Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAS10098
Name of Student	:Bhavadharani.S
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SMEC
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Write a function factorial(n) that calculates the factorial of a non-negative integer n (n!).	04/09/25	
2	Write a function is_palindrome(n) that checks if a number reads the same forwards and backwards.	04/09/25	
3	Write a function mean_of_digits(n) that returns the average of all digits in a number.	04/09/25	
4	Write a function digital_root(n) that repeatedly sums the digits of a number until a single digit is obtained.	04/09/25	
5	Write a function is_abundant(n) that returns True if the sum of proper divisors of n is greater than n.	04/09/25	
6	Write a function is_deficient(n) that returns True if the sum of proper divisors of n is less than n.	11/10/25	
7	Write a function for harshad number is_harshad(n) that checks if a number is divisible by the sum of its digits.	11/10/25	
8	Write a function is_automorphic(n) that checks if a number's square ends with the number itself.	11/10/25	
9	Write a function is_pronic(n) that checks if a number is the product of two consecutive integers.	11/10/25	
10	Write a function prime_factors(n) that returns the list of	11/10/25	

	prime factors of a number.		
11	Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.	19/10/25	
12	Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.	19/10/25	
13	Write a function is_mersenne_prime(p) that checks if 2p - 1 is a prime number (given that p is prime).	19/10/25	
14	Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.	19/10/25	
15	Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.	19/10/25	
16	Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).	01/1i/25	
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).	01/1i/25	
18	Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.	01/1i/25	
19	Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.	01/1i/25	

20	Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.	01/1i/25	
21	Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.	02/11/25	
22	Write a function chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences x ≡ ri mod mi.	02/11/25	
22	Write a function chinese Remainder Theorem Solver crt(remainders, moduli) that solves a system of congruences x ≡ ri mod mi.	02/11/25	
23	Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if x2 ≡ a mod p has a solution.	02/11/25	
24	Write a function order_mod(a, n) that finds the smallest positive integer k such that ak ≡ 1 mod n.	02/11/25	
25	Write a function Fibonacci Prime Check is_fibonacci_prime(n) that checks if a number is both Fibonacci and prime.	09/11/25	
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).	09/11/25	
27	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.	09/11/25	

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.	09/11/25	
29	Write a function Polygonal Numbers polygonal_number(s, n) that returns the n-th s-gonal number.	09/11/25	
30	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.	09/11/25	
31	Implement the probabilistic Miller-Rabin test is_prime_miller_rabin(n, k) with k rounds.	16/11/25	
32	Implement pollard_rho(n) for integer factorization using Pollard's rho algorithm.	16/11/25	
33	Write a function zeta_approx(s, terms) that approximates the Riemann zeta function ζ(s) using the first 'terms' of the series.	16/11/25	
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.	16/11/25	

Assignment

Date: 19-11-2025

TITLE: Run functions in Python without using external Library

AIM/OBJECTIVE(s): 1.Write a function factorial(n) that calculates the factorial of

a non-negative integer n (n!).

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

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

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

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

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

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

8. Write a function is_automorphic(n) that checks if a number's square ends with the number itself.

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

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

11. Write a function count_distinct_prime_factors(n) that returns how many unique prime factors a number has.
12. Write a function is_prime_power(n) that checks if a number can be expressed as pk where p is prime and k ≥ 1.
13. Write a function is_mersenne_prime(p) that checks if 2p - 1 is a prime number (given that p is prime).
14. Write a function twin_primes(limit) that generates all twin prime pairs up to a given limit.
15. Write a function Number of Divisors (d(n)) count_divisors(n) that returns how many positive divisors a number has.
16. Write a function aliquot_sum(n) that returns the sum of all proper divisors of n (divisors less than n).
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).
18. Write a function multiplicative_persistence(n) that counts how many steps until a number's digits multiply to a single digit.
19. Write a function is_highly_composite(n) that checks if a number has more divisors than any smaller number.
20. Write a function for Modular Exponentiation mod_exp(base, exponent, modulus) that efficiently calculates (baseexponent) % modulus.
21. Write a function Modular Multiplicative Inverse mod_inverse(a, m) that finds the number x such that (a * x) ≡ 1 mod m.

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

23. Write a function Quadratic Residue

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

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

25. Write a function Fibonacci Prime

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

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).

27. 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.

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.

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

30. 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.

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

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

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

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.

METHODOLOGY & TOOL USED: 1.

```
def factorial(n):
      product = 1
      for i in range(1,n+1):

        product  *= (i)
        return product
print(factorial(10))
```

2.

```
def is_palindrome(n):
    s = str(n)
    return s == s[::-1]
```

```
print(is_palindrome(121))

print(is_palindrome(123))

v = (1,2,3,4,5,6,7)

n = len(v)

i = sum(v)

print(i/n)
```

3.

```
v = (1,2,3,4,5,6,7)

def mean_of_digit(v):


 n = len(v)

 i = sum(v)

 return(i/n)

print(mean_of_digit(v))
```

4.

```
def digital_root(n):

    while n >= 10:

        n = sum(int(digit) for digit in str(n))

    return n

print(digital_root(1234))
```

5.

```
def is_abundant(n):

   if n < 1:

        return False

    divisors_sum = 1
```

```
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors_sum += i
            if i != n // i:
                divisors_sum += n // i
    return divisors_sum > n
print(is_abundant(12))
```

6.

```
def is_deficient(n):
    if n <= 1:
        return True
    proper_divisor_sum = sum(i for i in range(1, n) if n % i == 0)
    return proper_divisor_sum < n
print(is_deficient(20))
```

7.

```
def is_harshad(n):
    digit_sum = sum(int(d) for d in str(n))
    return n % digit_sum == 0
print(is_harshad(15))
```

8.

```
def is_automorphic(n):
    square = n * n


   return str(square).endswith(str(n))
print(is_automorphic(5))
```

9.

```
def is_pronic(n):
    if n < 0:
        return False
    i = 0
    while i * (i + 1) <= n:
        if i * (i + 1) == n:
            return True
        i += 1
    return False
print(is_pronic(22))
```

10.

```
def prime_factors(n):
    factors = []
    while n % 2 == 0:
        factors.append(2)
        n //= 2
    i = 3
    while i * i <= n:
        while n % i == 0:
            factors.append(i)
            n //= i
        i += 2
    if n > 2:
        factors.append(n)
```

```
    return factors

print(prime_factors(24))
```

11.

```
def count_distinct_prime_factors(n):
    if n < 2:
        raise ValueError


    count = 0
    temp = n




    if temp % 2 == 0:
        count += 1
        while temp % 2 == 0:
            temp //= 2



    factor = 3
    while factor * factor <= temp:
        if temp % factor == 0:
            count += 1
            while temp % factor == 0:
                temp //= factor
        factor += 2


    if temp > 1:
        count += 1
```

```
    return count
print(count_distinct_prime_factors(2310))
```

12.

```
def is_prime_power(n):

    if n < 2:
        raise ValueError("Input must be a positive integer greater than 1")

    max_exponent = n.bit_length()
    for k in range(1, max_exponent + 1):

        root = round(n ** (1 / k))


        for candidate in [root - 1, root, root + 1]:
            if candidate < 2:
                continue
            power = candidate ** k
            if power == n and is_prime(candidate):
                return True

    return False

def is_prime(num):
```

```
    if num < 2:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False

    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return False
    return True
print(is_prime_power(27))
```

13.

```
def is_mersenne_prime(p):
    if p < 2:
        raise ValueError("Input p must be at least 2")

    mersenne = 2**p - 1

    known_mersenne_primes = {2, 3, 5, 7, 13, 17, 19, 31, 61, 89,
107, 127}
    if p in known_mersenne_primes:
        return True

    return is_prime_mersenne(mersenne)
```

```
def is_prime_mersenne(n):


    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False


    limit = int(n**0.5) + 1


    k = 1
    divisor = 2 * k * (n.bit_length()) + 1


    while divisor <= limit:
        if n % divisor == 0:
            return False
        k += 1
        divisor = 2 * k * (n.bit_length()) + 1


    return True
print(is_mersenne_prime(7))
```

14.

```
def twin_primes(limit):


    if limit < 3:
        raise ValueError("Limit must be at least 3")
```

```
    primes = sieve_of_eratosthenes(limit)



    twin_pairs = []
    for i in range(len(primes) - 1):
        if primes[i + 1] - primes[i] == 2:
            twin_pairs.append((primes[i], primes[i + 1]))

    return twin_pairs

def sieve_of_eratosthenes(n):

    if n < 2:
        return []

    sieve = [True] * (n + 1)
    sieve[0] = sieve[1] = False

    for i in range(2, int(n**0.5) + 1):
        if sieve[i]:
            sieve[i*i : n+1 : i] = [False] * len(sieve[i*i : n+1 :
i])

    return [i for i, is_prime in enumerate(sieve) if is_prime]
print("Twin primes up to 20:")
print(twin_primes(20))
```

15.

```
def count_divisors(n):
    if n <= 0:
        raise ValueError("Input must be a positive integer")

    if n == 1:
        return 1

    count = 1
    i = 2
    while i * i <= n:
        if n % i == 0:

            if i * i == n:
                count += 1
            else:
                count += 2
        i += 1

    return count + 1
print(f"d(12) = {count_divisors(12)}")
```

16.

```
if  n <= 0:
    return "enter a positive integer"

  x = 0
```

```
  for i in range(1, n // 2 + 1):
    if n % i == 0:
      x += i

  return x
```

17.

```
def are_amicable(a,b):
    if a<0:
        return "enter a positive number"

    x=0
    for i in range (1,a//2+1):
        if a % i ==0:
            x+=i
        if x==b:
            return True
```

18.

```
def multiplicative_persistence(n):
    if n <10:
        return 0


    persistence=0
```

```
    number=n


    while number>=10:
        product=1
        for i in str(number):
            product *= int(i)
        number= product
        persistence += 1


    return persistence
```

19.

```
import math


def count_divisors(num):


    if num <= 0:
        return 0
    divisor_count = 0
    for i in range(1, int(math.sqrt(num) + 1)):
        if num % i == 0:
            if num / i == i:
                divisor_count += 1
            else:
                divisor_count += 2
    return divisor_count


def is_highly_composite(n):
```

```
    if n <= 0:
        return False

    original_divisor_count = count_divisors(n)



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

    return True
```

20.

```
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
```

21.

```
def mod_inverse(a, m):
    a = a % m
    m0 = m
    x = 0
    y = 1

    if m == 1:
        return 0

    while a > 1:
        q = m // a
        m, a = a, m % a
        x, y = y - q * x, x

    if a != 1:
        raise ValueError("Modular inverse does not exist (a and m
are not coprime)")

    if y < 0:
        y += m0
```

```
    return y


# Single use case
A = 3
M = 11
X = mod_inverse(A, M)
print(f"The modular multiplicative inverse of {A} mod {M} is {X}")
print(f"Check: ({A} * {X}) mod {M} = {(A * X) % M}")
```

22.

```
def mod_inverse(a, m):
    a = a % m
    m0 = m
    x = 0
    y = 1

    if m == 1:
        return 0

    while a > 1:
        q = m // a
        m, a = a, m % a
        x, y = y - q * x, x

    if a != 1:
        raise ValueError("Modular inverse does not exist (a and m
are not coprime)")

    if y < 0:
```

```
        y += m0

    return y

def crt(remainders, moduli):
    """Solves a system of congruences using the Chinese Remainder Theorem."""
    if len(remainders) != len(moduli):
        raise ValueError("Remainders and moduli lists must have the same length")

    # N is the product of all moduli
    N = 1
    for m in moduli:
        N *= m

    result = 0
    for r_i, m_i in zip(remainders, moduli):
        # n_i is the product of all moduli except m_i
        n_i = N // m_i

        try:
            x_i = mod_inverse(n_i, m_i)
        except ValueError as e:
            raise ValueError(f"Moduli are not pairwise coprime or modular inverse failed for {n_i} mod {m_i}: {e}")

        result += r_i * n_i * x_i
```

```
    return result % N


R = [2, 3, 2]

M = [3, 5, 7]

X = crt(R, M)


print(f"System of congruences: x ≡ {R[0]} mod {M[0]}, x ≡ {R[1]} mod {M[1]}, x ≡ {R[2]} mod {M[2]}")

print(f"The unique solution modulo {M[0] * M[1] * M[2]} is x = {X}")

print(f"Check: {X} mod 3 = {X % 3}, {X} mod 5 = {X % 5}, {X} mod 7 = {X % 7}")
```

23.

```
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_quadratic_residue(a, p):
```

```
    a = a % p


    if not is_prime(p):
        raise ValueError("Modulus p must be a prime number.")

    if a == 0:
        return True

    if p == 2:
        return True


    exponent = (p - 1) // 2
    result = pow(a, exponent, p)


    return result == 1


A_RESIDUE = 9
P_PRIME = 17
A_NON_RESIDUE = 3

print(f"Is {A_RESIDUE} a quadratic residue mod {P_PRIME}?
{is_quadratic_residue(A_RESIDUE, P_PRIME)}")
print(f"Is {A_NON_RESIDUE} a quadratic residue mod {P_PRIME}?
{is_quadratic_residue(A_NON_RESIDUE, P_PRIME)}")
```

24.

```
import math


def order_mod(a, n):


    if n <= 1:

        raise ValueError("Modulus n must be greater than 1.")


    if math.gcd(a, n) != 1:

        raise ValueError(f"The order is undefined because {a} and
{n} are not coprime (gcd(a, n) != 1).")




    k = 1


    current_power = a % n




    while current_power != 1:


        current_power = (current_power * a) % n

        k += 1


    return k



A = 3

N = 7

K = order_mod(A, N)
```

```
print(f"The order of {A} modulo {N} is k = {K}")
print(f"Check: {A}^{K} mod {N} = {pow(A, K, N)}")
```

25.

```
def is_prime_internal(n):

    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    # Check for factors up to sqrt(n)
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

def is_fibonacci_number_internal(n):

    if n < 0:
        return False

    def is_perfect_square(k):
```

```
        if k < 0:
            return False
        # Calculate the integer square root
        s = int(k**0.5)
        return s * s == k

    # Check the two conditions for the Fibonacci property
    # 5*n^2 + 4 or 5*n^2 - 4 must be a perfect square.
    if is_perfect_square(5 * n * n + 4) or is_perfect_square(5 * n *
n - 4):
        return True
    return False

def is_fibonacci_prime(n):

    if not is_prime_internal(n):
        return False

    if not is_fibonacci_number_internal(n):
        return False

    return True

# Single use case
TEST_NUMBER_1 = 13
TEST_NUMBER_2 = 7
TEST_NUMBER_3 = 8
TEST_NUMBER_4 = 29
```

```
print(f"{TEST_NUMBER_1} is Fibonacci and prime?
{is_fibonacci_prime(TEST_NUMBER_1)}")

print(f"{TEST_NUMBER_2} is Fibonacci and prime?
{is_fibonacci_prime(TEST_NUMBER_2)}")

print(f"{TEST_NUMBER_3} is Fibonacci and prime?
{is_fibonacci_prime(TEST_NUMBER_3)}")

print(f"{TEST_NUMBER_4} is Fibonacci and prime?
{is_fibonacci_prime(TEST_NUMBER_4)}")
```

26.

```
def lucas_sequence(n):


    if n <= 0:

        return

    a, b = 2, 1




    if n >= 1:

        yield a



    if n >= 2:

        yield b



    for _ in range(2, n):

        c = a + b

        yield c

        a, b = b, c


n_terms = 10
```

```
lucas_gen = lucas_sequence(n_terms)


print(f"The first {n_terms} Lucas numbers are:")


print(list(lucas_gen))
```

27.

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

    max_a = int(n**0.5)

    for a in range(2, max_a + 1):
        if n % a == 0:
            power = a * a
            while power <= n:
                if power == n:
                    return True
                power = power * a

    return False
```

```
print(f"Is 8 a perfect power? {is_perfect_power(8)}")
```

28.

```
def collatz_length(n):
    if n < 1:
        return 0

    steps = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        steps += 1

    return steps

start_num = 6
print(f"The Collatz sequence length for {start_num} is:
{collatz_length(start_num)}")
```

29.

```
def polygonal_number(s, n):

    if s < 3 or n < 1:
        return None
```

```
    numerator = (s - 2) * (n ** 2) - (s - 4) * n



    return numerator // 2



print(f"The 5th 3-gonal (triangular) number is: {polygonal_number(3,
5)}")
```

30.

```
def _is_prime(n):


    if n < 2:
        return False
    if n == 2 or 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 _get_prime_factors(n):


    factors = set()




    if n % 2 == 0:
        factors.add(2)
        while n % 2 == 0:
            n //= 2



    i = 3
    while i * i <= n:
        if n % i == 0:
            factors.add(i)
            while n % i == 0:
                n //= i
        i += 2



    if n > 2:
        factors.add(n)


    return factors


def is_carmichael(n):
```

```
    if n < 2 or _is_prime(n):
        return False


    factors = _get_prime_factors(n)



    product_of_factors = 1
    for p in factors:
        product_of_factors *= p


    if product_of_factors != n:
        return False



    for p in factors:
        if (n - 1) % (p - 1) != 0:
            return False
```

31.

```
def is_prime_miller_rabin(n, k=20):



    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False
```

```
s = 0
d = n - 1
while d % 2 == 0:
    d //= 2
    s += 1



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



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



    for _ in range(s - 1):
        x = (x * x) % n
        if x == 1:
            return False

        if x == n - 1:
            break
    else:
```

```
            return False



    return True
```

32.

```
import random
import math




def modular_pow(base, exponent,modulus):




    result = 1

    while (exponent > 0):



        if (exponent & 1):
            result = (result * base) % modulus




        exponent = exponent >> 1
```

```
        base = (base * base) % modulus

    return result



def pollard_rho(n):



    if (n == 1):
        return n



    if (n % 2 == 0):
        return 2



    x = (random.randint(0, 2) % (n - 2))
    y = x



    c = (random.randint(0, 1) % (n - 1))
```

```
    d = 1



    while (d == 1):



        x = (modular_pow(x, 2, n) + c + n)%n



        y = (modular_pow(y, 2, n) + c + n)%n
        y = (modular_pow(y, 2, n) + c + n)%n



        d = math.gcd(abs(x - y), n)



        if (d == n):
            return pollard_rho(n)

    return d
```

33.

```
def zeta_approx(s, terms):
```

```
    if terms <= 0:
        return 0.0

    zetasum = 0.0
    for n in range(1, terms + 1):
        zetasum += 1.0 / (n ** s)
    return zetasum
```

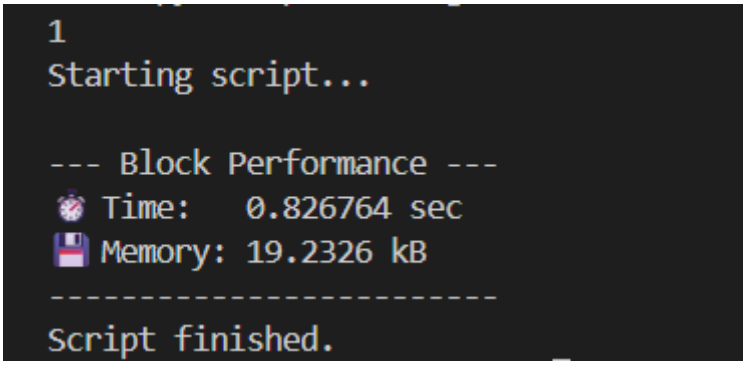
34.

```
def partitions(n):

    p = [0] * (n + 1)




    p[0] = 1

    for i in range(1, n + 1):
        k = 1
        while ((k * (3 * k - 1)) / 2 <= i) :
            p[i] += ((1 if k % 2 else -1) *
                    p[i - (k * (3 * k - 1)) // 2])

            if (k > 0):
                k *= -1
            else:
                k = 1 - k
```

```
return p[n]
```

Results Achieved :

Outputs :



```
1
Starting script...

--- Block Performance ---
⏱ Time:   0.826764 sec
💾 Memory: 19.2326 kB
-------------------------
Script finished.
```

```
True
Starting script...

--- Block Performance ---
⏱ Time:   0.869899 sec
💾 Memory: 19.2326 kB
-------------------------
Script finished.
PS D:\python prol\assignw2>
```

```
4.0
Starting script...

--- Block Performance ---
⏱ Time:   0.828105 sec
💾 Memory: 19.2326 kB
-------------------------
Script finished.
PS D:\python prol\assignw2>
```

```
1
Starting script...

--- Block Performance ---
⏱️ Time:   0.870759 sec
💾 Memory: 19.2326 kB
-------------------------
Script finished.
PS D:\python prol\assignw2>
```

```
Result: True
Current memory usage: 0.00 KiB
Starting script...

--- Block Performance ---
⏱️ Time:   1.565043 sec
💾 Memory: 19.2326 kB
-------------------------
Script finished.
```

```
Result: False
Time elapsed: 0.001456 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
PS D:\python prol\assignw3>
```

```
Result: True
Time elapsed: 0.000742 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
PS D:\python prol\assignw3>
```

```
Result: True
Time elapsed: 0.000052 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
PS D:\python prol\assignw3>
```

```
Result: False
Time elapsed: 0.000046 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
PS D:\python prol\assignw3>
```

```
Result: [2, 2, 2, 3]
Time elapsed: 0.000043 seconds
Current memory usage: 0.00 KB
Peak memory usage: 0.00 KB
PS D:\python prol\assignw3>
```

```
 PERFORMANCE MEASUREMENT: count_distinct_prime_factors
========================================
  Time: 0.0000 seconds
 Memory: 0.06 KB (peak)
 Result: 5
========================================
PS D:\python prol\assignw4>
```

```
 PERFORMANCE MEASUREMENT: is_mersenne_prime
========================================
  Time: 0.0000 seconds
 Memory: 0.71 KB (peak)
 Result: True
========================================
PS D:\python prol\assignw4>
```

```
Twin primes up to 20:
[(3, 5), (5, 7), (11, 13), (17, 19)]

 PERFORMANCE MEASUREMENT: twin_primes
========================================
  Time: 0.0001 seconds
 Memory: 0.34 KB (peak)
 Result: [(3, 5), (5, 7), (11, 13), (17, 19)]
========================================
PS D:\python prol\assignw4>
```

```
  PERFORMANCE MEASUREMENT: is_prime_power
 ========================================
   Time: 0.0011 seconds
  Memory: 0.16 KB (peak)
  Result: True
 ========================================
 PS D:\python prol\assignw4>
```

```
 PERFORMANCE MEASUREMENT: count_divisors
========================================
  Time: 0.0000 seconds
 Memory: 0.00 KB (peak)
 Result: 6
========================================
PS D:\python prol\assignw4>
```

```
16
6
0
Execution time: 0.00025349999941681745
PS D:\python prol\assignment 5>
```

```
None
None
True
duration: 0.0006744000002072426
PS D:\python prol\assignment 5>
```

```
 3
 3
 0
 duration: 0.0006625000005442416
 PS D:\python prol\assignment 5>
```

```
True
True
False
Execution time: 0.000420 seconds
PS D:\python prol\assignment 5>
```

```
(2^3) % 5 = 3
Execution time: 0.000223 seconds
PS D:\python prol\assignment 5>
```

```
--- Performance Report: 'heavy_computation' ---
⏱️ Execution Time: 1.801394 seconds
💾 Peak Memory:    38.5671 MB
📉 Current Memory: 0.0000 MB
----------------------------------------

Result: 499999500000
PS D:\python prol\assignw6>
```

```
System of congruences: x ≡ 2 mod 3, x ≡ 3 mod 5, x ≡ 2 mod 7
The unique solution modulo 105 is x = 93
Check: 93 mod 3 = 0, 93 mod 5 = 3, 93 mod 7 = 2
Processing 1000000 items...

--- Performance Report: 'heavy_computation' ---
⏱️ Execution Time: 1.757879 seconds
💾 Peak Memory:    38.5671 MB
📉 Current Memory: 0.0000 MB
----------------------------------------

Result: 499999500000
PS D:\python prol\assignw6>
```

```
Is 9 a quadratic residue mod 17? True
Is 3 a quadratic residue mod 17? False
Processing 1000000 items...

--- Performance Report: 'heavy_computation' ---
⏱️ Execution Time: 1.745900 seconds
💾 Peak Memory:    38.5671 MB
📉 Current Memory: 0.0000 MB
----------------------------------------

Result: 499999500000
PS D:\python prol\assignw6>
```

```
The order of 3 modulo 7 is k = 6
Check: 3^6 mod 7 = 1
Processing 1000000 items...

--- Performance Report: 'heavy_computation' ---
⏱️ Execution Time: 1.743787 seconds
💾 Peak Memory:    38.5671 MB
📉 Current Memory: 0.0000 MB
----------------------------------------

Result: 499999500000
PS D:\python prol\assignw6>
```

```
13 is Fibonacci and prime? True
7 is Fibonacci and prime? False
8 is Fibonacci and prime? False
29 is Fibonacci and prime? False
Processing 1000000 items...

--- Performance Report: 'heavy_computation' ---
⏱️ Execution Time: 1.759614 seconds
💾 Peak Memory:    38.5671 MB
📉 Current Memory: 0.0000 MB
----------------------------------------

Result: 499999500000
PS D:\python prol\assignw6>
```

```
The first 10 Lucas numbers are:
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
Time taken: 0.0017 seconds
Max Memory: 0.00 MB
PS D:\python prol\assignw7>
```

```
Is 8 a perfect power? True
Time taken: 0.0002 seconds
Max Memory: 0.00 MB
PS D:\python prol\assignw7>
```

```
The Collatz sequence length for 6 is: 8
Time taken: 0.0002 seconds
Max Memory: 0.00 MB
PS D:\python prol\assignw7>
```

```
PS D:\python prol\assignw7> & C:/Users/hp/App
The 5th 3-gonal (triangular) number is: 15
Time taken: 0.0002 seconds
Max Memory: 0.00 MB
PS D:\python prol\assignw7>
```

```
False
Time taken: 0.0003 seconds
Max Memory: 0.00 MB
PS D:\python prol\assignw7>
```

```
True
False
duration:0.001187
PS D:\python prol\assignment 8 file>
```

```
the prime factors of 12 are: 2
the prime factors of 4 are: 2
the prime factors of 187 are: 11
duration:0.000340
PS D:\python prol\assignment 8 file>
```

```
8 py3.py
1.2020564036593433
duration:0.000248
PS D:\python prol\assignment 8 file>
```

```
15
627
56
duration:0.000486
PS D:\python prol\assignment 8 file>
```

BRIEF DESCRIPTION:

factorial(n): Calculates the product of all integers from 1 to *n*. Used frequently in permutations and combinations.

is_palindrome(n): Checks if the number reads the same forwards and backwards (e.g., 121), usually by reversing the digits or converting to a string.

mean_of_digits(n): Extracts every digit from a number, sums them up, and divides by the count of digits to find the average value.

digital_root(n): A recursive process where you sum the digits of a number, then sum the digits of the result, repeating until only a single digit remains.

is_abundant(n): A number is "abundant" if the sum of its proper divisors (excluding the number itself) is **greater** than the number (e.g., 12).

is_deficient(n): The opposite of abundant; the sum of proper divisors is **less** than the number (e.g., 10).

is_harshad(n): Checks if the number is divisible by the sum of its own digits (e.g., 18 is divisible by 1+8=9).

is_automorphic(n): Checks if the square of the number ends with the number itself (e.g., $5^2 = 25$, $25^2 = 625$).

is_pronic(n): Checks if a number is the product of two consecutive integers, $n = x \times (x+1)$ (e.g., 12 is 3×4).

10. **prime_factors(n)**: Decomposes a number into its fundamental prime building blocks (e.g., 12 returns [2, 2, 3]).
11. **count_distinct_prime_factors(n)**: Similar to the above, but counts unique primes only (e.g., 12 has factors 2 and 3, so the count is 2).
12. **is_prime_power(n)**: Checks if a number consists of a single prime raised to a power k (e.g., $8 = 2^3$, $27 = 3^3$).
13. **is_mersenne_prime(p)**: Checks if a number of the form $2^p - 1$ is prime. These are often the largest known prime numbers.
14. **twin_primes(limit)**: Finds pairs of prime numbers that are exactly 2 apart (e.g., 3 and 5, 11 and 13).
15. **count_divisors(n)**: Counts how many numbers divide n evenly.
16. **aliquot_sum(n)**: Calculates the sum of all proper divisors of n. This is used to determine if a number is Perfect, Abundant, or Deficient.
17. **are_amicable(a, b)**: Checks for a friendly relationship where the aliquot sum of a equals b, and the aliquot sum of b equals a (e.g., 220 and 284).
18. **multiplicative_persistence(n)**: Calculates how many times you must multiply the digits of a number together until you reach a single digit.
19. **is_highly_composite(n)**: Checks if a number has more divisors than any positive integer smaller than it. These are essentially "anti-primes."
20. **mod_exp(base, exponent, modulus)**: Efficiently calculates $(base^{exponent}) \% modulus$. This is the core algorithm behind modern cryptography (RSA).

mod_inverse(a, m): Finds a number x such that $a \cdot x \equiv 1 \pmod m$. This is essential for "decrypting" in cryptography.

crt(remainders, moduli) (Chinese Remainder Theorem): Solves a system of equations to find a number that leaves specific remainders when divided by specific divisors.

is_quadratic_residue(a, p): Determines if a number a is a perfect square modulo p. Important in cryptography and determining solvability of equations.

order_mod(a, n): Finds the smallest power k such that $a^k \equiv 1 \pmod n$. This relates to the cycles of numbers.

is_fibonacci_prime(n): Checks if a number exists in the Fibonacci sequence AND is also a prime number.

lucas_sequence(n): Generates numbers similar to Fibonacci ($L_n = L_{n-1} + L_{n-2}$) but starting with 2 and 1 instead of 0 and 1.

is_perfect_power(n): Checks if a number can be written as integer a raised to integer power b (where $b > 1$).

collatz_length(n): Simulates the "3n + 1" problem and counts how many steps it takes for a number to collapse down to 1.

polygonal_number(s, n): Calculates numbers that can be arranged in shapes (triangles, squares, pentagons) based on the formula for s-sided polygons.

30. Detects "trick" numbers that are composite (not prime) but pass Fermat's basic primality test.
31. A probabilistic test used by computers to check if huge numbers are prime very quickly with high accuracy.
32. An integer factorization algorithm that is faster than simple trial division, used to break small to medium encryption keys.
33. : Approximates the Riemann Zeta function, a complex infinite series with deep connections to the distribution of prime numbers.
34. Calculates the number of unique ways an integer can be written as a sum of positive integers (e.g., for 4: 4, 3+1, 2+2, 2+1+1, 1+1+1+1).

DIFFICULTY FACED BY STUDENT: Confusing the difference between numeric and string representations, which can lead to errors when reversing the input. Making mistakes in iterative digit summing, such as not stopping when reaching a single digit.Inefficiency in loops and not optimizing for performance, which can cause slow or time-consuming execution for high input values

SKILLS ACHIEVED: Converting numbers to strings and vice versa, slicing strings, and handling numeric operations builds strong data handling skills

Mastery of loops for repeated operations such as summing digits or checking divisors enhances control flow understanding.

Understanding properties of numbers, such as palindromes, digital roots, and abundant numbers, deepens knowledge of number theory and arithmetic

concepts.