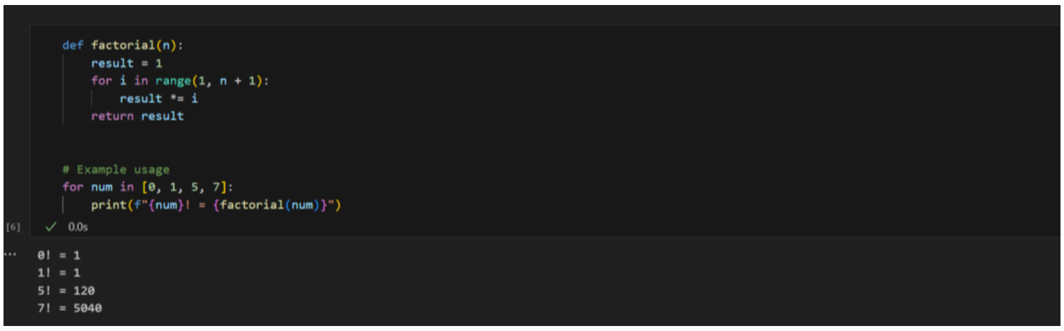
CSE VITyarthi Project

1. Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).

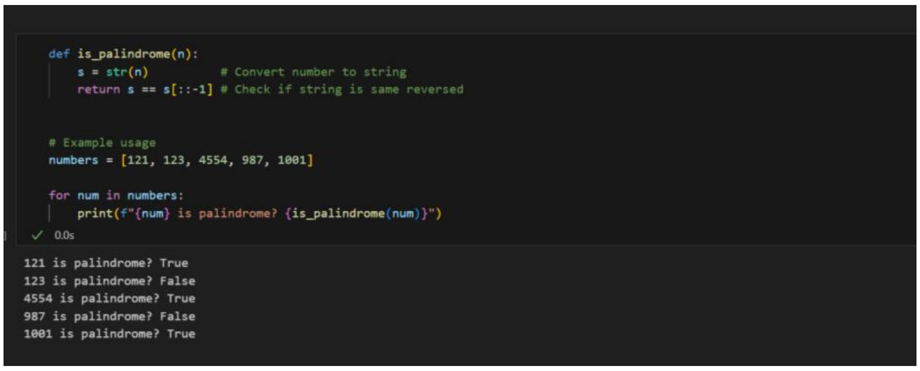


```python
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result


# Example usage
for num in [0, 1, 5, 7]:
    print(f"{num}! = {factorial(num)}")
```

[6] ✓ 0.0s

```
0! = 1
1! = 1
5! = 120
7! = 5040
```

2. Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.

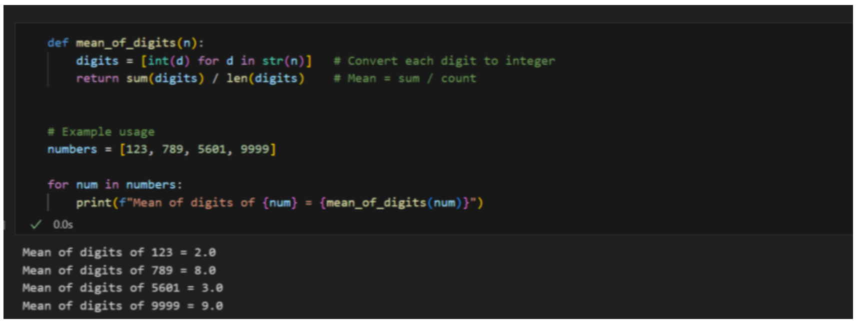


```python
def is_palindrome(n):
    s = str(n)           # Convert number to string
    return s == s[::-1] # Check if string is same reversed


# Example usage
numbers = [121, 123, 4554, 987, 1001]

for num in numbers:
    print(f"{num} is palindrome? {is_palindrome(num)}")
```

✓ 0.0s

```
121 is palindrome? True
123 is palindrome? False
4554 is palindrome? True
987 is palindrome? False
1001 is palindrome? True
```

3. Write a function mean_of_digits(n) that returns the average of all digits in a number.

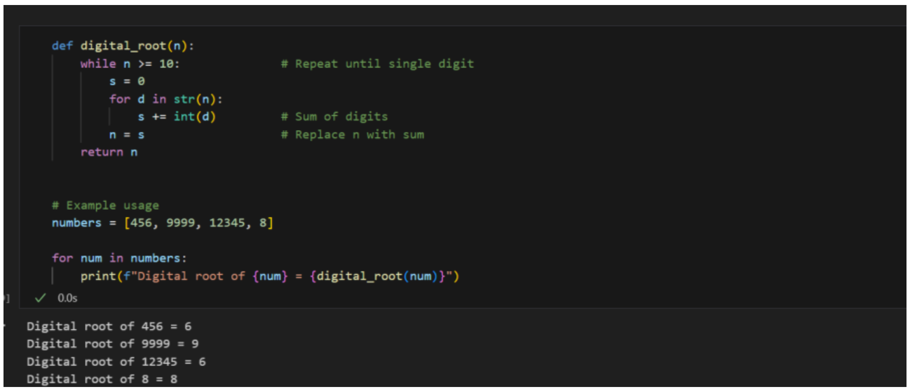


```python
def mean_of_digits(n):
    digits = [int(d) for d in str(n)]   # Convert each digit to integer
    return sum(digits) / len(digits)    # Mean = sum / count


# Example usage
numbers = [123, 789, 5601, 9999]

for num in numbers:
    print(f"Mean of digits of {num} = {mean_of_digits(num)}")
```

✓ 0.0s

```
Mean of digits of 123 = 2.0
Mean of digits of 789 = 8.0
Mean of digits of 5601 = 3.0
Mean of digits of 9999 = 9.0
```

4. Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.

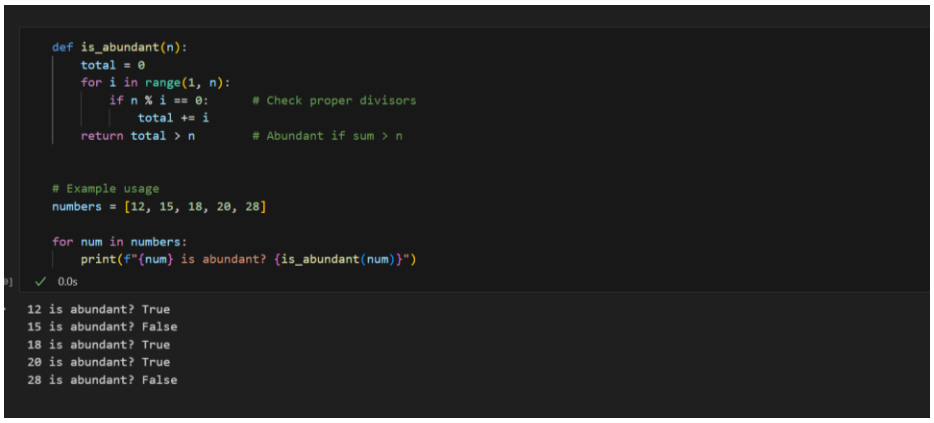


```python
def digital_root(n):
    while n >= 10:              # Repeat until single digit
        s = 0
        for d in str(n):
            s += int(d)         # Sum of digits
        n = s                   # Replace n with sum
    return n


# Example usage
numbers = [456, 9999, 12345, 8]

for num in numbers:
    print(f"Digital root of {num} = {digital_root(num)}")
```

✓ 0.0s

```
Digital root of 456 = 6
Digital root of 9999 = 9
Digital root of 12345 = 6
Digital root of 8 = 8
```

5. Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n.

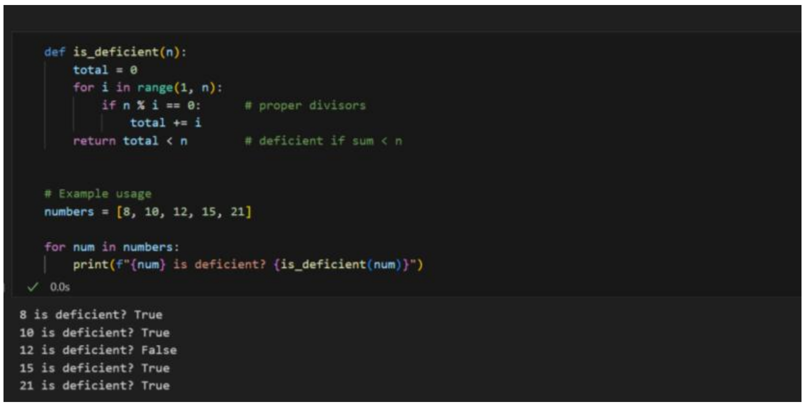


```python
def is_abundant(n):
    total = 0
    for i in range(1, n):
        if n % i == 0:      # Check proper divisors
            total += i
    return total > n        # Abundant if sum > n


# Example usage
numbers = [12, 15, 18, 20, 28]

for num in numbers:
    print(f"{num} is abundant? {is_abundant(num)}")
```

✓ 0.0s

```
12 is abundant? True
15 is abundant? False
18 is abundant? True
20 is abundant? True
28 is abundant? False
```

6. Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n

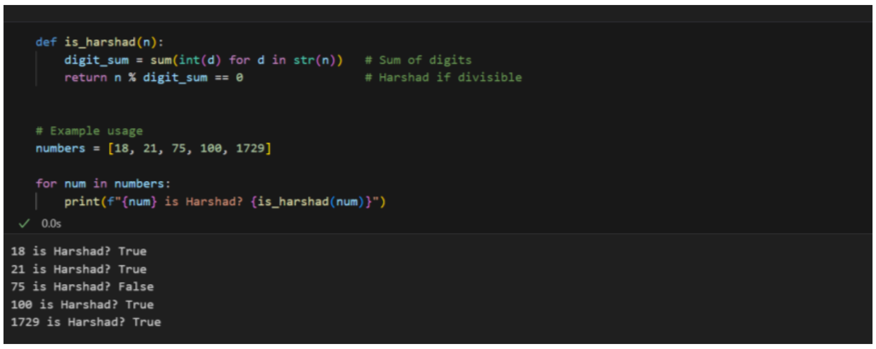


```python
def is_deficient(n):
    total = 0
    for i in range(1, n):
        if n % i == 0:      # proper divisors
            total += i
    return total < n        # deficient if sum < n


# Example usage
numbers = [8, 10, 12, 15, 21]

for num in numbers:
    print(f"{num} is deficient? {is_deficient(num)}")
```

✓ 0.0s

```
8 is deficient? True
10 is deficient? True
12 is deficient? False
15 is deficient? True
21 is deficient? True
```

7. Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.



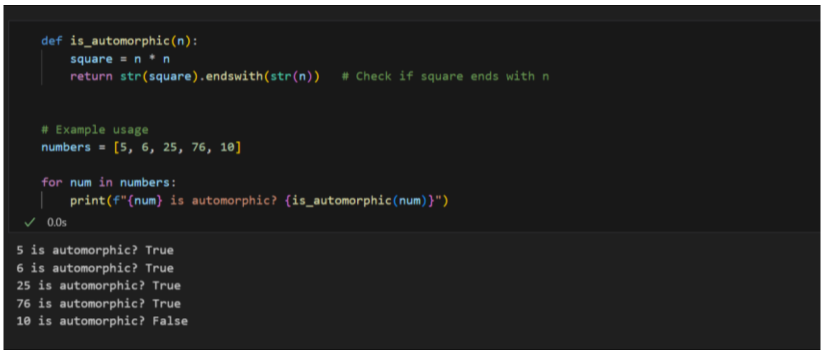
```python
def is_harshad(n):
    digit_sum = sum(int(d) for d in str(n))    # Sum of digits
    return n % digit_sum == 0                  # Harshad if divisible


# Example usage
numbers = [18, 21, 75, 100, 1729]

for num in numbers:
    print(f"{num} is Harshad? {is_harshad(num)}")
```

✓ 0.0s

```
18 is Harshad? True
21 is Harshad? True
75 is Harshad? False
100 is Harshad? True
1729 is Harshad? True
```

8. Write a function is_automorphic(n) that checks if a

number's square ends with the number itself.



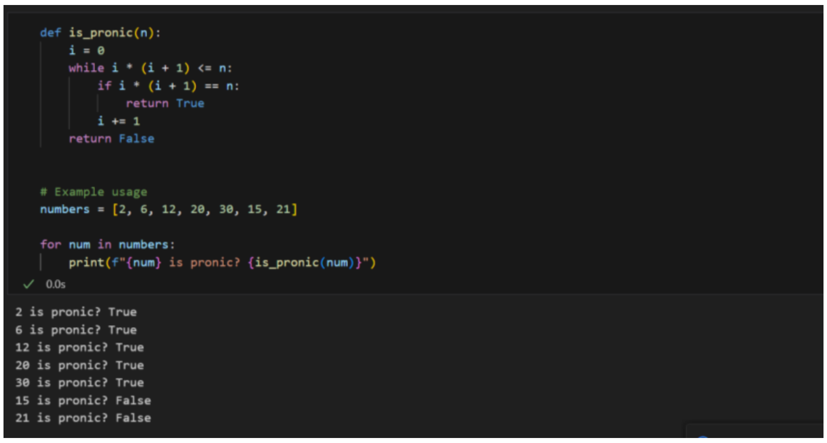
```python
def is_automorphic(n):
    square = n * n
    return str(square).endswith(str(n))   # Check if square ends with n


# Example usage
numbers = [5, 6, 25, 76, 10]

for num in numbers:
    print(f"{num} is automorphic? {is_automorphic(num)}")
```

✓ 0.0s

```
5 is automorphic? True
6 is automorphic? True
25 is automorphic? True
76 is automorphic? True
10 is automorphic? False
```

9. Write a function is_pronic(n) that checks if a number is

the product of two consecutive integers.



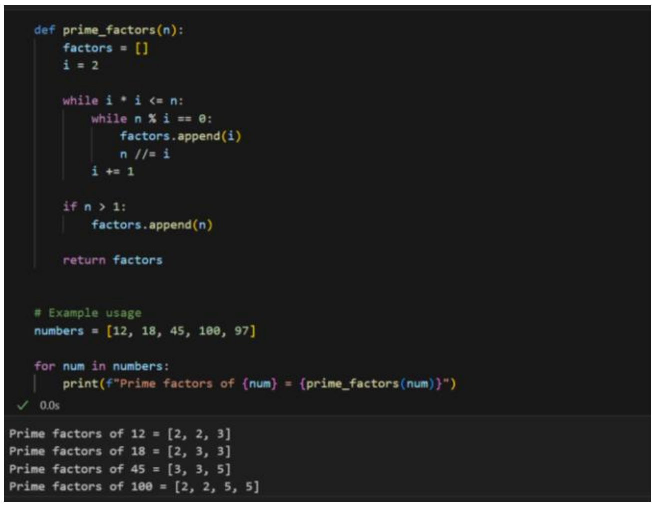
```python
def is_pronic(n):
    i = 0
    while i * (i + 1) <= n:
        if i * (i + 1) == n:
            return True
        i += 1
    return False


# Example usage
numbers = [2, 6, 12, 20, 30, 15, 21]

for num in numbers:
    print(f"{num} is pronic? {is_pronic(num)}")
```

✓ 0.0s

```
2 is pronic? True
6 is pronic? True
12 is pronic? True
20 is pronic? True
30 is pronic? True
15 is pronic? False
21 is pronic? False
```

10. Write a function prime_factors(n) that returns the list of

prime factors of a number.

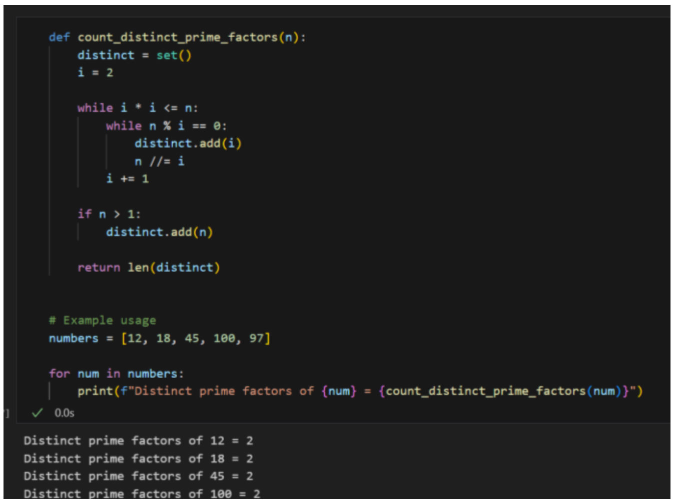


```
def prime_factors(n):
    factors = []
    i = 2

    while i * i <= n:
        while n % i == 0:
            factors.append(i)
            n //= i
        i += 1

    if n > 1:
        factors.append(n)

    return factors


# Example usage
numbers = [12, 18, 45, 100, 97]

for num in numbers:
    print(f"Prime factors of {num} = {prime_factors(num)}")
```

```
✓ 0.0s
```

```
Prime factors of 12 = [2, 2, 3]
Prime factors of 18 = [2, 3, 3]
Prime factors of 45 = [3, 3, 5]
Prime factors of 100 = [2, 2, 5, 5]
```

11. Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.

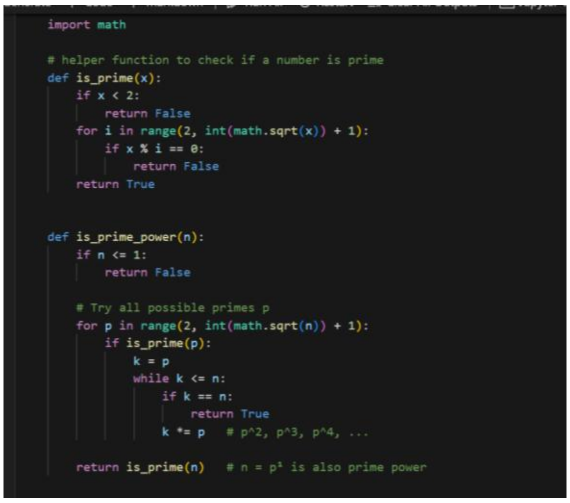


```
def count_distinct_prime_factors(n):
    distinct = set()
    i = 2

    while i * i <= n:
        while n % i == 0:
            distinct.add(i)
            n //= i
        i += 1

    if n > 1:
        distinct.add(n)

    return len(distinct)


# Example usage
numbers = [12, 18, 45, 100, 97]

for num in numbers:
    print(f"Distinct prime factors of {num} = {count_distinct_prime_factors(num)}")
```

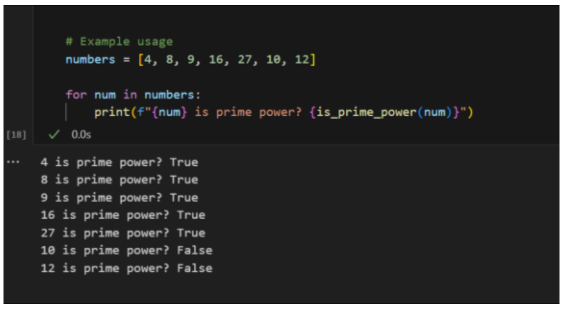
```
✓ 0.0s
```

```
Distinct prime factors of 12 = 2
Distinct prime factors of 18 = 2
Distinct prime factors of 45 = 2
Distinct prime factors of 100 = 2
```

12.Write a function is_prime_power(n) that checks if a number can be expressed as p^k where p is prime and $k \geq 1$.



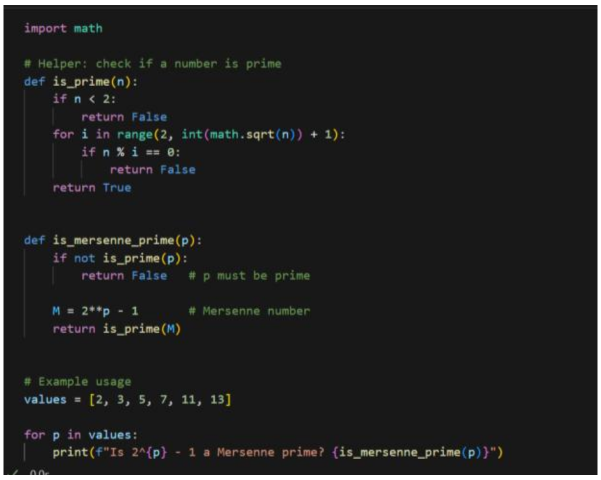
```
import math

# helper function to check if a number is prime
def is_prime(x):
    if x < 2:
        return False
    for i in range(2, int(math.sqrt(x)) + 1):
        if x % i == 0:
            return False
    return True


def is_prime_power(n):
    if n <= 1:
        return False

    # Try all possible primes p
    for p in range(2, int(math.sqrt(n)) + 1):
        if is_prime(p):
            k = p
            while k <= n:
                if k == n:
                    return True
                k *= p   # p^2, p^3, p^4, ...

    return is_prime(n)   # n = p¹ is also prime power
```



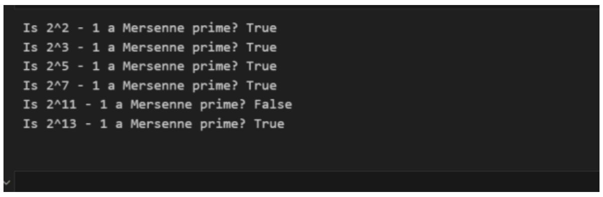
```python
# Example usage
numbers = [4, 8, 9, 16, 27, 10, 12]

for num in numbers:
    print(f"{num} is prime power? {is_prime_power(num)}")
```

```
4 is prime power? True
8 is prime power? True
9 is prime power? True
16 is prime power? True
27 is prime power? True
10 is prime power? False
12 is prime power? False
```

13. Write a function is_mersenne_prime(p) that checks if $2^p - 1$ is a prime number (given that p is prime).

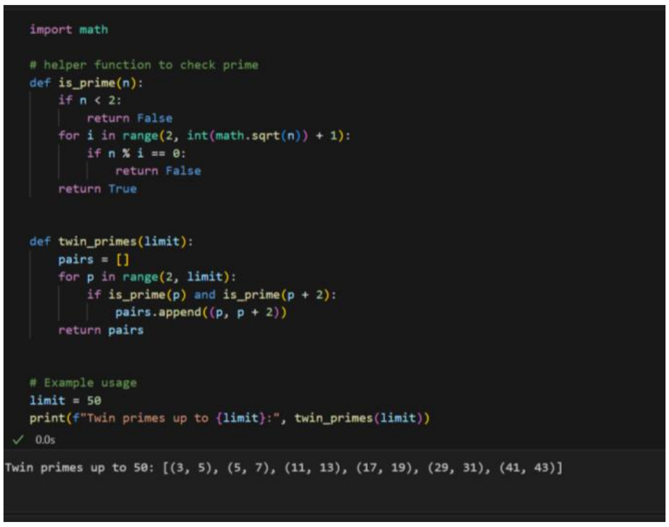


```python
import math

# Helper: check if a number is prime
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True


def is_mersenne_prime(p):
    if not is_prime(p):
        return False   # p must be prime

    M = 2**p - 1       # Mersenne number
    return is_prime(M)


# Example usage
values = [2, 3, 5, 7, 11, 13]

for p in values:
    print(f"Is 2^{p} - 1 a Mersenne prime? {is_mersenne_prime(p)}")
```



```
Is 2^2 - 1 a Mersenne prime? True
Is 2^3 - 1 a Mersenne prime? True
Is 2^5 - 1 a Mersenne prime? True
Is 2^7 - 1 a Mersenne prime? True
Is 2^11 - 1 a Mersenne prime? False
Is 2^13 - 1 a Mersenne prime? True
```

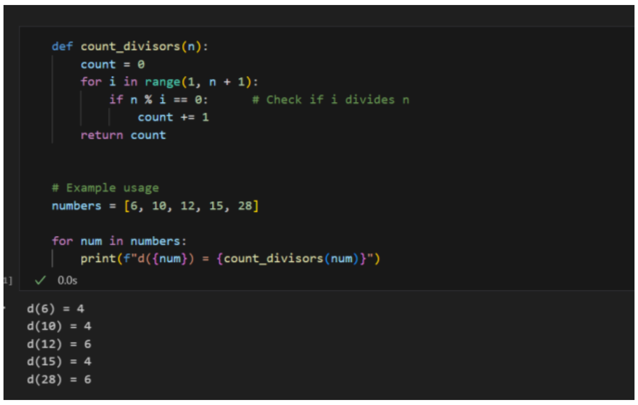
14. Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.



```python
import math

# helper function to check prime
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True


def twin_primes(limit):
    pairs = []
    for p in range(2, limit):
        if is_prime(p) and is_prime(p + 2):
            pairs.append((p, p + 2))
    return pairs


# Example usage
limit = 50
print(f"Twin primes up to {limit}:", twin_primes(limit))
```

```
Twin primes up to 50: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43)]
```

15. Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.

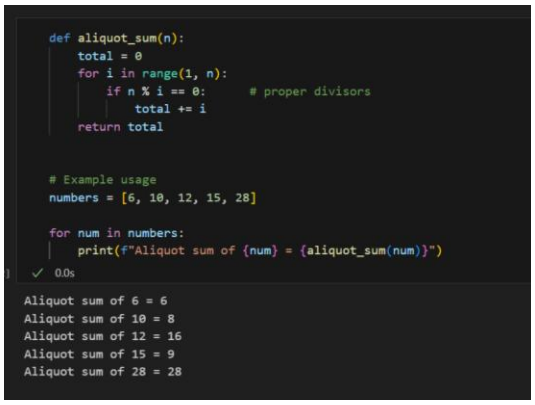


```python
def count_divisors(n):
    count = 0
    for i in range(1, n + 1):
        if n % i == 0:      # Check if i divides n
            count += 1
    return count


# Example usage
numbers = [6, 10, 12, 15, 28]

for num in numbers:
    print(f"d({num}) = {count_divisors(num)}")
```

```
✓ 0.0s
```

```
d(6) = 4
d(10) = 4
d(12) = 6
d(15) = 4
d(28) = 6
```

16. Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).

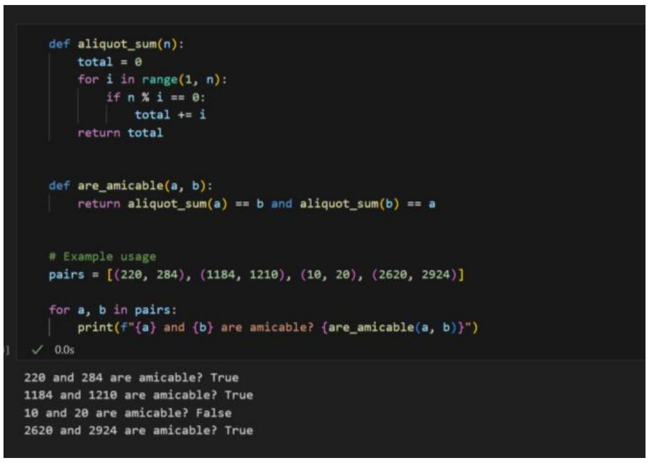


```python
def aliquot_sum(n):
    total = 0
    for i in range(1, n):
        if n % i == 0:      # proper divisors
            total += i
    return total


# Example usage
numbers = [6, 10, 12, 15, 28]

for num in numbers:
    print(f"Aliquot sum of {num} = {aliquot_sum(num)}")
```

```
✓ 0.0s
```

```
Aliquot sum of 6 = 6
Aliquot sum of 10 = 8
Aliquot sum of 12 = 16
Aliquot sum of 15 = 9
Aliquot sum of 28 = 28
```

17. Write a function are_amicable(a, b) that checks if two numbers are amicable (sum of proper divisors of a equals b and vice versa).

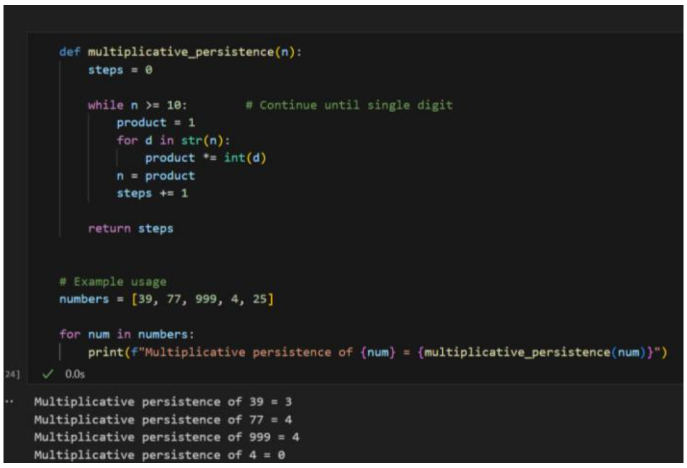


```python
def aliquot_sum(n):
    total = 0
    for i in range(1, n):
        if n % i == 0:
            total += i
    return total


def are_amicable(a, b):
    return aliquot_sum(a) == b and aliquot_sum(b) == a


# Example usage
pairs = [(220, 284), (1184, 1210), (10, 20), (2620, 2924)]

for a, b in pairs:
    print(f"{a} and {b} are amicable? {are_amicable(a, b)}")
```

```
✓ 0.0s
```

```
220 and 284 are amicable? True
1184 and 1210 are amicable? True
10 and 20 are amicable? False
2620 and 2924 are amicable? True
```

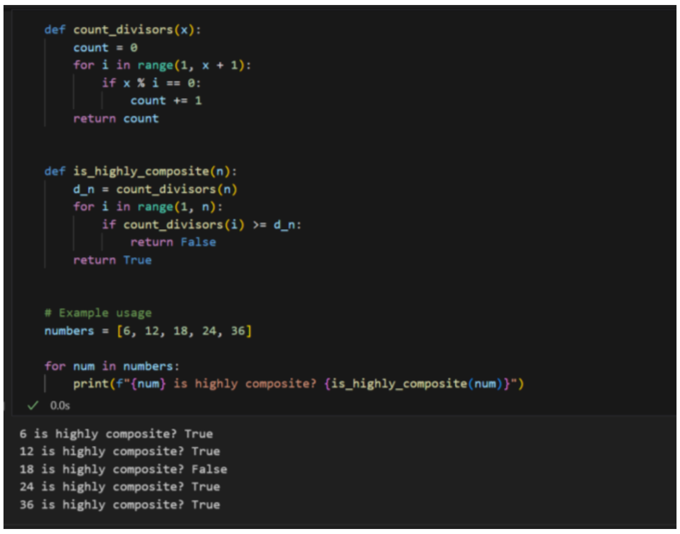
18. Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit



```python
def multiplicative_persistence(n):
    steps = 0

    while n >= 10:          # Continue until single digit
        product = 1
        for d in str(n):
            product *= int(d)
        n = product
        steps += 1

    return steps


# Example usage
numbers = [39, 77, 999, 4, 25]

for num in numbers:
    print(f"Multiplicative persistence of {num} = {multiplicative_persistence(num)}")
```

```
Multiplicative persistence of 39 = 3
Multiplicative persistence of 77 = 4
Multiplicative persistence of 999 = 4
Multiplicative persistence of 4 = 0
```

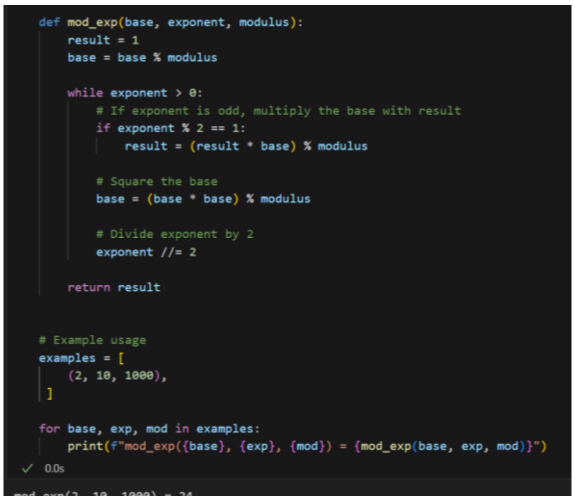
19. Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number



```python
def count_divisors(x):
    count = 0
    for i in range(1, x + 1):
        if x % i == 0:
            count += 1
    return count


def is_highly_composite(n):
    d_n = count_divisors(n)
    for i in range(1, n):
        if count_divisors(i) >= d_n:
            return False
    return True


# Example usage
numbers = [6, 12, 18, 24, 36]

for num in numbers:
    print(f"{num} is highly composite? {is_highly_composite(num)}")
```

```
6 is highly composite? True
12 is highly composite? True
18 is highly composite? False
24 is highly composite? True
36 is highly composite? True
```

20. Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates ($base^{exponent}$) % modulus

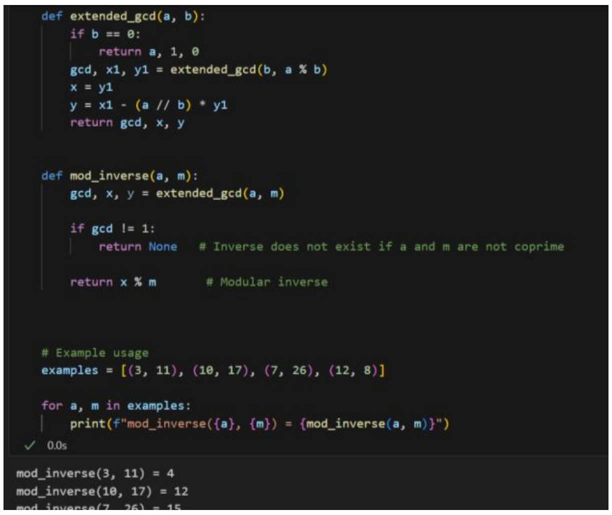


```python
def mod_exp(base, exponent, modulus):
    result = 1
    base = base % modulus

    while exponent > 0:
        # If exponent is odd, multiply the base with result
        if exponent % 2 == 1:
            result = (result * base) % modulus

        # Square the base
        base = (base * base) % modulus

        # Divide exponent by 2
        exponent //= 2

    return result


# Example usage
examples = [
    (2, 10, 1000),
]

for base, exp, mod in examples:
    print(f"mod_exp({base}, {exp}, {mod}) = {mod_exp(base, exp, mod)}")
```

✓ 0.0s

```
mod_exp(2, 10, 1000) = 24
```

21.Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.

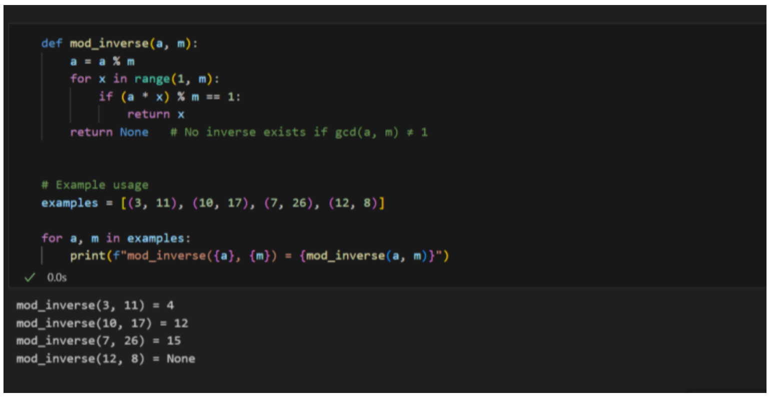


```python
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y


def mod_inverse(a, m):
    gcd, x, y = extended_gcd(a, m)

    if gcd != 1:
        return None   # Inverse does not exist if a and m are not coprime

    return x % m       # Modular inverse



# Example usage
examples = [(3, 11), (10, 17), (7, 26), (12, 8)]

for a, m in examples:
    print(f"mod_inverse({a}, {m}) = {mod_inverse(a, m)}")
```

✓ 0.0s

```
mod_inverse(3, 11) = 4
mod_inverse(10, 17) = 12
mod_inverse(7, 26) = 15
```

22. Write a function chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences x ≡ r_i mod m_i.

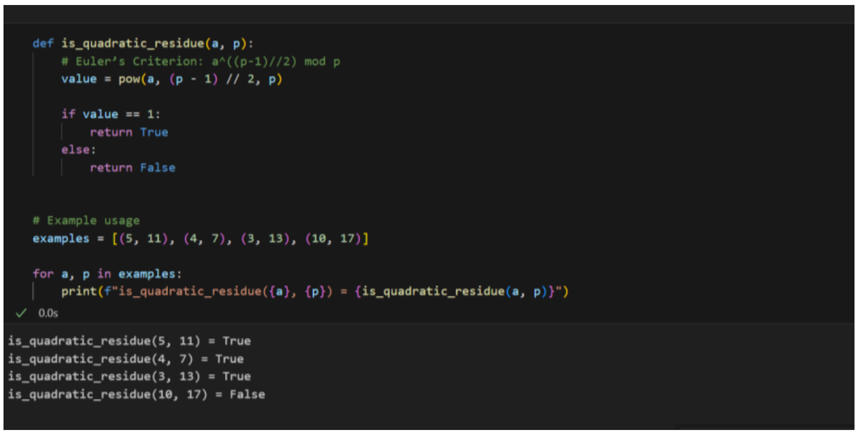


```python
def mod_inverse(a, m):
    a = a % m
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None   # No inverse exists if gcd(a, m) ≠ 1


# Example usage
examples = [(3, 11), (10, 17), (7, 26), (12, 8)]

for a, m in examples:
    print(f"mod_inverse({a}, {m}) = {mod_inverse(a, m)}")
```

✓ 0.0s

```
mod_inverse(3, 11) = 4
mod_inverse(10, 17) = 12
mod_inverse(7, 26) = 15
mod_inverse(12, 8) = None
```

23. Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if $x^2 \equiv a \bmod p$ has a solution



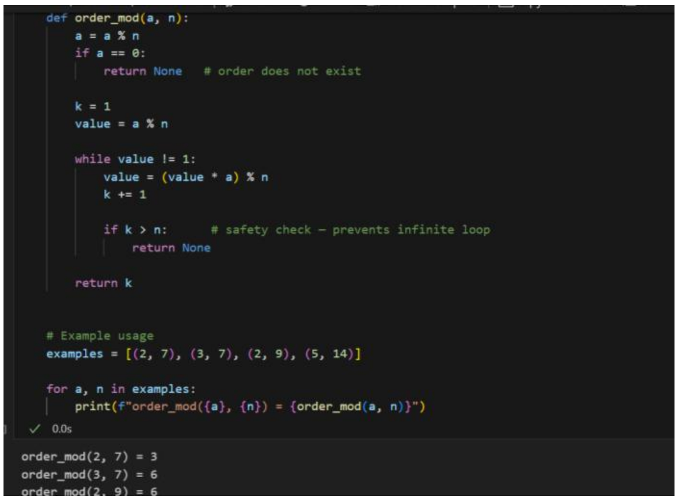
```python
def is_quadratic_residue(a, p):
    # Euler's Criterion: a^((p-1)//2) mod p
    value = pow(a, (p - 1) // 2, p)

    if value == 1:
        return True
    else:
        return False


# Example usage
examples = [(5, 11), (4, 7), (3, 13), (10, 17)]

for a, p in examples:
    print(f"is_quadratic_residue({a}, {p}) = {is_quadratic_residue(a, p)}")
```

```
✓ 0.0s
```

```
is_quadratic_residue(5, 11) = True
is_quadratic_residue(4, 7) = True
is_quadratic_residue(3, 13) = True
is_quadratic_residue(10, 17) = False
```

24. Write a function order_mod(a, n) that finds the smallest

positive integer k such that $a^k \equiv 1 \bmod n$.

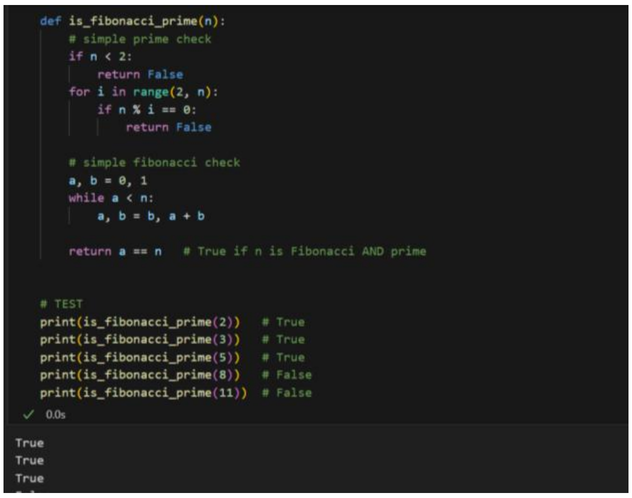


```python
def order_mod(a, n):
    a = a % n
    if a == 0:
        return None   # order does not exist

    k = 1
    value = a % n

    while value != 1:
        value = (value * a) % n
        k += 1

        if k > n:      # safety check – prevents infinite loop
            return None

    return k


# Example usage
examples = [(2, 7), (3, 7), (2, 9), (5, 14)]

for a, n in examples:
    print(f"order_mod({a}, {n}) = {order_mod(a, n)}")
```

```
✓ 0.0s
```

```
order_mod(2, 7) = 3
order_mod(3, 7) = 6
order_mod(2, 9) = 6
```

25. Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.

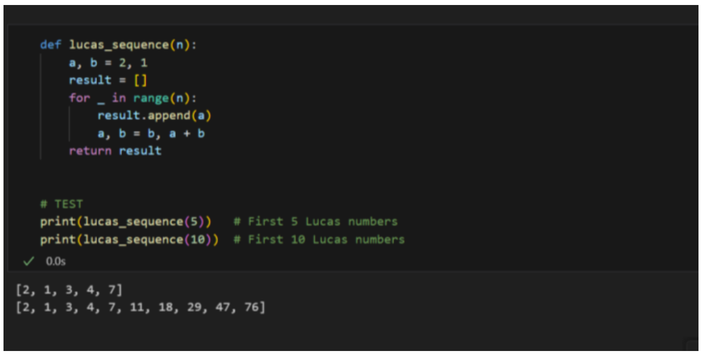


```python
def is_fibonacci_prime(n):
    # simple prime check
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False

    # simple fibonacci check
    a, b = 0, 1
    while a < n:
        a, b = b, a + b

    return a == n   # True if n is Fibonacci AND prime


# TEST
print(is_fibonacci_prime(2))   # True
print(is_fibonacci_prime(3))   # True
print(is_fibonacci_prime(5))   # True
print(is_fibonacci_prime(8))   # False
print(is_fibonacci_prime(11))  # False
```

```
✓ 0.0s
```

```
True
True
True
```

26. Write a function Lucas Numbers Generator lucas_sequence(n) that generates the first n Lucas numbers (similar to Fibonacci but starts with 2, 1)

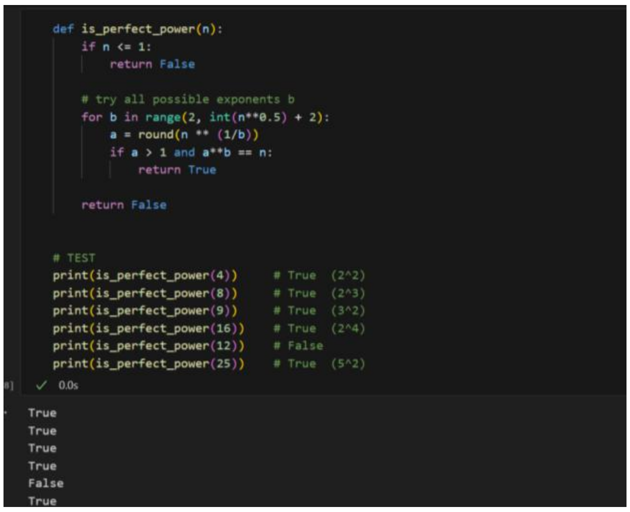


```python
def lucas_sequence(n):
    a, b = 2, 1
    result = []
    for _ in range(n):
        result.append(a)
        a, b = b, a + b
    return result


# TEST
print(lucas_sequence(5))   # First 5 Lucas numbers
print(lucas_sequence(10))  # First 10 Lucas numbers
```

✓ 0.0s

```
[2, 1, 3, 4, 7]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

27.Write a function for Perfect Powers Check is_perfect_power(n) that checks if a number can be expressed as a^b where a > 0 and b > 1.

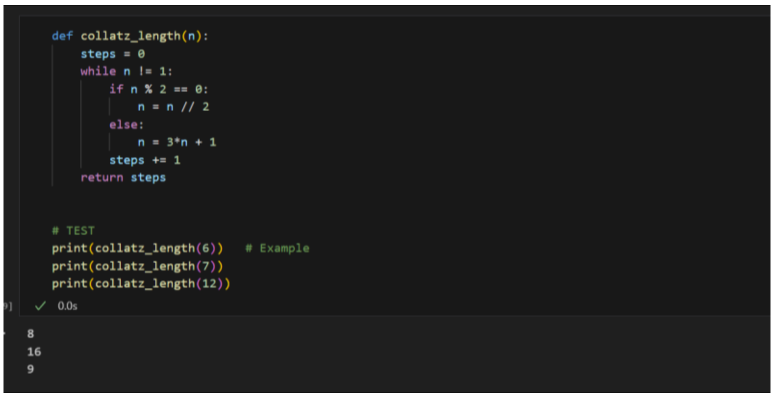


```python
def is_perfect_power(n):
    if n <= 1:
        return False

    # try all possible exponents b
    for b in range(2, int(n**0.5) + 2):
        a = round(n ** (1/b))
        if a > 1 and a**b == n:
            return True

    return False


# TEST
print(is_perfect_power(4))     # True  (2^2)
print(is_perfect_power(8))     # True  (2^3)
print(is_perfect_power(9))     # True  (3^2)
print(is_perfect_power(16))    # True  (2^4)
print(is_perfect_power(12))    # False
print(is_perfect_power(25))    # True  (5^2)
```

✓ 0.0s

```
True
True
True
True
False
True
```

28.Write a function Collatz Sequence Length collatz_length(n) that returns the number of steps for n to reach 1 in the Collatz conjecture.

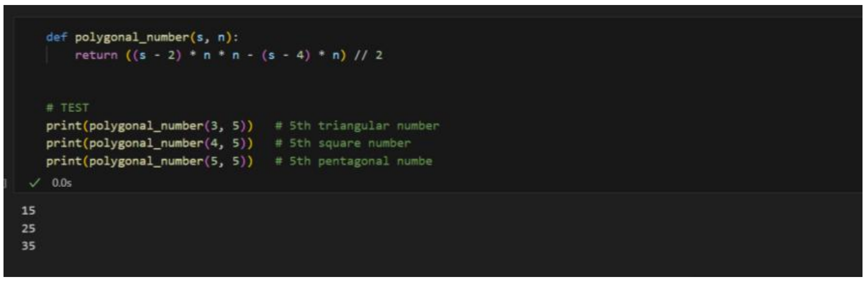


```python
def collatz_length(n):
    steps = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3*n + 1
        steps += 1
    return steps


# TEST
print(collatz_length(6))   # Example
print(collatz_length(7))
print(collatz_length(12))
```

✓ 0.0s

```
8
16
9
```

29. Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.

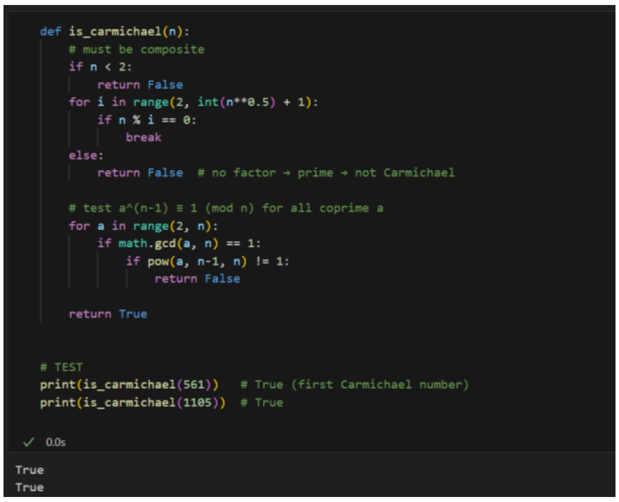


```
def polygonal_number(s, n):
    return ((s - 2) * n * n - (s - 4) * n) // 2


# TEST
print(polygonal_number(3, 5))   # 5th triangular number
print(polygonal_number(4, 5))   # 5th square number
print(polygonal_number(5, 5))   # 5th pentagonal numbe
```

```
✓ 0.0s
```

```
15
25
35
```

30. Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies $a^{n-1} \equiv 1$ mod n for all a coprime to n.

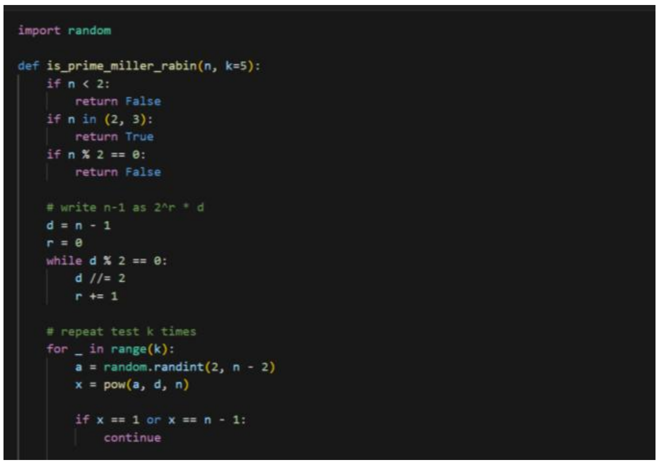


```
def is_carmichael(n):
    # must be composite
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            break
    else:
        return False  # no factor → prime → not Carmichael

    # test a^(n-1) ≡ 1 (mod n) for all coprime a
    for a in range(2, n):
        if math.gcd(a, n) == 1:
            if pow(a, n-1, n) != 1:
                return False

    return True


# TEST
print(is_carmichael(561))   # True (first Carmichael number)
print(is_carmichael(1105))  # True
```

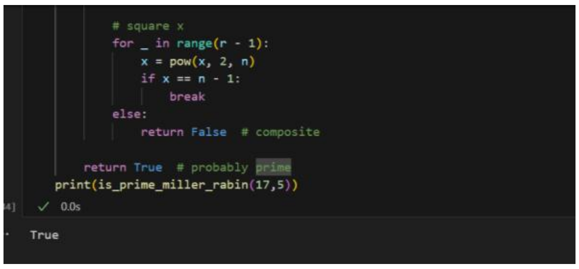
```
✓ 0.0s
```

```
True
True
```

31. Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.

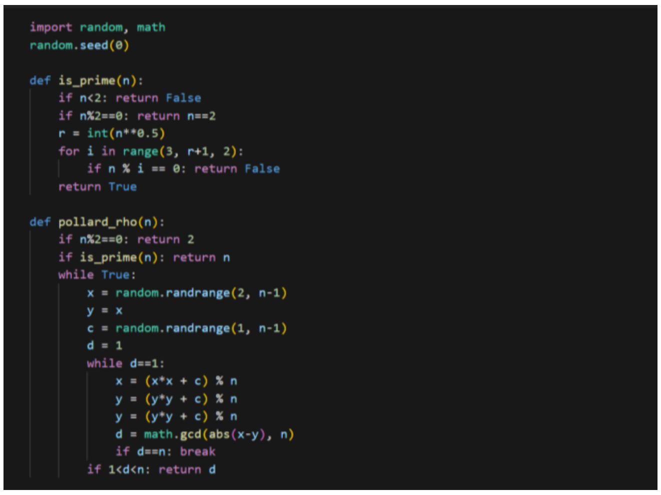


```
import random

def is_prime_miller_rabin(n, k=5):
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0:
        return False

    # write n-1 as 2^r * d
    d = n - 1
    r = 0
    while d % 2 == 0:
        d //= 2
        r += 1

    # repeat test k times
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)

        if x == 1 or x == n - 1:
            continue
```

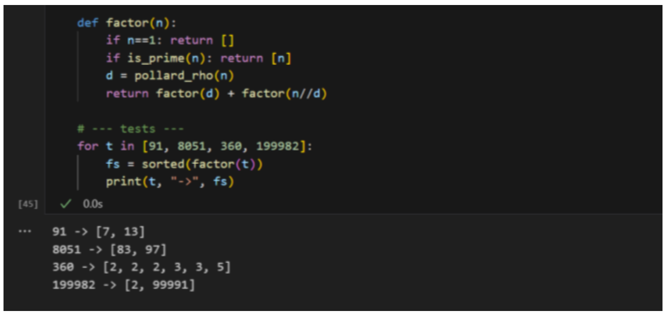


```
            # square x
            for _ in range(r - 1):
                x = pow(x, 2, n)
                if x == n - 1:
                    break
            else:
                return False  # composite

        return True  # probably prime
    print(is_prime_miller_rabin(17,5))
4]  ✓ 0.0s

    True
```

32. Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.

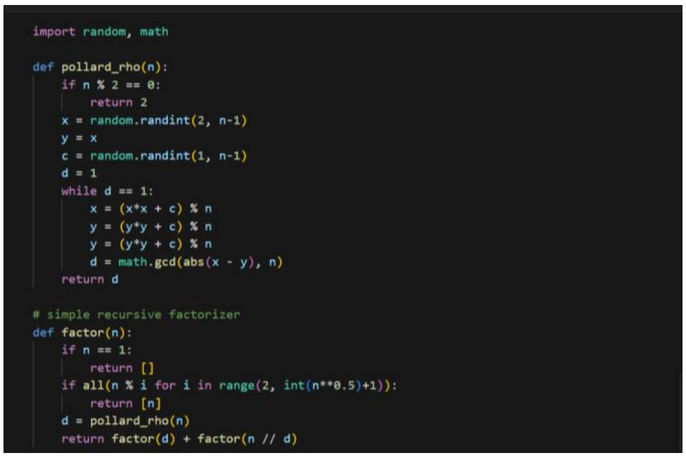


```
import random, math
random.seed(0)

def is_prime(n):
    if n<2: return False
    if n%2==0: return n==2
    r = int(n**0.5)
    for i in range(3, r+1, 2):
        if n % i == 0: return False
    return True

def pollard_rho(n):
    if n%2==0: return 2
    if is_prime(n): return n
    while True:
        x = random.randrange(2, n-1)
        y = x
        c = random.randrange(1, n-1)
        d = 1
        while d==1:
            x = (x*x + c) % n
            y = (y*y + c) % n
            y = (y*y + c) % n
            d = math.gcd(abs(x-y), n)
            if d==n: break
        if 1<d<n: return d
```

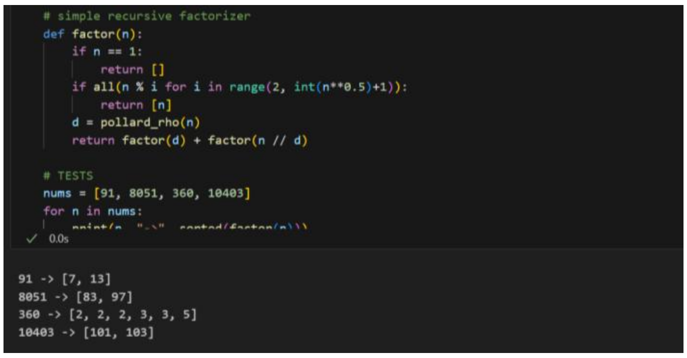


```
    def factor(n):
        if n==1: return []
        if is_prime(n): return [n]
        d = pollard_rho(n)
        return factor(d) + factor(n//d)

    # --- tests ---
    for t in [91, 8051, 360, 199982]:
        fs = sorted(factor(t))
        print(t, "->", fs)
[45]  ✓ 0.0s

...  91 -> [7, 13]
     8051 -> [83, 97]
     360 -> [2, 2, 2, 3, 3, 5]
     199982 -> [2, 99991]
```

33. Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.



```python
import random, math

def pollard_rho(n):
    if n % 2 == 0:
        return 2
    x = random.randint(2, n-1)
    y = x
    c = random.randint(1, n-1)
    d = 1
    while d == 1:
        x = (x*x + c) % n
        y = (y*y + c) % n
        y = (y*y + c) % n
        d = math.gcd(abs(x - y), n)
    return d

# simple recursive factorizer
def factor(n):
    if n == 1:
        return []
    if all(n % i for i in range(2, int(n**0.5)+1)):
        return [n]
    d = pollard_rho(n)
    return factor(d) + factor(n // d)
```

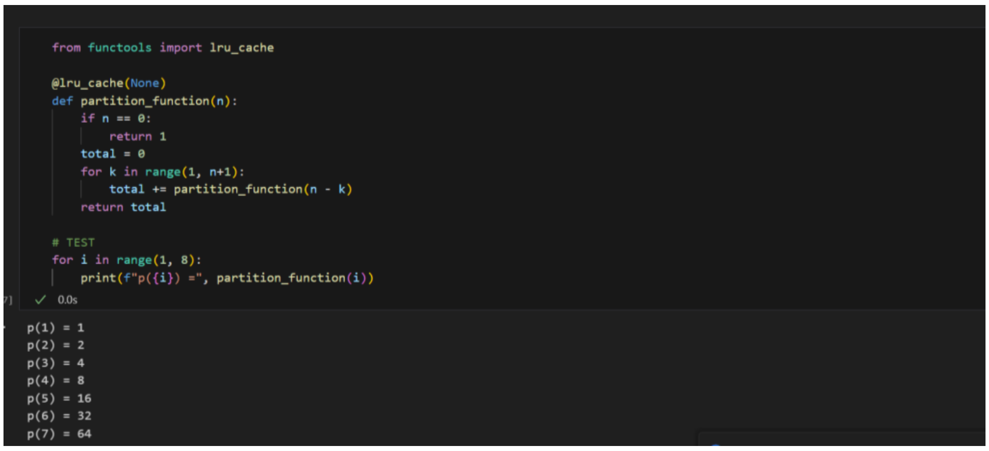


```python
# simple recursive factorizer
def factor(n):
    if n == 1:
        return []
    if all(n % i for i in range(2, int(n**0.5)+1)):
        return [n]
    d = pollard_rho(n)
    return factor(d) + factor(n // d)

# TESTS
nums = [91, 8051, 360, 10403]
for n in nums:
    print(n, "->", sorted(factor(n)))
```

✓ 0.0s

```
91 -> [7, 13]
8051 -> [83, 97]
360 -> [2, 2, 2, 3, 3, 5]
10403 -> [101, 103]
```

34. Write a function Partition Function p(n) partition_function(n) that calculates the number of distinct ways to write n as a sum of positive integers



```python
from functools import lru_cache

@lru_cache(None)
def partition_function(n):
    if n == 0:
        return 1
    total = 0
    for k in range(1, n+1):
        total += partition_function(n - k)
    return total

# TEST
for i in range(1, 8):
    print(f"p({i}) =", partition_function(i))
```

✓ 0.0s

```
p(1) = 1
p(2) = 2
p(3) = 4
p(4) = 8
p(5) = 16
p(6) = 32
p(7) = 64
```