1. Write a function called euler_phi(n) that calculates Euler's Totient Function, ϕ(n). This function counts the number of integers up to n that are coprime with n (i.e., numbers k for which ged(n, k) = 1).

```
n = int(input('Enter a positive integer :'))

def gcd(a, n):

   if a == 0:

      return n

   return gcd(n % a, a)

def etf(n):

   b = 1

   for a in range(2, n):

      if gcd(a, n) == 1:

         b += 1

   return b

print(f'number of integers up to {n} that are coprime with {n} are {etf(n)}')
```

2. Write a function called mobius(n) that calculates the Möbius function, μ(n). The function is defined as: μ(n) = 1 if n is a square-free positive integer with an even number of prime factors. μ(n) =-1 if n is a square-free positive integer with an odd number of prime factors. μ(n) = 0 if n has a squared prime factor

```
def mobius(n):
   if n <= 0:
      return 0
   if n == 1:
      return 1
   prime_factors_count = 0
   p = 2
   while p * p <= n:
      if n % p == 0:
         prime_factors_count += 1
         n //= p
         if n % p == 0:
            return 0
      p += 1
   if n > 1:
```

```
        prime_factors_count += 1
    if prime_factors_count % 2 == 0:
        return 1
    else:
        return -1
```

3. Write a function called divisor_sum(n) that calculates the sum of all positive divisors of n (including 1 and n itself). This is often denoted by σ(n).

```
n = int(input('Enter a positive integer: '))
def divisor_sum(n):
    if n <= 0:
        return 0
    total_sum = 0
    for i in range(1, n + 1):
        if n % i == 0:
            total_sum += i
    return total_sum
print(f"The sum of the divisors of {n} is {divisor_sum(n)}")
```

4. Write a function called prime_pi(n) that approximates the prime-counting function, π(n). This function returns the number of prime numbers less than or equal to n

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

def prime_pi(n):
    if n < 2:
        return 0
    count = 0
    for number in range(2, n + 1):
        if is_prime(number):
            count += 1
    return count

print(f"Number of prime numbers <= {n} are {prime_pi(n)}")
```

5. Write a function called legendre_symbol(a, p) that calculates the Legendre symbol (a/p), which is a useful function in quadratic reciprocity. It is defined for an odd prime p and an integer a not divisible by p as: (a / p) = 1 if a is a quadratic residue modulo p (i.e., there exists an integer x such that x2 = a (mod p)). (a / p) = - 1 if a is a quadratic non-residue modulo p.

```
def main(a, p):
```

```
m = 0
y = 0
x = 0
if a % p != 0:
    if p % 2 != 0:
        for i in range(1, p + 1):
            if p % i == 0:
                m += 1
        if m == 2:
            y = a % p
            for i in range(1, y + 1):
                if i * i == y:
                    x = 1
                    break
            else:
                x = -1
        if x == 1:
            print("a/p=1")
        else:
            print("a/p= -1")
```

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

```
 x=int(input( 'Enter a non-negative ihteger' )
def factorial(x):
   f = 1
   if x < 0:
       print('wrong input')
   else:
       for i in range(1, x + 1):
           f *= i
```

```
        print( 'FACTORIAL:' , f)
factorial(x)
```

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

```
n = int(input('Enter an integer: '))

def palindrome(n):
    k = n
    l = 0
    while k != 0:
        j = k % 10
        l = l * 10 + j
        k = k // 10

    if l == n:
        print("palindrome")
    else:
        print("not palindrome")

palindrome(n)
```

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

```
n= int(input("enter an integer "))
def mean_of_digits(n):
    c = 0
    m = 0
    k = n
```

```
    while k != 0:
        j = k % 10
        m += j
        c += 1
        k //= 10  # integer division

    avg = m / c
    print(avg)

mean_of_digits(n)
```

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

```
 n=int(input("enter an integer "))
def digital_root(n):
    while n >= 10:  # keep going until n is a single digit
        s = 0
        while n > 0:
            s += n % 10
            n //= 10
        n = s
    return n

digital_root(n)
```

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

```
 n=int(input("enter an integer "))
def is_abundant(n):
    k = n
```

```
    s = 0
    for i in range(1, k):
        if k % i == 0:
            s += i

    if s > n:
        print("true")
    else:
        print("false")

is_abundant(n)
```

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

```
n = int(input("Enter a positive integer: "))
def is_deficient(n):
    s = 1
    i = 2
    while i * i <= n:
        if n % i == 0:
            s += i
            if i * i != n:
                s += n // i
        i += 1
    return s < n
result = is_deficient(n)
print(result)
```

12. Write a function for harshad number is_harshad(n) that

checks if a number is divisible by the sum of its digits.

```
def is_harshad(n):

 n = int(n)
 original_n = n
 sum_of_digits = 0
 while n > 0:
   digit = n % 10
   sum_of_digits += digit
   n //= 10

 if (original_n % sum_of_digits == 0):
   print( original_n ,"is divisible by the sum of its digits")
 else:
    print( original_n ,"is not divisible by the sum of its digits")

num = int(input("Enter a number: "))

is_harshad(num)
```

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

number's square ends with the number itself.

```
def is_automorphic(n):
   sq= n**2
   n2=n
   a=0
   while (n!=0):
       a=a+1
       n=n/10
```

```
    last=sq%a
    if( last==n2):
        print( n2, "is an automorphic number")
    else:
        print( n2, "is not an automorphic number")

num=int(input("enter a number "))

is_automorphic(num)
```

14. Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.

```
def is_pronic(n):
    a=0
    for i in range
        (1,n): if(
        i*(i+1)==n):
            a=1
            break
        else:
            a=0

    return a

num= int(input("enter a number "))
if (is_pronic(num)==1):
  print( num , "is a pronic number")
elif (is_pronic(num)==0):
  print( num , "is not a pronic number")
```

15. Write a function prime_factors(n) that returns the list of prime factors of a number.

```
n = int(input("Enter a positive integer: "))
def prime_factors(n):
    factors = []

    while n % 2 == 0:
        factors += [2]
        n //= 2

    i = 3
    while i * i <= n:
        while n % i == 0:
            factors += [i]
            n //= i
        i += 2

if n > 2:
        factors += [n]

    return factors

if n <= 1:
    print([])
else:
    print(prime_factors(n))
```

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

```
def count_distinct_prime_factorial(n):
  if n < 2:
    return 0  # 0 and 1 have no prime factors

  count = 0
  i = 2
  while i * i <= n:
    if n % i == 0:
      count += 1
      while n % i == 0:
        n //= i
    i += 1
  if n > 1:
    count += 1  # n itself is a prime number

  return count
print(count_distinct_prime_factorial(60))
```

17. Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.

```
import time
import os
import psutil

def memory_usage():
  process = psutil.Process(os.getpid())
  return process.memory_info().rss
def is_prime_power(n):
```

```
    if n < 2:
        return False
    for p in range(2, int(n**0.5)+1):
        # check if p is prime
        is_prime = True
        for i in range(2, int(p**0.5)+1):
            if p % i == 0:
                is_prime = False
                break
        if not is_prime:
            continue
        # try different k values
        k = 1
        value = p
        while value < n:
            value *= p
            k += 1
        if value == n:
            return True
    return False
start_time = time.time()
n = int(input("Enter a number to check if it's a prime power: "))
if is_prime_power(n):
    print(f"{n} is a prime power.")
else:
    print(f"{n} is not a prime power.")


print(f"memory usage :{memory_usage()} bytes")
end_time = time.time()
print(f"\nExecution Time: {end_time - start_time:.6f} seconds")
```

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

```
def memory_usage():
    process = psutil.Process(os.getpid())
    return process.memory_info().rss
def is_mersenne_prime(p):

    a=2 ** p - 1
    b=0
    c=0
    if p < 2:
        return 0
    for i in range(2,p):
        if p%i==0:
            b=1
            break
        else:
            b=0
        for j in range(2,a):
            if a%i==0:
                c=1
                break
            else:
                c=0
    if b==0 and c==0:
        return True
    else:
        return False
```

```python
start_time = time.time()

p = int(input("Enter a prime number to check if it's a Mersenne prime: "))

if is_mersenne_prime(p):

   print(f"2^{p} - 1 is a Mersenne prime.")

else:

   print(f"2^{p} - 1 is not a Mersenne prime.")


print(f"memory usage :{memory_usage()} bytes")

end_time = time.time()

print(f"\nExecution Time: {end_time - start_time:.6f} seconds")
```

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

```python
import time

import sys

start_time = time.time()

def twin_prime(limit):

   def is_prime(n):

      if n < 2:

         return False

      for i in range(2, int(n ** 0.5) + 1):

         if n % i == 0:

            return False

      return True


   twin_primes = []

   for num in range(2, limit - 1):

      if is_prime(num) and is_prime(num + 2):

         twin_primes.append((num, num + 2))

   return twin_primes
```

```
limit = 130

end_time = time.time()

time_taken = end_time-start_time

print(f"The twin pairs are:{twin_prime(130)}")

print("time taken to run this code:",time_taken,"seconds")

print("memory usage:",sys.getsizeof(twin_prime(limit)),"KB")

# Output: [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
```

20. Write a function Number of Divisors

(d(n)) count_divisors(n) that returns how many positive

divisors a number has.

```
import time

import os

import psutil


def memory_usage():

    process = psutil.Process(os.getpid())

    return process.memory_info().rss

def count_divisors(n):

    count = 0

    for i in range(1, int(n**0.5)+1):

        if n % i == 0:

            if i * i == n:

                count += 1

            else:

                count += 2

    return count

start_time = time.time()

n = int(input("Enter a number to count its divisors: "))
```

```
print(f"Number of divisors of {n}: {count_divisors(n)}")

print(f"memory usage :{memory_usage()} bytes")

end_time = time.time()

print(f"\nExecution Time: {end_time - start_time:.6f} seconds")
```

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

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

n = int(input("Enter a number: "))
print(f"The aliquot sum of {n} is: {aliquot_sum(n)}")
```

22. 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).

```
def are_amicable(a,b):
    s1 = 0
    for i in range(1, a):
        if (a % i == 0):
            s1 += i

    s2 = 0
    for j in range(1, b):
        if (b % j == 0):
            s2 += j

    if (s1 == b and s2 == a):
        print("numbers are amicable")
    else:
        print("numbers are not amicable")

a = int(input("Enter a number: "))
b = int(input("Enter another number: "))
are_amicable(a, b)
```

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

```
def multiplicative_persistence(n):
    c = 0
    while n >= 10:
        p = 1
        temp_n = n
        while temp_n > 0:
            digit = temp_n % 10
            p *= digit
            temp_n //= 10
        n = p
        c += 1
    return c

n = int(input("Enter a number: "))
print(multiplicative_persistence(n))
```

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

```
def count_divisors(num):
    if num <= 0:
        return 0
    if num == 1:
        return 1

    count = 0
    i = 1
    while i * i <= num:
        if num % i == 0:
            if i * i == num:
                count += 1
            else:
                count += 2
        i += 1
    return count

def is_highly_composite(n):
    if n <= 0:
        return False
    if n == 1:
```

```
        return True

    divisors_of_n = count_divisors(n)

    for i in range(1, n):
        divisors_of_i = count_divisors(i)
        if divisors_of_i >= divisors_of_n:
            return False

    return True

number = int(input("Enter a positive integer: "))

if is_highly_composite(number):
    print(f"\n{number} IS highly composite.")
else:
    print(f"\n{number} is NOT highly composite.")

if number > 0:
    print(f"Divisors: {count_divisors(number)}")
```

25.Write a function for ModularExponentiation mod_exp(base, exponent, modulus) thatefficiently calculates (base^exponent) % modulus.prime factors of a number.

```
def mod_exp(base, exponent, modulus):

    if modulus == 1:
        return 0

    result = 1
    base %= modulus

    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus

        base = (base * base) % modulus
        exponent //= 2

    return result
b = int(input("Enter the base: "))
e = int(input("Enter the exponent: "))
m = int(input("Enter the modulus: "))
```

```
if e < 0:
    print("This function does not handle negative exponents.")
else:
    final_result = mod_exp(b, e, m)
    print(f"\nResult: ({b}^{e}) mod {m} = {final_result}")
```

26. Write a function Modular Multiplicative

Inverse mod_inverse(a, m) that finds the number x such

that (a * x) ≡ 1 mod m.

```
import time
import sys
import resource


def mod_inverse(a, m):
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y
    gcd, x, _ = extended_gcd(a % m, m)
    if gcd != 1:
        return None
    return (x % m + m) % m


# Test
start_time = time.time()
result = mod_inverse(3, 26)
end_time = time.time()
runtime = end_time - start_time
```

```
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024  # bytes
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

27. Write a function chinese Remainder Theorem

Solver crt(remainders, moduli) that solves a system of

congruences x ≡ ri mod mi.

```
import time
import sys
import resource
from math import gcd

def crt(remainders, moduli):
    if len(remainders) != len(moduli):
        return None
    prod = 1
    for m in moduli:
        prod *= m
    result = 0
    for rem, mod in zip(remainders, moduli):
        p = prod // mod
        result += rem * extended_inverse(p, mod) * p
    return result % prod

def extended_inverse(a, m):
    m0, x0, x1 = m, 1, 0
    if m == 1:
        return 0
    while a > 1:
        q = a // m
        m, a = a % m, m
```

```
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Test
start_time = time.time()
result = crt([2, 3], [3, 5])
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

28. Write a function Quadratic Residue

Check is_quadratic_residue(a, p) that checks if x2 ≡ a mod p

has a solution.

```
import time
import sys
import resource

def is_quadratic_residue(a, p):
    if p == 2:
        return True if a % 2 == 0 or a % 2 == 1 else False
    if a % p == 0:
        return True
    return pow(a, (p - 1) // 2, p) == 1

# Test
start_time = time.time()
result = is_quadratic_residue(2, 7)
end_time = time.time()
runtime = end_time - start_time
```

```
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

29. Write a function order_mod(a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.

```
import time
import sys
import resource

def order_mod(a, n):
    if gcd(a, n) != 1:
        return None
    k = 1
    pow_a = a % n
    while pow_a != 1:
        pow_a = (pow_a * a) % n
        k += 1
        if k > n:  # Cycle detection
            return None
    return k

from math import gcd

start_time = time.time()
result = order_mod(2, 7)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

30. Write a function Fibonacci Prime

Check is_fibonacci_prime(n) that checks if a number is

both Fibonacci and prime.

```
import time
import sys
import resource

def is_fibonacci(n):
    if n < 0:
        return False
    a, b = 0, 1
    while b < n:
        a, b = b, a + b
    return b == n

def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_fibonacci_prime(n):
```

```
    return is_fibonacci(n) and is_prime(n)


start_time = time.time()
result = is_fibonacci_prime(13)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

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

```
import time
import sys
import resource

def lucas_sequence(n):
    if n == 0:
        return [2]
    if n == 1:
        return [2, 1]
    seq = [2, 1]
    for i in range(2, n + 1):
        seq.append(seq[-1] + seq[-2])
    return seq[:n]


start_time = time.time()
```

```
result = lucas_sequence(10)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

32. Write a function for Perfect Powers

Check is_perfect_power(n) that checks if a number can be

expressed as ab where a > 0 and b > 1.

```
import time
import sys
import resource
from math import log, sqrt, ceil

def is_perfect_power(n):
    if n < 1:
        return False
    for b in range(2, int(log(n, 2)) + 1):
        a = round(n ** (1 / b))
        if a ** b == n:
            return True
    return False


start_time = time.time()
result = is_perfect_power(16)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

33. Write a function Collatz Sequence

Length collatz_length(n) that returns the number of steps

for n to reach 1 in the Collatz conjecture.

```
import time
import sys
import resource

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



start_time = time.time()
result = collatz_length(27)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

34. Write a function Polygonal Numbers polygonal_number(s,

n) that returns the n-th s-gonal number.

```
import time

import sys

import resource


def polygonal_number(s, n):

    if s < 3 or n < 1:

        return None

    return n * (n - 1) * (s - 2) // 2 + n




start_time = time.time()

result = polygonal_number(5, 3)  # Pentagon

end_time = time.time()

runtime = end_time - start_time

mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024

print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

35. Write a function Carmichael Number

Check is_carmichael(n) that checks if a composite number

n satisfies an−1 ≡ 1 mod n for all a coprime to n.

```
import time

import sys

import resource

from math import gcd


def is_carmichael(n):

    if n <= 1 or is_prime(n):

        return False

    for a in range(2, n):

        if gcd(a, n) == 1 and pow(a, n - 1, n) != 1:
```

```
            return False
    return True


def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True



start_time = time.time()
result = is_carmichael(561)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

36. Implement the probabilistic Miller-Rabin

test is_prime_miller_rabin(n, k) with k rounds.

```
import time
import sys
import resource
```

```
from random import randrange


def is_prime_miller_rabin(n, k=40):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True



start_time = time.time()
result = is_prime_miller_rabin(13)
end_time = time.time()
```

```
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

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

```
import time
import sys
import resource
from math import gcd
from random import randint

def pollard_rho(n):
    if n % 2 == 0:
        return 2
    x = randint(1, n - 1)
    y = x
    c = 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 = gcd(abs(x - y), n)
    if d == n:
        return None
    return d

start_time = time.time()
```

```python
result = pollard_rho(315)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

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

```python
import time
import sys
import resource


def zeta_approx(s, terms=1000):
    if s <= 1:
        return float('inf')
    total = 0.0
    for k in range(1, terms + 1):
        total += 1 / (k ** s)
    return total



start_time = time.time()
result = zeta_approx(2, 1000)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss * 1024
print(f"Result: {result:.6f}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```

39. 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.

```
import time
import sys
import resource

def partition_function(n):
    if n < 0:
        return 0
    p = [0] * (n + 1)
    p[0] = 1
    for i in range(1, n + 1):
        j = 1
        while True:
            pent = j * (3 * j - 1) // 2
            if pent > i:
                break
            sign = 1 if (j % 2 == 1) else -1
            p[i] += sign * p[i - pent]
            pent_neg = (-j) * (3 * (-j) - 1) // 2
            if pent_neg <= i:
                sign = 1 if (j % 2 == 0) else -1
                p[i] += sign * p[i - pent_neg]
            j += 1
    return p[n]


start_time = time.time()
result = partition_function(10)
end_time = time.time()
runtime = end_time - start_time
mem_usage = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss *
1024
print(f"Result: {result}, Runtime: {runtime:.6f}s, Memory: {mem_usage} bytes")
```