals the correctly reduced fraction $\frac{n}{d}$.

▷ Second, we check if the tens digit of n is equal to the units digit of d (i.e., `n_1 == d_0`), and if canceling these digits yields the same reduced fraction $\frac{\texttt{n\_0}}{\texttt{d\_1}}$.

When either of these conditions is met, we multiply the fraction $\frac{n}{d}$ into a cumulative product of all such fractions. This allows us to calculate the combined effect of all non-trivial examples.
After finding the product of these fractions, we print the denominator of the fraction in its lowest terms, which is our final answer.

Code [GitHub]

```python
from fractions import Fraction as fr

product = 1
for n in range(10, 99):
    for d in range(n + 1, 100):
        n_1, n_0 = divmod(n, 10)
        d_1, d_0 = divmod(d, 10)
        f = fr(n, d)
        conds = [
            d_0 != 0 and n_0 == d_1 and f == fr(n_1, d_0),
            n_1 == d_0 and f == fr(n_0, d_1)
        ]
        if conds[0] or conds[1]: product *= f

print(product.denominator)
```

34 Digit Factorials

Problem [Project Euler]

145 is a curious number, as $1! + 4! + 5! = 1 + 24 + 120 = 145$.
Find the sum of all numbers which are equal to the sum of the factorial of their digits.
NOTE: As $1! = 1$ and $2! = 2$ are not sums, they are not included.

Solution [Buy me a beer]

To solve this, we first create a list `f` that stores the factorials of the digits 0 through 9. We initialize `f` with `1`, corresponding to 0!, and then compute each subsequent factorial up to 9! by multiplying the current digit by the last factorial in the list.
Next, we define a function `dfsum(n)` that calculates the sum of the factorials of the digits of a given number n. This function converts n to a string, maps each character to an integer, and then uses the `f` list to retrieve the factorial of each digit, summing these values.
We also define a lambda function `is_factorion` to check if a number n is equal to the sum of the factorials of its digits.
To determine an upper limit for our search, we note that for any d-digit number, the maximum sum of factorials of its digits is $d \times 9!$. When d becomes large, this sum becomes insufficient to represent numbers with d digits, creating a natural upper bound. Using $7 \times 9!$ as this upper limit, we can be certain that any factorion will lie within this range.
Finally, we use `filter` to identify numbers in the range from 3 up to $7 \times 9!$ that satisfy the `is_factorion` condition, and we compute the sum of these numbers to obtain our result.

Code [GitHub]

```python
f = [1]
for n in range(1, 10): f.append(n * f[-1])

def dfsum(n):
    return sum(f[d] for d in map(int, str(n)))

is_factorion = lambda n: dfsum(n) == n
print(sum(filter(is_factorion, range(3, 7 * f[9]))))
```

35 Circular Primes

Problem [Project Euler]

The number, 197, is called a circular prime because all rotations of the digits: 197, 971, and 719, are themselves prime.
There are thirteen such primes below 100:

$$2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79, 97$$

How many circular primes are there below one million?

Solution [Buy me a beer]

To solve this, we first generate all primes below one million using `primerange` from the `sympy` library, which efficiently produces a list of prime numbers. We convert these primes into strings and store them in a `set` called `primes` for fast lookup, as checking for membership in a set is efficient.
Next, we define a function `rotations` to yield all rotations of a given number n. This function iterates over each possible rotation by slicing the string representation of n and recombining its parts.
We then initialize an empty set `circular_primes` to store all unique circular primes found. For each prime p in `primes`, we proceed as follows:

1. If p is already in `circular_primes`, we skip it since we have already processed it as part of another circular prime group.
2. If p contains any even digit (0, 2, 4, 6, or 8), we skip it as well, since it cannot be a circular prime.
3. For each valid prime p, we generate all rotations using `rotations`. If any rotation is not in `primes`, we stop and discard this prime; otherwise, if all rotations are in `primes`, we add all rotations of p to `circular_primes`.

Finally, we print the total number of circular primes by taking the length of `circular_primes` and adding 1 to include the prime 2, which is the only even-digit circular prime under one million.
This approach efficiently identifies circular primes by filtering out non-candidates early, minimizing unnecessary checks and leveraging set operations for quick lookups.

Code [GitHub]

```python
from sympy import primerange

def rotations(n):
    for i in range(len(n)):
        yield n[i:] + n[:i]

primes = set(map(str, primerange(10**6)))

circular_primes = set()
for p in primes:
    if p in circular_primes: continue
    if set(p) & set("02468"): continue
    rots = []
    for r in rotations(p):
        if r not in primes: break
        rots.append(r)
    else: circular_primes.update(rots)

print(len(circular_primes) + 1)
```

36 Double-base Palindromes

Problem [Project Euler]

The decimal number, $585 = 1001001001_2$ (binary), is palindromic in both bases.
Find the sum of all numbers, less than one million, which are palindromic in base 10 and base 2.
NOTE: The palindromic number, in either base, may not include leading zeros.

Solution [Buy me a beer]

To solve this, we define a helper function `is_pal(n, base)` that checks if a number n is palindromic in a given base. This function uses Python's formatted string syntax to convert n into the specified base: using `n:{base}` with `base` set to `'d'` for decimal or `'b'` for binary. It then checks if the resulting string representation is equal to its reverse, which indicates that the number is palindromic in that base.

Next, we define `is_double_pal(n)` to check if a number n is palindromic in both base 10 and base 2 by calling `is_pal(n, 'd')` and `is_pal(n, 'b')`. If both conditions are satisfied, the function returns `True`, indicating that n is a "double palindromic" number.
Finally, we use `filter` to apply `is_double_pal` to all numbers from 1 to $10^6 - 1$, identifying all double palindromic numbers in this range. We then calculate the sum of these numbers and print the result.

Code [GitHub]

```python
def is_pal(n, base):
    n = f"{n:{base}}"
    return n == n[::-1]

def is_double_pal(n):
    return is_pal(n, 'd') and is_pal(n, 'b')

print(sum(filter(is_double_pal, range(10**6))))
```

37 Truncatable Primes

Problem [Project Euler]

The number 3797 has an interesting property. Being prime itself, it is possible to continuously remove digits from left to right, and remain prime at each stage: 3797, 797, 97, and 7. Similarly, we can work from right to left: 3797, 379, 37, and 3.
Find the sum of the only eleven primes that are both truncatable from left to right and right to left.
NOTE: 2, 3, 5, and 7 are not considered to be truncatable primes.

Solution [Buy me a beer]

To solve this, we start by defining a generator function `left_truncatables` that generates left-truncatable primes by building on existing primes. The initial list of primes is set to `"3"`, `"5"`, and `"7"`, which are the only allowable single-digit primes for forming left-truncatable primes. At each iteration, we prepend each of these primes with a new digit (from `1` to `9`), forming larger numbers that are potentially prime. If the resulting number is prime

(checked with `isprime` from the `sympy` library), we add it to the list of `new_primes` to be extended in the next round. The generator yields each left-truncatable prime as it is found.
Next, we define a helper function `is_right_truncatable` to check if a given number n is right-truncatable. This function works recursively by removing the rightmost digit at each stage and verifying that the resulting number remains prime until only a single digit is left. If at any point the truncated number is not prime, the function returns `False`.
With both functions in place, we filter the left-truncatable primes produced by `left_truncatables` to retain only those that are also right-truncatable using `is_right_truncatable`. The problem states that there are only eleven such numbers, so we calculate the sum of these primes and adjust by subtracting 15 to exclude 3, 5, and 7, which should not be counted according to the problem statement.

Code [GitHub]

```python
from sympy import isprime

def left_truncatables():
    primes = ["3", "5", "7"]
    while primes:
        yield from map(int, primes)
        new_primes = []
        for prime in primes:
            for digit in map(str, range(1, 10)):
                new_prime = digit + prime
                if isprime(int(new_prime)):
                    new_primes.append(new_prime)
        primes = new_primes

def is_right_truncatable(n):
    if n < 10: return isprime(n)
    n //= 10
    return is_right_truncatable(n) if isprime(n) else False

primes = filter(is_right_truncatable, left_truncatables())
print(sum(primes, -15))
```

38 Pandigital Multiples

Problem [Project Euler]

Take the number 192 and multiply it by each of 1, 2, and 3:

$$\begin{aligned}192 \times 1 &= 192\\192 \times 2 &= 384\\192 \times 3 &= 576\end{aligned}$$

By concatenating each product, we get the 1 to 9 pandigital, 192384576. We will call 192384576 the concatenated product of 192 and $(1, 2, 3)$.
The same can be achieved by starting with 9 and multiplying by 1, 2, 3, 4, and 5, giving the pandigital, 918273645, which is the concatenated product of 9 and $(1, 2, 3, 4, 5)$.
What is the largest 1 to 9 pandigital 9-digit number that can be formed as the concatenated product of an integer with $(1, 2, \ldots, n)$ where $n > 1$?

Solution [Buy me a beer]

To solve this, we proceed as follows:

1. For each integer x, we initialize an empty string to build the concatenated product.
2. We multiply x by each element in the sequence $(1, 2, \ldots, n)$, appending each product to the concatenated string.
3. If the concatenated string reaches exactly 9 digits, we check if it is pandigital (i.e., if it contains each digit from 1 to 9 exactly once). If it is, we record it as a candidate for the largest pandigital number found so far.
4. If the concatenated string exceeds 9 digits, we stop further concatenation for the current integer x, as any additional products will only increase the length beyond the desired 9 digits.

By iterating through possible integers x and checking their concatenated products, we identify the largest 1 to 9 pandigital number that can be formed under these conditions.

Code [GitHub]

```python
from itertools import permutations, count
from string import digits

pans = set(''.join(p) for p in permutations(digits[1:]))

def products():
    for n in range(1, 10**5):
        concat = ""
        for k in count(1):
            concat = concat + str(n * k)
            if (l := len(concat)) >= 9: break
        if l == 9 and concat in pans: yield concat

print(max(map(int, products())))
```

39 Integer Right Triangles

Problem [Project Euler]

If p is the perimeter of a right-angled triangle with integral length sides, $\{a, b, c\}$, there are exactly three solutions for $p = 120$.

$$\{20, 48, 52\}, \{24, 45, 51\}, \{30, 40, 50\}$$

For which value of $p \leq 1000$, is the number of solutions maximized?

Solution [Buy me a beer]

To solve this, we:

1. Define a function `triples` to generate all primitive Pythagorean triples. We use two integers, m and n (where $m > n$), which generate all primitive Pythagorean triples as follows:

   $$a = m^2 - n^2, \quad b = 2mn, \quad c = m^2 + n^2.$$

   To ensure (a, b, c) is a primitive Pythagorean triple, we:

   ▷ Require m and n to be coprime ($\texttt{gcd}(m, n) = 1$).

   ▷ Require m and n to have opposite parity (one even, one odd).

2. After generating each primitive triple, we use integer multiples of it to find all triples that can sum to each perimeter $p \leq 1000$. For each triple (a, b, c), we generate multiples $k \times (a, b, c)$ until the perimeter $p = k \times (a + b + c)$ exceeds 1000.

3. Use a `Counter` called `perims` to count each perimeter. For each perimeter p we calculate, we increment its count in `perims`.

4. Finally, we find the perimeter p with the highest count by applying `max` to `perims` with a custom key that retrieves the perimeter with the maximum count.

Code [GitHub]

```python
from itertools import count
from math import gcd
from collections import Counter

LIMIT = 1000

def triples():
    for m in count(2):
        for n in range(1, m):
            if (m + n) % 2 != 1: continue
            if gcd(m, n) != 1: continue
            a = m**2 - n**2
            b = 2*m*n
            c = m**2 + n**2
            if a + b + c > LIMIT: return
            yield a, b, c

perims = Counter()
for a, b, c in triples():
    for k in count(1):
        ka, kb, kc = k*a, k*b, k*c
        if (p := ka + kb + kc) > LIMIT: break
        perims[p] += 1

print(max(perims, key=lambda p: perims[p]))
```

40 Champernowne's Constant

Problem [Project Euler]

An irrational decimal fraction is created by concatenating the positive integers:

$$0.12345678910\mathbf{1}112131415161718192021\ldots$$

It can be seen that the 12th digit of the fractional part is 1.
If d_n represents the n^{th} digit of the fractional part, find the value of the following expression:

$$d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000}$$

Solution [Buy me a beer]

To solve this, we generate the digits of the concatenated sequence one by one using an incremental approach:

1. We define a generator function, `gen()`, that starts with the number $n = 1$ and uses a deque `dq` to store the digits of n as strings.

2. In each loop iteration, if `dq` is empty, we convert n to a string, add each of its digits to `dq`, and increment n by 1.

3. We then yield the leftmost digit in `dq` by popping it from the left, providing each successive digit of the concatenated decimal fraction.

We specify the target digit positions as `indexes` $= [10^k \mid k = 0, 1, \ldots, 6]$, representing the positions 1, 10, 100, 1000, 10000, 100000, and 1000000.
Next, we initialize `prod` to 1 and iterate over the digits produced by `gen()` using `enumerate` to keep track of the current position i. When i matches any target position in `indexes`, we multiply `prod` by the current digit d.
Once i reaches the last target position, 1000000, we break the loop.
The final value of `prod` is the product of the specified digits, yielding the answer to the problem.

Code [GitHub]

```python
from collections import deque

def gen():
    n, dq = 1, deque()
    while True:
        if not dq:
            dq.extend(str(n))
            n += 1
        yield int(dq.popleft())

indexes = [10**k for k in range(7)]

prod = 1
for i, d in enumerate(gen(), 1):
    if i in indexes: prod *= d
    if i == indexes[-1]: break

print(prod)
```

41 Pandigital Prime

Problem [Project Euler]

We shall say that an n-digit number is pandigital if it makes use of all the digits 1 to n exactly once. For example, 2143 is a 4-digit pandigital and is also prime.
What is the largest n-digit pandigital prime that exists?

Solution [Buy me a beer]

To solve this, we proceed as follows:

1. We define a function `find_pandigital_prime` that generates all n-digit pandigital numbers using the digits from 1 to n.

2. We generate all permutations of the digits `"1"` through `str(n)` using `permutations(digits[1:n + 1])`. To avoid even numbers (which cannot be prime if they end in an even digit), we skip any permutation ending in `"2"`, `"4"`, `"6"`, or `"8"`.

3. For each valid permutation, we convert it to an integer and add it to

the list `odd_pandigitals`.

4. We then sort `odd_pandigitals` in descending order so that we can check the largest pandigital numbers first.

5. We check each number in `odd_pandigitals` to see if it is prime using `isprime` from the `sympy` library. If we find a prime, we return it as the result.

We start with $n = 9$, the maximum length for a pandigital number, and decrease n down to 4, since the problem already indicates the existence of a 4-digit pandigital prime. This order allows us to examine larger pandigital numbers first, increasing our chance of finding the largest prime early.

Code [GitHub]

```python
from string import digits
from itertools import permutations
from sympy import isprime

def find_pandigital_prime(n):
    odd_pandigitals = []
    for perm in permutations(digits[1:n + 1]):
        if perm[-1] in "2468": continue
        odd_pandigitals.append(int(''.join(perm)))

    odd_pandigitals.sort(reverse=True)

    for p in odd_pandigitals:
        if isprime(p): return p

for n in range(9, 3, -1):
    if (p := find_pandigital_prime(n)): break

print(p)
```

42 Coded Triangle Numbers

Problem [Project Euler]

The n^{th} term of the sequence of triangle numbers is given by $t_n = \frac{1}{2}n(n+1)$; so the first ten triangle numbers are:

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \ldots$$

By converting each letter in a word to a number corresponding to its alphabetical position and adding these values, we form a word value. For example, the word value for SKY is $19 + 11 + 25 = 55 = t_{10}$. If the word value is a triangle number, then we shall call the word a triangle word.
Using words.txt, a 16K text file containing nearly two-thousand common English words, how many are triangle words?

Solution [Buy me a beer]

This problem is similar to Problem 22, where we calculated name scores based on letter positions and used these values in further calculations.
To solve this, we proceed as follows:

1. We first read the words from the file `words.txt` using regular expressions to extract each word. This gives us a list of words to evaluate.
2. We then create a dictionary, `values`, that maps each uppercase letter to its corresponding position in the alphabet.
3. The function `get_value(name)` computes the word value by summing the values of each character in the word, using the `values` dictionary for lookup.
4. Next, we generate a set `T` containing the first 100 triangle numbers using the formula $t_n = \frac{1}{2}n(n+1)$. This set provides an efficient way to check if a word's value is a triangle number.

Finally, we count the number of words whose computed value is in the set `T`. This is done using a generator expression that applies `get_value(w)` to each word `w` in `words` and checks for membership in `T`. The final count represents the number of triangle words, which we print as the result.
This approach is efficient, leveraging set membership for fast triangle number checking and precomputed letter values for quick lookup.

Code [GitHub]

```python
from re import findall
from string import ascii_uppercase as letters

with open("0042_words.txt") as f:
    words = findall(r'\w+', f.read())

values = {letters[i]: i + 1 for i in range(len(letters))}

def get_value(name):
    return sum(values[char] for char in name)

T = {n * (n + 1) // 2 for n in range(100)}

result = sum(1 for w in words if get_value(w) in T)
print(result)
```

43 Sub-string Divisibility

Problem [Project Euler]

The number, 1406357289, is a 0 to 9 pandigital number because it is made up of each of the digits 0 to 9 in some order, but it also has a rather interesting sub-string divisibility property.
Let d_1 be the 1st digit, d_2 be the 2nd digit, and so on. In this way, we note the following:

$$\begin{aligned}
d_2d_3d_4 &= 406 \quad \text{is divisible by } 2\\
d_3d_4d_5 &= 063 \quad \text{is divisible by } 3\\
d_4d_5d_6 &= 635 \quad \text{is divisible by } 5\\
d_5d_6d_7 &= 357 \quad \text{is divisible by } 7\\
d_6d_7d_8 &= 572 \quad \text{is divisible by } 11\\
d_7d_8d_9 &= 728 \quad \text{is divisible by } 13\\
d_8d_9d_{10} &= 289 \quad \text{is divisible by } 17
\end{aligned}$$

Find the sum of all 0 to 9 pandigital numbers with this property.

Solution [Buy me a beer]

To solve this, we:

1. Generate all permutations of the digits `0` through `9` to produce potential pandigital numbers.
2. Use quick checks to filter permutations that cannot satisfy the divisibility property. Specifically:
   - ▷ We skip permutations where the first digit is `0`, as this would not form a valid 10-digit number.
   - ▷ We skip permutations where the fourth digit (d_4) is odd, as $d_2d_3d_4$ must be divisible by 2, implying that d_4 itself must be even.
   - ▷ We skip permutations where the sixth digit (d_6) is not 0 or 5, as $d_4d_5d_6$ must be divisible by 5.
3. Define a function `is_valid(n)` that checks whether each relevant 3-digit substring of n satisfies the divisibility property with respect to the primes list `[2, 3, 5, 7, 11, 13, 17]`.
4. For each valid permutation, we convert it to an integer and add it to our cumulative sum `result`.

Finally, we print `result` as the sum of all pandigital numbers satisfying the given properties. This approach efficiently filters and verifies pandigital numbers, ensuring that only those meeting the divisibility conditions are included in the final sum.

Code [GitHub]

```python
from string import digits
from itertools import permutations

primes = [2, 3, 5, 7, 11, 13, 17]

def is_valid(n):
    for i in range(len(n) - 3):
        if int(n[i+1:i+4]) % primes[i]: return False
    return True

result = 0
for p in permutations(digits):
    if p[0] == '0' or p[3] in "1357" or p[5] not in "05":
        continue
    pandigital = ''.join(p)
    if is_valid(pandigital): result += int(pandigital)

print(result)
```

44 Pentagon Numbers

Problem [Project Euler]

Pentagonal numbers are generated by the formula, $P_n = \frac{n(3n-1)}{2}$. The first ten pentagonal numbers are:

$$1, 5, 12, 22, 35, 51, 70, 92, 117, 145, \ldots$$

It can be seen that $P_4 + P_7 = 22 + 70 = 92 = P_8$. However, their difference, $70 - 22 = 48$, is not pentagonal.
Find the pair of pentagonal numbers, P_j and P_k, for which their sum and difference are pentagonal and $D = |P_k - P_j|$ is minimized; what is the value of D?

Solution [Buy me a beer]

To solve this, we:

1. Generate a set `P` of pentagonal numbers up to a specified limit, using `LIMIT` = 2400 in our code. This ensures that we have enough pentagonal numbers to find pairs satisfying the conditions, as we are searching for both the sum and the difference to be pentagonal.
2. Use `combinations` to iterate over all pairs (a, b) of pentagonal numbers within `P`.
3. For each pair (a, b), we check if both the sum $a + b$ and the absolute difference $D = |a - b|$ are present in `P`.
4. If both conditions are met, we print D as it represents a valid difference where both the sum and difference of the pentagonal numbers are also pentagonal.

Code [GitHub]

```python
from itertools import combinations

LIMIT = 2400

P = {n * (3*n - 1) // 2 for n in range(1, LIMIT + 1)}

for a, b in combinations(P, 2):
    if a + b in P and (D := abs(a - b)) in P: print(D)
```

45 Triangular, Pentagonal, and Hexagonal

Problem [Project Euler]

Triangle, pentagonal, and hexagonal numbers are generated by the following formulae:

Triangle	$T_n = \frac{n(n+1)}{2}$	1, 3, 6, 10, 15, ...
Pentagonal	$P_n = \frac{n(3n-1)}{2}$	1, 5, 12, 22, 35, ...
Hexagonal	$H_n = n(2n-1)$	1, 6, 15, 28, 45, ...

It can be verified that $T_{285} = P_{165} = H_{143} = 40755$.
Find the next triangle number that is also pentagonal and hexagonal.

Solution [Buy me a beer]

To solve this, we:

1. Define a general function $f(k, n)$ to generate polygonal numbers of type k with n sides, where $k = 3$ for triangular, $k = 5$ for pentagonal, and $k = 6$ for hexagonal numbers. The formula is:

$$f(k, n) = \frac{n((k-2)n - k + 4)}{2}$$

2. Generate sets of triangular, pentagonal, and hexagonal numbers up to a sufficiently large limit $n = 10^5$ for each type. This gives us large sets for each polygonal number sequence.

3. Calculate the intersection of these sets using $S[3] \cap S[5] \cap S[6]$, which finds numbers common to all three sequences.

4. Sort the resulting intersection and retrieve the second element, as the first element is 40755, which we already know. The second element is the answer we seek.

This approach efficiently finds the next number that appears in all three sequences by leveraging set operations for quick intersection and lookup.

Code [GitHub]

```python
f = lambda k, n: n * ((k - 2) * n - k + 4) // 2
S = {k: {f(k, n) for n in range(2, 10**5)} for k in [3,5,6]}

print(sorted(S[3] & S[5] & S[6])[1])
```

46 Goldbach's Other Conjecture

Problem [Project Euler]

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

$$\begin{aligned}9 &= 7 + 2 \times 1^2\\ 15 &= 7 + 2 \times 2^2\\ 21 &= 3 + 2 \times 3^2\\ 25 &= 7 + 2 \times 3^2\\ 27 &= 19 + 2 \times 2^2\\ 33 &= 31 + 2 \times 1^2\end{aligned}$$

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?

Solution [Buy me a beer]

To solve this, we:

1. Generate a list of primes up to a specified limit (in this case, 10^4) using `primerange` from the `sympy` library. This list will be used to check potential prime values for the conjecture.

2. Generate a list of squares up to the integer square root of the limit, as these squares will be used to compute values for $2 \times k^2$.

3. Create a set S to store all numbers that can be written as the sum of a prime and twice a square, by iterating over pairs of primes and squares using `product`. For each combination (p, k^2), we compute $p + 2 \times k^2$ and add it to S.

After generating all possible sums, we check each odd composite number starting from 9 using `count(9, 2)` to increment by 2 and skip even numbers. For each candidate n, we verify:

- ▷ If n is prime, we skip it as it does not meet the criteria of an odd composite number.

- ▷ If n is not in S, this means it cannot be written as the sum of a prime and twice a square, making it the smallest counterexample to the conjecture.

Code [GitHub]

```python
from sympy import primerange, isprime
from math import isqrt
from itertools import product, count, filterfalse

LIMIT = 10**4

primes = list(primerange(LIMIT))
squares = [n**2 for n in range(1, isqrt(LIMIT))]

S = set()
for p, sq in product(primes, squares):
    S.add(p + 2*sq)

for n in filterfalse(isprime, count(9, 2)):
    if n not in S: break

print(n)
```

47 Distinct Primes Factors

Problem [Project Euler]

The first two consecutive numbers to have two distinct prime factors are:

$$14 = 2 \times 7$$
$$15 = 3 \times 5$$

The first three consecutive numbers to have three distinct prime factors are:

$$644 = 2^2 \times 7 \times 23$$
$$645 = 3 \times 5 \times 43$$
$$646 = 2 \times 17 \times 19$$

Find the first four consecutive integers to have four distinct prime factors each. What is the first of these numbers?

Solution [Buy me a beer]

To solve this, we:

1. Define an upper limit, `LIMIT`, and use `primerange` from `sympy` to generate a list of primes up to this limit. These primes will help us count the distinct prime factors for each integer up to `LIMIT`.

2. Create a list `factors` initialized to zero, where each index i in `factors` represents the number of distinct prime factors for the integer i.

3. For each prime p in the list of primes, we iterate over its multiples m (i.e., $p, 2p, 3p, \ldots$) up to `LIMIT`. For each multiple m, we increment `factors[m]` by 1, which counts p as a distinct prime factor of m.

After counting the distinct prime factors for each integer up to `LIMIT`, we use `windowed` from the `more_itertools` library to examine consecutive windows of four integers. For each window, we check if all four numbers have exactly four distinct prime factors by verifying if the set of values in the window is $\{4\}$. If so, we have found our sequence, and we print the starting integer n of this window as the answer.
This approach efficiently finds the first four consecutive integers with four distinct prime factors by leveraging prime factor counting and sliding windows for consecutive checking.

Code [GitHub]

```python
from sympy import primerange
from more_itertools import windowed

LIMIT = 2*10**5

primes = primerange(LIMIT)
factors = [0] * (LIMIT + 1)

for p in primes:
    for m in range(p, LIMIT + 1, p):
        factors[m] += 1

for n, w in enumerate(windowed(factors, 4)):
    if set(w) == {4}: break

print(n)
```

48 Self Powers

Problem [Project Euler]

The series, $1^1 + 2^2 + 3^3 + \cdots + 10^{10} = 10405071317$.
Find the last ten digits of the series, $1^1 + 2^2 + 3^3 + \cdots + 1000^{1000}$.

Solution [Buy me a beer]

To solve this, we only need the last ten digits of the result, which is equivalent to finding the sum modulo 10^{10}.
To accomplish this:

1. We define `MOD` as 10^{10}, as we are only interested in the last ten digits.

2. For each integer n from 1 to 1000, we calculate $n^n \mod$ `MOD` using the built-in `pow` function. This function, when called as `pow(base, exp, mod)`, efficiently computes $\texttt{base}^{\texttt{exp}} \mod$ `mod` without generating large intermediate values, which is essential given the size of 1000^{1000}.

3. We sum all results from $n^n \mod$ `MOD` for n in the range 1 to 1000. Taking the sum modulo 10^{10} gives the last ten digits of the series.

Finally, we print `result` as the answer, which represents the last ten digits of the sum. This approach ensures we avoid handling excessively large numbers by using modular arithmetic throughout the calculation.

Code [GitHub]

```python
MOD = 10**10

result = sum(pow(n, n, MOD) for n in range(1, 1001)) % MOD

print(result)
```

49 Prime Permutations

Problem [Project Euler]

The arithmetic sequence, $1487, 4817, 8147$, in which each of the terms increases by 3330, is unusual in two ways: (i) each of the three terms are prime, and (ii) each of the 4-digit numbers are permutations of one another.
There are no arithmetic sequences made up of three 1-, 2-, or 3-digit primes exhibiting this property, but there is one other 4-digit increasing sequence.
What 12-digit number do you form by concatenating the three terms in this sequence?

Solution [Buy me a beer]

To solve this, we:

1. Generate all 4-digit prime numbers using `primerange` from the `sympy` library. For convenience, we convert each prime to a string so that we can sort and group primes by their digits.

2. Use a `defaultdict` called `primes` to store lists of primes grouped by their sorted digit tuples. For example, a prime $p = 8147$ would be stored in the list associated with the key $(1, 4, 7, 8)$ (the sorted digits of 8147).

3. For each list S in `primes.values()`, we check if it contains at least three primes. If so, we proceed to check combinations of three primes from S to find any arithmetic sequences.

4. For each combination (a, b, c) in S, we check if it forms an arithmetic

sequence by verifying that $2b = a + c$. We also exclude the known sequence $1487, 4817, 8147$ by ensuring $a \neq 1487$.

5. When a valid sequence a, b, c is found, we print the concatenated string `f"{a}{b}{c}"`, which forms the desired 12-digit number.

This approach efficiently groups 4-digit primes by their digit permutations, then checks each group for possible arithmetic sequences. By using combinations and set-based grouping, we identify the required sequence and output the concatenated result.

Code [GitHub]

```python
from collections import defaultdict
from sympy import primerange
from itertools import combinations

primes = defaultdict(list)
for p in map(str, primerange(10**3, 10**4)):
    primes[tuple(sorted(p))].append(int(p))

for S in primes.values():
    if len(S) < 3: continue
    for comb in combinations(sorted(S), 3):
        a, b, c = comb
        if 2*b == a + c and a != 1487:
            print(f"{a}{b}{c}")
```

50 Consecutive Prime Sum

Problem [Project Euler]

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?

Solution [Buy me a beer]

To solve this, we:

1. Generate a list of all primes below one million using `primerange` from the `sympy` library. This list will allow us to identify consecutive prime sums and check if a sum is prime.

2. Compute a cumulative sum array, `cumsum`, where each element represents the sum of all primes up to that position. This allows us to calculate the sum of any consecutive subarray of primes efficiently as:

   $$\texttt{cumsum}[j + i] - \texttt{cumsum}[j]$$

   which gives the sum of primes from index j to $j + i - 1$.

The function `find_prime()` iterates over possible subarray lengths i starting from the longest (the length of the primes list) down to 2. For each length i, we compute all possible sums of i consecutive primes:

▷ If a sum s exceeds the limit (one million), we break out of the loop for that length.

▷ If s is within the limit and also in our list of primes, we return s as it is the longest sum of consecutive primes resulting in a prime.

Finally, we print the result of `find_prime()`, which is the prime below one million that can be expressed as the sum of the most consecutive primes.
This approach is efficient as it leverages cumulative sums for quick subarray calculations and starts with the longest subarrays first, allowing it to identify the prime with the maximum number of consecutive terms quickly.

Code [GitHub]

```python
from sympy import primerange

LIMIT = 10**6
primes = list(primerange(LIMIT))

cumsum = [0]
for p in primes: cumsum.append(cumsum[-1] + p)

def find_prime():
    for i in range(l := len(primes), 1, -1):
        for j in range(l - i + 1):
            if (s := cumsum[j+i] - cumsum[j]) > LIMIT: break
            if s in primes: return s

print(find_prime())
```

51 Prime Digit Replacements

Problem [Project Euler]

By replacing the 1st digit of the 2-digit number *3, it turns out that six of the nine possible values: 13, 23, 43, 53, 73, and 83, are all prime.
By replacing the 3rd and 4th digits of 56**3 with the same digit, this 5-digit number is the first example having seven primes among the ten generated numbers, yielding the family: 56003, 56113, 56333, 56443, 56663, 56773, and 56993. Consequently, 56003, being the first member of this family, is the smallest prime with this property.
Find the smallest prime which, by replacing part of the number (not necessarily adjacent digits) with the same digit, is part of an eight prime value family.

Solution [Buy me a beer]

To solve this, we:

1. Generate all 6-digit primes using `primerange` from `sympy`.
2. Define a function `pattern_gen(p)` to generate digit replacement patterns for a given prime p. This function:
   - ▷ Converts p to a list of digits for easy manipulation.
   - ▷ Uses `combinations` to select sets of three digits to replace, generating a unique pattern where those selected positions are replaced with '*'.
3. Define a function `matching(pattern)` to generate all possible numbers for a given pattern by replacing the '*' character with each digit from 0 to 9:
   - ▷ If '*' is the last digit of the pattern, we restrict replacements to odd digits, as even numbers cannot be prime.
   - ▷ If '*' is the first digit, we exclude '0' to avoid generating numbers with leading zeros.
4. Define a cached function `isprime(n)` to check the primality of numbers, improving efficiency by avoiding redundant primality tests.

To identify the required prime:

1. For each prime p, we generate patterns using `pattern_gen`.
2. For each pattern, we generate potential numbers by replacing '*' with each allowed digit.
3. We count the number of primes within each family of generated numbers. If a pattern yields exactly eight primes, we store the smallest prime in this family and return it as the solution.

The result of `print(find_prime())` is the smallest prime that, by replacing some digits, forms a family of eight prime values. This approach is efficient due to the use of caching for primality tests and the generation of only relevant patterns for digit replacement.

Code [GitHub]

```python
from sympy import primerange, isprime
from itertools import combinations
from functools import cache

primes = map(str, primerange(10**5, 10**6))

def pattern_gen(p):
    p, l = list(p), len(p)
    for indexes in combinations(range(l), 3):
        pattern = p.copy()
        for i in indexes: pattern[i] = '*'
        yield ''.join(pattern)

def matching(pattern):
    digits = set("0123456789")
    if pattern[-1] == '*': digits -= set("02468")
    if pattern[0] == '*': digits -= {'0'}

    for d in digits: yield pattern.replace('*', d)

def find_prime():
    @cache
    def is_prime(n): return isprime(int(n))

    for p in primes:
        for pattern in pattern_gen(p):
            not_prime_count, p_list = 0, []
            for p in matching(pattern):
                if is_prime(p): p_list.append(p)
                else:
                    not_prime_count += 1
                    if not_prime_count > 2: break
            else:
                if len(p_list) == 8: return min(p_list)

print(find_prime())
```

52 Permuted Multiples

Problem [Project Euler]

It can be seen that the number, 125874, and its double, 251748, contain exactly the same digits, but in a different order.
Find the smallest positive integer, x, such that $2x$, $3x$, $4x$, $5x$, and $6x$, contain the same digits.

Solution [Buy me a beer]

To solve this, we:

1. Define a function `has_same_digits(n)` that uses `Counter` from Python's `collections` module. This function creates a digit frequency count for n and compares it with the digit frequencies of each multiple $2n$, $3n$, $4n$, $5n$, and $6n$. The `Counter` efficiently checks if the multiples contain the same digits by comparing frequency counts.

2. Start a loop with $n = 1$ and increment n until we find the solution.

3. For each n, check if `has_same_digits(n)` returns `True` for all multiples up to $6n$. If it does, n is the desired solution.

Once we find the smallest n meeting the condition, we print n as the result. Using `Counter` ensures that digit frequencies are accurately compared, making this approach efficient and straightforward. By incrementing n in a loop and verifying each multiple, we find the smallest integer that meets the criteria.

Code [GitHub]

```python
from collections import Counter
from itertools import count

def same_digits(n):
    digits = Counter(str(n))
    for k in range(2, 7):
        if Counter(str(k*n)) != digits: return False
    return True

for n in count(1):
    if same_digits(n): break
print(n)
```

53 Combinatoric Selections

Problem [Project Euler]

There are exactly ten ways of selecting three from five, 12345:

$$123,\ 124,\ 125,\ 134,\ 135,\ 145,\ 234,\ 235,\ 245,\ \text{and } 345$$

In combinatorics, we use the notation, $\binom{5}{3} = 10$.

In general, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$, where $r \le n$, $n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$, and $0! = 1$.

It is not until $n = 23$, that a value exceeds one million: $\binom{23}{10} = 1144066$.

How many, not necessarily distinct, values of $\binom{n}{r}$ for $1 \le n \le 100$, are greater than one million?

Solution [Buy me a beer]

To solve this, we use Pascal's Triangle to efficiently generate the binomial coefficients up to $n = 100$:

1. Define a function `binomials(n)` that constructs Pascal's Triangle up to row n, where each row represents the binomial coefficients for that n value.

2. Start by initializing Pascal's Triangle with [1], representing the top of the triangle.

3. For each row i from 1 to n:

   ▷ Initialize the row with 1 (the first binomial coefficient of each row).

   ▷ For each position j from 1 to $i - 1$, calculate the binomial coefficient as the sum of the two coefficients directly above it:

   $$\text{row}[j] = \text{tri}[i-1][j-1] + \text{tri}[i-1][j].$$

   ▷ Append 1 as the last element of each row.

   ▷ Append this row to Pascal's Triangle.

4. Yield each value from Pascal's Triangle using `flatten` from `more_itertools` to handle the values as a single sequence.

To count values greater than one million, we use:

```
sum(1 for b in binomials(100) if b > 10**6).
```

This counts each binomial coefficient generated by `binomials(100)` that exceeds one million, providing the final answer.
Using Pascal's Triangle is efficient because it avoids recalculating factorials and instead builds each binomial coefficient from previous values.

Code [GitHub]

```python
from more_itertools import flatten

def binomials(n):
    tri = [[1]]

    for i in range(1, n + 1):
        row = [1]
        for j in range(1, i):
            row.append(tri[i - 1][j - 1] + tri[i - 1][j])
        row.append(1)
        tri.append(row)

    yield from flatten(tri)

print(sum(1 for b in binomials(100) if b > 10**6))
```

54 Poker Hands

Problem [Project Euler]

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 the 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:

Hand	**Player 1**	**Player 2**	**Winner**
1	5♥ 5♣ 6♠ 7♠ K♦	2♣ 3♠ 8♠ 8♦ 10♦	Player 2
2	5♦ 8♣ 9♠ J♠ A♣	2♣ 5♣ 7♦ 8♠ Q♥	Player 1
3	2♦ 9♣ A♠ A♥ A♣	3♦ 6♦ 7♦ 10♦ Q♦	Player 2
4	4♦ 6♠ 9♥ Q♥ Q♣	3♦ 6♦ 7♥ Q♦ Q♠	Player 1
5	2♥ 2♦ 4♣ 4♦ 4♠	3♣ 3♦ 3♠ 9♠ 9♦	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?

Solution [Buy me a beer]

To solve this problem, we analyze poker hands by ranking them according to the specified categories: high card, one pair, two pairs, three of a kind, straight, flush, and full house. We observe that there are no hands in the file containing a four of a kind, straight flush, or royal flush, which allows us to simplify our code by excluding these classifications when determining each hand's winner. Additionally, we note that the only draws are cases in which both players have one pair or both have high card, which further reduces the need for complex tie-breaking logic.
We define helper functions to parse and classify each hand. First, the `counts` function takes in a hand and produces a count of each card's value and suit. Using this count, the `classify` function categorizes the hand according to the possible rankings. For instance, if a hand has both three of one value and two of another, it is classified as a "full house"; if all five cards are of the same suit, it is a "flush"; and if the values are consecutive, it is classified as a "straight." In addition, the function can detect cases like two pairs or a single pair.
The comparison function `wins_player1` uses these classifications to determine the outcome of each hand. By ranking the hand types from lowest to highest, it assigns a numeric score to each hand, making it straightforward to compare two hands. When the hands are of the same type, we further compare the relevant card values, such as the highest card for "high card" hands or the value of the pair for "one pair" hands, to break ties and determine the winner.
Finally, we process the hands by reading the file and using the `batched` function to separate the sequence of cards into groups of two hands of five cards each, one for each player. We then apply our rules to each pair of hands and count the number of times Player 1 wins, giving us the final result.

Code [GitHub]

```python
from collections import Counter
from itertools import batched, starmap
import re

with open("0054_poker.txt") as f:
    cards = re.findall(r'\w{2}', f.read())

def counts(hand):
    values = {'T': 10, 'J': 11, 'Q': 12, 'K': 13, 'A': 14}
    v_c, s_c = Counter(), Counter()
    for card in hand:
        value, suit = card
        if value in values: v_c[values[value]] += 1
        else: v_c[int(value)] += 1
        s_c[suit] += 1
    return v_c, s_c

def classify(hand):
    v_c, s_c = counts(hand)
    if 3 in v_c.values() and 2 in v_c.values(): return "ful"
    if 5 in s_c.values(): return "flush"
    sorted_values = sorted(v_c.keys())
    if (sorted_values[-1] - sorted_values[0] == 4) and len(
       ↪set(v_c.values())) == 1: return "straight"
    if 3 in v_c.values(): return "three of a kind"
    if list(v_c.values()).count(2) == 2: return "two pairs"
    if 2 in v_c.values(): return "one pair"
    return "high card"

def wins_player1(h1, h2):
    scores = ["high card", "one pair", "two pairs", "three
       ↪of a kind", "straight", "flush", "ful"]
    s1, s2 = scores.index(classify(h1)), scores.index(
       ↪classify(h2))
    if s1 > s2: return True
    if s1 == s2:
        v_c1, _ = counts(h1)
        v_c2, _ = counts(h2)
        if s1 == 0:
            return max(v_c1.keys()) > max(v_c2.keys())
        elif s1 == 1:
            return v_c1.most_common(1)[0][0] > v_c2.
               ↪most_common(1)[0][0]
    return False

game_gen = batched(batched(cards, 5), 2)
print(sum(starmap(wins_player1, game_gen)))
```

55 Lychrel Numbers

Problem [Project Euler]

If we take 47, reverse and add, $47 + 74 = 121$, which is palindromic.
Not all numbers produce palindromes so quickly. For example,

$$\begin{aligned} 349 + 943 &= 1292 \\ 1292 + 2921 &= 4213 \\ 4213 + 3124 &= 7337 \end{aligned}$$

That is, 349 took three iterations to arrive at a palindrome.
Although no one has proved it yet, it is thought that some numbers, like 196, never produce a palindrome. A number that never forms a palindrome through the reverse and add process is called a Lychrel number. Due to the theoretical nature of these numbers, and for the purpose of this problem, we shall assume that a number is Lychrel until proven otherwise. In addition, you are given that for every number below ten-thousand, it will either (i) become a palindrome in less than fifty iterations, or, (ii) no one, with all the computing power that exists, has managed so far to map it to a palindrome. In fact, 10677 is the first number to be shown to require over fifty iterations before producing a palindrome: 4668731596684224866951378664 (53 iterations, 28-digits).
Surprisingly, there are palindromic numbers that are themselves Lychrel numbers; the first example is 4994.
How many Lychrel numbers are there below ten-thousand?

Solution [Buy me a beer]

To solve this, we:

1. Define a helper function `is_palindromic(n)` that checks if a number n is palindromic by converting it to a string and comparing it to its reverse.

2. Define the main function `is_lychrel(n)` to determine if n is a Lychrel number:

   ▷ For up to fifty iterations, we add n to its reverse.

   ▷ After each addition, we check if the result is palindromic using `is_palindromic`.

- ▷ If a palindrome is produced at any point, we return `False`, indicating n is not a Lychrel number.
- ▷ If no palindrome is produced within fifty iterations, we return `True`, indicating that n is a Lychrel number.

3. Use `sum(map(is_lychrel, range(10_000)))` to count all numbers from 1 to 9999 that are Lychrel numbers.

Code [GitHub]

```python
def is_palindromic(n):
    return str(n) == str(n)[::-1]

def is_lychrel(n):
    for _ in range(50):
        n += int(str(n)[::-1])
        if is_palindromic(n): return False
    return True

print(sum(map(is_lychrel, range(10_000))))
```

56 Powerful Digit Sum

Problem [Project Euler]

A googol (10^{100}) is a massive number: one followed by one-hundred zeros; 100^{100} is almost unimaginably large: one followed by two-hundred zeros. Despite their size, the sum of the digits in each number is only 1.
Considering natural numbers of the form a^b, where $a, b < 100$, what is the maximum digital sum?

Solution [Buy me a beer]

To solve this, we:

1. Define a helper function `digit_sum(n)` that calculates the sum of the digits of n by converting n to a string, mapping each character to an integer, and summing the resulting values.

2. Initialize `max_dsum` to 0, which will store the maximum digital sum found.

3. Use `product(range(100), repeat=2)` from `itertools` to iterate over all pairs (a, b) where $0 \leq a < 100$ and $0 \leq b < 100$.

4. For each pair (a, b):

   ▷ Calculate a^b.

   ▷ Compute the digital sum of a^b using `digit_sum`.

   ▷ Update `max_dsum` to the maximum of the current `max_dsum` and the calculated digital sum.

Finally, we print `max_dsum`, which is the maximum digital sum for any a^b with $a, b < 100$.

Code [GitHub]

```python
from itertools import product

def digit_sum(n): return sum(map(int, str(n)))

max_dsum = 0
for a, b in product(range(100), repeat=2):
    max_dsum = max(max_dsum, digit_sum(a**b))

print(max_dsum)
```

57 Square Root Convergents

Problem [Project Euler]

It is possible to show that the square root of two can be expressed as an infinite continued fraction.

$$\sqrt{2} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \dots}}}$$

By expanding this for the first four iterations, we get:

$$1 + \frac{1}{2} = \frac{3}{2} = 1.5$$
$$1 + \frac{1}{2 + \frac{1}{2}} = \frac{7}{5} = 1.4$$

$$1 + \frac{1}{2 + \frac{1}{2+\frac{1}{2}}} = \frac{17}{12} = 1.41666\ldots$$

$$1 + \frac{1}{2 + \frac{1}{2+\frac{1}{2+\frac{1}{2}}}} = \frac{41}{29} = 1.41379\ldots$$

The next three expansions are $\frac{99}{70}$, $\frac{239}{169}$, and $\frac{577}{408}$, but the eighth expansion, $\frac{1393}{985}$, is the first example where the number of digits in the numerator exceeds the number of digits in the denominator.
In the first one-thousand expansions, how many fractions contain a numerator with more digits than the denominator?

Solution [Buy me a beer]

To solve this, we:

1. Use Python's `Fraction` class from the `fractions` module to maintain exact fractions.
2. Initialize a list `convergents` with the value 1, representing the first convergent. This list will store the expanded fractions.
3. Initialize `count` to 0, which will keep track of the number of times the numerator has more digits than the denominator.
4. For each expansion up to 1000:
   - ▷ Calculate the next convergent as:

     $$1 + \frac{1}{1 + \texttt{convergents}[-1]}$$

   - ▷ Extract the numerator and denominator of this fraction using `new_conv.numerator` and `new_conv.denominator`.
   - ▷ Convert both the numerator and denominator to strings and check if the numerator has more digits than the denominator. If it does, increment `count`.
   - ▷ Append `new_conv` to `convergents` for use in the next iteration.

After processing all 1000 expansions, we print `count`, which represents the total number of times the numerator has more digits than the denominator within the first 1000 expansions.

Code [GitHub]

```python
from fractions import Fraction

convergents, count = [1], 0
for _ in range(1000):
    new_conv = 1 + Fraction(1, 1 + convergents[-1])
    n, d = new_conv.numerator, new_conv.denominator
    if len(str(n)) > len(str(d)): count += 1
    convergents.append(new_conv)

print(count)
```

58 Spiral Primes

Problem [Project Euler]

Starting with 1 and spiraling anticlockwise in the following way, a square spiral with side length 7 is formed.

37	36	35	34	33	32	**31**
38	**17**	16	15	14	**13**	30
39	18	**5**	4	**3**	12	29
40	19	6	1	2	11	28
41	20	**7**	8	9	10	27
42	21	22	23	24	25	26
43	44	45	46	47	48	49

It is interesting to note that the odd squares lie along the bottom right diagonal, but what is more interesting is that 8 out of the 13 numbers lying along both diagonals are prime; that is, a ratio of $\frac{8}{13} \approx 62\%$.
If one complete new layer is wrapped around the spiral above, a square spiral with side length 9 will be formed. If this process is continued, what is the side length of the square spiral for which the ratio of primes along both diagonals first falls below 10%?

Solution [Buy me a beer]

To solve this, we:

1. Define a generator function, `corner_gen()`, to yield the relevant corner values for each layer in the spiral:
   - ▷ For each odd side length n starting from 3, we calculate n^2, which is the bottom-right corner of the current layer.
   - ▷ We determine the step size as:
     $$\text{step} = \frac{n^2 - (n-2)^2}{4}.$$
   - ▷ We iterate over each corner in the layer, yielding only the top-right, top-left, and bottom-left corners (ignoring the bottom-right corner, as it is always a perfect square and therefore not prime for $n > 1$).
   - ▷ For each corner, we decrement `sq` by `step` to get the next corner.
2. Initialize `prime_count` to 0, which will track the number of primes found among the diagonal numbers.
3. Use a loop with `batched` to process values from `corner_gen()` in groups of three (one for each of the three relevant corners).
4. For each batch, we count the primes using `isprime` from `sympy` and add the result to `prime_count`.
5. Calculate the total count of numbers along the diagonals as $4i + 1$, where i is the current layer index. This is because each layer adds four new numbers to the diagonals.
6. Check if the ratio of primes along the diagonals, `prime_count`$/(4i+1)$, falls below 0.1. If it does, break out of the loop, as we have found the required side length.

Finally, we print $2i + 1$, which gives the side length of the spiral when the prime ratio first falls below 10%.
This approach is efficient because it selectively generates and checks only the necessary diagonal values for primality, stops as soon as the ratio requirement is met, and avoids redundant calculations.

Code [GitHub]

```python
from itertools import count, batched
from sympy import isprime

def corner_gen():
    for n in count(3, 2):
        sq = n**2
        step = (sq - (n - 2)**2) // 4
        for _ in range(3):
            sq -= step
            yield sq

prime_count = 0
for i, corners in enumerate(batched(corner_gen(), 3), 1):
    prime_count += sum(map(isprime, corners))
    if prime_count / (4*i + 1) < 0.1: break

print(2*i + 1)
```

59 XOR Decryption

Problem [Project Euler]

Each character on a computer is assigned a unique code, and the preferred standard is ASCII (American Standard Code for Information Interchange). For example, uppercase A = 65, asterisk (*) = 42, and lowercase k = 107.
A modern encryption method is to take a text file, convert the bytes to ASCII, then XOR each byte with a given value, taken from a secret key. The advantage of the XOR function is that using the same encryption key on the cipher text restores the plain text; for example, $65 \operatorname{XOR} 42 = 107$, then $107 \operatorname{XOR} 42 = 65$.
For unbreakable encryption, the key is the same length as the plain text message, and the key is made up of random bytes. The user would keep the encrypted message and the encryption key in different locations, and without both "halves", it is impossible to decrypt the message.
Unfortunately, this method is impractical for most users, so the modified method is to use a password as a key. If the password is shorter than the message, which is likely, the key is repeated cyclically throughout the message. The balance for this method is using a sufficiently long password key for security, but short enough to be memorable.

Your task has been made easy, as the encryption key consists of three lowercase characters. Using 0059_cipher.txt, a file containing the encrypted ASCII codes, and the knowledge that the plain text must contain common English words, decrypt the message and find the sum of the ASCII values in the original text.

Solution [Buy me a beer]

To solve this, we:

1. Load the encrypted ASCII codes from the file `"0059_cipher.txt"` and store them as a list of integers in `cipher`.
2. Define `allowed`, a set of printable characters to ensure that only valid characters appear in the decrypted text.
3. Define `words`, a list of common English words that we expect to find in the decrypted message, such as "the" and "is".
4. Use `product(map(ord, letters), repeat=3)` to generate all possible three-character keys by mapping each lowercase letter to its ASCII code and cycling through all combinations of three letters.
5. For each key `password`:
   - ▷ Set up a repeating sequence `rep` of the key to match the length of the message.
   - ▷ Use the XOR operation between each ASCII code in the encrypted message and the corresponding repeated key value, converting the result to a character using `chr`.
   - ▷ Check if each decrypted character is in `allowed`. If any character is not allowed, we discard this key and move to the next one.
   - ▷ If all characters in the decrypted message are valid and the message contains all the common English words in `words`, we have found the correct key.
6. Finally, calculate the sum of the ASCII values of the decrypted text by applying `ord` to each character in the plaintext and summing the results.

This approach works by systematically generating and testing each potential three-character key and checking the decrypted message for validity. By leveraging the knowledge of common English words, we ensure that we select the correct key. The result, `sum(map(ord, plain))`, gives the sum of ASCII values in the original decrypted message.

Code [GitHub]

```python
from itertools import count, product
from string import ascii_lowercase as letters, printable
from operator import xor

with open("0059_cipher.txt") as f:
    cipher = list(map(int, f.read().split(',')))

allowed = set(printable[:94] + ' ')
words = ["the", "of", "is", "by", "to", "this"]

pass_gen = product(map(ord, letters), repeat=3)
for password in pass_gen:
    plain, rep = "", (password[i % 3] for i in count())
    for char in map(chr, map(xor, cipher, rep)):
        if char not in allowed: break
        plain += char
    else:
        if all(word in plain for word in words): break

print(sum(map(ord, plain)))
```

60 Prime Pair Sets

Problem [Project Euler]

The primes 3, 7, 109, and 673, are quite remarkable. By taking any two primes and concatenating them in any order, the result will always be prime. For example, taking 7 and 109, both 7109 and 1097 are prime. The sum of these four primes, 792, represents the lowest sum for a set of four primes with this property.
Find the lowest sum for a set of five primes for which any two primes concatenate to produce another prime.

Solution [Buy me a beer]

To solve this, we:

1. Generate a list of primes up to 10^4 using `primerange` from `sympy`. We convert each prime to a string for easy concatenation.

2. Define a helper function `combines(p, q)` that checks if two primes p and q form a prime when concatenated in both orders, $p + q$ and $q + p$. This function returns `True` if both concatenations are prime, and `False` otherwise.

3. Use a `defaultdict` of sets, `friends`, to store pairs of primes that pass the concatenation test. For each pair of primes (p, q), if `combines(p, q)` is `True`, we add q to the set of friends for p and p to the set of friends for q.

4. Define a recursive function `groups(group)` that attempts to build groups of five primes:

   ▷ If the length of the current group reaches five, we yield the group as a valid solution.

   ▷ Otherwise, we find the intersection of sets in `friends` for each prime in `group`. This intersection gives the primes that can be added to `group` without violating the concatenation condition.

   ▷ For each new prime in this intersection, we recursively call `groups` with the new prime added to the group.

5. Initialize `min_sum` to infinity. We then iterate over each prime p, using `groups((p,))` to find all valid groups starting with p.

6. For each valid group, we calculate the sum of its elements. If this sum is less than `min_sum`, we update `min_sum`.

Finally, we print `min_sum`, which represents the lowest sum of a set of five primes where any two primes concatenate to form another prime. This approach is efficient due to the use of intersections to limit candidates and recursive grouping to build sets satisfying the required conditions.

Code [GitHub]

```python
from sympy import primerange, isprime
from itertools import combinations
from collections import defaultdict

primes = list(map(str, primerange(3, 10**4)))

def combines(p, q):
    return isprime(int(p + q)) and isprime(int(q + p))

friends = defaultdict(set)
for p, q in combinations(primes, 2):
    if combines(p, q):
        friends[p].add(q)
        friends[q].add(p)

def groups(group):
    if len(group) == 5:
        yield group
        return
    common = set.intersection(*(friends[p] for p in group))
    for new_p in common: yield from groups(group + (new_p,))

min_sum = float('inf')
for p in primes:
    for group in groups((p,)):
        if (s := sum(map(int, group))) > min_sum: continue
        min_sum = s

print(min_sum)
```

61 Cyclical Figurate Numbers

Problem [Project Euler]

Triangle, square, pentagonal, hexagonal, heptagonal, and octagonal numbers are all figurate (polygonal) numbers and are generated by the following formulae:

Triangle	$P_{3,n} = \frac{n(n+1)}{2}$	$1, 3, 6, 10, 15, \ldots$
Square	$P_{4,n} = n^2$	$1, 4, 9, 16, 25, \ldots$
Pentagonal	$P_{5,n} = \frac{n(3n-1)}{2}$	$1, 5, 12, 22, 35, \ldots$
Hexagonal	$P_{6,n} = n(2n-1)$	$1, 6, 15, 28, 45, \ldots$
Heptagonal	$P_{7,n} = \frac{n(5n-3)}{2}$	$1, 7, 18, 34, 55, \ldots$
Octagonal	$P_{8,n} = n(3n-2)$	$1, 8, 21, 40, 65, \ldots$

The ordered set of three 4-digit numbers: 8128, 2882, 8281, has three interesting properties:

1. The set is cyclic, in that the last two digits of each number is the first two digits of the next number (including the last number with the first).

2. Each polygonal type: triangle ($P_{3,127} = 8128$), square ($P_{4,91} = 8281$), and pentagonal ($P_{5,44} = 2882$), is represented by a different number in the set.

3. This is the only set of 4-digit numbers with this property.

Find the sum of the only ordered set of six cyclic 4-digit numbers for which each polygonal type: triangle, square, pentagonal, hexagonal, heptagonal, and octagonal, is represented by a different number in the set.

Solution [Buy me a beer]

To solve this, we:

1. Initialize a `defaultdict numbers` where each key k (ranging from 3 to 8) corresponds to a polygonal type (triangle, square, etc.). Each entry will hold a list of tuples containing the polygonal type k and its corresponding four-digit number.

2. For each figurate type k, generate the polygonal numbers using the

formula:

$$P_{k,n} = \frac{n((k-2)n-k+4)}{2}.$$

3. For each number x generated:
   - ▷ Stop generating numbers when $x \geq 10^4$ (as we are only interested in four-digit numbers).
   - ▷ Store numbers between 10^3 and 10^4 in `numbers[k]` along with their type k and string representation of the number.
4. Initialize `paths` as a list of paths, each starting with an octagonal number, as these will be the base numbers of our sequences.
5. For each step, extend existing paths by iterating over all available four-digit figurate numbers:
   - ▷ If a number's type k is already in the path, skip it to ensure distinct figurate types.
   - ▷ Check the cyclic condition: if the first two digits of the new number match the last two digits of the current path's last number, append it to `new_paths`.
6. After constructing six-number paths, filter paths to keep only those that form a closed cycle (i.e., the last two digits of the final number in the path match the first two digits of the first number).
7. Calculate the result by summing the numeric values of the nodes in the first valid cyclic path.

Finally, we print `result`, which is the sum of the numbers in the unique six-term cyclic sequence.
This approach ensures that we explore only valid cyclic paths of distinct polygonal types, checking both the numeric constraints and the required cyclic property.

Code [GitHub]

```python
from itertools import count, chain
from collections import defaultdict

numbers = defaultdict(list)
for k in range(3, 9):
    for n in count():
        if (x := n*((k - 2)*n - k + 4) // 2) >= 10**4: break
        if x > 10**3: numbers[k].append((k, str(x)))

paths = [[x] for x in numbers[8]]
for _ in range(5):
    new_paths = []
    for path in paths:
        for k, x in chain(*numbers.values()):
            K = set(node[0] for node in path)
            if k in K: continue
            if x[:2] == path[-1][1][2:]:
                new_paths.append(path + [(k, x)])
    paths = new_paths

paths = [p for p in paths if p[-1][1][2:] == p[0][1][:2]]
result = sum(int(node[1]) for node in paths[0])
print(result)
```

62 Cubic Permutations

Problem [Project Euler]

The cube, 41063625 (345^3), can be permuted to produce two other cubes: 56623104 (384^3) and 66430125 (405^3). In fact, 41063625 is the smallest cube which has exactly three permutations of its digits which are also cubes. Find the smallest cube for which exactly five permutations of its digits are cubes.

Solution [Buy me a beer]

To solve this, we approach the problem as follows:

1. Initialize a `defaultdict` called `cubes`, where each key is a tuple representing the sorted digits of a cube, and each value is a set of cubes that share this digit pattern.
2. Set a limit (`LIMIT`) to 10^4 to check cubes for values of n up to 9999.
3. For each integer n from 0 to `LIMIT`:
   - ▷ Calculate n^3.
   - ▷ Sort the digits of n^3 and use this sorted tuple of digits as the key in `cubes`.
   - ▷ Add n^3 to the set associated with this key in `cubes`, effectively grouping cubes that are permutations of each other.
4. Initialize an empty set `new_cubes` to store the cubes that meet the condition.
5. Iterate over the values in `cubes`. For each set of cubes s:
   - ▷ If the length of s is exactly 5, add all cubes in s to `new_cubes`.
6. Finally, output the smallest value in `new_cubes` using `min(new_cubes)`, which gives the smallest cube with exactly five cubic permutations.

Code [GitHub]

```python
from collections import defaultdict

LIMIT = 10**4

cubes = defaultdict(set)
for n in range(LIMIT):
    digits = sorted(str(n**3))
    cubes[tuple(digits)].add(n**3)

new_cubes = set()
for s in cubes.values():
    if len(s) == 5: new_cubes |= s

print(min(new_cubes))
```

63 Powerful Digit Counts

Problem [Project Euler]

The 5-digit number, $16807 = 7^5$, is also a fifth power. Similarly, the 9-digit number, $134217728 = 8^9$, is a ninth power.
How many n-digit positive integers exist which are also an nth power?

Solution [Buy me a beer]

This problem becomes simpler with a key insight: an n-digit n-th power must satisfy the inequality $10^{n-1} \leq x^n < 10^n$. This inequality implies that:

- ▷ The base x must be less than 10, since higher bases would exceed the n-digit constraint.
- ▷ The exponent n has an upper bound of $\frac{1}{1-\log_{10} 9} \approx 21.85$, meaning we only need to consider n values up to 21.

With this insight, we can solve the problem by:

1. Initializing a counter `powers` to record the number of valid n-digit n-th powers.
2. Iterating over each base from 1 to 9 (since bases of 10 or higher would not satisfy the n-digit constraint).
3. For each base, looping over possible exponents n from 1 to 21.
4. Calculating x^n and checking if it has exactly n digits by comparing n to the length of `str`(x^n):
    - ▷ If x^n has n digits, we increment `powers`.

After examining all combinations of bases and exponents within these constraints, `powers` holds the total number of n-digit integers that are also n-th powers. The result, printed as the final answer, provides the count of all such numbers.

Code [GitHub]

```python
powers = 0
for base in range(1, 10):
    for n in range(1, 22):
        if len(str(base**n)) == n:
            powers += 1

print(powers)
```

64 Odd Period Square Roots

Problem [Project Euler]

All square roots are periodic when written as continued fractions and can be written in the form:

$$\sqrt{N} = a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \ldots}}}$$

For example, let us consider $\sqrt{23}$:

$$\sqrt{23} = 4 + \sqrt{23} - 4 = 4 + \cfrac{1}{\frac{1}{\sqrt{23}-4}} = 4 + \cfrac{1}{1 + \frac{\sqrt{23}-3}{7}}$$

If we continue, we would get the following expansion:

$$\sqrt{23} = 4 + \cfrac{1}{1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{8 + \ldots}}}}$$

The process can be summarised as follows:

$$\begin{aligned}
a_0 &= 4, \quad \frac{1}{\sqrt{23}-4} = \frac{\sqrt{23}+4}{7} = 1 + \frac{\sqrt{23}-3}{7} \\
a_1 &= 1, \quad \frac{7}{\sqrt{23}-3} = \frac{7(\sqrt{23}+3)}{14} = 3 + \frac{\sqrt{23}-3}{2} \\
a_2 &= 3, \quad \frac{2}{\sqrt{23}-3} = \frac{2(\sqrt{23}+3)}{14} = 1 + \frac{\sqrt{23}-4}{7} \\
a_3 &= 1, \quad \frac{7}{\sqrt{23}-4} = \frac{7(\sqrt{23}+4)}{7} = 8 + \sqrt{23} - 4
\end{aligned}$$

$$a_4 = 8, \quad \frac{1}{\sqrt{23}-4} = \frac{\sqrt{23}+4}{7} = 1 + \frac{\sqrt{23}-3}{7}$$

$$a_5 = 1, \quad \frac{7}{\sqrt{23}-3} = \frac{7(\sqrt{23}+3)}{14} = 3 + \frac{\sqrt{23}-3}{2}$$

$$a_6 = 3, \quad \frac{2}{\sqrt{23}-3} = \frac{2(\sqrt{23}+3)}{14} = 1 + \frac{\sqrt{23}-4}{7}$$

$$a_7 = 1, \quad \frac{7}{\sqrt{23}-4} = \frac{7(\sqrt{23}+4)}{7} = 8 + \sqrt{23}-4$$

It can be seen that the sequence is repeating. For conciseness, we use the notation $\sqrt{23} = [4;(1,3,1,8)]$, to indicate that the block (1,3,1,8) repeats indefinitely.
The first ten continued fraction representations of (irrational) square roots are:

$$\begin{aligned}
\sqrt{2} &= [1;(2)], \quad \text{period} = 1\\
\sqrt{3} &= [1;(1,2)], \quad \text{period} = 2\\
\sqrt{5} &= [2;(4)], \quad \text{period} = 1\\
\sqrt{6} &= [2;(2,4)], \quad \text{period} = 2\\
\sqrt{7} &= [2;(1,1,1,4)], \quad \text{period} = 4\\
\sqrt{8} &= [2;(1,4)], \quad \text{period} = 2\\
\sqrt{10} &= [3;(6)], \quad \text{period} = 1\\
\sqrt{11} &= [3;(3,6)], \quad \text{period} = 2\\
\sqrt{12} &= [3;(2,6)], \quad \text{period} = 2\\
\sqrt{13} &= [3;(1,1,1,1,6)], \quad \text{period} = 5
\end{aligned}$$

Exactly four continued fractions, for $N \leq 13$, have an odd period.
How many continued fractions for $N \leq 10,000$ have an odd period?

Solution [Buy me a beer]

We proceed as follows:

1. Define a helper function, `period(n)`, that calculates the period of the continued fraction for $\sqrt{n}$.

2. Initialize variables p and q as follows:

   ▷ p starts as the integer part of $\sqrt{n}$, and q as 1.

   ▷ If $\sqrt{n}$ is an integer (i.e., n is a perfect square), the function returns 0 since there is no period.

3. For each iteration:

   ▷ Update q using $q = \frac{n-p^2}{q}$.

   ▷ Calculate the integer part, `floor`, of $\frac{\sqrt{n}+p}{q}$, representing the next term in the continued fraction.

   ▷ Update p as $p = -(p - \texttt{floor} \cdot q)$.

   ▷ Check if the state (p, q) has occurred before (stored in the dictionary `states`). If it has, the sequence has started to repeat, and the period length is the difference between the current iteration and the recorded index of this state.

   ▷ Store the current state in `states` with the current iteration as its value.

4. Finally, calculate the answer by iterating through values $n = 1$ to 10,000 and summing those for which `period(n)` is odd.

This approach allows us to efficiently calculate the period of each continued fraction for values up to 10,000 and count those with odd periods. The result is printed as the answer.

Code [GitHub]

```python
from math import sqrt
from itertools import count

def period(n):
    p, q, states = int(root := sqrt(n)), 1, {}
    if root.is_integer(): return 0
    for i in count(1):
        q = (n - p**2) / q
        floor = int((root + p) / q)
        p = -1 * (p - (floor * q))
        if (p, q) in states: return i - states[p, q]
        states[p, q] = i

print(sum(1 for n in range(10**4) if period(n) % 2))
```

65 Convergents of e

Problem [Project Euler]

The square root of 2 can be written as an infinite continued fraction.

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \dots}}}}$$

The infinite continued fraction can be written, $\sqrt{2} = [1; (2)]$, where (2) indicates that 2 repeats *ad infinitum*. In a similar way, $\sqrt{23} = [4; (1, 3, 1, 8)]$. It turns out that the sequence of partial values of continued fractions for square roots provides the best rational approximations. Let us consider the convergents for $\sqrt{2}$.

$$1 + \frac{1}{2} = \frac{3}{2}$$
$$1 + \cfrac{1}{2 + \cfrac{1}{2}} = \frac{7}{5}$$
$$1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2}}} = \frac{17}{12}$$
$$1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2}}}} = \frac{41}{29}$$

Hence, the sequence of the first ten convergents for $\sqrt{2}$ are:

$$1, \frac{3}{2}, \frac{7}{5}, \frac{17}{12}, \frac{41}{29}, \frac{99}{70}, \frac{239}{169}, \frac{577}{408}, \frac{1393}{985}, \frac{3363}{2378}, \ldots$$

What is most surprising is that the important mathematical constant,

$$e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, \ldots, 1, 2k, 1, \ldots]$$

The first ten terms in the sequence of convergents for e are:

$$2, 3, \frac{8}{3}, \frac{11}{4}, \frac{19}{7}, \frac{87}{32}, \frac{106}{39}, \frac{193}{71}, \frac{1264}{465}, \frac{1457}{536}, \ldots$$

The sum of digits in the numerator of the 10th convergent is $1+4+5+7 = 17$. Find the sum of digits in the numerator of the 100th convergent of the continued fraction for e.

Solution [Buy me a beer]

To solve this, we:

1. Define a function `coeff(k)` to generate the coefficients of the continued fraction for e based on the index k:
   - ▷ If $k = 0$, `coeff` returns 2, the first term in the sequence.
   - ▷ For other values of k, if $(k+1) \mod 3 = 0$, we return $2 \times (k+1)/3$; otherwise, the coefficient is 1.
2. Define a function `conv(n)` to compute the n-th convergent of the continued fraction using recursion:

- ▷ If $n = 0$, the convergent is just the fraction with the coefficient from `coeff(0)`.
- ▷ Otherwise, we calculate the n-th convergent as `coeff(n)` $+ \frac{1}{\texttt{conv(n - 1)}}$, building the fraction structure iteratively.

3. Compute the 100-th convergent by calling `conv(99)`. The numerator of this fraction is then converted to a string, and we sum its digits using `sum(map(int, str(...)))`.

This approach leverages recursion and fractional arithmetic to build the convergents step-by-step. After obtaining the 100-th convergent, we sum the digits of its numerator and print the result, yielding the solution.

Code [GitHub]

```python
from fractions import Fraction as f

def coeff(k):
    if k == 0: return 2
    return 2 * (k + 1) // 3 if (k + 1) % 3 == 0 else 1

def conv(n):
    if n == 0: return f(coeff(0))
    else: return f(coeff(n)) + f(1, conv(n - 1))

print(sum(map(int, str(conv(99).numerator))))
```

66 Diophantine Equation

Problem [Project Euler]

Consider quadratic Diophantine equations of the form:

$$x^2 - Dy^2 = 1$$

For example, when $D = 13$, the minimal solution in x is $649^2 - 13 \times 180^2 = 1$.
It can be assumed that there are no solutions in positive integers when D is a square.
By finding minimal solutions in x for $D = \{2, 3, 5, 6, 7\}$, we obtain the following:

$$3^2 - 2 \times 2^2 = 1$$
$$2^2 - 3 \times 1^2 = 1$$
$$\mathbf{9}^2 - 5 \times 4^2 = 1$$
$$5^2 - 6 \times 2^2 = 1$$
$$8^2 - 7 \times 3^2 = 1$$

Hence, by considering minimal solutions in x for $D \leq 7$, the largest x is obtained when $D = 5$.
Find the value of $D \leq 1000$ in minimal solutions of x for which the largest value of x is obtained.

Solution [Buy me a beer]

This problem requires finding the value of $D \leq 1000$ for which the minimal solution of x in the equation $x^2 - Dy^2 = 1$ (a form of Pell's equation) is maximized. Pell's equation typically has integer solutions, and for non-square D, there exists a minimal solution in x and y that can be found through continued fraction expansions.
To solve this, we:

1. Define a helper function, `continued_fraction_gen(n)`, to generate the continued fraction terms for $\sqrt{n}$. These terms help us approximate $\sqrt{n}$ with convergents, leading to possible solutions for Pell's equation.
    - ▷ In `continued_fraction_gen(n)`, we initialize the denominator and integer part for generating each continued fraction term iteratively.
    - ▷ For each term, we calculate a new integer part using the formula `int(denominator * (sqrt(n) + integer_part) / (n - integer_part ** 2))` and update the denominator and integer part for the next iteration.
2. Initialize variables `x_max` and `D_max` to track the maximum value of x and the corresponding D that produces this value.
3. For each D from 2 to 1000 (skipping perfect squares), we calculate the continued fraction terms and use them to construct convergents until we find the minimal solution for x.
    - ▷ If the continued fraction length minus 2 is odd, we discard the last term; otherwise, we extend it with a repeat of the terms.
    - ▷ Using these terms, we build the convergent for x/y by setting the initial numerator and denominator, and iterating backward over the terms.

4. Finally, we check if x for the current D exceeds `x_max`. If it does, we update `x_max` and `D_max` with the new values.

The output is the value of D that produces the largest minimal x in the equation $x^2 - Dy^2 = 1$.

Code [GitHub]

```python
from math import sqrt, isqrt

def continued_fraction_gen(n):
    denominator, integer_part = 1, isqrt(n)
    current_term, final_term = 1, 2 * integer_part

    while current_term != final_term:
        current_term = int(denominator * (sqrt(n) + integer_part) / (n - integer_part ** 2))
        yield current_term
        denominator, integer_part = (n - integer_part ** 2) // denominator, \
                                    current_term * (n - integer_part ** 2) // denominator - integer_part

x_max, D_max = 0, None
for D in range(2, 1001):
    if sqrt(D).is_integer(): continue
    continued_fraction = [isqrt(D)] + list(continued_fraction_gen(D))

    if (len(continued_fraction) - 2) % 2 == 1: continued_fraction.pop()
    else: continued_fraction.extend(continued_fraction[1:-1])

    numerator, denominator = continued_fraction[-1], 1
    for term in reversed(continued_fraction[:-1]):
        numerator, denominator = term * numerator + denominator, numerator

    if numerator > x_max: x_max, D_max = numerator, D

print(D_max)
```

67 Maximum Path Sum II

Problem [Project Euler]

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

$$\begin{array}{ccccccc} & & & \mathbf{3} & & & \\ & & \mathbf{7} & & 4 & & \\ & 2 & & \mathbf{4} & & 6 & \\ 8 & & 5 & & \mathbf{9} & & 3 \end{array}$$

That is, $3 + 7 + 4 + 9 = 23$.
Find the maximum total from top to bottom in triangle.txt, a 15K text file containing a triangle with one-hundred rows.
NOTE: This is a much more difficult version of Problem 18. It is not possible to try every route to solve this problem, as there are 2^{99} altogether! If you could check one trillion (10^{12}) routes every second it would take over twenty billion years to check them all. There is an efficient algorithm to solve it. ;o)

Solution [Buy me a beer]

To solve this problem efficiently, we use a dynamic programming approach that works from the bottom of the triangle to the top. This method allows us to calculate the maximum path sum without having to evaluate each possible route individually, which would be computationally prohibitive.
Our solution begins by reading the triangle data from the file and converting it into a list of lists, where each list represents a row of the triangle.
The approach is as follows:

1. Starting from the second-to-last row of the triangle, we move upward.
2. For each element in the current row, we update it by adding the maximum of the two adjacent elements directly below it in the row below.
3. By the time we reach the top of the triangle, each element has been updated to represent the maximum path sum from that point to the bottom of the triangle.
4. The top element of the triangle then holds the maximum path sum from top to bottom.

The output of the program, T[0][0], provides the maximum total from the top to the bottom of the triangle.

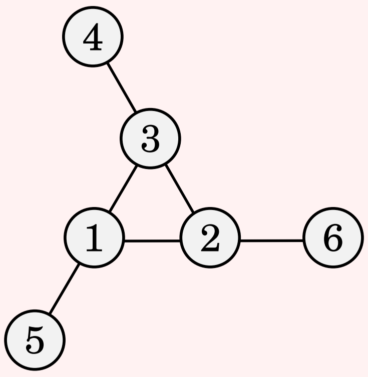
Code [GitHub]

```python
with open("0067_triangle.txt") as f:
    triangle = f.read()

T = [
    [int(n) for n in row.split()]
    for row in triangle.splitlines()
]

for i in range(len(T) - 2, -1, -1):
    for j in range(i + 1):
        T[i][j] += max(T[i + 1][j], T[i + 1][j + 1])

print(T[0][0])
```

68 Magic 5-gon Ring

Problem [Project Euler]

Consider the following "magic" 3-gon ring, filled with the numbers 1 to 6, and each line adding to nine.

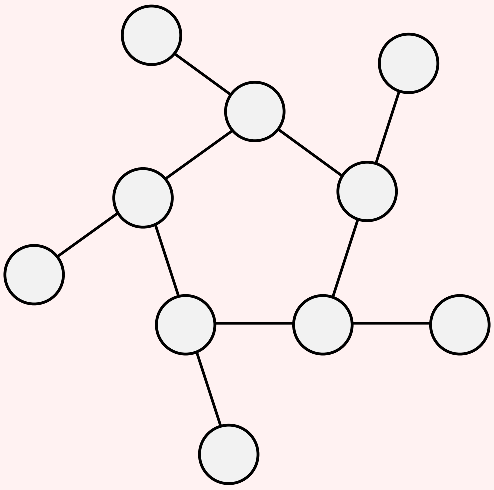


Working **clockwise**, and starting from the group of three with the numerically lowest external node (4,3,2 in this example), each solution can be described uniquely. For example, the above solution can be described by the set: 4,3,2; 6,2,1; 5,1,3.

It is possible to complete the ring with four different totals: 9, 10, 11, and 12. There are eight solutions in total.

Total	Solution Set
9	4,2,3; 5,3,1; 6,1,2
9	4,3,2; 6,2,1; 5,1,3
10	2,3,5; 4,5,1; 6,1,3
10	2,5,3; 6,3,1; 4,1,5
11	1,4,6; 3,6,2; 5,2,4
11	1,6,4; 5,4,2; 3,2,6
12	1,5,6; 2,6,4; 3,4,5
12	1,6,5; 3,5,4; 2,4,6

By concatenating each group, it is possible to form 9-digit strings; the maximum string for a 3-gon ring is 432621513.
Using the numbers 1 to 10, and depending on arrangements, it is possible to form 16- and 17-digit strings. What is the maximum **16-digit** string for a "magic" 5-gon ring?



Solution [Buy me a beer]

To solve this problem, we start by generating all permutations of the numbers 1 to 10, dividing each permutation into two groups: the outer and inner nodes of the ring.
By ensuring that the largest number, 10, is always in the outer ring, we guarantee that any resulting valid string will be exactly 16 digits long. This approach not only simplifies the search but also allows us to avoid constructing strings with lengths that don't match the required format.

Next, we calculate the sum of the numbers in one line of the ring, which we call the `target_sum`. For the configuration to be a "magic" ring, each of the five lines formed by a combination of an outer node, an inner node, and the next inner node in sequence must add up to this `target_sum`. We use the `all()` function to verify that each line meets this criterion. If not, we move on to the next permutation.
Once we identify a valid ring, we find the starting position by choosing the outer node with the smallest value. This ensures a unique representation of each solution, as the problem specifies.
Finally, we construct a string `string` by traversing the ring clockwise, starting from the determined outer node, and concatenate the values of each line in sequence. If this string is greater than our current maximum `maximum`, we update `maximum`. After checking all permutations, `maximum` holds the maximum 16-digit solution.

Code [GitHub]

```python
from itertools import permutations

maximum = ""
for x in permutations(range(1, 11), 10):
    out, inn = x[:5], x[5:]
    if 10 in inn: continue

    target_sum = out[4] + inn[4] + inn[0]
    if not all(out[i] + inn[i] + inn[(i + 1) % 5] == target_sum for i in range(4)): continue

    i0 = min(range(5), key=lambda i: out[i])
    string = "".join(
        f"{out[(i0+i)%5]}{inn[(i0+i)%5]}{inn[(i0+i+1)%5]}"
        for i in range(5)
    )

    if string > maximum: maximum = string

print(maximum)
```

69 Totient Maximum

Problem [Project Euler]

Euler's totient function, $\phi(n)$ (sometimes called the phi function), is defined as the number of positive integers not exceeding n which are relatively prime to n. For example, as 1, 2, 4, 5, 7, and 8 are all less than or equal to nine and relatively prime to nine, $\phi(9) = 6$.

n	**Relatively Prime**	$\phi(n)$	$n/\phi(n)$
2	1	1	2
3	1,2	2	1.5
4	1,3	2	2
5	1,2,3,4	4	1.25
6	1,5	2	3
7	1,2,3,4,5,6	6	1.1666...
8	1,3,5,7	4	2
9	1,2,4,5,7,8	6	1.5
10	1,3,7,9	4	2.5

It can be seen that $n = 6$ produces a maximum $n/\phi(n)$ for $n \leq 10$.
Find the value of $n \leq 1,000,000$ for which $n/\phi(n)$ is a maximum.

Solution [Buy me a beer]

To solve this problem, we aim to maximize the ratio $\frac{n}{\phi(n)}$ for values of $n \leq$ 1,000,000. Euler's totient function $\phi(n)$ counts the number of integers up to n that are relatively prime to n. Calculating $\phi(n)$ for all numbers up to our limit efficiently requires us to leverage properties of prime numbers.
First, we initialize an array `phi` where each index represents a number n, and each element is initialized with the value n itself. This array will eventually store the totient values for each n.
To compute $\phi(n)$ values, we iterate through each integer p from 2 up to the limit. For each p where $\phi[p] = p$ (indicating that p is prime), we know that p is a divisor of the numbers in its multiples. For each multiple k of p, we update $\phi[k]$ by scaling it by the factor $(p-1)/p$, which effectively removes the contribution of p as a divisor in $\phi(k)$.
After populating the `phi` array, we calculate the ratio $\frac{n}{\phi(n)}$ for each n from 2 to the limit. If this ratio exceeds the current maximum, we update our maximum value and record the corresponding n.

Finally, we output the value of n that produced the maximum ratio. This efficient approach allows us to avoid redundant calculations and use properties of prime factors to optimize our solution.

Code [GitHub]

```python
LIMIT = 10**6
phi = list(range(LIMIT + 1))

for p in range(2, LIMIT + 1):
    if phi[p] == p:
        for k in range(p, LIMIT + 1, p):
            phi[k] *= p - 1
            phi[k] //= p

maximum = 0
for n in range(2, LIMIT + 1):
    if (ratio := n / phi[n]) > maximum:
        maximum, n_max = ratio, n

print(n_max)
```

70 Totient Permutation

Problem [Project Euler]

Euler's totient function, $\phi(n)$ (sometimes called the phi function), is used to determine the number of positive numbers less than or equal to n which are relatively prime to n. For example, as $1, 2, 4, 5, 7$, and 8, are all less than nine and relatively prime to nine, $\phi(9) = 6$.
The number 1 is considered to be relatively prime to every positive number, so $\phi(1) = 1$.
Interestingly, $\phi(87109) = 79180$, and it can be seen that 87109 is a permutation of 79180.
Find the value of n, $1 < n < 10^7$, for which $\phi(n)$ is a permutation of n and the ratio $n/\phi(n)$ produces a minimum.

Solution [Buy me a beer]

We start by initializing arrays to store Euler's totient values and their ratios. The sieving process for computing `phi` values is the same as in Problem 69: for each prime p, we iterate over its multiples and adjust each multiple based on p to accurately reflect its count of coprimes. Once the `phi` values are computed, we calculate the ratio $\frac{n}{\phi(n)}$ for each n, storing it for later comparison.

To determine if $\phi(n)$ is a permutation of n, we define a helper function that compares digit frequencies of n and $\phi(n)$. As we iterate through the values, we check this permutation condition and update our minimum ratio whenever we find a smaller value that satisfies it, tracking the corresponding n.

By the end, the variable `n_min` holds the integer n that both meets the permutation condition and minimizes $\frac{n}{\phi(n)}$, which we then print as our result.

Code [GitHub]

```python
LIMIT = 10**7
phi = list(range(LIMIT + 1))
ratio = {}

for p in range(2, LIMIT + 1):
    if phi[p] == p:
        for k in range(p, LIMIT + 1, p):
            phi[k] *= p - 1
            phi[k] //= p
    ratio[p] = p / phi[p]

def is_permutation(n):
    return sorted(str(n)) == sorted(str(phi[n]))

minimum = float('inf')
for n in range(2, LIMIT + 1):
    if ratio[n] < minimum and is_permutation(n):
        minimum, n_min = ratio[n], n

print(n_min)
```

71 Ordered Fractions

Problem [Project Euler]

Consider the fraction, $\frac{n}{d}$, where n and d are positive integers. If $n < d$ and HCF$(n, d) = 1$, it is called a reduced proper fraction.
If we list the set of reduced proper fractions for $d \le 8$ in ascending order of size, we get:

$$\frac{1}{8}, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \frac{3}{8}, \mathbf{\frac{2}{5}}, \frac{3}{7}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{5}{8}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, \frac{7}{8}$$

It can be seen that $\frac{2}{5}$ is the fraction immediately to the left of $\frac{3}{7}$.
By listing the set of reduced proper fractions for $d \le 1\,000\,000$ in ascending order of size, find the numerator of the fraction immediately to the left of $\frac{3}{7}$.

Solution [Buy me a beer]

We solve this problem by hand using properties of the Farey sequence and fraction approximation to efficiently find the fraction immediately to the left of $\frac{3}{7}$ among reduced proper fractions with denominators up to $d \le 1\,000\,000$.
To approximate $\frac{3}{7}$ as closely as possible from the left, we choose a denominator d close to our limit. We calculate $k = \left\lfloor \frac{10^6}{7} \right\rfloor = 142\,857$, which is the largest integer we can multiply by 7 without exceeding 10^6. We then set $d = 7 \times k = 999\,999$, giving us a fraction with denominator just below the limit and very close to $\frac{3}{7}$.
This choice yields the fraction $\frac{428\,570}{999\,999}$, where the numerator $n = 3 \times k - 1 = 428\,570$ approximates $\frac{3}{7}$ as closely as possible from the left. Notice that since $\frac{3 \times k}{999\,999} = \frac{3}{7}$, the fraction $\frac{428\,570}{999\,999}$ is extremely close to $\frac{3}{7}$ while remaining just to the left.
To confirm that $\frac{428\,570}{999\,999}$ is in lowest terms, we check $\gcd(428\,570, 999\,999) = 1$, ensuring it is a reduced proper fraction. This makes $\frac{428\,570}{999\,999}$ the closest valid fraction immediately to the left of $\frac{3}{7}$ in the Farey sequence for $d \le 1\,000\,000$.

Code [GitHub]

```python
from math import gcd

LIMIT = 10**6

k = LIMIT // 7
n, d = 3*k - 1, 7*k

if gcd(n, d) == 1: print(n)
```

72 Counting Fractions

Problem [Project Euler]

Consider the fraction, $\frac{n}{d}$, where n and d are positive integers. If $n < d$ and $\text{HCF}(n, d) = 1$, it is called a reduced proper fraction.
If we list the set of reduced proper fractions for $d \le 8$ in ascending order of size, we get:

$$\frac{1}{8}, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \frac{3}{8}, \frac{2}{5}, \frac{3}{7}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{5}{8}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, \frac{7}{8}$$

It can be seen that there are 21 elements in this set.
How many elements would be contained in the set of reduced proper fractions for $d \le 1\,000\,000$?

Solution [Buy me a beer]

To solve this problem, we use Euler's totient function $\phi(n)$, which counts the number of integers less than n that are coprime to n. For each denominator d, the totient value $\phi(d)$ gives the count of reduced proper fractions of the form $\frac{n}{d}$ where $n < d$ and $\text{HCF}(n, d) = 1$.
To determine the total number of reduced proper fractions with denominators up to a given limit, we sum $\phi(d)$ for each d from 2 to the limit. This gives:

$$\sum_{n=2}^{\texttt{LIMIT}} \varphi(n) = \Phi(\texttt{LIMIT}) - 1$$

where $\Phi(n)$ is the cumulative sum of the totient values up to n, representing the total count of reduced proper fractions. The subtraction by 1 excludes the fraction $\frac{1}{1}$, which is not part of our count of fractions with $d \geq 2$. So, by summing the values in `phi` from 2 to `LIMIT`, we obtain the solution, `result`, which we print as the answer.

Code [GitHub]

```python
LIMIT = 10**6
phi = list(range(LIMIT + 1))

for p in range(2, LIMIT + 1):
    if phi[p] == p:
        for k in range(p, LIMIT + 1, p):
            phi[k] *= p - 1
            phi[k] //= p

result = sum(phi[n] for n in range(2, LIMIT + 1))
print(result)
```

73 Counting Fractions in a Range

Problem [Project Euler]

Consider the fraction, $\dfrac{n}{d}$, where n and d are positive integers. If $n < d$ and $\mathrm{HCF}(n, d) = 1$, it is called a reduced proper fraction.
If we list the set of reduced proper fractions for $d \leq 8$ in ascending order of size, we get:

$$\frac{1}{8}, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \mathbf{\frac{3}{8}}, \mathbf{\frac{2}{5}}, \mathbf{\frac{3}{7}}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{5}{8}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, \frac{7}{8}$$

It can be seen that there are 3 fractions between $\dfrac{1}{3}$ and $\dfrac{1}{2}$.

How many fractions lie between $\dfrac{1}{3}$ and $\dfrac{1}{2}$ in the sorted set of reduced proper fractions for $d \leq 12\,000$?

Solution [Buy me a beer]

First, we set a variable `LIMIT` to 12000, as specified in the problem. This allows us to loop over each possible denominator d from 2 up to 12000.
For each denominator d, we define a range of numerators n such that the fraction $\frac{n}{d}$ is between $\frac{1}{3}$ and $\frac{1}{2}$:

- ▷ To ensure $\frac{n}{d} > \frac{1}{3}$, we calculate the minimum possible n as $n_{\text{min}} = \left\lfloor \frac{d}{3} \right\rfloor + 1$.
- ▷ Similarly, to ensure $\frac{n}{d} < \frac{1}{2}$, we calculate the maximum possible n as $n_{\text{max}} = \left\lfloor \frac{d-1}{2} \right\rfloor$.

With this range of n values, we loop through each possible numerator n from n_{min} to n_{max}. For each n, we check if $\frac{n}{d}$ is in reduced form by confirming that $\gcd(n, d) = 1$. If this condition is met, $\frac{n}{d}$ is a valid reduced fraction, so we increment our count.
Finally, we output the total count after examining all fractions within the specified limits.

Code [GitHub]

```python
from math import gcd

LIMIT = 12000

count = 0
for d in range(2, LIMIT + 1):
    n_min, n_max = d // 3 + 1, (d - 1) // 2
    for n in range(n_min, n_max + 1):
        if gcd(n, d) == 1: count += 1

print(count)
```

74 Digit Factorial Chains

Problem [Project Euler]

The number 145 is well known for the property that the sum of the factorial of its digits is equal to 145:

$$1! + 4! + 5! = 1 + 24 + 120 = 145.$$

Perhaps less well known is 169, in that it produces the longest chain of numbers that link back to 169; it turns out that there are only three such loops that exist:

$$\begin{aligned}
&169 \to 363601 \to 1454 \to 169\\
&871 \to 45361 \to 871\\
&872 \to 45362 \to 872
\end{aligned}$$

It is not difficult to prove that EVERY starting number will eventually get stuck in a loop. For example,

$$\begin{aligned}
&69 \to 363600 \to 1454 \to 169 \to 363601 (\to 1454)\\
&78 \to 45360 \to 871 \to 45361 (\to 871)\\
&540 \to 145 (\to 145)
\end{aligned}$$

Starting with 69 produces a chain of five non-repeating terms, but the longest non-repeating chain with a starting number below one million is sixty terms.
How many chains, with a starting number below one million, contain exactly sixty non-repeating terms?

Solution [Buy me a beer]

We start by observing that numbers with the same digit frequencies yield identical sums of factorials of their digits. This lets us treat numbers with the same digit composition as equivalent, allowing efficient caching and reducing redundant calculations.
First, we precompute the factorials for each digit 0 through 9 and store them in a dictionary f. With this setup, we define a function `dfsum(n)` to calculate the sum of factorials for the digits of a number n. We return the result as a sorted tuple of digits, which provides a unique representation for numbers with the same digit frequencies. This approach enables caching by digit pattern using `@cache`, avoiding recalculations for repeated patterns.

Next, we implement `chain_len` to compute the length of the chain starting from any number n. This function iteratively calculates each link in the chain using `dfsum`, adding each link to a list called `chain` to detect loops. If a loop is detected or if the chain exceeds 60 terms, the function stops. Chains that reach exactly 60 unique terms without forming a loop add their initial digit pattern to a set called `target`.
To populate `target`, we generate all possible digit combinations for numbers up to six digits using `combinations_with_replacement`. For each combination, we determine if it forms a chain of exactly 60 terms. If so, we add this pattern to `target`.
Finally, we check each number from 1 to $999,999$ by sorting its digits and checking if this pattern exists in `target`. If it does, we increment `count`. The final result, stored in `count`, gives the number of starting numbers below one million that produce a chain with exactly 60 non-repeating terms.

Code [GitHub]

```python
from math import factorial
from functools import cache
from itertools import combinations_with_replacement
from string import digits

f = {str(n): factorial(n) for n in range(10)}

@cache
def dfsum(n):
    return tuple(sorted(str(sum(f[d] for d in n))))

def chain_len(n, chain=[]):
    if not chain: chain = [n]
    if len(chain) > 60: return None
    if (dfs := dfsum(n)) in chain: return len(chain)
    return chain_len(dfs, chain + [dfs])

target = set()
for l in range(1, 7):
    for comb in combinations_with_replacement(digits, l):
        s = tuple(sorted(comb))
        if chain_len(comb) == 60: target.add(s)

count = 0
for n in map(str, range(1, 10**6)):
    if tuple(sorted(n)) in target:
        count += 1

print(count)
```

75 Singular Integer Right Triangles

Problem [Project Euler]

It turns out that 12 cm is the smallest length of wire that can be bent to form an integer-sided right angle triangle in exactly one way, but there are many more examples.

$$\begin{aligned}
\mathbf{12}\,\text{cm}&: (3,4,5)\\
\mathbf{24}\,\text{cm}&: (6,8,10)\\
\mathbf{30}\,\text{cm}&: (5,12,13)\\
\mathbf{36}\,\text{cm}&: (9,12,15)\\
\mathbf{40}\,\text{cm}&: (8,15,17)\\
\mathbf{48}\,\text{cm}&: (12,16,20)
\end{aligned}$$

In contrast, some lengths of wire, like 20 cm, cannot be bent to form an integer-sided right angle triangle, and other lengths allow more than one solution to be found; for example, using 120 cm it is possible to form exactly three different integer-sided right angle triangles.

$$\mathbf{120}\,\text{cm}: (30,40,50),\ (20,48,52),\ (24,45,51)$$

Given that L is the length of the wire, for how many values of $L \le 1\,500\,000$ can exactly one integer-sided right angle triangle be formed?

Solution [Buy me a beer]

To solve this problem, we start by generating all Pythagorean triples with a perimeter $L \le 1{,}500{,}000$. We use the well-known formula for generating primitive Pythagorean triples:

$$a = m^2 - n^2, \quad b = 2mn, \quad c = m^2 + n^2$$

where m and n are positive integers such that $m > n$, m and n are coprime, and $m + n$ is odd. This ensures that a, b, and c are integers and form a Pythagorean triple.
For each generated triple (a, b, c), we calculate the perimeter $L = a + b + c$. We then consider all integer multiples $k \cdot (a, b, c)$ for $k \ge 1$ as long as $kL \le 1{,}500{,}000$. This allows us to count all integer-sided right-angle triangles of a given perimeter.
We use a counter to keep track of how many times each perimeter L appears. Finally, we count the perimeters that appear exactly once, as these

correspond to wire lengths that can form exactly one unique integer-sided right-angle triangle.

Code [GitHub]

```python
from itertools import count
from math import gcd
from collections import Counter

LIMIT = 1_500_000

def triples():
    for m in count(2):
        if 2 * m * (m + 1) > LIMIT: break
        for n in range(1, m):
            if (m + n) % 2 != 1: continue
            if gcd(m, n) != 1: continue
            a = m**2 - n**2
            b = 2 * m * n
            c = m**2 + n**2
            if (perimeter := a + b + c) > LIMIT: break
            yield a, b, c

perims = Counter()
for a, b, c in triples():
    for k in count(1):
        ka, kb, kc = k * a, k * b, k * c
        if (p := ka + kb + kc) > LIMIT: break
        perims[p] += 1

print(sum(1 for p in perims if perims[p] == 1))
```

Counting Summations

Problem [Project Euler]

It is possible to write five as a sum in exactly six different ways:

$$\begin{aligned}&4+1\\&3+2\\&3+1+1\\&2+2+1\\&2+1+1+1\\&1+1+1+1+1\end{aligned}$$

How many different ways can one hundred be written as a sum of at least two positive integers?

Solution [Buy me a beer]

First, we define a recursive function `p(n,k)` that counts the ways to split `n` into exactly `k` parts, also called restricted integer partitions. The recursive setup follows the rule:

$$p_k(n) = p_k(n-k) + p_{k-1}(n-1)$$

with base cases $p_k(0) = 1$ and $p_k(n) = 0$ if n or k is negative (unless both are zero). For a visual proof of this recurrence, I recommend that you watch this video. Using caching to store results, each unique `p(n, k)` is only calculated once, making it efficient.
Next, to find the number of ways to write 100 as a sum of at least two positive parts, we sum `p(100, k)` for all `k` from 2 up to 100. This counts all valid partitions, and we print the result as the answer.

Code [GitHub]

```python
from functools import cache

@cache
def p(n, k):
    if n == 0 and k == 0: return 1

    if not (n == 0 and k == 0):
        if n <= 0 or k <= 0:
            return 0

    return p(n - k, k) + p(n - 1, k - 1)

n = 100
print(sum(p(n, k) for k in range(2, n + 1)))
```

77 Prime Summations

Problem [Project Euler]

It is possible to write ten as the sum of primes in exactly five different ways:

$$
\begin{aligned}
&7+3\\
&5+5\\
&5+3+2\\
&3+3+2+2\\
&2+2+2+2+2
\end{aligned}
$$

What is the first value which can be written as the sum of primes in over five thousand different ways?

Solution [Buy me a beer]

We start by generating a list of primes up to a reasonable limit. Using recursion, we define a function `pparts(n, i)` that calculates the number of ways n can be written as a sum of primes up to the i-th prime in our list. The recursion is structured as follows:

▷ If $n = 0$, we have found a valid partition, so we return 1.

▷ If $n < 0$ or $i < 0$, there is no valid partition, so we return 0.

▷ Otherwise, we compute the sum of partitions that include the i-th prime (`pparts(n - primes[i], i)`) and those that exclude it (`pparts(n, i - 1)`).

Finally, we apply this function to each integer n, using `first_true` to find the first n for which `pparts(n)` > 5000.

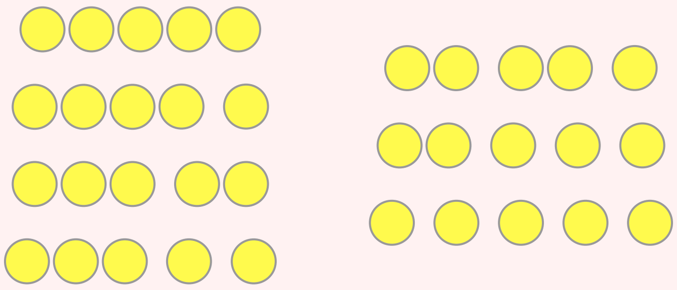
Code [GitHub]

```python
from sympy import primerange
from more_itertools import first_true

primes = list(primerange(1000))

def pparts(n, i=len(primes) - 1):
    if n == 0: return 1
    if n < 0 or i < 0: return 0

    with_prime = pparts(n - primes[i], i)
    without_prime = pparts(n, i - 1)
    return with_prime + without_prime

print(first_true(primes, pred=lambda p: pparts(p) > 5000))
```

78 Coin Partitions

Problem [Project Euler]

Let $p(n)$ represent the number of different ways in which n coins can be separated into piles. For example, five coins can be separated into piles in exactly seven different ways, so $p(5) = 7$.



Find the least value of n for which $p(n)$ is divisible by one million.

Solution [Buy me a beer]

To solve this problem, we start by recognizing that the partition function $p(n)$ represents the number of ways n can be separated into piles. Our goal is to find the smallest n for which $p(n)$ is divisible by one million.
We approach this by creating a generator function, `p_gen`, that calculates $p(n)$ iteratively. We initialize our sequence with $p(0) = 1$, the base case for partitions, which we yield first. Then, for each n from 1 onward, we calculate $p(n)$ by using the pentagonal number theorem, which helps us express each partition number in terms of previous values in the sequence. By iterating through generalized pentagonal numbers and alternating addition and subtraction based on their indices, we can build up each partition value efficiently.
With `p_gen` set up to yield each partition number as it's calculated, we use `takewhile` to continue generating partition values until we reach the first $p(n)$ that's divisible by one million. At that point, `takewhile` stops, and we use `ilen` to count how many terms were generated up to this point, effectively giving us the smallest n for which $p(n)$ meets the divisibility requirement.
This approach allows us to avoid calculating $p(n)$ individually and instead stream partition values until our condition is met. By taking advantage of the properties of partition numbers and the efficiency of the generator, we find the answer without storing unnecessary values, making the solution both compact and efficient.

Code [GitHub]

```python
from itertools import count, takewhile
from more_itertools import ilen

def p_gen():
    p_values = [1]
    yield 1
    for n in count(1):
        total = 0
        for k in count(1):
            g_k = k * (3 * k - 1) // 2
            g_neg_k = k * (3 * k + 1) // 2

            if g_k > n: break

            sign = (2 * (k % 2) - 1)
            total += sign * p_values[n - g_k]
            if g_neg_k <= n:
                total += sign * p_values[n - g_neg_k]

        p_values.append(total)
        yield total

print(ilen(takewhile(lambda x: x % 10**6, p_gen())))
```

79 Passcode Derivation

Problem [Project Euler]

A common security method used for online banking is to ask the user for three random characters from a passcode. For example, if the passcode was 531278, they may ask for the 2nd, 3rd, and 5th characters; the expected reply would be: 317.
The text file, keylog.txt, contains fifty successful login attempts.
Given that the three characters are always asked for in order, analyse the file so as to determine the shortest possible secret passcode of unknown length.

Solution [Buy me a beer]

First, we load the list of successful login attempts from `keylog.txt` into `codes`. Each entry in this list represents a sequence of three digits observed in order, which gives clues about the correct passcode structure.
To build the relative order of each digit, we use a `defaultdict` named `order`, where each key represents a digit in the passcode, and its value is a set of digits that must follow it. For each entry `(a, b, c)` in `codes`, we update the `order` dictionary so that:

- ▷ `b` and `c` must follow `a`,
- ▷ and `c` must follow `b`.

Once we have established these relationships, we create a list of the digits (`passcode`) from `order.keys()`. Then, we sort this list by the size of each digit's set (from `order`), in descending order, so that digits with more following dependencies come earlier in the passcode.
Finally, we convert the sorted list of digits to a string and print it as the passcode. This represents the shortest possible passcode that satisfies all observed three-digit sequences.

Code [GitHub]

```python
from collections import defaultdict

with open('0079_keylog.txt') as f:
    codes = f.read().splitlines()

order = defaultdict(set)

for a, b, c in codes:
    order[a].update({b, c})
    order[b].add(c)
    if not order[c]: order[c] = set()

passcode = list(order.keys())
passcode.sort(key=lambda d: len(order[d]), reverse=True)
print(''.join(passcode))
```

80 Square Root Digital Expansion

Problem [Project Euler]

It is well known that if the square root of a natural number is not an integer, then it is irrational. The decimal expansion of such square roots is infinite without any repeating pattern at all.
The square root of two is $1.41421356237309504880\ldots$, and the digital sum of the first one hundred decimal digits is 475.
For the first one hundred natural numbers, find the total of the digital sums of the first one hundred decimal digits for all the irrational square roots.

Solution [Buy me a beer]

First, we filter out numbers whose square roots are integers by defining a helper function, `is_not_square`, that checks if a number is not a perfect square. For each irrational square root, we then approximate the first 100 decimal digits using a modified Newton's method.
To obtain these digits accurately:

- ▷ We multiply the number by 10^{200} to shift the decimal point, allowing us to approximate the first 100 decimal digits directly.
- ▷ We then apply Newton's method iteratively to find an approximation

x such that $x^2 \approx n \times 10^{200}$.

▷ Once we have this approximation, we calculate the digital sum of the first 100 digits using the `dsum` function.

Finally, we sum these digital sums for all irrational square roots from 1 to 100 and print the result.

Code [GitHub]

```python
from math import isqrt

def dsum(n): return sum(map(int, str(n)[:100]))

def sqrt_dsum(n):
    n *= 10**200

    x, f = 1, lambda x: (x + n // x) // 2
    while (new_x := f(x)) != x: x = new_x

    return dsum(x)

is_not_square = lambda n: isqrt(n)**2 != n
not_squares = filter(is_not_square, range(1, 101))
print(sum(map(sqrt_dsum, not_squares)))
```

81 Path Sum: Two Ways

Problem [Project Euler]

In the 5 × 5 matrix below, the minimal path sum from the top left to the bottom right, by **only moving to the right and down**, is indicated in bold red and is equal to 2427.

$$\begin{pmatrix} \mathbf{131} & 673 & 234 & 103 & 18 \\ \mathbf{201} & \mathbf{96} & \mathbf{342} & 965 & 150 \\ 630 & 803 & \mathbf{746} & \mathbf{422} & 111 \\ 537 & 699 & 497 & \mathbf{121} & 956 \\ 805 & 732 & 524 & \mathbf{37} & \mathbf{331} \end{pmatrix}$$

Find the minimal path sum from the top left to the bottom right by only moving right and down in matrix.txt, a 31K text file containing an 80 × 80 matrix.

Solution [Buy me a beer]

To solve this problem, we apply dynamic programming to compute the minimal path sum from the top-left to the bottom-right of the matrix, with movements restricted to the right and downward directions.

- ▷ We initialize a dynamic programming matrix, `dp`, with the same dimensions as the input matrix. The cell `dp`$[0][0]$ is set to the value of the top-left cell in the input matrix.
- ▷ For the first row and first column, we compute cumulative sums since these cells can only be reached from the left (for the first row) or from above (for the first column).
- ▷ For each remaining cell (i, j), we calculate the minimal path sum using:

  $$\texttt{dp}[i][j] = \texttt{matrix}[i][j] + \min(\texttt{dp}[i-1][j], \texttt{dp}[i][j-1])$$

  This ensures each cell holds the minimum sum needed to reach it from the top-left.
- ▷ The result, `dp`$[-1][-1]$, contains the minimal path sum from the top-left to the bottom-right of the matrix.

Code [GitHub]

```python
from itertools import product

with open("0081_matrix.txt") as f:
    matrix = [
        [int(n) for n in row.split(",")]
        for row in f.read().strip().split("\n")
    ]

SIZE = len(matrix)

dp = [[0] * SIZE for _ in range(SIZE)]
dp[0][0] = matrix[0][0]

for k in range(1, SIZE):
    dp[0][k] = dp[0][k-1] + matrix[0][k]
    dp[k][0] = dp[k-1][0] + matrix[k][0]

```

```python
for i, j in product(range(1, SIZE), repeat=2):
    dp[i][j] = matrix[i][j] + min(dp[i-1][j], dp[i][j-1])
print(dp[-1][-1])
```

82 Path Sum: Three Ways

Problem [Project Euler]

NOTE: This problem is a more challenging version of Problem 81.
The minimal path sum in the 5×5 matrix below, by starting in any cell in the left column and finishing in any cell in the right column, and only moving up, down, and right, is indicated in red and bold; the sum is equal to 994.

$$\begin{pmatrix} 131 & 673 & \mathbf{234} & \mathbf{103} & \mathbf{18} \\ \mathbf{201} & \mathbf{96} & \mathbf{342} & 965 & 150 \\ 630 & 803 & 746 & 422 & 111 \\ 537 & 699 & 497 & 121 & 956 \\ 805 & 732 & 524 & 37 & 331 \end{pmatrix}$$

Find the minimal path sum from the left column to the right column in matrix.txt, a 31K text file containing an 80×80 matrix.

Solution [Buy me a beer]

To solve this problem, we extend the dynamic programming approach used in Problem 81 to allow movement in three directions: right, up, and down. This approach is necessary because we are finding the minimal path sum from any cell in the left column to any cell in the right column, and the added flexibility requires additional adjustments to ensure the path cost is minimized.

▷ We initialize a dynamic programming matrix, `dp`, where $\texttt{dp}[i][0]$ is set to the value of the corresponding cell in the first column of the input matrix, as each path starts from the left column.

▷ For each subsequent column j, we first compute minimal path sums by moving right from the previous column:

$$\texttt{dp}[i][j] = \texttt{dp}[i][j-1] + \texttt{matrix}[i][j]$$

▷ Next, we update each cell by propagating minimal path sums upwards:

$$\mathtt{dp}[i][j] = \min(\mathtt{dp}[i][j], \mathtt{dp}[i-1][j] + \mathtt{matrix}[i][j])$$

and then downwards:

$$\mathtt{dp}[i][j] = \min(\mathtt{dp}[i][j], \mathtt{dp}[i+1][j] + \mathtt{matrix}[i][j])$$

to ensure all possible paths are considered.

▷ Finally, we find the minimal path sum reaching the right column by taking the minimum value in the last column of `dp`, representing the smallest cost path from any starting cell in the left column to any ending cell in the right column.

Code [GitHub]

```python
with open("0082_matrix.txt") as f:
    matrix = [
        [int(n) for n in row.split(",")]
        for row in f.read().strip().split("\n")
    ]

SIZE = len(matrix)

dp = [[float('inf')] * SIZE for _ in range(SIZE)]
for i in range(SIZE): dp[i][0] = matrix[i][0]

for j in range(1, SIZE):
    for i in range(SIZE):
        dp[i][j] = dp[i][j-1] + matrix[i][j]

    for i in range(1, SIZE):
        dp[i][j] = min(dp[i][j], dp[i-1][j] + matrix[i][j])

    for i in range(SIZE - 2, -1, -1):
        dp[i][j] = min(dp[i][j], dp[i+1][j] + matrix[i][j])

result = min(dp[i][-1] for i in range(SIZE))
print(result)
```

83 Path Sum: Four Ways

Problem [Project Euler]

NOTE: This problem is a significantly more challenging version of Problem 81.
In the 5×5 matrix below, the minimal path sum from the top left to the bottom right, by moving left, right, up, and down, is indicated in bold red and is equal to 2297.

$$\begin{pmatrix} \mathbf{131} & 673 & \mathbf{234} & \mathbf{103} & \mathbf{18} \\ \mathbf{201} & \mathbf{96} & \mathbf{342} & 965 & \mathbf{150} \\ 630 & 803 & 746 & \mathbf{422} & \mathbf{111} \\ 537 & 699 & 497 & \mathbf{121} & 956 \\ 805 & 732 & 524 & \mathbf{37} & \mathbf{331} \end{pmatrix}$$

Find the minimal path sum from the top left to the bottom right by moving left, right, up, and down in matrix.txt, a 31K text file containing an 80×80 matrix.

Solution [Buy me a beer]

To solve this problem, we use Dijkstra's algorithm to find the minimal path sum from the top-left to the bottom-right cell of the matrix, allowing movement in all four directions: up, down, left, and right.

▷ We start by initializing a priority queue and setting the distance to the starting cell, `dist[0][0]`, as its matrix value. All other distances are set to infinity.

▷ We then enter a loop where we continually pop the cell with the current smallest path sum from the queue. For each cell, we explore its neighbors by moving up, down, left, and right.

▷ For each valid neighbor, we calculate the potential new path sum, `new_sum`. If `new_sum` is less than the recorded distance to this neighbor in `dist`, we update `dist` and push the neighbor to the priority queue.

▷ The loop continues until we reach the bottom-right cell, at which point we break out of the loop since we have found the minimal path sum to this destination.

▷ Finally, we output the minimal path sum stored in dist[SIZE-1][SIZE-1], representing the minimal path sum from the top-left to the bottom-right of the matrix.

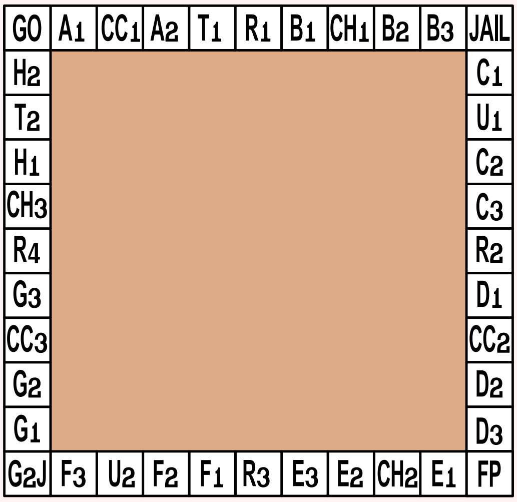
Code [GitHub]

```
from heapq import heappop, heappush

with open("0083_matrix.txt") as f:
    matrix = [
        [int(n) for n in row.split(",")]
        for row in f.read().strip().split("\n")
    ]

SIZE = len(matrix)
directions = (0, 1), (1, 0), (0, -1), (-1, 0)

dist = [[float('inf')] * SIZE for _ in range(SIZE)]
dist[0][0] = matrix[0][0]

queue = [(matrix[0][0], 0, 0)]

while queue:
    current_sum, x, y = heappop(queue)

    if (x, y) == (SIZE - 1, SIZE - 1): break

    for dx, dy in directions:
        nx, ny = x + dx, y + dy

        if 0 <= nx < SIZE and 0 <= ny < SIZE:
            new_sum = current_sum + matrix[nx][ny]

            if new_sum < dist[nx][ny]:
                dist[nx][ny] = new_sum
                heappush(queue, (new_sum, nx, ny))

print(current_sum)
```

84 Monopoly Odds

Problem [Project Euler]

In the game, **Monopoly**, the standard board is set up in the following way:



A player starts on the GO square and adds the scores on two 6-sided dice to determine the number of squares they advance in a clockwise direction. Without any further rules, we would expect to visit each square with equal probability: 2.5%. However, landing on G2J (Go To Jail), CC (Community Chest), and CH (Chance) changes this distribution.
In addition to G2J, and one card from each of CC and CH that orders the player to go directly to jail, if a player rolls three consecutive doubles, they do not advance the result of their 3rd roll. Instead, they proceed directly to jail.
At the beginning of the game, the CC and CH cards are shuffled. When a player lands on CC or CH, they take a card from the top of the respective pile, and after following the instructions, it is returned to the bottom of the pile. There are sixteen cards in each pile, but for the purpose of this problem, we are only concerned with cards that order a movement; any instruction not concerned with movement will be ignored and the player will remain on the CC/CH square.

▷ Community Chest (2/16 cards):

1. Advance to GO

2. Go to JAIL

- ▷ Chance (10/16 cards):

1. Advance to GO
2. Go to JAIL
3. Go to C1
4. Go to E3
5. Go to H2
6. Go to R1
7. Go to next R (railway company)
8. Go to next R
9. Go to next U (utility company)
10. Go back 3 squares.

The heart of this problem concerns the likelihood of visiting a particular square. That is, the probability of finishing at that square after a roll. For this reason, it should be clear that, with the exception of G2J, for which the probability of finishing on it is zero, the CH squares will have the lowest probabilities, as 5/8 request a movement to another square, and it is the final square that the player finishes at on each roll that we are interested in. We shall make no distinction between "Just Visiting" and being sent to JAIL, and we shall also ignore the rule about requiring a double to "get out of jail", assuming that they pay to get out on their next turn.
By starting at GO and numbering the squares sequentially from 00 to 39, we can concatenate these two-digit numbers to produce strings that correspond with sets of squares.
Statistically, it can be shown that the three most popular squares, in order, are JAIL (6.24%) = Square 10, E3 (3.18%) = Square 24, and GO (3.09%) = Square 00. So these three most popular squares can be listed with the six-digit modal string: 102400.
If, instead of using two 6-sided dice, two 4-sided dice are used, find the six-digit modal string.

Solution [Buy me a beer]

To solve this problem, we model the Monopoly game as a Markov chain with a transition matrix that captures the probabilities of moving between squares based on rolls and card draws. The goal is to compute the steady-state distribution of this matrix, as this distribution indicates the long-term probabilities of landing on each square.

- ▷ We first calculate the probabilities of rolling each possible total with two 4-sided dice. Using these probabilities, we initialize a transition matrix, P, for the 40 board squares.
- ▷ For each square i, we compute transition probabilities to subsequent squares based on possible outcomes:
  - If a player lands on G2J (Go to Jail), they are moved directly to JAIL (square 10).
  - For Community Chest (CC) and Chance (CH) squares, we handle card draws that lead to specific movements, with probabilities reflecting the proportion of movement cards.
- ▷ After setting up all transitions in P, we apply matrix exponentiation by repeatedly multiplying a vector representing the initial probabilities by P. After sufficient iterations, this process converges, giving us the steady-state distribution.
- ▷ Finally, we identify the three most frequently visited squares by sorting the steady-state probabilities in descending order and formatting these squares as the six-digit modal string.

This approach leverages the properties of a Markov chain to model the game's random processes and efficiently compute the desired probabilities.

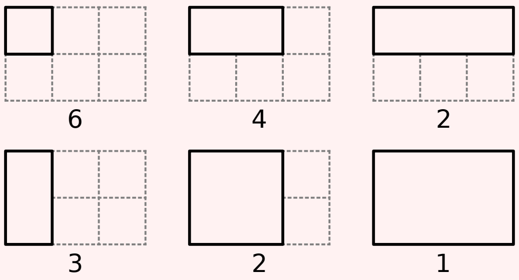
Code [GitHub]

```python
import numpy as np
from collections import defaultdict
from itertools import product

results = defaultdict(float)
P = np.zeros((40, 40))
prob_vec = np.zeros(40); prob_vec[0] = 1.0

for i, j in product(range(1, 5), repeat=2):
    results[i + j] += 1 / 16

for i in range(40):
    for value, p in results.items():
        nxt = (i + value) % 40
        if nxt == 30: P[i][10] += p
        elif nxt in {2, 17, 33}:
            P[i][0] += p / 16
            P[i][10] += p / 16
            P[i][nxt] += 14 * p / 16
        elif nxt in {7, 22, 36}:
            for j in [0, 5, 10, 11, 12, 15, 24, 39]:
                P[i][j] += p / 16
            P[i][(nxt - 3) % 40] += p / 16
            P[i][nxt] += 6 * p / 16
        else: P[i][nxt] += p

for _ in range(200): prob_vec = prob_vec @ P

most_popular = np.argsort(prob_vec)[-3:][::-1]
print(''.join(f"{square:02}" for square in most_popular))
```

85 Counting Rectangles

Problem [Project Euler]

By counting carefully, it can be seen that a rectangular grid measuring 3×2 contains eighteen rectangles:



Although there exists no rectangular grid that contains exactly two million rectangles, find the area of the grid with the nearest solution.

Solution [Buy me a beer]

The number of rectangles in a grid of size $m \times n$ is given by $\binom{m+1}{2}\binom{n+1}{2}$, since each rectangle is determined by choosing two horizontal and two vertical grid lines.
We start by defining a helper function, `rectangles`, to calculate the total number of rectangles in a grid with given dimensions `width` and `height`.
To find the grid with an area closest to containing exactly two million rectangles, we set a `LIMIT` to control the maximum grid dimensions and initialize `min_diff` to store the smallest difference from our target of two million.
Next, we iterate over all possible pairs of `width` and `height` up to `LIMIT`, calculating the rectangle count for each. For each combination, we check the difference between our calculated rectangle count and two million, updating `min_diff` and `area` if this new difference is smaller.
Finally, we print the area of the grid that has the rectangle count closest to two million.

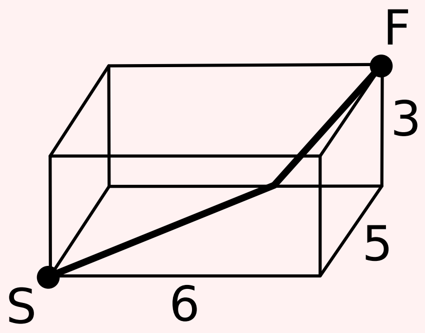
Code [GitHub]

```python
from math import comb

def rectangles(width, height):
    return comb(width + 1, 2) * comb(height + 1, 2)

LIMIT = 10**3

min_diff = float('inf')
for w in range(1, LIMIT + 1):
    for h in range(w, LIMIT + 1):
        diff = abs(2_000_000 - rectangles(w, h))
        if diff < min_diff:
            min_diff, area = diff, w * h

print(area)
```

86 Cuboid Route

Problem [Project Euler]

A spider, S, sits in one corner of a cuboid room, measuring $6 \times 5 \times 3$, and a fly, F, sits in the opposite corner. By travelling on the surfaces of the room, the shortest "straight line" distance from S to F is 10, and the path is shown on the diagram.



However, there are up to three "shortest" path candidates for any given cuboid, and the shortest route doesn't always have integer length.
It can be shown that there are exactly 2060 distinct cuboids, ignoring rotations, with integer dimensions, up to a maximum size of $M \times M \times M$, for

which the shortest route has integer length when $M = 100$. This is the least value of M for which the number of solutions first exceeds two thousand; the number of solutions when $M = 99$ is 1975.
Find the least value of M such that the number of solutions first exceeds one million.

Solution [Buy me a beer]

In this solution, we seek the smallest dimension M of a cuboid such that the number of distinct cuboids with integer shortest paths from one corner to the opposite exceeds one million. The shortest path from one corner to the opposite can be computed on the cuboid's surfaces, effectively giving us paths of the form $\sqrt{M^2 + N^2}$ where $N = b + c$ for dimensions (a, b, c) such that $a \leq b \leq c \leq M$.
We precompute squares up to a large limit to quickly determine if a path length is an integer. For each value of M, we compute the possible paths by iterating over values of N, ensuring that $M^2 + N^2$ forms a perfect square. Each valid integer path adds to the count based on the number of possible (b, c) pairs that sum to N and satisfy $b \leq c \leq M$.
By incrementally summing these counts starting from $M = 1$, we find the smallest M such that the cumulative count of cuboids with integer paths exceeds one million.

Code [GitHub]

```python
from itertools import count, takewhile

squares = (n**2 for n in count(1))
squares = set(takewhile(lambda x: x < 2*10**8, squares))
is_square = lambda x: x in squares

def sols(M):
    total = 0
    for N in range(2, 2 * M + 1):
        if is_square(M**2 + N**2):
            total += M+1 - (N+1) // 2 if N > M+1 else N // 2
    return total

cum_total = 0
for M in count(1):
    cum_total += sols(M)
    if cum_total > 10**6: break

print(M)
```

87 Prime Power Triples

Problem [Project Euler]

The smallest number expressible as the sum of a prime square, prime cube, and prime fourth power is 28. In fact, there are exactly four numbers below fifty that can be expressed in such a way:

$$\begin{aligned}28 &= 2^2 + 2^3 + 2^4\\ 33 &= 3^2 + 2^3 + 2^4\\ 49 &= 5^2 + 2^3 + 2^4\\ 47 &= 2^2 + 3^3 + 2^4\end{aligned}$$

How many numbers below fifty million can be expressed as the sum of a prime square, prime cube, and prime fourth power?

Solution [Buy me a beer]

We start by generating primes up to the square root of the limit, which we use to create the necessary prime powers. By setting up a generator for each power, we ensure that only values below the limit are included, resulting in three lists of possible values for squares, cubes, and fourth powers.
With these lists, we then examine all possible sums by combining one value from each list. For any valid sum below the limit, we add it to a set to ensure uniqueness. This approach allows us to handle large combinations efficiently by breaking out of loops whenever partial sums exceed the limit. In the end, the size of the set gives us the count of unique numbers below fifty million that can be expressed in this way.

Code [GitHub]

```python
from sympy import primerange
from itertools import takewhile

LIMIT = 50_000_000

primes = list(primerange(int(LIMIT**0.5)))

pow_gen = lambda exp: (p**exp for p in primes)
powers = {
    k: list(takewhile(lambda x: x < LIMIT, pow_gen(k)))
    for k in [2, 3, 4]
}
```

```

numbers = set()
for a in powers[2]:
    for b in powers[3]:
        if (n := a + b) >= LIMIT: break
        for c in powers[4]:
            if (m := n + c) >= LIMIT: break
            numbers.add(m)

print(len(numbers))
```

88 Product-sum Numbers

Problem [Project Euler]

A natural number, N, that can be written as the sum and product of a given set of at least two natural numbers, $\{a_1, a_2, \ldots, a_k\}$, is called a product-sum number:

$$N = a_1 + a_2 + \cdots + a_k = a_1 \times a_2 \times \cdots \times a_k.$$

For example, $6 = 1 + 2 + 3 = 1 \times 2 \times 3$.
For a given set of size k, we shall call the smallest N with this property a minimal product-sum number. The minimal product-sum numbers for sets of size $k = 2, 3, 4, 5$, and 6 are as follows.

$$\begin{aligned}
k = 2&: \quad 4 = 2 \times 2 = 2 + 2\\
k = 3&: \quad 6 = 1 \times 2 \times 3 = 1 + 2 + 3\\
k = 4&: \quad 8 = 1 \times 1 \times 2 \times 4 = 1 + 1 + 2 + 4\\
k = 5&: \quad 8 = 1 \times 1 \times 2 \times 2 \times 2 = 1 + 1 + 2 + 2 + 2\\
k = 6&: \quad 12 = 1 \times 1 \times 1 \times 1 \times 2 \times 6 = 1 + 1 + 1 + 1 + 2 + 6
\end{aligned}$$

Hence for $2 \le k \le 6$, the sum of all the minimal product-sum numbers is $4 + 6 + 8 + 12 = 30$; note that 8 is only counted once in the sum.
In fact, as the complete set of minimal product-sum numbers for $2 \le k \le 12$ is $\{4, 6, 8, 12, 15, 16\}$, the sum is 61.
What is the sum of all the minimal product-sum numbers for $2 \le k \le 12000$?

Solution [Buy me a beer]

To find all minimal *product-sum numbers* for set sizes $2 \leq k \leq 12000$, we define a recursive approach that leverages the unique relationship between the sum and product of integers in a set. We initialize an array N of size `LIMIT + 1` to store the smallest product-sum N for each k, initially set to a large placeholder.

The recursive function iterates over potential factors to calculate both the sum and product of elements in each set, thereby determining the smallest product-sum N for any k where $N = a_1 + a_2 + \cdots + a_k = a_1 \times a_2 \times \cdots \times a_k$. By updating the N array only when a smaller product-sum is found for a given k, we efficiently build minimal values without unnecessary recalculations. The function is designed to maintain ascending factor order to prevent redundant or reversed sets.

In the end, we extract unique values from N for $2 \leq k \leq 12000$, ensuring only minimal product-sum numbers are summed. This provides an optimal solution by focusing solely on the smallest values, resulting in a fast and effective calculation of the desired sum.

Code [GitHub]

```python
LIMIT = 12000

N = [2 * LIMIT] * (LIMIT + 1)

def productsum(prod, suma, n, i0):
    if (k := n + prod - suma) > LIMIT: return
    if prod < N[k]: N[k] = prod
    for i in range(i0, 2 * LIMIT // prod + 1):
        productsum(prod * i, suma + i, n + 1, i)

productsum(1, 0, 0, 2)
print(sum(set(N[2:])))
```

89 Roman Numerals

Problem [Project Euler]

For a number written in Roman numerals to be considered valid, there are basic rules which must be followed. Even though the rules allow some numbers to be expressed in more than one way, there is always a "best" way of writing a particular number.
For example, it would appear that there are at least six ways of writing the number sixteen:

IIIIIIIIIIIIIIII
VIIIIIIIIIII
VVIIIIII
XIIIIII
VVVI
XVI

However, according to the rules, only XIIIIII and XVI are valid, and the last example is considered to be the most efficient, as it uses the least number of numerals.
The 11K text file, roman.txt, contains one thousand numbers written in valid, but not necessarily minimal, Roman numerals; see About... Roman Numerals for the definitive rules for this problem.
Find the number of characters saved by writing each of these in their minimal form.
NOTE: You can assume that all the Roman numerals in the file contain no more than four consecutive identical units.

Solution [Buy me a beer]

First, we define a function, `to_decimal`, which allows us to convert each Roman numeral to a decimal by iterating through each pair of characters and adjusting based on whether each character should be added or subtracted. This conversion gives us a reliable decimal representation for each numeral.
Next, we create `roman_len` to calculate the minimal number of characters required to represent a given decimal in Roman numerals. Using an array, we map each digit to its minimal representation in Roman characters, making it straightforward to compute the minimal length needed for any numeral, even in cases with thousands.

With these functions in place, we open the file, read each line, convert each numeral to its minimal form, and calculate the difference in characters between the original and minimal forms. We accumulate the character savings across all entries, then print the total saved.

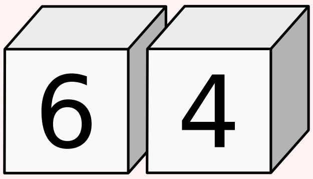
Code [GitHub]

```
R = {
    'I': 1, 'V': 5, 'X': 10, 'L': 50,
    'C': 100, 'D': 500, 'M': 1000
    }
L = [0, 1, 2, 3, 2, 1, 2, 3, 4, 2]

def to_decimal(r):
    dec = R[r[-1]]
    for i in range(len(r) - 1):
        a, b = r[i:i + 2]
        sgn = 1 if R[a] >= R[b] else -1
        dec += sgn * R[a]
    return dec

def roman_len(n):
    l = i = 0
    while n > 0:
        n, d = divmod(n, 10)
        l += L[d] if i < 3 else d
        i += 1
    return l

with open("0089_roman.txt") as f:
    romans = f.read().splitlines()

saved = lambda r: len(r) - roman_len(to_decimal(r))
print(sum(map(saved, romans)))
```

90 Cube Digit Pairs

Problem [Project Euler]

Each of the six faces on a cube has a different digit (0 to 9) written on it; the same is done to a second cube. By placing the two cubes side-by-side in different positions, we can form a variety of two-digit numbers.
For example, the square number 64 could be formed:



In fact, by carefully choosing the digits on both cubes, it is possible to display all of the square numbers below one hundred: 01, 04, 09, 16, 25, 36, 49, 64, and 81.
For example, one way this can be achieved is by placing $\{0, 5, 6, 7, 8, 9\}$ on one cube and $\{1, 2, 3, 4, 8, 9\}$ on the other cube.
However, for this problem, we shall allow the 6 or 9 to be turned upside-down, so that an arrangement like $\{0, 5, 6, 7, 8, 9\}$ and $\{1, 2, 3, 4, 6, 7\}$ allows for all nine square numbers to be displayed; otherwise, it would be impossible to obtain 09.
In determining a distinct arrangement, we are interested in the digits on each cube, not the order.

$$\{1, 2, 3, 4, 5, 6\} \text{ is equivalent to } \{3, 6, 4, 1, 2, 5\}$$
$$\{1, 2, 3, 4, 5, 6\} \text{ is distinct from } \{1, 2, 3, 4, 5, 9\}$$

But because we are allowing 6 and 9 to be reversed, the two distinct sets in the last example both represent the extended set $\{1, 2, 3, 4, 5, 6, 9\}$ for the purpose of forming two-digit numbers.
How many distinct arrangements of the two cubes allow for all of the square numbers to be displayed?

Solution [Buy me a beer]

First, we generate the set of two-digit squares under 100. For easy comparison, we convert each square to a sorted list of its digits. This lets us later match two-digit combinations formed by the cubes with our required squares.
Next, we create all unique 6-digit combinations from the digits 0 through 9. For each cube arrangement, we treat any cube containing a 6 or a 9 as if it contains both, since these digits are interchangeable by being rotated. This adjustment ensures we can form combinations like 09 and 06 without needing separate configurations for both 6 and 9 on each cube.
After that, we examine each unique pair of cubes. For each pair, we form all possible two-digit numbers by choosing one digit from each cube, checking if these combinations cover the list of required squares. If a pair successfully represents all required squares, we count it as valid.
Finally, we tally the total number of valid pairs, which gives us the number of distinct cube configurations that can display every square below 100.

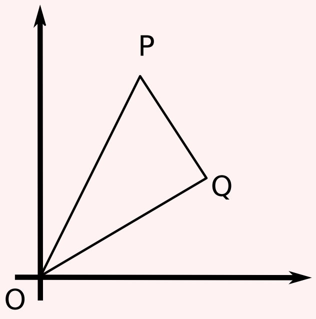
Code [GitHub]

```python
from itertools import combinations, product
from string import digits

squares = []
for n in range(1, 10):
    sq = sorted(f"{n**2:02}")
    squares.append(sq)

count, cube_gen = 0, map(set, combinations(digits, 6))
for cubes in map(list, combinations(cube_gen, 2)):
    for i in range(2):
        if '6' in cubes[i] or '9' in cubes[i]:
            cubes[i].update({'6', '9'})

    unseen = squares.copy()
    for num in map(sorted, product(*cubes)):
        if num in unseen: unseen.remove(num)
        if not unseen: break
    else: continue
    count += 1

print(count)
```

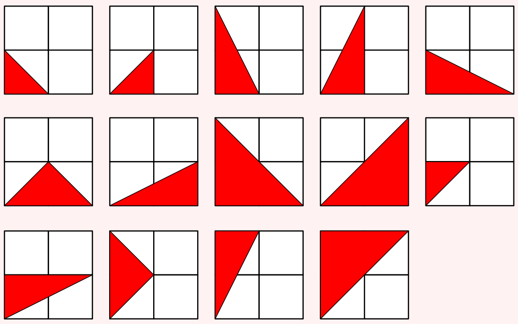
91 Right Triangles with Integer Coordinates

Problem [Project Euler]

The points $P(x_1, y_1)$ and $Q(x_2, y_2)$ are plotted at integer coordinates and are joined to the origin, $O(0,0)$, to form $\triangle OPQ$.



There are exactly fourteen triangles containing a right angle that can be formed when each coordinate lies between 0 and 2 inclusive; that is, $0 \le x_1, y_1, x_2, y_2 \le 2$.



Given that $0 \le x_1, y_1, x_2, y_2 \le 50$, how many right triangles can be formed?

Solution [Buy me a beer]

To solve this problem, we aim to count the number of right triangles $\triangle OPQ$ that can be formed by integer points $P(x_1, y_1)$ and $Q(x_2, y_2)$ in a grid where $0 \leq x_1, y_1, x_2, y_2 \leq 50$, with the right angle located at any of the points O, P, or Q.
We start by generating all points in the grid, excluding the origin $O(0, 0)$. For each unique pair of points P and Q, we first check that they are not collinear with O, which is necessary to form a valid triangle. Collinearity is determined by the condition:

$$x_1 y_2 = x_2 y_1.$$

If the points P and Q are not collinear with O, we proceed to check if they form a right triangle. We calculate the distances OP, PQ, and OQ, then sort these distances to identify the hypotenuse. To verify if the triangle is a right triangle, we confirm that the Pythagorean theorem holds:

$$a^2 + b^2 = c^2,$$

where c is the longest side.
By iterating through all pairs of points and applying these checks, we count the total number of right triangles satisfying the criteria and obtain the final result.

Code [GitHub]

```python
from itertools import product, combinations as combs
from math import dist as d

grid = set()
for p in product(range(51), repeat=2):
    if p != (0, 0): grid.add(p)

p0, count = (0, 0), 0
for p1, p2 in combs(grid, 2):
    if p1[0]*p2[1] == p2[0]*p1[1]: continue
    a, b, c = d(p0, p1), d(p1, p2), d(p2, p0)
    a, b, c = sorted([a, b, c])
    if abs(a**2 + b**2 - c**2) < 0.1: count += 1

print(count)
```

92 Square Digit Chains

Problem [Project Euler]

A number chain is created by continuously adding the square of the digits in a number to form a new number until it has been seen before.
For example,

$$44 \to 32 \to 13 \to 10 \to \mathbf{1} \to \mathbf{1}$$
$$85 \to \mathbf{89} \to 145 \to 42 \to 20 \to 4 \to 16 \to 37 \to 58 \to \mathbf{89}$$

Therefore, any chain that arrives at 1 or 89 will become stuck in an endless loop. What is most amazing is that **EVERY** starting number will eventually arrive at 1 or 89.
How many starting numbers below ten million will arrive at 89?

Solution [Buy me a beer]

- ▷ First, we define a function, `sqdsum`, to compute the sum of the squares of a number's digits. This function helps us generate new numbers in the chain.
- ▷ We consider only the unique digit combinations of length up to 7 (since 10^7 has at most 7 digits). Using `combinations_with_replacement`, we generate all possible combinations of digits from 1 to 9 for chains, as leading zeros do not alter the results.
- ▷ For each combination of digits, we calculate the total number of distinct numbers represented by that combination. This is achieved by permuting the digits appropriately, using factorial calculations to account for duplicates.
- ▷ We then create a `Counter` object to store the occurrences of each sum of squares of digits. We recursively process this `Counter`, summing the values for chains that reach 89 and updating the counts for other sums until no new numbers remain to process.
- ▷ Finally, the accumulated count for chains reaching 89 gives the answer, which we print.

This approach efficiently counts the desired chains without iterating through every number below ten million, reducing computation by exploiting digit permutations and recursive counting.

Code [GitHub]

```python
from itertools import combinations_with_replacement as combs
from collections import Counter
from math import factorial as f, prod

def sqdsum(n):
    if n == 0: return 0
    q, r = divmod(n, 10)
    return r**2 + sqdsum(q)

def numbers(combo):
    combo = Counter(combo)
    combo['0'] = 7 - combo.total()
    return f(7) // prod(map(f, combo.values()))

def get_total(cnt, total=0):
    if len(cnt) <= 1: return total
    new_cnt = Counter()
    for k, v in cnt.items(): new_cnt[sqdsum(k)] += v
    total += new_cnt.pop(89)
    return get_total(new_cnt, total)

cnt = Counter()
for num_digits in range(1, 8):
    for combo in combs(map(str, range(1, 10)), num_digits):
        cnt[sqdsum(int(''.join(combo)))] += numbers(combo)

print(get_total(cnt))
```

93 Arithmetic Expressions

Problem [Project Euler]

By using each of the digits from the set, $\{1, 2, 3, 4\}$, exactly once, and making use of the four arithmetic operations $(+, -, \times, /)$ and brackets/parentheses, it is possible to form different positive integer targets.
For example,

$$\begin{aligned} 8 &= (4 \times (1 + 3))/2 \\ 14 &= 4 \times (3 + 1/2) \\ 19 &= 4 \times (2 + 3) - 1 \\ 36 &= 3 \times 4 \times (2 + 1) \end{aligned}$$

Note that concatenations of the digits, like $12 + 34$, are not allowed.
Using the set, $\{1, 2, 3, 4\}$, it is possible to obtain thirty-one different target numbers of which 36 is the maximum, and each of the numbers 1 to 28 can be obtained before encountering the first non-expressible number.
Find the set of four distinct digits, $a < b < c < d$, for which the longest set of consecutive positive integers, 1 to n, can be obtained, giving your answer as a string: *abcd*.

Solution [Buy me a beer]

To solve this problem, we first define all possible operations, including non-commutative variations for subtraction and division, where we consider both $a-b$ and $b-a$, and similarly for division. This gives us six distinct operations to choose from.
Next, we generate all combinations of three operators to apply to each set of four digits. For each set of digits, we test every possible ordering of the digits, using both the standard and reversed versions of each operation.
To evaluate the expressions, we iterate over all permutations of the digits $\{a, b, c, d\}$ and apply the selected operators in every possible way. For each expression, we check if the result is a positive integer by ensuring the denominator is 1 and the value is greater than zero. These valid integer results are collected in a set.
After generating results for each combination, we examine the set of integers to determine the longest sequence of consecutive positive integers starting from 1. The length of this sequence is recorded for each four-digit set.
Finally, by comparing the lengths of these sequences across all four-digit combinations, we identify the set that produces the longest sequence. We return this set as a concatenated string to form our answer.

Code [GitHub]

```python
from itertools import product, combinations, permutations
from operator import add, sub, mul, truediv as div
from fractions import Fraction

sub2 = lambda x, y: sub(y, x)
div2 = lambda x, y: div(y, x)

all_ops = [add, sub, sub2, mul, div, div2]
operators = list(product(all_ops, repeat=3))
digs = list(combinations(map(Fraction, range(1, 10)), 4))

def integers(digits: tuple) -> set:
    ints = set()
    for ops, ds in product(operators, permutations(digits)):
        try:
            x = ops[0](ds[0], ds[1])
            for i in range(1, 3): x = ops[i](x, ds[i + 1])
            if x.denominator == 1 and x.numerator > 0:
                ints.add(x.numerator)
        except ZeroDivisionError: continue
    return ints

def consecutive(ints: set) -> int:
    for n in range(1, len(ints) + 1):
        if n not in ints: return n - 1
    return len(ints)

max_length, best_digits = 0, None
for digits in digs:
    consecutive_length = consecutive(integers(digits))
    if consecutive_length > max_length:
        max_length, best_digits = consecutive_length, digits

print(''.join(str(n) for n in best_digits))
```

94 Almost Equilateral Triangles

Problem [Project Euler]

It is easily proved that no equilateral triangle exists with integral length sides and integral area. However, the *almost equilateral triangle* 5-5-6 has an area of 12 square units.
We shall define an *almost equilateral triangle* to be a triangle for which two sides are equal and the third differs by no more than one unit.
Find the sum of the perimeters of all *almost equilateral triangles* with integral side lengths and area and whose perimeters do not exceed one billion (1 000 000 000).

Solution [Buy me a beer]

In this solution, we aim to find the sum of the perimeters of all *almost equilateral triangles* with integer side lengths and integer areas, where the perimeter does not exceed one billion. An *almost equilateral triangle* is defined as a triangle with two equal sides and a third side differing by no more than one unit from these.
Our approach leverages the fact that the areas of certain integer-sided triangles—specifically, Heronian triangles that are almost equilateral—can be generated using recurrence relations. This sequence, available in the *Online Encyclopedia of Integer Sequences* (OEIS), allows us to calculate subsequent triangles by relying on known initial values.
The solution follows these steps:

1. We begin with a known base case: the triangle with sides 5, 5, and 6.
2. Using recurrence relations, we iteratively generate the next *almost equilateral triangle* by:
   - ▷ Calculating the next area based on previous area values.
   - ▷ Alternating the sign for the recurrence relation to adjust the length of the third side.
3. For each generated triangle, we compute the perimeter as $3 \times \text{side} + \text{sign}$. If this perimeter is within the one-billion limit, we add it to a cumulative total.

The loop stops once the perimeter exceeds the specified limit. Finally, the cumulative sum of the perimeters for all qualifying triangles is printed as the solution.

Code [GitHub]

```python
total_perimeter_sum = 0
side, prev_area, next_area, sign = 5, 8, 28, 1

while (perimeter := 3 * side + sign) <= 10**9:
    total_perimeter_sum += perimeter

    prev_area, next_area, sign = next_area, 4 * next_area -
        ↪prev_area, -sign
    side = (2 * sign + next_area) // 6

print(total_perimeter_sum)
```

95 Amicable Chains

Problem [Project Euler]

The proper divisors of a number are all the divisors excluding the number itself. For example, the proper divisors of 28 are 1, 2, 4, 7, and 14. As the sum of these divisors is equal to 28, we call it a perfect number.
Interestingly, the sum of the proper divisors of 220 is 284 and the sum of the proper divisors of 284 is 220, forming a chain of two numbers. For this reason, 220 and 284 are called an amicable pair.
Perhaps less well known are longer chains. For example, starting with 12496, we form a chain of five numbers:

$$12496 \to 14288 \to 15472 \to 14536 \to 14264 (\to 12496 \to \cdots)$$

Since this chain returns to its starting point, it is called an amicable chain.
Find the smallest member of the longest amicable chain with no element exceeding one million.

Solution [Buy me a beer]

▷ First, we initialize an array, `divsum`, where each index i contains the sum of proper divisors of i. To fill this array efficiently, we iterate over each integer n from 1 up to our limit. For each n, we add n to all multiples of n (i.e., $2n, 3n, \ldots$), thus accumulating the sum of divisors for each number.

▷ Next, we define a function, `get_chain`, which takes an initial number (origin) and constructs its divisor chain. Starting with this number, we repeatedly look up the next number in the chain using `divsum`. If a number in the chain exceeds one million or repeats without forming a complete cycle, we discard that chain. Otherwise, if the chain loops back to the origin, we return it as an amicable chain.

▷ We then apply `get_chain` to each number up to one million and identify the longest chain by comparing chain lengths. Using Python's `max` function, we select the chain with the maximum length, and we return the smallest number in this chain as the result.

▷ Finally, we print the smallest member of the longest chain found.

Code [GitHub]

```python
LIMIT = 10**6

divsum = [0] * (LIMIT + 1)

for n in range(1, LIMIT + 1):
    for m in range(2*n, LIMIT + 1, n):
        divsum[m] += n

def get_chain(origin):
    chain = [origin]
    while True:
        if (nxt := divsum[chain[-1]]) > 10**6: return []
        if nxt in chain:
            return chain if nxt == chain[0] else []
        chain.append(nxt)

longest_chain = max(map(get_chain, range(LIMIT + 1)), key=
    ↪lambda x: len(x))
print(min(longest_chain))
```

96 Su Doku

Problem [Project Euler]

Su Doku (Japanese meaning *number place*) is the name given to a popular puzzle concept. Its origin is unclear, but credit must be attributed to Leonhard Euler who invented a similar, and much more difficult, puzzle idea called Latin Squares. The objective of Su Doku puzzles, however, is to replace the blanks (or zeros) in a 9 by 9 grid such that each row, column, and 3 by 3 box contains each of the digits 1 to 9. Below is an example of a typical starting puzzle grid and its solution grid.

0	0	3	0	2	0	6	0	0
9	0	0	3	0	5	0	0	1
0	0	1	8	0	6	4	0	0
0	0	8	1	0	2	9	0	0
7	0	0	0	0	0	0	0	8
0	0	6	7	0	8	2	0	0
0	0	2	6	0	9	5	0	0
8	0	0	2	0	3	0	0	9
0	0	5	0	1	0	3	0	0

4	8	3	9	2	1	6	5	7
9	6	7	3	4	5	8	2	1
2	5	1	8	7	6	4	9	3
5	4	8	1	3	2	9	7	6
7	2	9	5	6	4	1	3	8
1	3	6	7	9	8	2	4	5
3	7	2	6	8	9	5	1	4
8	1	4	2	5	3	7	6	9
6	9	5	4	1	7	3	8	2

A well-constructed Su Doku puzzle has a unique solution and can be solved by logic, although it may be necessary to employ "guess and test" methods in order to eliminate options (there is much contested opinion over this). The complexity of the search determines the difficulty of the puzzle; the example above is considered *easy* because it can be solved by straightforward direct deduction.
The 6K text file, sudoku.txt, contains fifty different Su Doku puzzles ranging in difficulty, but all with unique solutions (the first puzzle in the file is the example above).
By solving all fifty puzzles, find the sum of the 3-digit numbers found in the top left corner of each solution grid; for example, 483 is the 3-digit number found in the top left corner of the solution grid above.

Solution [Buy me a beer]

We begin by reading the file and parsing its contents into individual 9x9 grids, where each Sudoku is represented as a list of lists of integers. Each row in the grid becomes a list, allowing us to treat each puzzle as a structured 2D array.
To solve each puzzle, we implement a backtracking approach. For each cell, we attempt to place numbers from 1 to 9, checking if each placement is valid according to Sudoku's rules. Our `is_valid` function handles these checks by confirming that no number appears more than once in the same row, column, or 3x3 subgrid. If a number is valid, we place it temporarily and recursively try to complete the puzzle with the remaining empty cells. If we encounter a dead end, we backtrack by resetting the cell and testing the next possible number, eventually finding a valid configuration due to the uniqueness of each puzzle's solution.
Once a puzzle is solved, we extract the 3-digit number from the top-left corner by interpreting the first three values in the first row as a hundreds, tens, and units place. We calculate this as `s[0][0] * 100 + s[0][1] * 10 + s[0][2]`. We add this 3-digit number

to our running total for each puzzle.
After processing all the puzzles, the accumulated total in `result` represents the sum of the top-left corner values for each solved Sudoku. We then print `result`, which provides the final answer.

Code [GitHub]

```python
import re
from itertools import product

with open("p096_sudoku.txt") as f:
    rows = re.findall(r'\d{9}', f.read())

sudokus = [
    [list(map(int, rows[9*i + j])) for j in range(9)]
    for i in range(len(rows) // 9)
]

def is_valid(sudoku, i, j, n):
    for k in range(9):
        if sudoku[i][k] == n or sudoku[k][j] == n:
            return False

    s_i, s_j = 3 * (i // 3), 3 * (j // 3)
    ranges = range(s_i, s_i + 3), range(s_j, s_j + 3)
    for i, j in product(*ranges):
        if sudoku[i][j] == n: return False
    return True

def solve(sudoku):
    for i, j in product(range(9), repeat=2):
        if sudoku[i][j] != 0: continue
        for n in range(1, 10):
            if is_valid(sudoku, i, j, n):
                sudoku[i][j] = n
                if solve(sudoku): return True
                sudoku[i][j] = 0
        return False
    return True

result = 0
for s in sudokus:
    solve(s)
    result += sum(s[0][j] * 10**(2 - j) for j in range(3))

print(result)
```

97 Large Non-Mersenne Prime

Problem [Project Euler]

The first known prime found to exceed one million digits was discovered in 1999, and is a Mersenne prime of the form $2^{6972593} - 1$; it contains exactly 2 098 960 digits. Subsequently, other Mersenne primes, of the form $2^p - 1$, have been found which contain more digits.
However, in 2004, there was found a massive non-Mersenne prime which contains 2 357 207 digits:

$$28433 \times 2^{7830457} + 1.$$

Find the last ten digits of this prime number.

Solution [Buy me a beer]

To find the last ten digits of the massive non-Mersenne prime $28433 \times 2^{7830457} + 1$, we focus on modular arithmetic.
First, we set `MOD = 10**10` to work only within the last ten digits. Using Python's `pow` function, we compute $2^{7830457} \mod 10^{10}$ directly, which gives us the last ten digits of $2^{7830457}$ without handling the entire number. Next, we multiply this result by 28433, add 1, and take it modulo 10^{10} again to ensure we're still within the last ten digits.
This final result is printed, providing exactly the last ten digits of $28433 \times 2^{7830457} + 1$. This approach keeps calculations efficient by only working with the digits we need.

Code [GitHub]

```python
MOD = 10**10

digits = 28_433 * pow(2, 7_830_457, MOD) + 1

print(digits % MOD)
```

98 Anagramic Squares

Problem [Project Euler]

By replacing each of the letters in the word CARE with 1, 2, 9, and 6 respectively, we form a square number: $1296 = 36^2$. What is remarkable is that, by using the same digital substitutions, the anagram, RACE, also forms a square number: $9216 = 96^2$. We shall call CARE (and RACE) a square anagram word pair and specify further that leading zeroes are not permitted, and neither may a different letter have the same digital value as another letter.
Using words.txt, a 16K text file containing nearly two thousand common English words, find all the square anagram word pairs (a palindromic word is NOT considered to be an anagram of itself).
What is the largest square number formed by any member of such a pair?
NOTE: All anagrams formed must be contained in the given text file.

Solution [Buy me a beer]

First, we load all words from the file and use a dictionary to group anagrams, where each key is a sorted tuple of the letters in the word, and the value is a list of words sharing that sorted letter pattern. This way, words with the same letters end up together.
Next, we gather pairs of words from each anagram group that has more than one word, ignoring single-word groups since they can't form pairs.
Then, we define a helper function, `is_square`, to check if a number is a perfect square. For each anagram pair, we generate all possible digit mappings for their unique letters, assigning a unique digit to each letter.
We apply each mapping to convert the letters in each word of the pair into numbers. If a number has a leading zero, we skip that mapping (since leading zeroes aren't allowed).
Finally, we check if both numbers in the pair are squares. If they are, we add them to a set of squares. Once all pairs are processed, we output the maximum square found among them.

Code [GitHub]

```python
from itertools import combinations, permutations
from collections import defaultdict
from string import digits
from math import sqrt
from re import findall

with open("0098_words.txt") as f:
    words = findall(r'\w+', f.read())

groups = defaultdict(list)
for word in words: groups[tuple(sorted(word))].append(word)

anagrams = []
for group in groups.values():
    if len(group) == 1: continue
    for comb in combinations(group, 2):
        anagrams.append(comb)

is_square = lambda n: sqrt(n).is_integer()

squares = set()
for pair in anagrams:
    letters = sorted(set(pair[0]))

    for perm in permutations(digits, l := len(letters)):
        mapping = {letters[i]: perm[i] for i in range(l)}
        numbers = [''.join(mapping[word[i]] for i in range(l
            ↪)) for word in pair]

        if any(n[0] == '0' for n in numbers): continue

        numbers = set(map(int, numbers))
        if all(map(is_square, numbers)): squares |= numbers

print(max(squares))
```

99 Largest Exponential

Problem [Project Euler]

Comparing two numbers written in index form like 2^{11} and 3^7 is not difficult, as any calculator would confirm that $2^{11} = 2048 < 3^7 = 2187$.
However, confirming that $632382^{518061} > 519432^{525806}$ would be much more difficult, as both numbers contain over three million digits.
Using base_exp.txt, a 22K text file containing one thousand lines with a base/exponent pair on each line, determine which line number has the greatest numerical value.
NOTE: The first two lines in the file represent the numbers in the example given above.

Solution [Buy me a beer]

First, we open the file `base_exp.txt` to access each line containing a base and an exponent. Each line gives a number in the form `base,exponent`, so we iterate over each line directly.
Next, we initialize `max_line` to store the line number with the highest computed value so far and `max_value` to store that computed value. We calculate a comparable value for each base-exponent pair by using the formula `exp` $\cdot \log($`base`$)$. This method is based on the idea that comparing powers a^x and b^y can be simplified by comparing $x \cdot \log(a)$ and $y \cdot \log(b)$.
As we go through each line, we:

1. Extract `base` and `exp` by splitting the line at the comma and converting each to an integer.
2. Compute the value of `exp` $\cdot \log($`base`$)$.
3. If this value is greater than `max_value`, we update `max_value` and set `max_line` to the current line number.

After processing all lines, `max_line` will hold the line number of the base-exponent pair with the greatest numerical value, which we print as our final answer. This solution avoids calculating large powers directly by using logarithms, which is much more efficient.

Code [GitHub]

```python
from math import log

with open("0099_base_exp.txt") as f:
    basexps = f.readlines()

def value(n):
    base, exp = map(int, basexps[n - 1].split(','))
    return exp * log(base)

print(max(range(1, len(basexps) + 1), key=value))
```

100 Arranged Probability

Problem [Project Euler]

If a box contains twenty-one coloured discs, composed of fifteen blue discs and six red discs, and two discs were taken at random, it can be seen that the probability of taking two blue discs, $P(\mathrm{BB}) = \frac{15}{21} \times \frac{14}{20} = \frac{1}{2}$.
The next such arrangement, for which there is exactly 50% chance of taking two blue discs at random, is a box containing eighty-five blue discs and thirty-five red discs.
By finding the first arrangement to contain over $10^{12} = 1\,000\,000\,000\,000$ discs in total, determine the number of blue discs that the box would contain.

Solution [Buy me a beer]

To solve this problem, we begin with the requirement that the probability of drawing two blue discs is exactly $\frac{1}{2}$. Given that the total number of discs is $n = b + r$, where b represents the number of blue discs and r the number of red discs, we can set up the probability condition as follows:

$$P(\mathrm{BB}) = \frac{b}{n} \cdot \frac{b-1}{n-1} = \frac{1}{2}.$$

Rearranging this, we derive the Diophantine equation:

$$2b^2 - n^2 - 2b + n = 0.$$

Solving this equation directly is challenging, but it leads to a recurrence relation that generates solutions. With initial values $b = 15$ and $n = 21$, we derive the recurrence relations:

$$b' = 3b + 2n - 2, \quad n' = 4b + 3n - 3.$$

Starting from $b = 15$ and $n = 21$, we use these recurrence formulas iteratively to generate new values for b and n until n exceeds 10^{12}. The first b that satisfies this condition gives the required count of blue discs.
This approach avoids the need for brute force, leveraging the structure of the recurrence relation derived from the Diophantine equation to efficiently reach the solution.

Code [GitHub]

```python
b, n = 15, 21
while n <= 10**12:
    b, n = 3*b + 2*n - 2, 4*b + 3*n - 3

print(b)
```

101 Optimum Polynomial

Problem [Project Euler]

If we are presented with the first k terms of a sequence, it is impossible to determine with certainty the value of the next term, as there are infinitely many polynomial functions that can model the sequence.
As an example, consider the sequence of cube numbers. This is defined by the generating function, $u_n = n^3$: $1, 8, 27, 64, 125, 216, \ldots$
Suppose we were only given the first two terms of this sequence. Working on the principle that "simple is best," we might assume a linear relationship and predict the next term to be 15 (common difference 7). Even if we had the first three terms, by the same principle of simplicity, we would assume a quadratic relationship.
We define $\mathrm{OP}(k, n)$ as the n-th term of the optimum polynomial generating function for the first k terms of a sequence. It is clear that $\mathrm{OP}(k, n)$ will accurately generate the terms of the sequence for $n \leq k$, and potentially the *first incorrect term* (FIT) will be $\mathrm{OP}(k, k+1)$; in which case we shall

call it a *bad OP* (BOP).
As a basis, if we are only given the first term of a sequence, it makes sense to assume constancy; that is, for $n \geq 2$, $\mathrm{OP}(1, n) = u_1$.
Thus, we obtain the following OPs for the cubic sequence:

$\mathrm{OP}(1, n) = 1$	1, **1**, 1, 1, . . .
$\mathrm{OP}(2, n) = 7n - 6$	1, 8, **15**, . . .
$\mathrm{OP}(3, n) = 6n^2 - 11n + 6$	1, 8, 27, **58**, . . .
$\mathrm{OP}(4, n) = n^3$	1, 8, 27, 64, 125, . . .

Clearly, no BOPs exist for $k \geq 4$.
By considering the sum of FITs generated by the BOPs (indicated in **red** above), we obtain $1 + 15 + 58 = 74$.
Consider the following tenth-degree polynomial generating function:

$$u_n = 1 - n + n^2 - n^3 + n^4 - n^5 + n^6 - n^7 + n^8 - n^9 + n^{10}.$$

Find the sum of FITs for the BOPs.

Solution [Buy me a beer]

From here on, you're on your own. Good luck!
Don't forget to add me to your friends list with this key:

```
2152595_2ijDAbuxbUQikEeHElW44nRzIqNbLSnn
```

A Execution Times

	time (s)
1	0.02
2	0.02
3	0.02
4	0.18
5	0.02
6	0.02
7	0.43
8	0.02
9	0.08
10	0.80
11	0.02
12	0.28
13	0.02
14	1.25
15	0.02
16	0.03
17	0.04
18	0.06
19	0.02
20	0.02
21	0.06
22	0.03
23	2.67
24	0.02
25	0.04
26	0.03
27	2.22
28	0.02
29	0.03
30	0.45
31	0.02
32	0.87
33	0.03
34	3.19
35	1.40
36	0.38
37	0.45
38	0.28
39	0.02
40	0.42
41	0.51
42	0.03
43	1.04
44	0.33
45	0.14
46	0.39
47	0.69
48	0.02
49	0.37
50	1.34
51	0.51
52	0.52
53	0.02
54	0.04
55	0.05
56	0.08
57	0.04
58	1.85
59	0.19
60	2.31
61	0.46
62	0.04
63	0.02
64	0.16
65	0.02
66	0.03
67	0.02
68	2.71
69	1.41
70	17.52
71	0.02
72	1.27
73	2.89
74	0.82
75	1.33
76	0.03
77	0.42
78	4.16
79	0.02
80	0.05
81	0.03
82	0.04
83	0.04
84	0.14
85	0.30
86	1.73
87	0.94
88	0.18
89	0.03
90	0.38
91	4.15
92	0.08
93	3.00
94	0.02
95	6.85
96	11.89
97	0.02
98	27.47
99	0.02
100	0.02