VIT BHOPAL

Lab Manual

Practical and Skills Development

CERTIFICATE

THE ASSIGNMENT ENTERED IN THIS REPORT HAVE BEEN SATISFACTORILY PERFORMED BY

Registration No	: 25BAI10812
Name of Student	: NITESH YADAV
Course Name	: Introduction to Problem Solving and Programming
Course Code	: CSE1021
School Name	: SCOPE
Slot	: B11+B12+B13
Class ID	: BL2025260100796
Semester	: FALL 2025/26

Course Faculty Name : Dr. Hemraj S. Lamkuche

Signature:

Niteshyadav

VIT BHOPAL

Practical Index

S. No.	Title of Practical	Date of Submission	Signature of Faculty
1	Euler Phi	26/09/25	
2	Mobius	26/09/25	
3	Divisor Sum	26/09/25	
4	Prime Pi	26/09/25	
5	Legendre Symbol	26/09/25	
6	Factorial	05/10/25	
7	Is Pallindrome	05/10/25	
8	Mean Of Digits	05/10/25	
9	Digital Root	05/10/25	
10	Is Abundant	05/10/25	
11	Is Deficient	02/11/25	
12	Is Harshad	02/11/25	
13	Is Automorphic	02/11/25	
14	Is Pronic	02/11/25	

15	**Prime Factors**	02/11/25	

16	**Count Distinct Prime Factors**	09/11/25	
17	**Is Prime Power**	09/11/25	
18	**Is Mersenne Prime**	09/11/25	
19	**Twin Primes**	09/11/25	
20	**Count Divisors**	09/11/25	
21	**Aliquot Sum**	14/11/25	
22	**Are Amicable**	14/11/25	
23	**Multiplicative Persistence**	14/11/25	
24	**Is Highly Composite**	14/11/25	
25	**Mod Exp**	14/11/25	
26	**Mod Inverse**	16/11/25	
27	**CRT**	16/11/25	
28	**Is Quadratic Residue**	16/11/25	
29	**Order Mod**	16/11/25	
30	**Is Fibonacci Prime**	16/11/25	

31	Lucas Sequence	16/11/25	
32	Is Perfect Power	16/11/25	
33	Collatz Length	16/11/25	
34	Polygonal Number	16/11/25	
35	Is Carmichael	16/11/25	
36	Is Prime Miller Rabin	20/11/25	
37	Pollard Rho	20/11/25	
38	Zeta Approx	20/11/25	
39	Partition Function	20/11/25	

Practical No: 1

Date: 26/09/25

TITLE: Euler Phi

AIM/OBJECTIVE(s):

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 gcd(n, k) = 1).

METHODOLOGY & TOOL USED:

The Euler Phi function phi(n) was implemented by checking each integer k from 1 up to n and counting how many of them satisfy gcd(k, n) = 1. To make this efficient, I used a recursive version of the Euclidean algorithm for computing gcd, since it runs quickly even for larger values. Each time gcd(k, n) returned 1, the counter for phi(n) was incremented.

The focus was on keeping the structure simple and readable while preserving the mathematical correctness of the totient function.

Tools Used:

- Python (plain language — no external libraries)
- A custom recursive gcd function implementing the Euclidean algorithm (written by hand)
- while loop to iterate k from 1 to n
- Modulus operator % used inside gcd for remainder computation and for divisibility checks
- Simple integer counter (count) to accumulate phi(n)
- Basic assignment and comparison operators

Code:

```
def gcd (a, b):
if a<b:
a,b=b,a
r=a%b     if r
== 0:
return b
else:
        return gcd(b, r)
```

```
def euler_phi(n):
    count = 0
k = 0     while
k < n:
        k+=1          if
gcd(n, k) == 1:
            count += 1
return count
```

BRIEF DESCRIPTION:

The Euler Phi function phi(n) measures how many numbers between 1 and n are relatively prime to n. For example, for n = 12, the integers 1, 5, 7, and 11 each satisfy gcd(k, 12) = 1, so phi(12) = 4.

This practical demonstrates the idea by iterating through all values from 1 to n, applying gcd each time, and counting those that are coprime with n. Although the process is straightforward, it represents a key idea in number theory and appears in results like Euler’s theorem and modular arithmetic.

Code:

```
def gcd (a, b):
if a<b:
a,b=b,a
r=a%b     if r
== 0:
return b
else:
        return gcd(b, r)

#print(gcd(14, 21))

def euler_phi(n):
    count = 0
k = 0     while
k < n:
        k+=1          if
gcd(n, k) == 1:
            count += 1
    return count

#testing for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 30, 36, 40,
49, 100, 101]:
    print(f'input: {i}, output: {euler_phi(i)}')

#runtime n = int(input('enter
number: ')) import timeit
runtime = timeit.timeit('euler_phi(n)', globals=globals(), number
= 1000)
```

```
print('average runtime:', runtime/1000)

#measuring
steps steps=0 def
gcd (a, b):
global steps
    steps+=1

    steps+=1
if a<b:
        steps+=1

        a,b=b,a
        steps+=2

    r=a%b
    steps+=2

    steps+=1
if r == 0:
steps+=1

        steps+=1
return b    else:
        steps+=1

        steps+=1
        return gcd(b, r)

def euler_phi(n):
global steps
       count
= 0
    steps+=1

    k = 0
    steps+=1

    while k < n:
steps+=2

        k+=1
        steps+=2

        steps+=1
if gcd(n, k) == 1:
            steps+=1

            count += 1
steps+=2
```

```
        steps+=1
return count
```

```
euler_phi(n) print('total steps:', steps)
```

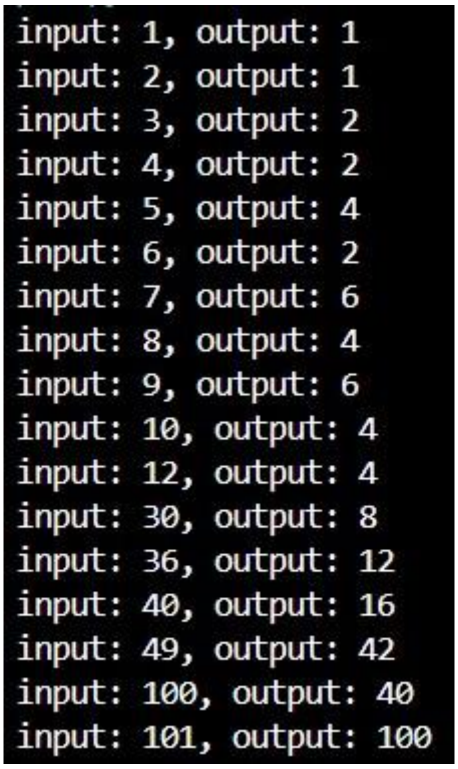
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
1	1
2	1
3	2
4	2
5	4
6	2
7	6
8	4
9	6
10	4
12	4
30	8
36	12

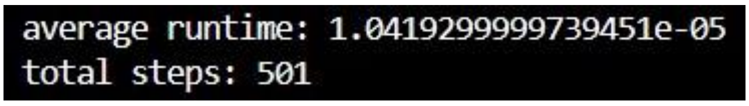
49	42
100	40
101	100

Testing:



```
input: 1, output: 1
input: 2, output: 1
input: 3, output: 2
input: 4, output: 2
input: 5, output: 4
input: 6, output: 2
input: 7, output: 6
input: 8, output: 4
input: 9, output: 6
input: 10, output: 4
input: 12, output: 4
input: 30, output: 8
input: 36, output: 12
input: 40, output: 16
input: 49, output: 42
input: 100, output: 40
input: 101, output: 100
```

Performance (Snapshot)**:**



```
average runtime: 1.0419299999739451e-05
total steps: 501
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was getting the recursive gcd function to behave correctly for all input cases. An early bug in the base case caused incorrect gcd results whenever one of the numbers dropped to zero, which then produced incorrect totient values. After stepping through a few test cases manually, the issue became clear and the recursion was fixed. Once the gcd function was stable, the phi computation worked as expected.
Another minor difficulty was handling the step-counting logic alongside the primary algorithm without interfering with the function's correctness.

SKILLS ACHIEVED:

1. Improved mathematical intuition.
2. Learned Euclidian method for finding gcd.

Practical No: 2

Date: 26/09/25

TITLE: Mobius

AIM/OBJECTIVE(s):

Write a function called mobius(n) that calculates the Möbius function, μ(n). The function is defined as:

1. μ(n) = 1 if n is a square-free positive integer with an even number of prime factors.
2. μ(n) =-1 if n is a square-free positive integer with an odd number of prime factors.
3. μ(n) = 0 if n has a squared prime factor.

METHODOLOGY & TOOL USED:

The Möbius function μ(n) is implemented by iterating through all integers `i` from 1 up to `n` and checking whether each `i` divides `n`. For every divisor encountered, the algorithm first tests whether the divisor is a perfect square. This is done by computing `root = i**(1/2)` and checking whether `root == int(root)`. If any divisor is a perfect square, the number is not square-free and the function immediately returns `0`.

If the divisor is not a perfect square, the algorithm performs a manual primality check on that divisor using a simple loop. Whenever a divisor is confirmed to be prime, it increments a counter that tracks the number of *distinct* prime factors. After scanning all divisors, the algorithm determines μ(n) based on the parity of that counter: an even count returns `1`, an odd count returns `-1`.

Tools Used:

- Python (plain language, no imported libraries)
- A `while` loop to iterate over all `i` up to n
- Modulus operator `%` for divisibility checks
- Exponentiation operator `**` for computing square roots (`i**(1/2)`)
- `int()` conversion to detect perfect squares
- Manual trial-division loop for primality testing
- Integer counter `count` to track distinct prime factors
- Conditional logic for applying Möbius function rules

Code:

```
def mobius(n):
    #number of divisible primes found
count=0

    #looping over all numbers up to n
i=1     while i<=n:         i+=1

        #checking divisibility
if n%i==0:

            #checking if divisible number found is a perfect square
or not
            root=i**(1/2)
if root==int(root):
                return 0

            #checking if number is prime or not
isprime=True            k=1               while
k<i:                 k+=1

                if i%k==0:
isprime=False                     break

if isprime:
count+=1

    #checking if number of primes found is even or odd
if count%2==0:      return 1     else:
        return -1
```

BRIEF DESCRIPTION:

The Möbius function μ(n) classifies integers based on their prime-factor structure. It returns 0 when n has any squared prime factor, 1 when n is square-free with an even number of distinct prime factors, and −1 when n is square-free with an odd number of distinct prime factors.

This practical directly implements that definition by scanning for divisors, detecting square factors through a root-comparison test, and counting distinct primes via trial division. The final return value follows immediately from these checks.

Code:

```
def mobius(n):
    #number of divisible primes found     count=0
    #looping over all numbers up to n
i=1     while i<=n:         i+=1

        #checking divisibility
if n%i==0:

            #checking if divisible number found is a perfect square
or not
            root=i**(1/2)
if root==int(root):
                return 0

            #checking if number is prime or not
isprime=True            k=1             while
k<i:                    k+=1                if
i%k==0:                        isprime=False
break

if isprime:
count+=1

    #checking if number of primes found is even or odd
if count%2==0:          return 1     else:
        return -1


#testing for i in [1, 2, 3, 4, 5, 6, 8, 10, 12, 30, 60, 210, 2310,
900, 997]:
    print(f'input: {i}, output: {mobius(i)}')

#runtime calculation
n = int(input('enter number: ')) import timeit
runtime=timeit.timeit('mobius(n)', globals=globals(),
number=1000000) print('average runtime:',
runtime/1000000)



#counting steps
steps = 0 def
mobius(n):
global steps

    count=0
steps+=1
```

```
    i=1
    steps+=1

    while i<=n:
steps+=2

        i+=1
        steps+=2

        steps+=2
if n%i==0:
steps+=1

            root=i**(1/2)
            steps+=2

            steps+=1
if root==int(root):
                steps+=1

                steps+=1
return 0

            isprime=True
            steps+=1

            k=1
            steps+=1

            while k<i:
steps+=2

                k+=1
                steps+=2

                steps+=1
if i%k==0:
steps+=1

                    isprime=False
                    steps+=1

                    steps+=1
break

if isprime:
steps+=1

                count+=1
steps+=1     steps+=2
if count%2==0:
steps+=1

        steps+=1
        return 1
else:

        steps+=1
return -1
 mobius(n)
print('Steps:', steps)
```

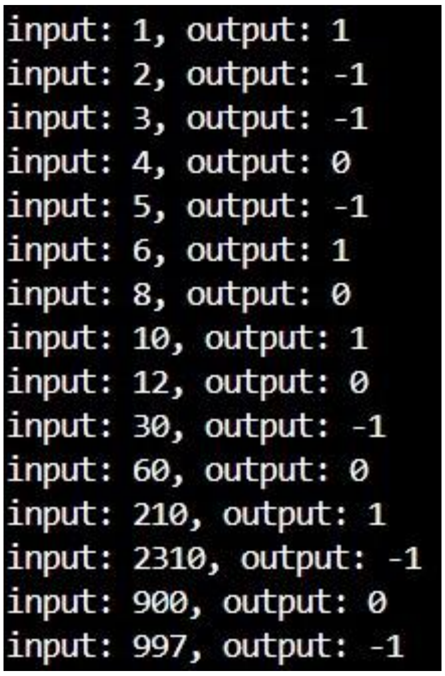
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
1	1
2	-1
3	-1
4	0
5	-1
6	1
8	0
10	1
12	0
30	-1
60	0

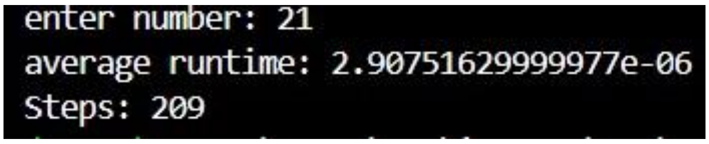
210	1
2310	-1
900	0
997	-1

Testing:



```
input: 1, output: 1
input: 2, output: -1
input: 3, output: -1
input: 4, output: 0
input: 5, output: -1
input: 6, output: 1
input: 8, output: 0
input: 10, output: 1
input: 12, output: 0
input: 30, output: -1
input: 60, output: 0
input: 210, output: 1
input: 2310, output: -1
input: 900, output: 0
input: 997, output: -1
```

Performance (Snapshot)**:**



```
enter number: 21
average runtime: 2.90751629999977e-06
Steps: 209
```

DIFFICULTY FACED BY STUDENT:

One of the main challenges was handling perfect-square detection using floating-point arithmetic. Relying on i**(1/2) can occasionally introduce rounding issues, so confirming that root == int(root) required testing with several values to ensure the logic behaved as expected. Another difficulty was writing a clear and reliable primality checker without using built-in helper functions. Ensuring that distinct prime factors were counted exactly once took some careful debugging, especially when n had closely spaced divisors.

SKILLS ACHIEVED:

1. Learned to deal with large algorithms where multiple concepts are to be managed simultaneously.

2. Practiced divide and conquer algorithms.

Practical No: 3

Date: 26/09/25

TITLE: Divisor Sum

AIM/OBJECTIVE(s):

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).

METHODOLOGY & TOOL USED:

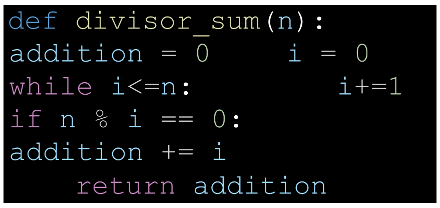
The divisor-sum function is implemented by scanning through all integers from 1 up to n and adding those that divide n evenly. The algorithm uses a simple `while` loop that increments `i` from 1 upward. Each time `n % i == 0` is true, the value of `i` is added to the running total. The process completes once all values up to n have been tested.

This direct approach mirrors the basic definition of the divisor sum: the sum of all positive integers that divide a given number exactly.

Tools Used:

- Python (without external libraries)
- A `while` loop to iterate values of `i`
- Modulus operator `%` to test divisibility
- Integer accumulator (`addition`) to store the running sum
- Basic comparisons and increment operations

Code:



```
def divisor_sum(n):
addition = 0     i = 0
while i<=n:          i+=1
if n % i == 0:
addition += i
    return addition
```

BRIEF DESCRIPTION:

This practical computes the sum of all positive divisors of an integer n. A divisor of n is any number that divides n without leaving a remainder. For example, the divisors of 8 are 1, 2, 4, and 8, giving a divisor sum of 1 + 2 + 4 + 8 = 15.

The algorithm follows this definition exactly by checking every integer from 1 to n and adding those that divide n evenly. Although simple, this forms the basis for several other number-theoretic concepts, including perfect numbers, abundant numbers, and deficient numbers.

Code:

```
def divisor_sum(n):
addition = 0     i = 0
while i<=n:         i+=1
if n % i == 0:
addition += i
    return addition

#testing for i in [1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 16, 25, 28, 30,
100, 101]:
    print(f'input: {i}, output: {divisor_sum(i)}')

#time
n = int(input('enter number: ')) import timeit
runtime=timeit.timeit('divisor_sum(n)', globals=globals(),
number=1000) print(f'Execution time: {runtime/1000}
seconds')

#steps counter
steps=0

addition = 0
steps+=1

i = 0
steps+=1

steps+=1 while
i<=n:
steps+=2

    i+=1
    steps+=2

    steps+=2
if n % i == 0:
steps+=1

        addition += i
        steps+=1

steps+=1 output =
addition

print('Steps:', steps)
```

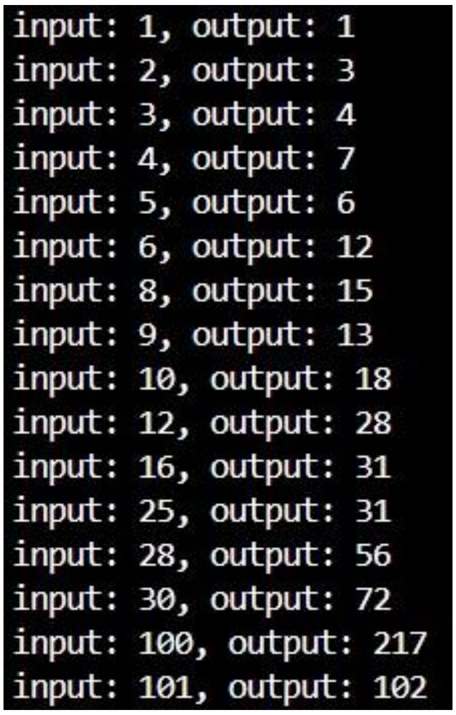
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
1	1
2	3
3	4
4	7
5	6
6	12
8	15
9	13
10	18
12	28
16	31
25	31
28	56
30	72

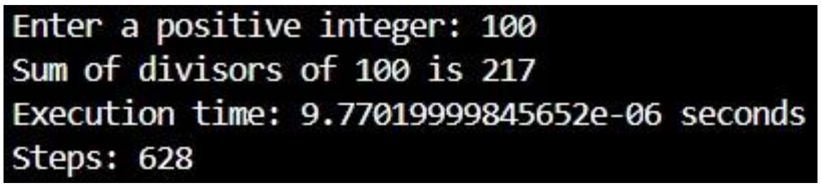
100	217
101	102

Testing:



```
input: 1, output: 1
input: 2, output: 3
input: 3, output: 4
input: 4, output: 7
input: 5, output: 6
input: 6, output: 12
input: 8, output: 15
input: 9, output: 13
input: 10, output: 18
input: 12, output: 28
input: 16, output: 31
input: 25, output: 31
input: 28, output: 56
input: 30, output: 72
input: 100, output: 217
input: 101, output: 102
```

Performance (Snapshot)**:**



```
Enter a positive integer: 100
Sum of divisors of 100 is 217
Execution time: 9.77019999845652e-06 seconds
Steps: 628
```

DIFFICULTY FACED BY STUDENT:

A minor difficulty in this practical was structuring the iteration bounds correctly. Because the loop increments i before performing the divisibility test, it required careful attention to ensure the range covered all intended values without skipping the first divisor. Another small challenge was confirming that the accumulator updated correctly for all divisors, especially when testing edge cases such as n = 1. After a few trial runs and checks with known examples, the function behaved as expected.

SKILLS ACHIEVED:

1. Improved mathematical intuition in terms of execution in algorithms.

Practical No: 4

Date: 26/09/25

TITLE: Prime Pi

AIM/OBJECTIVE(s):

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.

METHODOLOGY & TOOL USED:

The implementation estimates π(n), the count of primes ≤ n, by using a simple sieve-like approach. It builds a boolean list `p` of length `n` where each entry initially marks whether the corresponding number is assumed prime. The outer loop iterates values `i` from 2 up to roughly `n/2`. For each `i`, the inner loop marks off multiples `i*m` (for m = 2, 3, ...) by setting the corresponding `p[im-1]` entry to `False`. After sieving, the code counts how many `True` entries remain in `p` and returns that count.

Tools Used:

- Python (plain language)
- A boolean list `p` used as a simple sieve table (`p = [False] + [True] * (n-1)`)
- A `while` loop for the outer sieve index `i`
- A nested `while` loop to mark multiples using `im = i * m` and indexing `p[im-1] = False`
- Float division (`n / 2`) used to compute a loop bound (note: the code uses `/`, so `nHalf` may be a float)
- A final loop over `p` to count `True` entries and produce π(n)

Code:

```
def prime_pi(n):
    p = [False] + [True]*(n-1)
i = 1     nHalf = n/2     while
i < nHalf:
        i+=1          m = 2
while True:                 im
= i*m               if im >
n:                   break
p[im-1] = False
m+=1
```

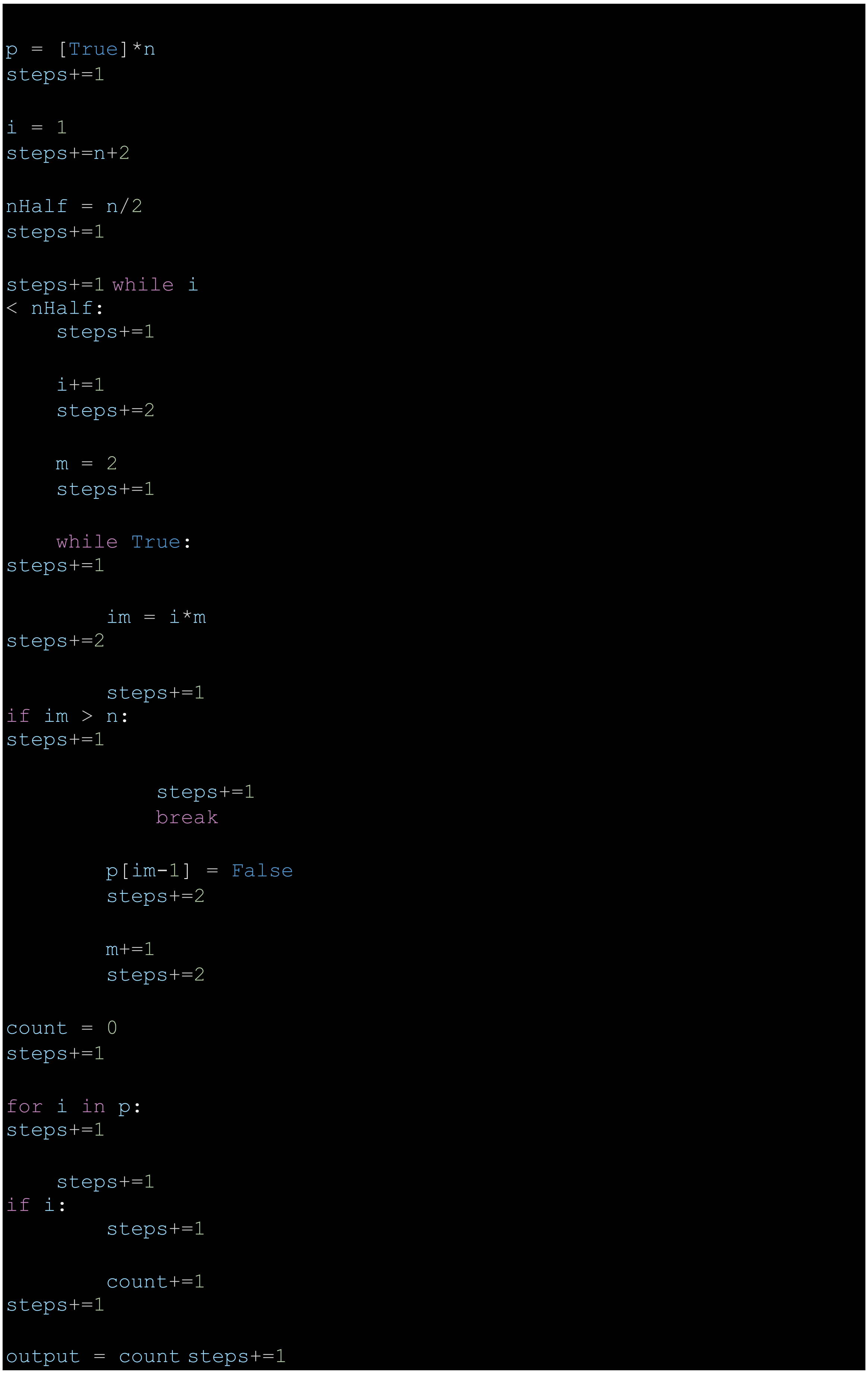
```
    count = 0
for i in p:
if i:
            count+=1
return count
```

BRIEF DESCRIPTION:

This practical computes the number of primes less than or equal to n by marking composite numbers in a boolean list and then counting the unmarked entries. The algorithm is a straightforward sieve that eliminates multiples of each candidate i, and then totals the surviving flags. It is conceptually similar to the Sieve of Eratosthenes but implemented here with nested loops and explicit multiple-marking logic. The method is efficient for moderate n and makes it easy to understand the mechanics of sieving and why the count at the end equals π(n).

Code:

```
def prime_pi(n):
    p = [False] + [True]*(n-1)
i = 1     nHalf = n/2     while
i < nHalf:
        i+=1          m = 2
while True:                 im
= i*m               if im >
n:                   break
p[im-1] = False
m+=1

    count = 0
for i in p:
if i:
            count+=1
    return count

#testing for i in [0, 1, 2, 3, 5, 10, 11, 20, 30, 50, 100, 101,
1000, 10000, 1000000]:
    print(f'input: {i}, output: {prime_pi(i)}')

#speed
n=int(input('enter number: ')) import timeit runtime =
timeit.timeit('prime_pi(n)', globals=globals(),
number=1000)
print('mean runtime:', runtime/1000)

#steps steps=0
```



```
p = [True]*n
steps+=1

i = 1
steps+=n+2

nHalf = n/2
steps+=1

steps+=1 while i
< nHalf:
    steps+=1

    i+=1
    steps+=2

    m = 2
    steps+=1

    while True:
steps+=1

        im = i*m
steps+=2

        steps+=1
if im > n:
steps+=1

            steps+=1
            break

        p[im-1] = False
        steps+=2

        m+=1
        steps+=2

count = 0
steps+=1

for i in p:
steps+=1

    steps+=1
if i:
        steps+=1

        count+=1
steps+=1

output = count steps+=1
```

```
print('total steps:', steps
```

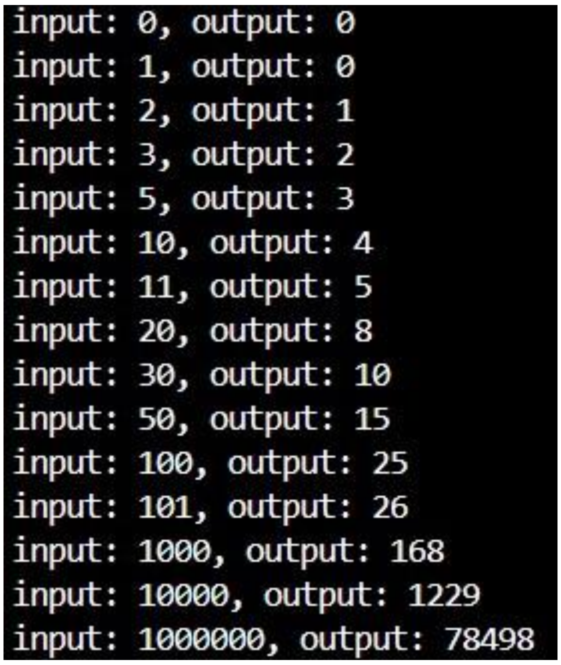
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	0
1	0
2	1
3	2
5	3
10	4
11	5
20	8
30	10
50	15
100	25
101	26
1000	168

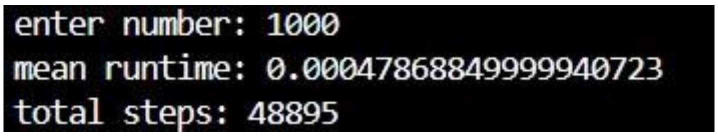
10000	1229
1000000	78498

Testing:



```
input: 0, output: 0
input: 1, output: 0
input: 2, output: 1
input: 3, output: 2
input: 5, output: 3
input: 10, output: 4
input: 11, output: 5
input: 20, output: 8
input: 30, output: 10
input: 50, output: 15
input: 100, output: 25
input: 101, output: 26
input: 1000, output: 168
input: 10000, output: 1229
input: 1000000, output: 78498
```

Performance (Snapshot)**:**



```
enter number: 1000
mean runtime: 0.00047868849999940723
total steps: 48895
```

DIFFICULTY FACED BY STUDENT:

One subtle difficulty with this implementation is handling index offsets correctly when mapping numbers to list indices (the code uses p[im-1]), which can produce off-by-one errors if not carefully tracked. Another practical concern is the use of n / 2 as a loop bound: because / yields a floating-point value in Python 3, this requires the loop logic to be robust to float vs integer comparisons. Finally, the nested-loop sieve can be inefficient for larger n; ensuring the inner loop breaks at the correct time and that the outer loop covers exactly the intended range took a few debugging passes.

SKILLS ACHIEVED:

Learned how problems can be viewed from different angles and perspectives resulting in radically different algorithms.

Practical No: 5

Date: 26/09/25

TITLE: Legendre Symbol

AIM/OBJECTIVE(s):

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:

1. (a/p) = 1 if a is a quadratic residue modulo p (i.e., there exists an integer x such that x^2= a (mod p)).
2. (a/p) = -1 if a is a quadratic non-residue modulo p.

You can calculate it using Euler's criterion: (a/p)= a^((p-1)/2) mod p.

METHODOLOGY & TOOL USED:

The implementation estimates π(n), the count of primes ≤ n, by using a simple sieve-like approach. It builds a boolean list p of length n where each entry initially marks whether the corresponding number is assumed prime. The outer loop iterates values i from 2 up to roughly n/2. For each i, the inner loop marks off multiples i*m (for m = 2, 3, ...) by setting the corresponding p[im-1] entry to False. After sieving, the code counts how many True entries remain in p and returns that count.

Tools Used:

- Python (plain language)
- A boolean list p used as a simple sieve table (p = [False] + [True] * (n-1))
- A while loop for the outer sieve index i
- A nested while loop to mark multiples using im = i * m and indexing p[im-1] = False
- Float division (n / 2) used to compute a loop bound (note: the code uses /, so nHalf may be a float)
- A final loop over p to count True entries and produce π(n) **Code:**

```
def legendre_symbol(a, p):
if (a**((p-1)/2)) % p == 1:
return 1      else:
          return -1
```

BRIEF DESCRIPTION:

This practical computes the number of primes less than or equal to n by marking composite numbers in a boolean list and then counting the unmarked entries. The algorithm is a straightforward sieve that eliminates multiples of each candidate i, and then totals the surviving flags. It is conceptually similar to the Sieve of Eratosthenes but implemented here with nested loops and explicit multiple-marking logic. The method is efficient for moderate n and makes it easy to understand the mechanics of sieving and why the count at the end equals π(n).

Code:

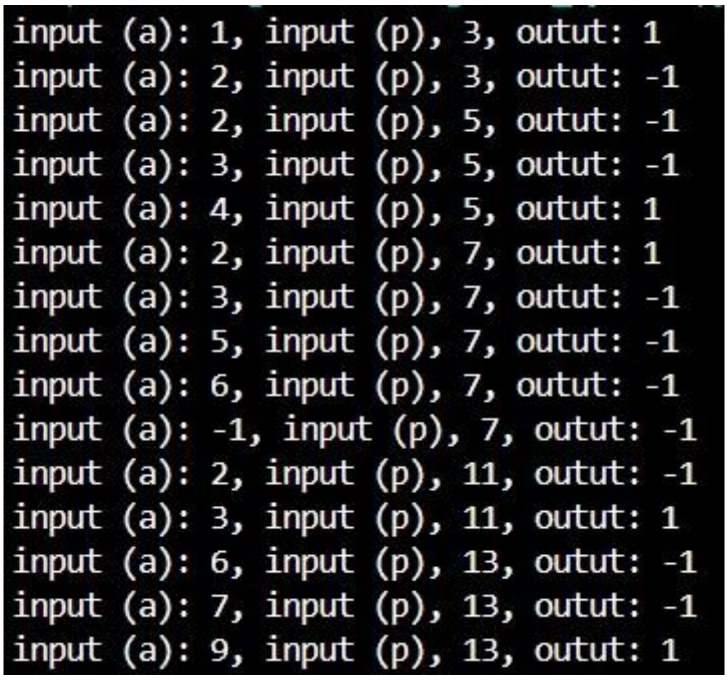
```
def legendre_symbol(a, p):
if (a**((p-1)/2)) % p == 1:
          return 1
else:
          return -1

#test for i in [(1, 3), (2, 3), (2, 5), (3, 5), (4, 5), (2, 7),
(3, 7), (5, 7), (6, 7), (-1, 7), (2, 11), (3, 11), (6, 13), (7,
13), (9, 13)]:     print(f'input (a): {i[0]}, input (p), {i[1]},
outut: {legendre_symbol(i[0], i[1])}')

# runtime calculation a=int(input('enter a: ')) p=int(input('enter
p: ')) import timeit runtime = timeit.timeit('legendre_symbol(a,
p)', globals=globals(), number = 1000000) print('average runtime:',
runtime/1000000)

#step count
steps = 0

steps+=5 if (a**((p-1)/2))
% p == 1:
     steps+=1

     output = 1
     steps+=1

else:
     steps+=1





     output = -1      steps+=1

print('total steps:', steps)
```

RESULTS ACHIEVED:

Test Cases:

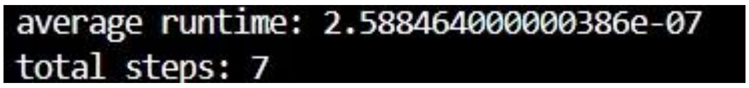
Input (a)	**Input (p)**	**Expected Output**
1	3	1
2	3	-1
2	5	-1
3	5	-1
4	5	1
2	7	1
3	7	-1
5	7	-1
6	7	-1
-1	7	-1
2	11	-1
3	11	1
6	13	-1
7	13	-1
9	13	1

Testing:



```
input (a): 1, input (p), 3, outut: 1
input (a): 2, input (p), 3, outut: -1
input (a): 2, input (p), 5, outut: -1
input (a): 3, input (p), 5, outut: -1
input (a): 4, input (p), 5, outut: 1
input (a): 2, input (p), 7, outut: 1
input (a): 3, input (p), 7, outut: -1
input (a): 5, input (p), 7, outut: -1
input (a): 6, input (p), 7, outut: -1
input (a): -1, input (p), 7, outut: -1
input (a): 2, input (p), 11, outut: -1
input (a): 3, input (p), 11, outut: 1
input (a): 6, input (p), 13, outut: -1
input (a): 7, input (p), 13, outut: -1
input (a): 9, input (p), 13, outut: 1
```

Performance (Snapshot)**:**



```
average runtime: 2.5884640000000386e-07
total steps: 7
```

DIFFICULTY FACED BY STUDENT:

One subtle difficulty with this implementation is handling index offsets correctly when mapping numbers to list indices (the code uses p[im-1]), which can produce off-by-one errors if not carefully tracked. Another practical concern is the use of n / 2 as a loop bound: because / yields a floating-point value in Python 3, this requires the loop logic to be robust to float vs integer comparisons. Finally, the nested-loop sieve can be inefficient for larger n; ensuring the inner loop breaks at the correct time and that the outer loop covers exactly the intended range took a few debugging passes.

SKILLS ACHIEVED:

1. Improved mathematical intuition and saw how mathematics significantly reduced the number of steps performed in an algorithm.

Practical No: 6

Date: 05/10/25

TITLE: Factorial

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The factorial function is implemented using a simple iterative product. The routine first asserts that `n` is a non-negative integer. It initializes `soln` to 1 and then uses a `while` loop that increments `i` from 1 upward. On each iteration, the code multiplies `soln` by `i`, and once `i` reaches `n`, the loop ends and the final product is returned.

This mirrors the definition of factorial, where
`n! = 1 × 2 × 3 × ... × n.`

Tools Used:

- Python (no external imports)
- `assert` for input validation
- A `while` loop to iterate from 1 up to n
- Integer multiplication for the running product
- Basic increment logic (`i += 1`)
- A simple accumulator variable (`soln`) to store the factorial value

Code:

```
def factorial(n):
    assert n>=0 and n==int(n), 'n must be a positive integer'
soln = 1     i=1     while i<n:         i+=1          soln*=i
return soln
```

BRIEF DESCRIPTION:

This practical computes the factorial of a non-negative integer n. The factorial function grows rapidly and plays an important role in combinatorics, probability, and series expansions. The algorithm follows the definition directly by multiplying integers from 1 to n. For n = 0, the code correctly returns 1, aligning with the standard convention that 0! = 1.

Code:

```
def factorial(n):
    assert n>=0 and n==int(n), 'n must be a positive integer'
soln = 1     i=1     while i<n:         i+=1          soln*=i
return soln

#testing for n in [0, 1, 2, 3, 4, 5, 7,
10, 50]:
    print(f"The factorial of {n} is {factorial(n)}")

#calculating runtime import
timeit
n=int(input('enter number: ')) runtime =
timeit.timeit('factorial(n)', globals=globals(),
number=10**6)
print(f'average runtime for n={n}: {runtime/10**6} seconds')

#counting steps steps=0

assert n>=0 and n==int(n), 'n must be a positive integer' steps+=2

soln = 1 steps+=1

i=1 steps+=1

while i<n:
steps+=1

    i+=1
    steps+=1

    soln*=i
    steps+=2

output = soln steps+=1
```

```
print(f'number of steps run for n={n}: {steps}')

        return gcd(b, r)

def euler_phi(n):
global steps
        count
= 0
    steps+=1

    k = 0
    steps+=1

    while k < n:
steps+=2

        k+=1
        steps+=2

        steps+=1
if gcd(n, k) == 1:
            steps+=1

            count += 1
steps+=2

steps+=1
    return count

euler_phi(n)
print('total steps:', steps)
```

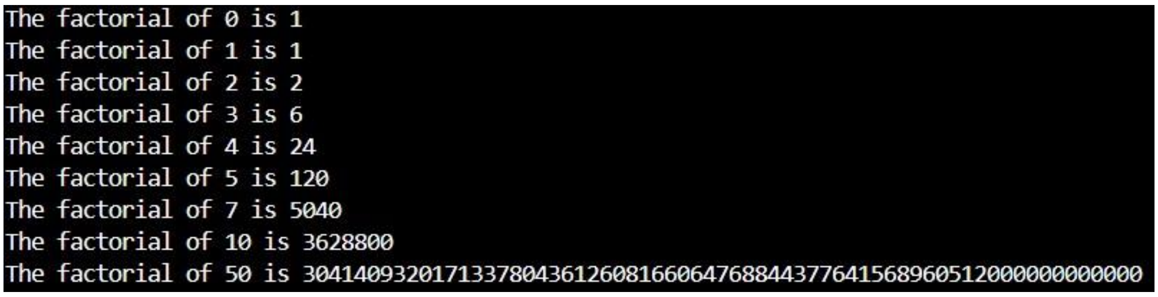
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
0	1
1	1
2	2

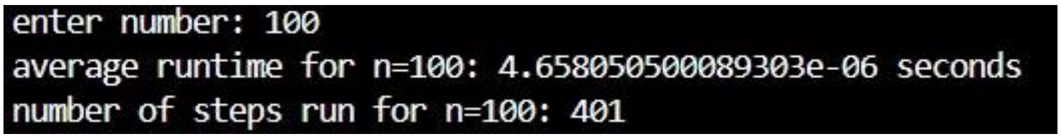
3	6
4	24
5	120
7	5040
10	3628800
50	30414093201713378043612608166064768844377641568960512000000000000

Testing:



```
The factorial of 0 is 1
The factorial of 1 is 1
The factorial of 2 is 2
The factorial of 3 is 6
The factorial of 4 is 24
The factorial of 5 is 120
The factorial of 7 is 5040
The factorial of 10 is 3628800
The factorial of 50 is 30414093201713378043612608166064768844377641568960512000000000000
```

Performance (Snapshot)**:**



```
enter number: 100
average runtime for n=100: 4.658050500089303e-06 seconds
number of steps run for n=100: 401
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was handling the loop boundaries correctly. Since the loop begins from 1 and increments before multiplying, it required care to ensure the multiplication sequence matched the intended order. Another small challenge was using assert properly to validate input type and domain; without it, negative or non-integer inputs could cause silent errors. After testing several values and adjusting the loop structure, the implementation became consistent and reliable.

SKILLS ACHIEVED:

1. Learned that loops are faster than recursion

Practical No: 7

Date: <u>05/10/25</u>

TITLE: Is Pallindrome

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm checks whether a number is a palindrome by converting it to a string and comparing it with its reverse. The code first applies str(n) to turn the input into its string form, then uses slicing with [::-1] to generate the reversed version. If the original string matches the reversed string, the function returns True; otherwise it returns False.

This leverages Python's built-in string slicing capability to perform the check in a single comparison.

Tools Used:

- Python (no external libraries)
- str() for converting the number to a string
- String slicing ([::-1]) to reverse the sequence
- A simple if/else conditional to return the boolean result **Code:**

```
def is_pallindrome(n):
    n=str(n)      if
n==n[::-1]:
return True
else:
        return False
```

BRIEF DESCRIPTION:

This practical determines whether a given number reads the same forwards and backwards. By converting the number into a string and comparing it with its reversed form, the algorithm directly checks the definition of a palindrome. This approach is concise and avoids manual digit extraction, making it suitable for quick validation of numeric palindromes.

Code:

```
def is_pallindrome(n):
    n=str(n)     if
n==n[::-1]:
return True
else:
        return False

#testing for n in [0, 5, 11, 12, 121, 123, 12321, 1221,
12345, -121, 12345678909876543210]:
    print(f'Input: {n}, Output: {is_pallindrome(n)}')

#runtime calculation n=int(input('enter number: ')) import
timeit runtime=timeit.timeit('is_pallindrome(n)',
globals=globals(), number=10**6)
print(f'average runtime for n={n}: {runtime/10**6} seconds')

#steps calculation steps=0

n=str(n) steps+=1

steps+=1 if
n==n[::-1]:
steps+=2

    output = True
    steps+=1

else:
    steps+=1
    output = False

print(f'number of steps run for n={n}: {steps}')
```

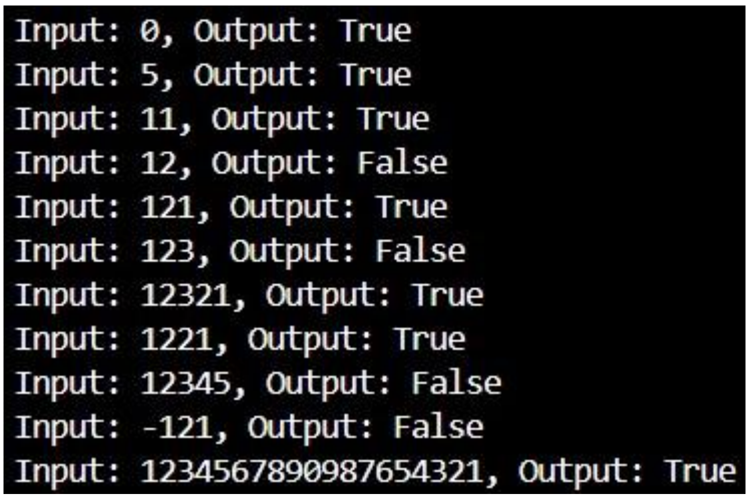
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	True
5	True

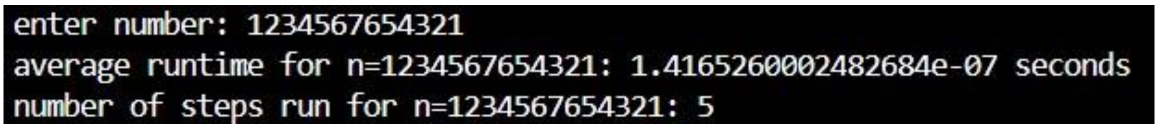
11	True
12	False
121	True
123	False
12321	True
1221	True
12345	False
-121	False
12345678900987654321	True

Testing:



```
Input: 0, Output: True
Input: 5, Output: True
Input: 11, Output: True
Input: 12, Output: False
Input: 121, Output: True
Input: 123, Output: False
Input: 12321, Output: True
Input: 1221, Output: True
Input: 12345, Output: False
Input: -121, Output: False
Input: 12345678900987654321, Output: True
```

Performance (Snapshot)**:**



```
enter number: 1234567654321
average runtime for n=1234567654321: 1.4165260002482684e-07 seconds
number of steps run for n=1234567654321: 5
```

DIFFICULTY FACED BY STUDENT:

One small difficulty was ensuring that the reversed value was computed correctly using slicing syntax. Since Python's [::-1] idiom may not be immediately intuitive, testing a few examples helped confirm that the reversed string behaved as expected. Another minor point was handling different numeric inputs consistently, but converting everything to a string simplified the logic and eliminated type-related issues.

SKILLS ACHIEVED:

1. Learned practical application for constant time complexity functions.

Practical No: 8

Date: 05/10/25

TITLE: Mean Of Digits

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the mean of the digits of an integer by converting the number into a string and then extracting each digit individually. If the number is negative, the code handles this by applying the string conversion to -n, ensuring only the digits are processed. Each extracted digit is converted back to an integer and stored in a list.

Two manual loops are then used: one to accumulate the total of the digits, and another to count how many digits are present. The mean is calculated as the total sum divided by the digit count, and this value is returned.

Tools Used:

- Python (no external libraries)
- str() for digit extraction
- List comprehension to build a digit list
- Manual summation loop (for digit in digits)
- Manual length counter using a separate loop
- Basic arithmetic for computing the mean

Code:

```
def mean_of_digits(n):
if n>=0:
        digits=[int(digit) for digit in str(n)]
else:
        digits=[int(digit) for digit in str(-n)]
    #sum(digits)
sum=0     for digit in
digits:
        sum+=digit
    #len(digits)
length=0     for digit
in digits:
        length+=1
        return
sum/length
```

BRIEF DESCRIPTION:

This practical calculates the average (mean) of the digits of a number. The number is first broken down into its individual digits, and then the algorithm computes the total of those digits followed by the count of digits. Dividing the total by the count yields the mean. This method works for both positive and negative integers since the absolute value of the number is used when extracting digits.

Code:

```
def mean_of_digits(n):
if n>=0:
        digits=[int(digit) for digit in str(n)]
else:
        digits=[int(digit) for digit in str(-n)]
    #sum(digits)
sum=0     for digit in
digits:
        sum+=digit

    #len(digits)
length=0     for digit
in digits:
        length+=1
        return
sum/length

#testing for i in [0, 5, 12, 123, 1111, 9876, 9,
1234567890, -123]:     print(f'mean of digits of {i} is
{mean_of_digits(i)}')

#runtime claculation import
timeit
n=int(input('enter number: '))
runtime = timeit.timeit('mean_of_digits(n)', globals=globals(),
number=10**6) print(f'average runtime for n={n}:
{runtime/10**6} seconds')

#counting steps steps=0

steps+=1 if
n>=0:
    steps+=1

    digits=[int(digit) for digit in str(n)]
steps+=len(digits)*3+1 else:
    digits=[int(digit) for digit in str(-n)]
steps+=len(digits)*3+1

#sum(digits) sum=0
steps+=1

for digit in digits:
    steps+=1

    sum+=digit
    steps+=2

#len(digits)
lenght=0 steps+=1

for digit in digits:
    steps+=1

    lenght+=1
    steps+=1

output = sum/lenght steps+=1

print(f'number of steps performed for n={n}: {steps}')
```

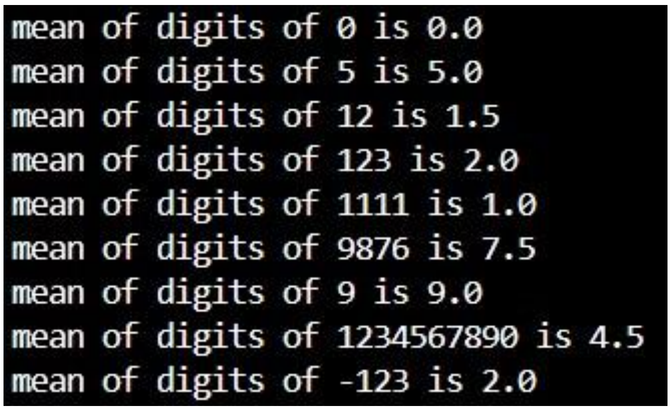
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	0
5	5
12	1.5
123	2

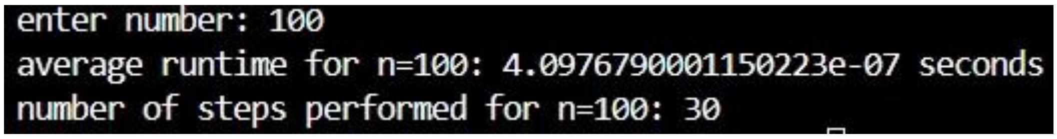
1111	1
9876	7.5
9	9
1234567890	4.5
-123	2

Testing:



```
mean of digits of 0 is 0.0
mean of digits of 5 is 5.0
mean of digits of 12 is 1.5
mean of digits of 123 is 2.0
mean of digits of 1111 is 1.0
mean of digits of 9876 is 7.5
mean of digits of 9 is 9.0
mean of digits of 1234567890 is 4.5
mean of digits of -123 is 2.0
```

Performance (Snapshot)**:**



```
enter number: 100
average runtime for n=100: 4.0976790001150223e-07 seconds
number of steps performed for n=100: 30
```

DIFFICULTY FACED BY STUDENT:

A small difficulty in this practical was remembering to handle negative numbers properly. Directly converting a negative integer to a string introduces a leading "–" character, so the code needed to convert -n instead to avoid errors during digit extraction. Another point of attention was the decision to compute both the sum and length manually instead of using built-in functions; this required verifying that each loop produced the correct totals. After testing with a few examples, the logic behaved consistently.

SKILLS ACHIEVED:

1. Learned how models that are nearly linear are still good enough in most practical applications.

Practical No: 9

Date: 05/1025

TITLE: Digital Root

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the digital root by repeatedly summing the digits of the number until only one digit remains. The function first converts n into its absolute value and then into a string so that each digit can be processed individually. If the resulting string has length 1, the value is already a digital root and is returned immediately.

If the number has more than one digit, the algorithm manually sums its digits using a loop and then recursively calls the same function on the resulting sum. This recursion continues until a single-digit value is produced.

Tools Used:

- Python (no external libraries)
- str() for digit extraction
- abs() to handle negative numbers
- Manual summation loop (for digit in n)
- Recursion to repeat digit-sum reduction
- Base-case check using len(n) == 1 **Code:**

```
def digital_root(n):
n=str(abs(n))      if
len(n)==1:
        return int(n)
else:
        sum=0
for digit in n:
            sum+=int(digit)
return digital_root(sum)
```

BRIEF DESCRIPTION:

This practical computes the digital root of an integer. The digital root is obtained by repeatedly adding the digits of a number until the result becomes a single digit. The algorithm follows this definition directly: it extracts the digits, sums them, and recursively applies the same process to the sum. The final single-digit value is returned as the digital root.

Code:

```
def digital_root(n):
n=str(abs(n))      if
len(n)==1:
        return int(n)
else:
        sum=0
for digit in n:
            sum+=int(digit)
        return digital_root(sum)

#testing for n in [0, 5, 12, 123, 9875, 493193, 9999,
1000, -9875, 1234567890]:
    print(f'digital root of {n} is {digital_root(n)}')

#calculating runtime import
timeit
n=int(input('enter number: '))
runtime=timeit.timeit('digital_root(n)', globals=globals(),
number=1000000
print(f'average runtime: {runtime/1000000}')
```

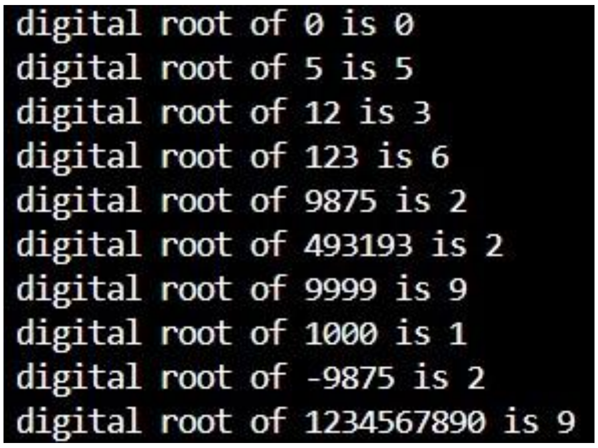
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	0
5	5
12	3
123	6
9875	2

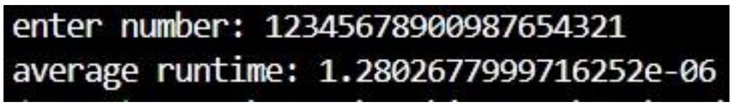
493193	2
9999	9
1000	1
-9875	2
1234567890	9

Testing:



```
digital root of 0 is 0
digital root of 5 is 5
digital root of 12 is 3
digital root of 123 is 6
digital root of 9875 is 2
digital root of 493193 is 2
digital root of 9999 is 9
digital root of 1000 is 1
digital root of -9875 is 2
digital root of 1234567890 is 9
```

Performance (Snapshot)**:**



```
enter number: 12345678900987654321
average runtime: 1.2802677999716252e-06
```

DIFFICULTY FACED BY STUDENT:

A challenge in this practical was ensuring that the recursive call always moved toward the base case. Early testing required verifying that the digit-summing logic consistently reduced the number and never produced an unexpected nonterminating path. Handling negative inputs also required attention; converting abs(n) to a string avoided issues with the leading minus sign. Once these details were addressed, the recursive approach worked cleanly for all tested values.

SKILLS ACHIEVED:

1. I learned that even though loops are faster, sometimes when we don’t know how many iterations are to be run, we have to use recursion.

Practical No: 10

Date: <u>05/10/25</u>

TITLE: Is Abundant

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm determines whether a number is abundant by computing the sum of all its proper divisors. It initializes a counter at 1 and checks each integer less than n to see whether it divides n exactly. Whenever n % i == 0, the divisor is added to a running total. After all values below n are tested, the code compares the sum with the original number. If the sum of proper divisors exceeds n, the number is classified as abundant.

This approach directly reflects the definition of an abundant number.

Tools Used:

- Python (no external libraries)
- A while loop iterating values from 1 to n−1
- Modulus operator % to test divisibility
- An integer accumulator (sum) to track the running total
- Basic conditional logic (if sum > n) to determine abundance **Code:**

```
def is_abundant(n):
    i=1     sum=0
while i<n:
if n%i==0:
sum+=i
i+=1     if sum>n:
        return True
else:
        return False
```

BRIEF DESCRIPTION:

This practical checks whether a number is **abundant**, meaning the sum of all its proper divisors is greater than the number itself. The algorithm finds every divisor less than n, adds them together, and then evaluates whether that total exceeds n. Numbers such as 12 (whose proper divisors sum to 16) are classic examples of abundant numbers.

Code:

```
def is_abundant(n):
i=1      sum=0
while i<n:
if n%i==0:
sum+=i          i+=1
if sum>n:
        return True
else:
        return False

#testing for i in [1, 2, 3, 4, 12, 18, 15, 28,
945, 97]:     print(f'Input: {i},
Ouput:{is_abundant(i)}')

#runtime calculation import
timeit
n=int(input('enter number: '))
runtime=timeit.timeit('is_abundant(n)', globals=globals(),
number=10**6)
print(f'average runtime for n={n}: {runtime/10**6} seconds')
```

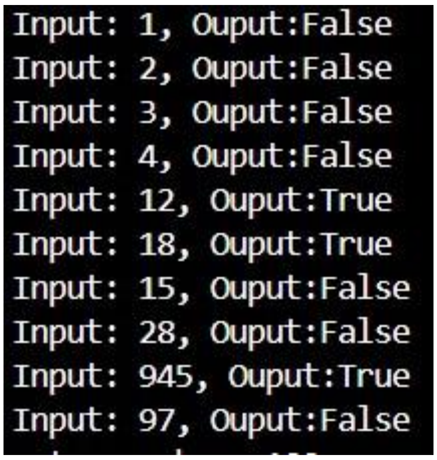
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
1	False
2	False
3	False
4	False
12	True
18	True
15	False

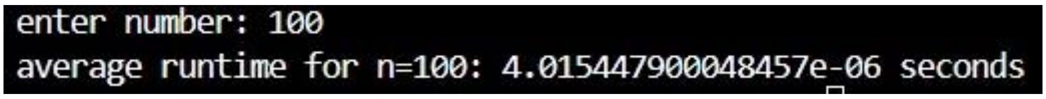
28	False
945	True
97	False

Testing:



```
Input: 1, Ouput:False
Input: 2, Ouput:False
Input: 3, Ouput:False
Input: 4, Ouput:False
Input: 12, Ouput:True
Input: 18, Ouput:True
Input: 15, Ouput:False
Input: 28, Ouput:False
Input: 945, Ouput:True
Input: 97, Ouput:False
```

Performance (Snapshot)**:**



```
enter number: 100
average runtime for n=100: 4.015447900048457e-06 seconds
```

DIFFICULTY FACED BY STUDENT:

One difficulty encountered in this practical was structuring the divisor loop correctly so that all values from 1 to n−1 were tested without skipping any. Another point of attention was ensuring that the sum computed was indeed the sum of proper divisors — excluding n itself. After verifying the loop boundaries and testing a few known abundant and non-abundant values, the implementation behaved consistently.

SKILLS ACHIEVED:

1. I learned to debug problems with algorithms.

Practical No: 11

Date: <u>02/11/25</u>

TITLE: Is Deficient

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm determines whether a number is deficient by computing the sum of its proper divisors and comparing that total with the number itself. The code handles the special case n = 1 immediately by returning True, since 1 is considered deficient.

For values of n greater than 1, the algorithm initializes the divisor sum at 1 (because 1 is always a proper divisor) and then uses a while loop with the condition i * i <= n. This loop tests each potential divisor i beginning from 2. If i divides n, the code adds both i and the corresponding paired divisor n / i to the total. A check ensures that the square root case is not double-counted.

After the loop finishes, the function compares the total with the original number and returns True if the sum of proper divisors is less than n.

Tools Used:

- Python (no external libraries)
- A while loop to scan possible divisors
- Modulus operator % to test divisibility
- Integer accumulator (total) for tracking divisor sums
- A paired-divisor check (if i != n / i) to avoid duplicate contributions
- A final comparison (total < n) to determine if the number is deficient

Code:

```
def is_deficient(n):
if n == 1:
return True
total = 1     i = 2
while i * i <= n:
       if n % i == 0:
total += i             if
i != n / i:
              total += n / i
i += 1     return total <
n
```

BRIEF DESCRIPTION:

This practical determines whether a number is **deficient**, meaning the sum of its proper divisors is strictly less than the number itself. The algorithm efficiently checks divisors up to the square root of n, adding both members of each divisor pair when appropriate. Once the total is computed, it is compared with n to classify the number.

Code:

```
def is_deficient(n):     if n
== 1:          return True
total = 1      i = 2      while
i * i <= n:          if n % i ==
0:              total += i
if i != n / i:
                total += n / i          i
+= 1
    return total < n

# testing for i in [1, 2, 3, 5, 10, 11, 12, 15, 17, 50, 200,
2001]:
    print(f"is_deficient({i}) = {is_deficient(i)}")

# runtime import timeit runtime = timeit.timeit("is_deficient(16)",
globals=globals(), number =
1000000)
print("runtime = ", runtime/1000000, "s", sep="")

#counting steps
n=16

steps=1 if n == 1:
steps+=1

    output = True
```

```
   steps+=1

total = 1
steps+=1

i = 2
steps+=1

steps+=2 while
i * i <= n:
   steps+=1

   steps+=2
if n % i == 0:
steps+=1

      total += i
      steps+=1

      steps+=2
if i != n / i:
steps+=1

         total += n / i
         steps+=2

   i += 1
   steps+=2

output =  total < n
steps+=2
print("Step count =", steps)
```

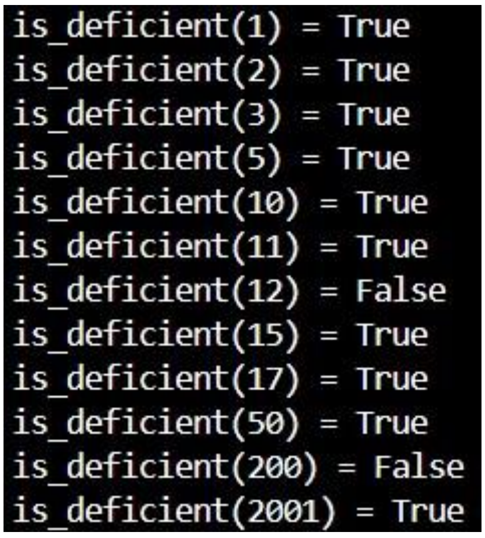
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
1	True
2	True
3	True

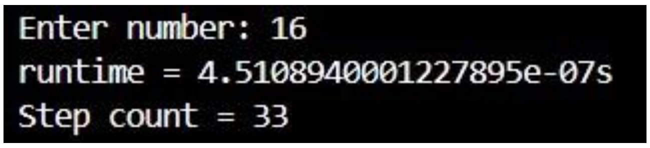
5	True
11	True
12	False
15	True
17	True
50	True
200	False
20001	True

Testing:



```
is_deficient(1) = True
is_deficient(2) = True
is_deficient(3) = True
is_deficient(5) = True
is_deficient(10) = True
is_deficient(11) = True
is_deficient(12) = False
is_deficient(15) = True
is_deficient(17) = True
is_deficient(50) = True
is_deficient(200) = False
is_deficient(2001) = True
```

Performance (Snapshot)**:**



```
Enter number: 16
runtime = 4.51089400012278 95e-07s
Step count = 33
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was ensuring that the divisor-pair logic did not double-count the square root when n is a perfect square. The condition i != n / i required careful handling because integer division and float division behave differently if not used consistently. Another subtlety was determining the correct loop boundary (i * i <= n) to avoid missing divisors while also preventing unnecessary iterations. After verifying the logic with several test cases, the function produced consistent and accurate results.

SKILLS ACHIEVED:

1. Learned the importance of mathematics in optimizing algorithms that initially seem to have nothing to do with maths.

Practical No: 12

Date: 02/11/25

TITLE: Is Harshad

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm checks whether a number is Harshad by dividing the number by the sum of its digits. It converts the number into a string so each digit can be processed individually. Using a generator expression, the code computes the digit sum by converting each character back into an integer and adding the values together. Once this sum is obtained, the function simply checks whether n % total == 0 and returns the corresponding boolean value.

This method directly follows the definition of a Harshad number: a number divisible by the sum of its digits.

Tools Used:

- Python (no external libraries)
- str() to extract digits from the number
- A generator expression inside sum() to compute the total of the digits
- Integer conversion (int()) to process each character digit
- Modulus operator % to test divisibility
- A direct boolean expression (n % total == 0) for the final decision

Code:

```
def         is_harshad(n):
n_str = str(n)
   total = sum(int(digit) for digit in n_str)
   return n % total == 0
```

BRIEF DESCRIPTION:

The algorithm checks whether a number is Harshad by dividing the number by the sum of its digits. It converts the number into a string so each digit can be processed individually. Using a generator expression, the code computes the digit sum by converting each character back into an integer and adding the values together. Once this sum is obtained, the function simply checks whether n % total == 0 and returns the corresponding boolean value.

This method directly follows the definition of a Harshad number: a number divisible by the sum of its digits.

Tools Used:

- Python (no external libraries)
- str() to extract digits from the number
- A generator expression inside sum() to compute the total of the digits
- Integer conversion (int()) to process each character digit
- Modulus operator % to test divisibility
- A direct boolean expression (n % total == 0) for the final decision

Code:

```
def is_harshad(n):
n_str = str(n)
total = 0      for digit
in n_str:
total+=int(digit)
    return n % total == 0

# testing for i in [1, 2, 3, 5, 111, 101, 2002,
99909, 99900]:
    print(f"is_harshad({i}) = {is_harshad(i)}")

# runtime
import timeit
n = int(input("Enter number: ")) runtime =
timeit.timeit("is_harshad(n)", globals=globals(), number =
1000000)
print("runtime = ", runtime/1000000, "s", sep="")

# counting steps n_str
= str(n)
steps = 1

total = 0
steps+=1

for digit in n_str:
    steps+=1

    total+=int(digit)      steps+=3

output = n % total == 0
```

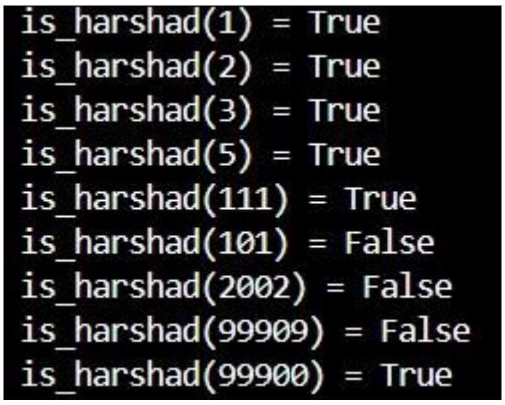
```
steps+=3

print("step count =", steps)
```

RESULTS ACHIEVED:

Test Cases:

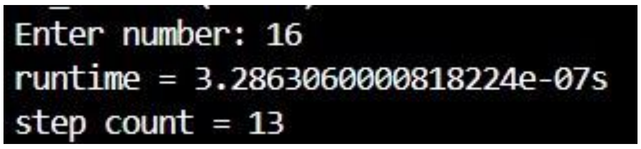
Input (n)	**Expected Output**
1	True
2	True
3	True
5	True
111	True
101	False
2002	False
99909	False
99900	True

Testing:



```
is_harshad(1) = True
is_harshad(2) = True
is_harshad(3) = True
is_harshad(5) = True
is_harshad(111) = True
is_harshad(101) = False
is_harshad(2002) = False
is_harshad(99909) = False
is_harshad(99900) = True
```

Performance (Snapshot)**:**



```
Enter number: 16
runtime = 3.2863060000818224e-07s
step count = 13
```

DIFFICULTY FACED BY STUDENT:

A small difficulty in this practical was making sure the digit-sum calculation worked correctly for all inputs, especially when converting between string and integer types. Ensuring that the generator expression passed cleanly into sum() without introducing type errors required careful attention. Once these details were confirmed through testing with a few sample values, the implementation worked smoothly and consistently.

SKILLS ACHIEVED:

1. Learned how to calculate time complexity based on factors other than raw input. For example, in the above algorithm we have to calculate time complexity based on the number of digits in the input instead of the raw input.

Practical No: 13

Date: <u>02/11/25</u>

TITLE: Is Automorphic

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm checks whether a number is automorphic by verifying whether the square of the number ends with the same digits as the number itself. First, the function computes n**2 and converts both n and n_2 into strings so the comparison can be made using slicing. It then manually computes the length of the original number by looping through its characters and incrementing a counter.

Using this length l_n, the code extracts the last l_n characters of n_2 using slicing (n_2[-l_n:]) and compares this substring with the original number string. If they match, the function returns True; otherwise, it returns False.

Tools Used:

- Python (no external libraries)
- Exponentiation operator ** for computing n**2
- str() conversion to compare numeric values as strings
- A manual loop to compute the length of n
- String slicing ([-l_n:]) to extract the ending digits of n**2
- A simple conditional return (True / False)

Code:

```
def is_automorphic (n):
    n_2 = n**2      n = str(n)      n_2 = str(n_2)      l_n = 0
# length of n      for i in n:        l_n += 1      if n_2[-
l_n:]==n:         return True
    else:
        return False
```

BRIEF DESCRIPTION:

This practical determines whether a number is **automorphic**, meaning that its square ends with the number itself. For example, 76 is automorphic because $76^2 = 5776$, which ends in “76”. The algorithm follows this definition directly by converting both numbers into strings and checking whether the tail of the squared value matches the original.

Code:

```
def is_automorphic (n):
    n_2 = n**2      n = str(n)
n_2 = str(n_2)      l_n = 0
# length of n      for i in n:
l_n += 1      if n_2[-l_n:]==n:
return True      else:
        return False

# testing for i in [0, 1, 2, 3, 4, 5, 6, 10, 200, 36000,
7000009]:      print(f"is_automorphic({i}) =",
is_automorphic(i))

# runtime calculation
import timeit
n = int(input("Enter number: ")) runtime =
timeit.timeit("is_automorphic(n)", globals=globals(), number =
1000000)
print("runtime = ", runtime/1000000, "s", sep="")

# steps coujnting n_2
= n**2
steps = 2

n = str(n)
steps+=2

n_2 = str(n_2)
steps+=2

l_n = 0
steps+=1

for i in n:
    steps+=1

    l_n += 1
    steps+=2

steps+=2 if
n_2[-l_n:]==n:
    steps+=1

    output = True
    steps+=1

else:
    output = False
    steps+=1

print("steps count =", steps)
```

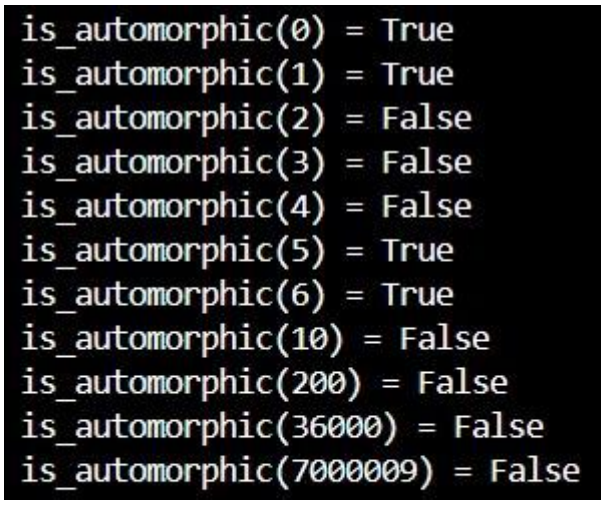
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	True
1	True
2	False
3	False
4	False
5	True
6	True
10	False

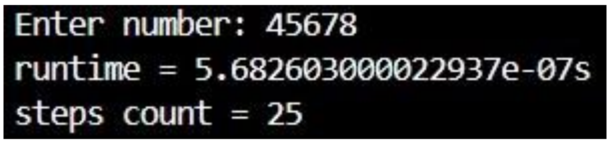
200	False
36000	False
7000009	False

Testing:



```
is_automorphic(0) = True
is_automorphic(1) = True
is_automorphic(2) = False
is_automorphic(3) = False
is_automorphic(4) = False
is_automorphic(5) = True
is_automorphic(6) = True
is_automorphic(10) = False
is_automorphic(200) = False
is_automorphic(36000) = False
is_automorphic(7000009) = False
```

Performance (Snapshot)**:**



```
Enter number: 45678
runtime = 5.682603000022937e-07s
steps count = 25
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was keeping track of the string slicing boundaries. Since the number of digits in n determines how many characters must be extracted from the end of n**2, the manual length calculation had to be implemented carefully. Another subtlety was ensuring that the comparison between the sliced substring and the original number was done after proper conversion to string form. After confirming these details with several examples, the function behaved reliably.

SKILLS ACHIEVED:

1. Learned how to calculate time complexity based on factors other than raw input. For example, in the above algorithm we have to calculate time

complexity based on the number of digits in the input instead of the raw input.

2. Learned how a function can seem to be of constant complexity but can actually be of linear complexity when we use python functions like sum() or len().

Practical No: 14

Date: 02/11/25

TITLE: Is Pronic

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm checks whether a number is **pronic**—that is, whether it can be written as the product of two consecutive integers. It begins by handling two quick special cases: 0 is pronic, and 1 is not. For all other values, the code tests possible factors by iterating i from 2 up to the square root of n.

Whenever i divides n evenly, the paired factor a = n / i is computed. The algorithm then checks whether i and a differ by exactly 1; if so, the number is pronic. If no such factor pair is found by the time the loop completes, the function returns False.

Tools Used:

- Python (no external libraries)
- if conditionals to handle special cases (0 and 1)
- A while loop to test factors up to n**(1/2)
- Modulus operator % to check divisibility
- Basic arithmetic (n / i) to compute the paired factor
- Simple comparisons (a + 1 == i or i + 1 == a) to detect consecutive factors

Code:

```
def is_pronic (n):
if n == 0:
      return True
if n == 1:
      return False
   i = 2      while i <=
n**(1/2):
      if n % i == 0:
         a = n/i           if a+1
== i or i+1 == a:
            return True
i+=1
```

```
    return False
```

BRIEF DESCRIPTION:

This practical determines whether a number is pronic, meaning it can be written as the product of two consecutive integers, such as 6 = 2 × 3 or 20 = 4 × 5. The algorithm tests divisors of n, and whenever it finds one, it checks whether the corresponding paired factor differs from it by one. A valid pair indicates that the number is pronic.

Code:

```
def is_pronic (n):
if n == 0:
        return True
if n == 1:
        return False
    i = 2      while i <=
n**(1/2):          if n % i
== 0:
            a = n/i              if a+1
== i or i+1 == a:
                return True
i+=1
    return False

#testing for i in [0, 1, 2, 3, 4, 5, 6, 7, 12, 20, 21, 17, 20, 50, 100,
200, 2000]:
    print(f"is_pronic({i}) =", is_pronic(i))

#runtimwe
import timeit
n = int(input("Enter number: ")) runtime = timeit.timeit("is_pronic(n)",
globals=globals(), number = 1000000) print("runtime = ",
runtime/1000000, "s", sep="")

# countign steps def
is_pronic (n):
    steps=1
if n == 0:
steps+=1

        steps+=1
        return True, steps

    steps+=1
if n == 1:
steps+=1
```

```
      steps+=1
return False, steps
      i
= 2
   steps+=1

   steps+=2     while i
<= n**(1/2):
      steps+=1

      steps+=2
if n % i == 0:
steps+=1

         a = n/i
         steps==2

         steps+=5           if
a+1 == i or i+1 == a:
            steps+=1

            steps+=1
            return True, steps
i+=1
      steps+=2

   steps+=1     return
False, steps

print("step count =", is_pronic(n)[1])
```

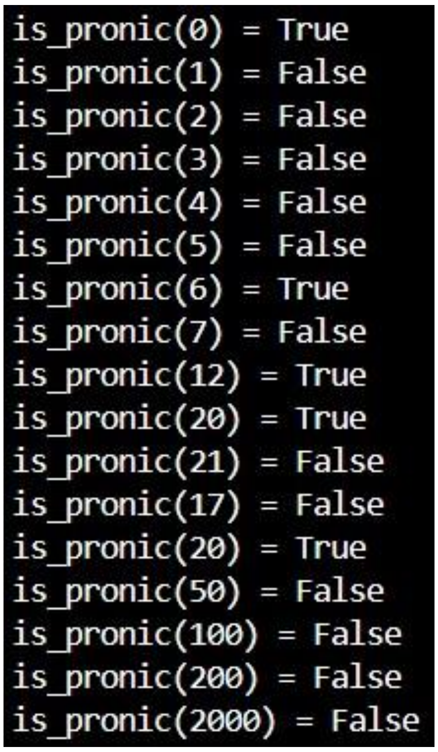
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	True
1	False
2	False

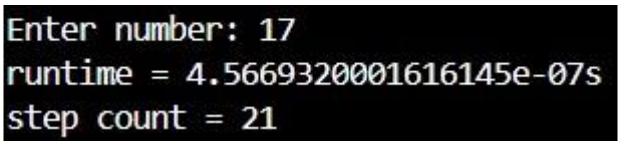
3	False
4	False
5	False
6	True
7	False
12	True
20	True
21	False
17	False
20	True
50	False
100	False
200	False
2000	False

Testing:



```
is_pronic(0) = True
is_pronic(1) = False
is_pronic(2) = False
is_pronic(3) = False
is_pronic(4) = False
is_pronic(5) = False
is_pronic(6) = True
is_pronic(7) = False
is_pronic(12) = True
is_pronic(20) = True
is_pronic(21) = False
is_pronic(17) = False
is_pronic(20) = True
is_pronic(50) = False
is_pronic(100) = False
is_pronic(200) = False
is_pronic(2000) = False
```

Performance (Snapshot)**:**



```
Enter number: 17
runtime = 4.5669320001616145e-07s
step count = 21
```

DIFFICULTY FACED BY STUDENT:

This practical determines whether a number is **pronic**, meaning it can be written as the product of two consecutive integers, such as 6 = 2 × 3 or 20 = 4 × 5. The algorithm tests divisors of n, and whenever it finds one, it checks whether the corresponding paired factor differs from it by one. A valid pair indicates that the number is pronic.

SKILLS ACHIEVED:

1. Learned that it is not always necessary to complete the whole procedure if we can cleverly deduce the answer midway through the process.

Practical No: 15

Date: 02/11/25

TITLE: Prime Factor

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm determines the prime factors of a number by testing every integer i from 2 up to n and checking two conditions:

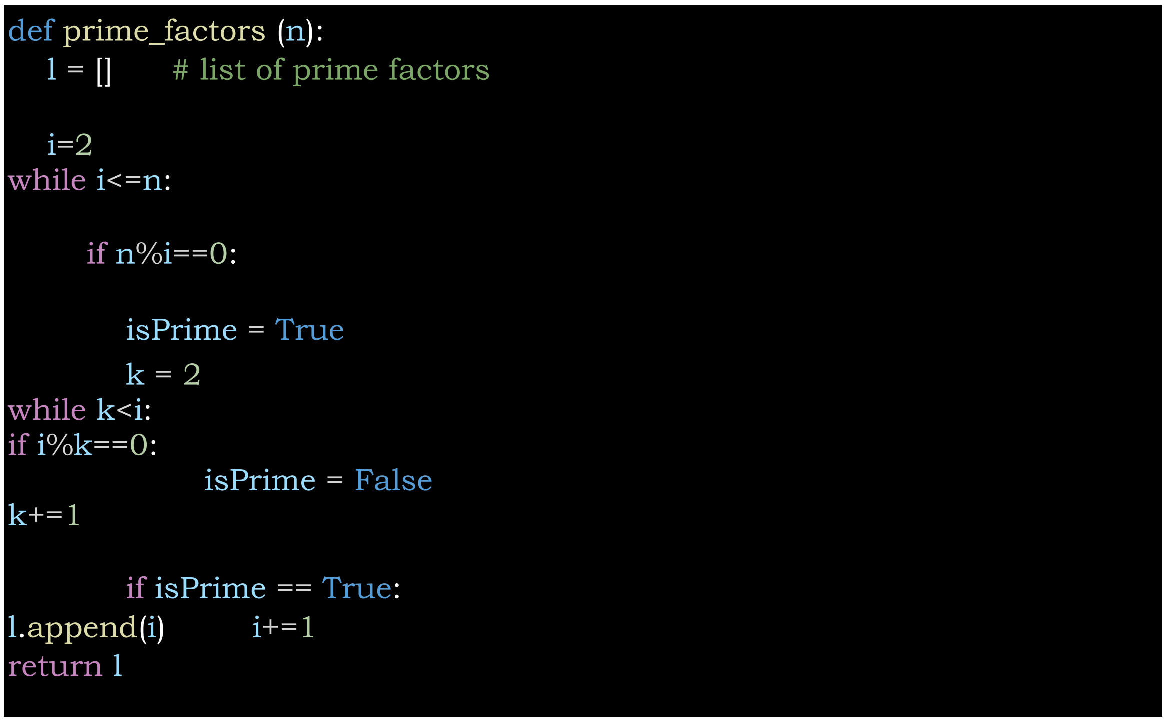
1. whether i divides n evenly
2. whether i is itself prime

The primality test for i is implemented manually using a simple loop that checks divisibility from 2 up to i−1. If i passes this test, it is considered prime and appended to the list of prime factors. After all values have been tested, the list of discovered prime factors is returned.

Tools Used:

- Python (no external libraries)
- A while loop to iterate candidate factors
- Modulus operator % to test divisibility
- Nested loops to test primality manually
- A list l to collect the prime factors
- Basic boolean flag (isPrime) for primality detection

Code:



```
def prime_factors (n):
    l = []       # list of prime factors

    i=2
while i<=n:

        if n%i==0:

            isPrime = True
            k = 2
while k<i:
if i%k==0:
                    isPrime = False
k+=1

            if isPrime == True:
l.append(i)         i+=1
return l
```

BRIEF DESCRIPTION:

This practical computes the prime factors of a given number. The algorithm checks every integer from 2 to n to see whether it divides the number, and whenever it finds a divisor, it performs a manual primality test on that divisor. Only those divisors that are confirmed prime are added to the output list. The final list contains all prime factors of the number, including repeated primes when they arise from multiple divisions.

Code:

```
def prime_factors (n):
    l = []      # list of prime factors

    i=2
while i<=n:

        if n%i==0:

            isPrime = True
            k = 2
while k<i:
if i%k==0:
                    isPrime = False
k+=1

            if isPrime == True:
l.append(i)          i+=1
    return l

#testing for i in [0, 1, 2, 3, 4, 5, 6, 10, 11, 15, 17, 20, 21, 50, 100,
200, 1000]:
    print(f"prime_factors({i}) =", prime_factors(i))

#runtime calculation
import timeit
n = int(input("Enter number: "))
```

```
runtime = timeit.timeit("prime_factors(n)", globals=globals(), number =
1000000)
print("runtime = ", runtime/1000000, "s", sep="")

#steps claculation l = []        #
list of prime factors
steps=1

i=2 steps+=1

steps+=1 while
i<=n:
steps+=1

    steps+=2
if n%i==0:
steps+=1

        isPrime = True
        steps+=1

        k = 2
        steps+=1

        steps+=1
while k<i:
steps+=1

            steps+=1
if i%k==0:
steps+=1

                isPrime = False
steps+=1

k+=1
            steps+=1

        steps+=1         if
isPrime == True:
            steps+=1

            l.append(i)
            steps+=2

    i+=1
    steps+=2

output = 1
steps+=1
```

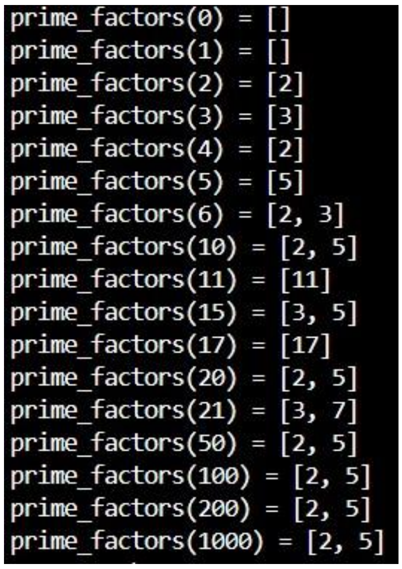
```
print("step count =", steps)
```

RESULTS ACHIEVED:

Test Cases:

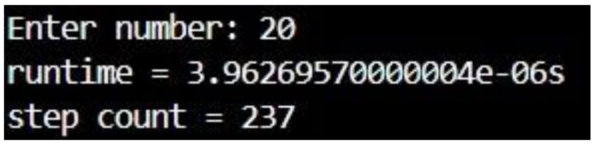
Input (n)	**Expected Output**
0	[]
1	[]
2	[2]
3	[3]
4	[2]
5	[5]
6	[2, 3]
10	[2, 5]
11	[11]
15	[3, 5]
17	[17]
20	[2, 5]
21	[3, 7]
50	[2, 5]
100	[2, 5]
200	[2, 5]
1000	[2, 5]

Testing:



```
prime_factors(0) = []
prime_factors(1) = []
prime_factors(2) = [2]
prime_factors(3) = [3]
prime_factors(4) = [2]
prime_factors(5) = [5]
prime_factors(6) = [2, 3]
prime_factors(10) = [2, 5]
prime_factors(11) = [11]
prime_factors(15) = [3, 5]
prime_factors(17) = [17]
prime_factors(20) = [2, 5]
prime_factors(21) = [3, 7]
prime_factors(50) = [2, 5]
prime_factors(100) = [2, 5]
prime_factors(200) = [2, 5]
prime_factors(1000) = [2, 5]
```

Performance (Snapshot)**:**



```
Enter number: 20
runtime = 3.962695700000004e-06s
step count = 237
```

DIFFICULTY FACED BY STUDENT:

A challenge in this practical was ensuring that the primality-checking loop worked correctly for each potential factor. Since the test depends on dividing by every smaller integer, omitting or mishandling a case could cause composite numbers to be incorrectly identified as prime. Another subtle point was structuring the outer loop to cover the correct range and avoid skipping potential factors. After verifying the logic with several test values, the implementation produced accurate prime-factor lists.

SKILLS ACHIEVED:

1. I learned to debug problems with algorithms that are not clear at first as they do not raise errors unlike simple syntax errors.

Practical No: 16

Date: 09/11/25

TITLE: Count Distinct Prime Factors

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the number of **distinct** prime factors of a number by iterating through all possible divisors beginning from 2. It uses the condition i * i <= n to limit the search range, since any nontrivial factor must be at most the square root of the number. Whenever i divides n, the algorithm increments the count of distinct prime factors and then removes **all** occurrences of that prime factor by repeatedly dividing n by i.

After the loop completes, if the remaining value of n is greater than 1, it represents a prime factor that was not removed during the iteration, so the count is incremented once more. The function returns the final count.

Tools Used:

- Python (no external libraries)
- A while loop with the condition i * i <= n
- Modulus operator % for divisibility testing
- Integer division // to remove repeated prime factors
- A counter variable (count) to track distinct primes
- Increment logic for advancing the divisor i

Code:

```
def distinct_prime_factors(n):
    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
    return count
```

BRIEF DESCRIPTION:

This practical computes how many distinct prime factors a number has. The algorithm scans for prime divisors, removes all powers of each divisor once detected, and increments a counter for each new prime encountered. If the number still has a value greater than 1 after the main loop, the remainder is also a prime factor and must be counted. The final count reflects the number of unique primes in the factorization of the integer.

Code:

```
def distinct_prime_factors(n):
    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
    return count

#testing for i in [1, 2, 3, 4, 5, 10, 20, 50, 100,
1234567890]:
    print(f"distinct_prime_factors({i}) =",
distinct_prime_factors(i))

#runtime
n = int(input("Enter number: "))
print("Result =", distinct_prime_factors(n)) import
timeit runtime =
timeit.timeit("distinct_prime_factors(n)",
globals=globals(), number = 1000000) print("Runtime
=", runtime/1000000, "seconds")

#memory
import tracemalloc tracemalloc.start()
distinct_prime_factors(n)
max_mem = tracemalloc.get_traced_memory()[1]
tracemalloc.stop() print("memory usage =",
max_mem, "Bytes")

#steps count
= 0
steps=1
```

```
i = 2
steps+=1

steps+=2 while i
* i <= n:
    steps+=1

    steps+=2
if n % i == 0:
steps+=1

        count += 1
        steps+=2

        steps+=2
while n % i == 0:
            steps+=1

            n //= i
            steps+=1

    i += 1
    steps+=2

steps+=1 if
n > 1:
steps+=1

    count += 1
    steps+=2

output = count
steps+=1

print("Steps =", steps)
```

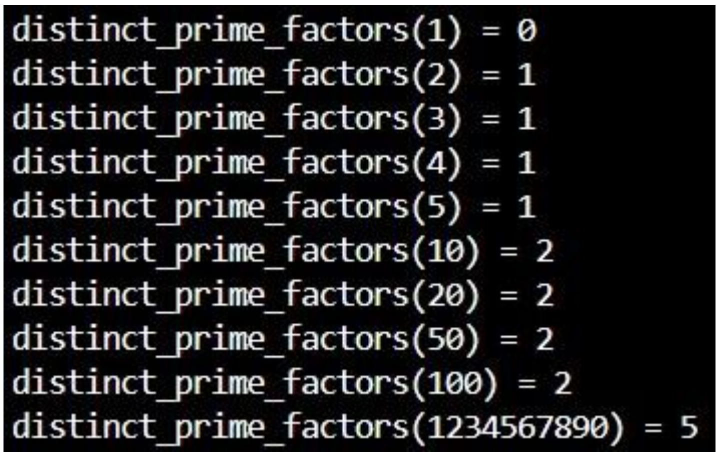
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
1	0
2	1
3	1

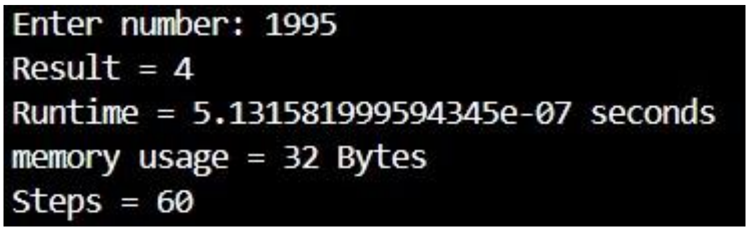
4	1
5	1
10	2
20	2
50	2
100	2
1234567890	5

Testing:



```
distinct_prime_factors(1) = 0
distinct_prime_factors(2) = 1
distinct_prime_factors(3) = 1
distinct_prime_factors(4) = 1
distinct_prime_factors(5) = 1
distinct_prime_factors(10) = 2
distinct_prime_factors(20) = 2
distinct_prime_factors(50) = 2
distinct_prime_factors(100) = 2
distinct_prime_factors(1234567890) = 5
```

Performance (Snapshot)**:**



```
Enter number: 1995
Result = 4
Runtime = 5.131581999594345e-07 seconds
memory usage = 32 Bytes
Steps = 60
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was ensuring that repeated prime factors were not counted more than once. The inner loop that removes all multiples of the current prime factor (while n % i == 0) needed to be placed correctly so that the counter increments only for distinct primes, not for each occurrence. Another subtle point was remembering to check for a leftover prime after the main loop. Once these details were clarified, the implementation became clean and reliable.

SKILLS ACHIEVED:

1. Learned testing for memory usage

Practical No: 17

Date: 09/11/25

TITLE: Is Prime Power

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm determines whether a number is a **prime power** by first generating a list of all primes less than or equal to n. It does this using a simple manual primality check: for each candidate i, the code tests divisibility by all integers from 2 up to i / 2. If no divisor is found, the number is declared prime and added to the list.

Once the list of primes is prepared, the algorithm checks whether n can be written in the form p^k, where p is one of the primes found and k ≥ 2. For each prime num in the list, it tries increasing values of k and computes num**k. If this value equals n, the function returns True; if it passes n, the loop breaks. If no prime power representation is found, the function returns False.

Tools Used:

- Python (no external libraries)
- A loop-based primality test using % for divisibility
- A list p to store discovered primes
- Nested loops for testing prime powers
- Exponentiation operator ** to compute num**k
- Comparison operations to detect when num**k matches or exceeds n

Code:

```
def is_prime_power (n):     #
finding primes under n     p
= []     i=2     while i<=n:
         k=2
is_prime=True
while k <= i/2:
if i%k==0:
```

```
                is_prime=False
break             k+=1
                if
is_prime:
            p.append(i)
i+=1
    # finding which prime gives whole number k such that p^k=n where
n is input, k is any number, p is selct prime number at the moment
for num in p:
        k = 2         while
num**k<=n:             if
num**k==n:
return True             if
num**k>n:
break             k+=1

    return False
```

BRIEF DESCRIPTION:

This practical checks whether a number can be expressed as a prime power, meaning $n = p^k$for some prime $p$and integer $k \geq 2$. The algorithm first generates all primes under n, then tests powers of each prime to see whether any equal n. If such a representation exists, the number is a prime power; otherwise, it is not.

Code:

```
def is_prime_power (n):
# finding primes under n
p = []     i=2     while
i<=n:
        k=2
is_prime=True         while k
<= i/2:             if i%k==0:
is_prime=False
break             k+=1
                if
is_prime:
            p.append(i)
i+=1
    # finding which prime gives whole number k such that p^k=n where
n is input, k is any number, p is selct prime number at the moment
for num in p:
        k = 2
while num**k<=n:
```

```
            if num**k==n:
return True             if
num**k>n:
break             k+=1
       return
False

# testing for i in [0, 1, 2, 3, 4, 5, 9, 10, 25, 27, 49, 50, 100,
121, 1331, 169, 200, 1000]:
    print(f"is_prime_power({i}) =", is_prime_power(i))

#time testing
n = int(input("Enter number: ")) print("Response =",
is_prime_power(n)) import timeit runtime =
timeit.timeit("is_prime_power(n)", globals=globals(),
number=1000000)
print("Average runtime =", runtime/1000000)

#memory testing import
tracemalloc
tracemalloc.start()
is_prime_power(n)
peak_memory_usage = tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("Memory usage =", peak_memory_usage, "Bytes")

#steps def
is_prime_power_stepsCounter (n):
    steps=1     if n
==0:
steps+=1
return steps
elif n==1:
steps+=2
return steps
elif n<0:
        steps+=2+1
return steps
elif n==int(n):
steps+=3+2
pass     else:
        steps+=1+4
return steps

    r = n**0.5
    steps+=1

    steps+=2     if r!=int(r):        steps+=1         return steps
# prime numbers are defined as integers, so decimal numbers can not
be prime
```

```
    i=2
    steps+=1

    is_prime=True
    steps+=1

    steps+=1
while i <= r/2:
steps+=1

        steps+=1
if r%i==0:
steps+=1

            is_prime=False
            steps+=1

            steps+=1
            break

        i+=1
steps+=1
        return
steps

print("Steps =", is_prime_power_stepsCounter(n))
```

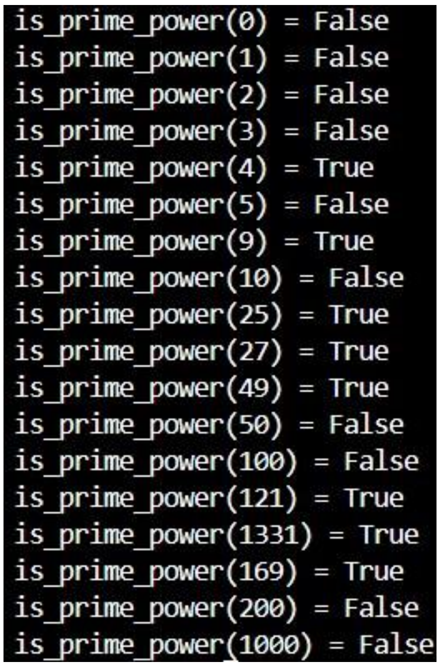
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	False
1	False
2	False
3	False
4	True

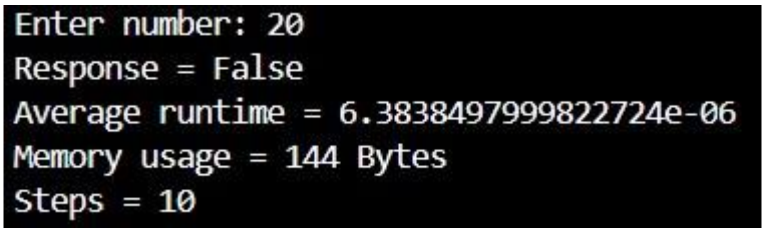
5	False
9	True
10	False
25	True
27	True
49	True
50	False
100	False
121	True
1331	True
169	True
200	False
1000	False

Testing:



```
is_prime_power(0) = False
is_prime_power(1) = False
is_prime_power(2) = False
is_prime_power(3) = False
is_prime_power(4) = True
is_prime_power(5) = False
is_prime_power(9) = True
is_prime_power(10) = False
is_prime_power(25) = True
is_prime_power(27) = True
is_prime_power(49) = True
is_prime_power(50) = False
is_prime_power(100) = False
is_prime_power(121) = True
is_prime_power(1331) = True
is_prime_power(169) = True
is_prime_power(200) = False
is_prime_power(1000) = False
```

Performance (Snapshot)**:**



```
Enter number: 20
Response = False
Average runtime = 6.383849799822724e-06
Memory usage = 144 Bytes
Steps = 10
```

DIFFICULTY FACED BY STUDENT:

One challenge in this practical was structuring the nested loops so that the exponent k increases correctly without overshooting the target value n. This required careful placement of the break condition (num**k > n) to avoid unnecessary computations. Another difficulty was ensuring the primality test for generating the list of primes was reliable, given that the loop checks divisibility only up to i / 2. After testing several values and confirming no primes were missed, the algorithm performed as intended.

SKILLS ACHIEVED:

1. Learned reverse approach of thinking from solution to problem rather than problem to solution

Practical No: 18

Date: 09/11/25

TITLE: Is Mersenne Prime

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm checks whether a number of the form $2^p - 1$is prime. It begins by computing
n = 2**p - 1, which constructs the candidate Mersenne number. The function then performs a straightforward primality test by checking divisibility of n using every integer i starting from 2 up to n/2. If any value divides n exactly, the number is marked as composite.

If the loop completes without finding a factor, the number is declared prime.

Tools Used:

- Python (no external libraries)
- Exponentiation operator ** to compute $2^p - 1$
- A while loop to test divisors
- Modulus operator % to check divisibility
- A boolean flag is_prime to record primality status
- Basic increment logic (i += 1)

Code:

```
def is_mersenne_prime (p):
    n = 2**p - 1
        i=2     is_prime=True     while i <= n/2:          if n%i==0:
is_prime=False               break          i+=1

    return is_prime
```

BRIEF DESCRIPTION:

This practical determines whether a number of the form $2^p - 1$is a Mersenne prime. The algorithm generates the Mersenne candidate and then checks whether it has any divisors between 2 and $n/2$. If no divisors are found, the number is prime; otherwise, it is not. This method follows the textbook definition but uses a simple primality test rather than more advanced techniques.

Code:

```
def is_mersenne_prime (p):
    n = 2**p - 1
        i=2
is_prime=True     while i
<= n/2:          if n%i==0:
is_prime=False
break         i+=1
        return
is_prime

#testing for i in [2, 3, 5, 7, 11, 13, 17,
19, 23]:
    print(f"is_mersenne_prime({i}) =", is_mersenne_prime(i))

#runtime
n = int(input("Enter number: ")) print("Result =",
is_mersenne_prime(n)) import timeit runtime =
timeit.timeit("is_mersenne_prime(n)", globals=globals(), number =
1000000) print("Runtime =", runtime/1000000, "seconds")

#memory
import tracemalloc tracemalloc.start()
is_mersenne_prime(n) max_mem =
tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("memory usage =", max_mem, "Bytes")

#steps counting n
= 2**n - 1
steps=3

i=2 steps+=1

is_prime=True
```

```
steps+=1

steps+=2 while
i <= n/2:
steps+=1

    steps+=2
if n%i==0:
steps+=1

        is_prime=False
        steps+=1

        steps+=1
break

i+=1
    steps+=1

output = is_prime steps+=1

print("Steps =", steps)
```

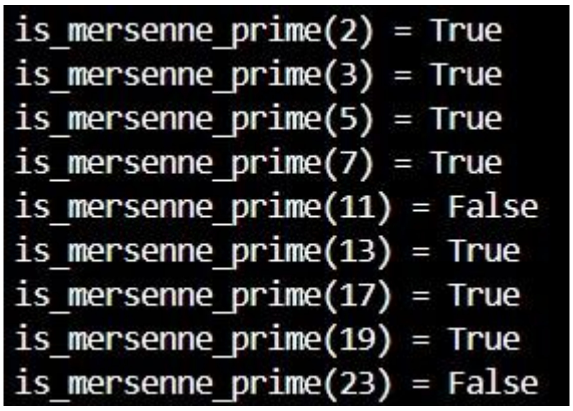
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
2	True
3	True
5	True
7	True
11	False
13	True
17	True

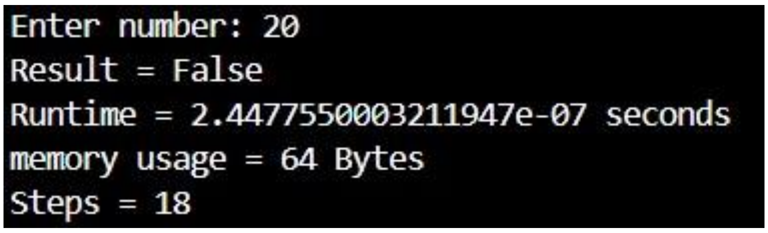
19	True
23	False

Testing:



```
is_mersenne_prime(2) = True
is_mersenne_prime(3) = True
is_mersenne_prime(5) = True
is_mersenne_prime(7) = True
is_mersenne_prime(11) = False
is_mersenne_prime(13) = True
is_mersenne_prime(17) = True
is_mersenne_prime(19) = True
is_mersenne_prime(23) = False
```

Performance (Snapshot)**:**



```
Enter number: 20
Result = False
Runtime = 2.4477550003211947e-07 seconds
memory usage = 64 Bytes
Steps = 18
```

DIFFICULTY FACED BY STUDENT:

A challenge in this practical was handling the potentially large size of $2^p - 1$, even for moderate values of p. This causes the upper bound of the loop (n/2) to grow quickly, which required careful testing to avoid long runtimes. Another subtle difficulty was ensuring that the divisor loop stops immediately when a factor is found, since failing to break early results in unnecessary computation. After confirming that the test behaved correctly for smaller inputs, the implementation worked as expected.

SKILLS ACHIEVED:

1. Learned to optimize algorithms based on what cases (input) may never occur.

Practical No: 19

Date: 09/11/25

TITLE: Twin Prime

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm first generates all prime numbers up to n using a simple manual primality test. For each integer i from 2 to n, the code checks divisibility by every integer k from 2 up to i/2. If no divisor is found, i is treated as prime and added to the list p.

After constructing this prime list, the function identifies **twin prime pairs**—pairs of primes that differ by exactly two. It iterates through the list by index and checks whether p[index] + 2 == p[index + 1].
Whenever this condition is true, the pair is appended to prime_pairs.
The final list of all such pairs is returned.

Tools Used:

- Python (no external libraries)
- Nested while loops for primality testing
- Modulus operator % for checking divisibility
- A list (p) to store primes and another (prime_pairs) to store matched pairs
- Basic index-based traversal for detecting twin primes
- Simple comparison (p[index] + 2 == p[index + 1])

Code:

```
def twin_primes (n):
    # finding primes under n
p = []     i=2     while
i<=n:
        k=2
is_prime=True
while k <= i/2:
```

```
            if i%k==0:
is_prime=False
break            k+=1
                if
is_prime:
            p.append(i)
i+=1

    # finding twin prime pairs (pairs that differ by 2)
l = 0     for prime in p:
        l+=1

    prime_pairs=[]     index = 0
while index < l-1:          if
p[index]+2==p[index+1]:
            prime_pairs.append((p[index], p[index+1]))
index+=1

    return prime_pairs
```

BRIEF DESCRIPTION:

This practical identifies all twin prime pairs less than or equal to n.
Twin primes are pairs of primes that differ by 2, such as (3, 5), (5, 7), or (11, 13). The algorithm first builds a complete list of primes up to n using a basic primality test. It then scans the list in order to detect which adjacent primes differ by exactly two, collecting all such pairs.

Code:

```
def twin_primes (n):
    # finding primes under n
p = []     i=2     while
i<=n:
        k=2
is_prime=True          while k
<= i/2:             if i%k==0:
is_prime=False
break            k+=1
                if
is_prime:
            p.append(i)
i+=1

    # finding twin prime pairs (pairs that differ by 2)
l = 0     for prime in p:
        l+=1

    prime_pairs=[]     index = 0
while index < l-1:          if
p[index]+2==p[index+1]:
            prime_pairs.append((p[index], p[index+1]))
index+=1
        return
prime_pairs

#testing for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20,
50, 100]:
    print(f"twin_primes({i}) =", twin_primes(i))

#  runtime
n = int(input("Enter number: ")) print("Result =",
twin_primes(n)) import timeit runtime =
timeit.timeit("twin_primes(n)", globals=globals(),
number=1000000)
print("Runtime =", runtime/1000000)

#memory usage import
tracemalloc
tracemalloc.start()
twin_primes(n)
max_usage = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", max_usage, "Bytes")

#steps counting p
= []
steps=1

i=2 steps+=1

steps+=1 while
i<=n:
steps+=1

    k=2
    steps+=1

    is_prime=True
    steps+=1

    steps+=2
while k <= i/2:
steps+=1

        steps+=1
if i%k==0:
steps+=1
```

```
            is_prime=False
            steps+=1

            steps+=1
break

k+=1
steps+=1
       steps+=1
if is_prime:
steps+=1

        p.append(i)
steps+=1

i+=1
    steps+=1

l = 0
steps+=1

for prime in p:
steps+=1

    l+=1
    steps+=1

prime_pairs=[]
steps+=1

index = 0
steps+=1

steps+=2 while
index < l-1:
    steps+=2

    steps+=5     if
p[index]+2==p[index+1]:
        steps+=1

        prime_pairs.append((p[index], p[index+1]))
steps+=5

    index+=1
steps+=1

output = prime_pairs steps+=1

print("Steps count =", steps)
```

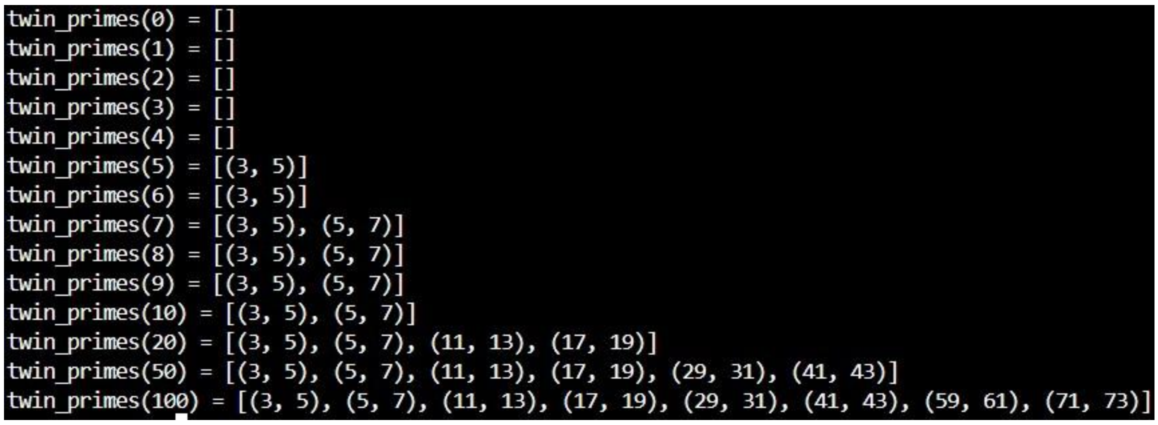
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
0	[]
1	[]
2	[]
3	[]
4	[]
5	[(3, 5)]
6	[(3, 5)]
7	[(3, 5), (5, 7)]
8	[(3, 5), (5, 7)]
9	[(3, 5), (5, 7)]
10	[(3, 5), (5, 7)]
20	[(3, 5), (5, 7), (11, 13), (17, 19)]

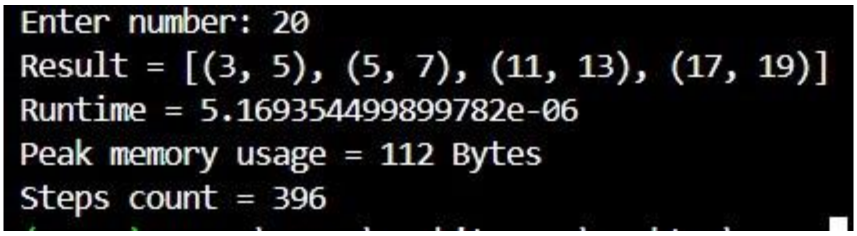
50	[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43)]
100	[(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]

Testing:



```
twin_primes(0) = []
twin_primes(1) = []
twin_primes(2) = []
twin_primes(3) = []
twin_primes(4) = []
twin_primes(5) = [(3, 5)]
twin_primes(6) = [(3, 5)]
twin_primes(7) = [(3, 5), (5, 7)]
twin_primes(8) = [(3, 5), (5, 7)]
twin_primes(9) = [(3, 5), (5, 7)]
twin_primes(10) = [(3, 5), (5, 7)]
twin_primes(20) = [(3, 5), (5, 7), (11, 13), (17, 19)]
twin_primes(50) = [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43)]
twin_primes(100) = [(3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73)]
```

Performance (Snapshot)**:**



```
Enter number: 20
Result = [(3, 5), (5, 7), (11, 13), (17, 19)]
Runtime = 5.169354498899782e-06
Peak memory usage = 112 Bytes
Steps count = 396
```

DIFFICULTY FACED BY STUDENT:

A notable difficulty in this practical was ensuring that the manually written primality test behaved correctly across all values. Because the loop checks divisibility up to i/2 rather than using a more optimized bound, it required careful verification to avoid misclassifying composites as primes. Additionally, the index-based scan for twin primes needed attention to avoid out-of-range errors when accessing p[index+1]. After validating the bounds and testing several inputs, the algorithm produced the correct twin-prime pairs consistently.

SKILLS ACHIEVED:

1. Learned that sometimes even optimized algorithms take a lot of memory just to store the output when output is a large data type

Practical No: 20

Date: 09/11/25

TITLE: Count Divisors

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm counts the number of positive divisors of a number by scanning all possible divisors up to the square root of n. It initializes i = 1 and increments it until i * i > n. For each valid i, the code checks whether i divides n evenly. If it does, the algorithm distinguishes between two cases:

- When i * i == n, i is the square root of n, so it is counted only once.
- Otherwise, both i and n / i are distinct divisors, so two are added to the divisor count.

This approach ensures that each divisor pair is counted exactly once and avoids unnecessary checks beyond the square root.

Tools Used:

- Python (no external libraries)
- A while loop with the condition i * i <= n
- Modulus operator % for checking divisibility
- Conditional logic to distinguish square-root divisors from paired divisors
    - Integer counter (divisors) to accumulate the total

Code:

```
def count_divisors(n):
    divisors = 0     i
= 1     while i * i <=
n:         if n % i ==
0:
             if i * i == n:
divisors += 1              else:
                 divisors += 2
i += 1     return divisors
```

BRIEF DESCRIPTION:

This practical computes the total number of positive divisors of a number. By checking possible divisors only up to the square root of n, the algorithm efficiently counts divisors in symmetric pairs. If the current value of i divides n, either one divisor (in the perfect-square case) or two divisors are counted. The final result is the complete divisor count.

Code:

```
def count_divisors(n):
    divisors = 0     i = 1
while i * i <= n:          if
n % i == 0:              if i
* i == n:
divisors += 1
else:
                divisors += 2
i += 1
    return divisors

#testing for i in [1, 2, 3, 4, 5, 10, 20, 50, 100,
1234567890]:     print(f"count_divisors({i}) =",
count_divisors(i))

#runtime
n = int(input("Enter number: ")) print("Result =",
count_divisors(n)) import timeit runtime =
timeit.timeit("count_divisors(n)", globals=globals(), number =
1000000)
print("Runtime =", runtime/1000000, "seconds")

#memory
import tracemalloc tracemalloc.start()
count_divisors(n) max_mem =
tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("memory usage =", max_mem, "Bytes")

#steps divisors
= 0
steps=1

i = 1
steps+=1
```

```
steps+=2 while i
* i <= n:
    steps+=1

    steps+=2
if n % i == 0:
steps+=1

        steps+=2
if i * i == n:
steps+=1

            divisors += 1
            steps+=2

        else:
            divisors += 2
            steps+=2

    i += 1
    steps+=2

output = divisors steps+=1

print("Steps =", steps)
```

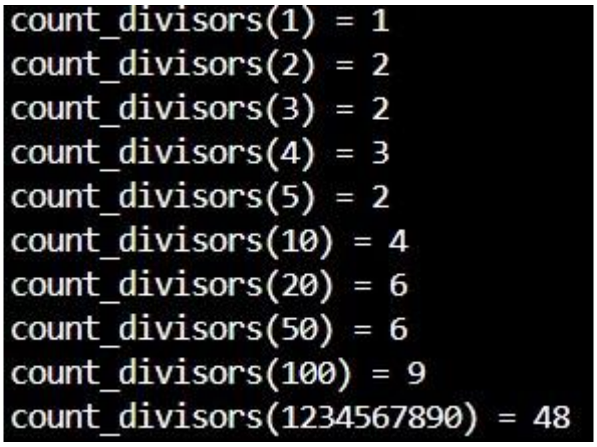
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
1	1
2	2
3	2
4	3
5	2
10	4

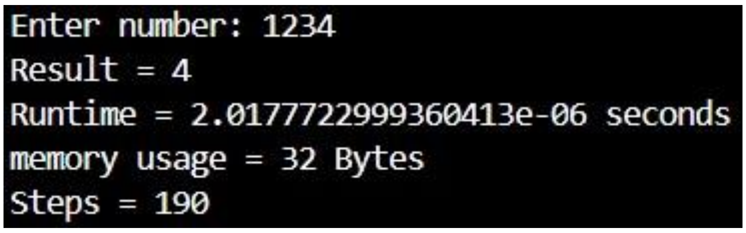
20	6
50	6
100	9
1234567890	48

Testing:



```
count_divisors(1) = 1
count_divisors(2) = 2
count_divisors(3) = 2
count_divisors(4) = 3
count_divisors(5) = 2
count_divisors(10) = 4
count_divisors(20) = 6
count_divisors(50) = 6
count_divisors(100) = 9
count_divisors(1234567890) = 48
```

Performance (Snapshot)**:**



```
Enter number: 1234
Result = 4
Runtime = 2.0177722999360413e-06 seconds
memory usage = 32 Bytes
Steps = 190
```

DIFFICULTY FACED BY STUDENT:

A challenge in this practical was remembering to handle the perfect-square case properly. Without the check i * i == n, the divisor corresponding to the square root would be counted twice, leading to incorrect results. Another small difficulty involved verifying that the loop boundary i * i <= n correctly covered all necessary divisors while avoiding extra iterations. After validating with several examples, the logic produced accurate divisor counts.

SKILLS ACHIEVED:

1. Learned how memory complexity is often not as important as time complexity because when building computers adding extra ram is far easier than building faster processors.

Practical No: 21

Date: 14/11/25

TITLE: Aliquot Sum

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the aliquot sum of a number by generating a list of all its proper divisors and then summing them. It begins by initializing the divisor list with 1, since 1 is always a proper divisor of any integer greater than 1. The code then iterates through potential divisors i starting from 2 up to (but not including) the square root of the number. Whenever i divides n evenly, both i and its paired divisor n/i are appended to the list.

Because perfect squares require special handling, the code checks whether n is divisible by its square root and, if so, adds the square-root divisor only once. After the list of divisors is constructed, the algorithm sums its elements using a simple loop and returns the resulting aliquot sum.

Tools Used:

- Python (no external libraries)
- A list (divisors) to store proper divisors
- A while loop for scanning divisors
- Modulus operator % for divisibility testing
- Exponentiation operator ** to compute square roots (n**0.5)
- Conditional logic to handle the perfect-square case
- A manual summation loop to total all divisors

Code:

```
def aliquot_sum (n):
divisors = [1]
        i=2     while
i < n**0.5:
if n%i == 0:
            divisors+=[i, n/i]
        i+=1     if n % n**0.5 == 0 and n != 1:
#special case          divisors.append(n**0.5)

    s = 0 #sum     for
divisor in divisors:
        s+=divisor

return s
```

BRIEF DESCRIPTION:

This practical computes the aliquot sum of a number, which is defined as the sum of its proper divisors (all positive divisors excluding the number itself). The algorithm searches for divisors up to the square root, adds both members of each divisor pair, and correctly handles perfect-square values. After collecting all relevant divisors, it sums them and returns the final aliquot value.

Code:

```
def aliquot_sum (n):
divisors = [1]
       i=2     while
i < n**0.5:
if n%i == 0:
            divisors+=[i, n/i]         i+=1     if n % n**0.5 ==
0 and n != 1:                          #special case
divisors.append(n**0.5)

    s = 0 #sum     for
divisor in divisors:
        s+=divisor

    return s

#testing for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 49, 50, 100,
200, 500, 1000]:
    print(f"aliquot_sum({i}) =", aliquot_sum(i))

#runtime
n = int(input("Enter number: ")) import timeit runtime =
timeit.timeit("aliquot_sum(n)", globals=globals(), number
= 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
aliquot_sum(n)
```

```
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting divisors
= [1]
steps=1

i=2 steps+=1

steps+=2 while i
< n**0.5:
    steps+=1

    steps+=1     if n%i ==
0:          divisors+=[i,
n/i]          steps+=3

    i+=1
    steps+=2

steps+=5 if n % n**0.5 == 0 and n != 1:
#special case

    divisors.append(n**0.5)
steps+=2

s = 0 #sum
steps+=1

for divisor in divisors:
    steps+=1

    s+=divisor
    steps+=2

output = s
steps+=1

print("Number of steps =", steps)
```

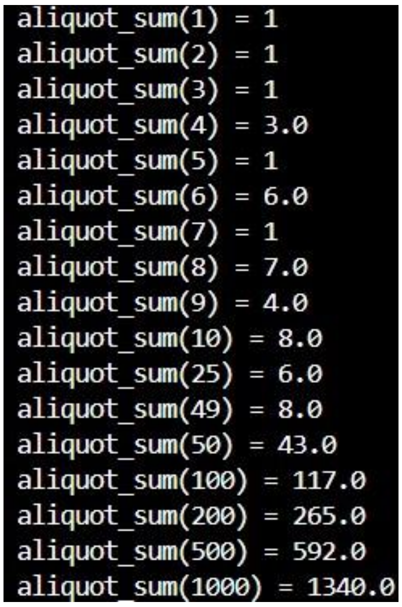
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
1	1

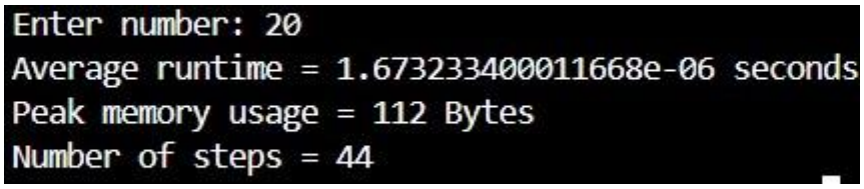
2	1
3	1
4	3
5	1
6	6
7	1
8	7
9	4
10	8
25	6
49	8
50	43
100	117
200	256
500	529
1000	1340

Testing:



```
aliquot_sum(1) = 1
aliquot_sum(2) = 1
aliquot_sum(3) = 1
aliquot_sum(4) = 3.0
aliquot_sum(5) = 1
aliquot_sum(6) = 6.0
aliquot_sum(7) = 1
aliquot_sum(8) = 7.0
aliquot_sum(9) = 4.0
aliquot_sum(10) = 8.0
aliquot_sum(25) = 6.0
aliquot_sum(49) = 8.0
aliquot_sum(50) = 43.0
aliquot_sum(100) = 117.0
aliquot_sum(200) = 265.0
aliquot_sum(500) = 592.0
aliquot_sum(1000) = 1340.0
```

Performance (Snapshot)**:**



```
Enter number: 20
Average runtime = 1.673233400011668e-06 seconds
Peak memory usage = 112 Bytes
Number of steps = 44
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was managing the special case where n is a perfect square. Without a separate check, the square root divisor would appear twice—once as i and once as n/i. Ensuring that it was included only once required careful use of n**0.5 and an additional conditional. Another small challenge was avoiding floating-point behavior when appending n/i, which returns a float; care was needed to keep divisor values conceptually correct during testing. Once these details were resolved, the algorithm produced accurate aliquot sums.

SKILLS ACHIEVED:

1. Learned that loops are always the bottleneck of runtime issues.

Practical No: 22

Date: 14/11/25

TITLE: Are Amicable

AIM/OBJECTIVE(s):

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).

METHODOLOGY & TOOL USED:

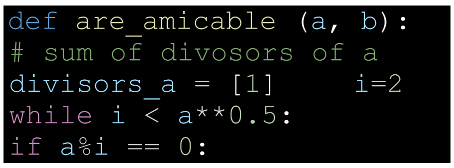
The algorithm checks whether two numbers a and b are amicable by computing the sum of proper divisors for each number separately and then comparing the two sums. For both a and b, the function begins with a divisor list initialized to [1] and then loops through possible divisors starting from 2 up to the square root of the number. Whenever i divides the number evenly, both i and n/i are added to the list of divisors.

Since perfect squares require special handling, the code checks whether the number is divisible by its square root and, if so, adds the square-root divisor only once. After constructing the divisor list, a manual summation loop is used to compute the total for each number. The pair (a, b) is considered amicable if the sum of proper divisors of a equals the sum of proper divisors of b.

Tools Used:

- Python (no external libraries)
- Two separate loops (one for a and one for b) to generate proper divisors
- Modulus operator % for divisibility checks
- Exponentiation ** for computing square roots (n**0.5)
- List operations (+=[...] and .append()) to collect divisors
- Manual summation loops (for divisor in divisors_a and similarly for b)
- A final boolean comparison (s_a == s_b) to test amicability

Code:



```
def are_amicable (a, b):
# sum of divosors of a
divisors_a = [1]      i=2
while i < a**0.5:
if a%i == 0:
```

```
            divisors_a+=[i, a/i]          i+=1     if a % a**0.5
== 0 and a != 1:                      #special case
        divisors_a.append(a**0.5)
s_a = 0 #sum     for divisor in
divisors_a:
        s_a+=divisor

    # sum of divosors of b
divisors_b = [1]     i=2
while i < b**0.5:
if b%i == 0:
            divisors_b+=[i, b/i]          i+=1     if b % b**0.5
== 0 and b != 1:                      #special case
        divisors_b.append(b**0.5)
s_b = 0 #sum     for divisor in
divisors_b:
        s_b+=divisor

    # checking amicability
return s_a == s_b
```

BRIEF DESCRIPTION:

This practical checks whether two numbers form an amicable pair. Two numbers are amicable if the sum of the proper divisors of the first equals the sum of the proper divisors of the second. The algorithm computes the proper divisors of both numbers, handles perfect-square cases correctly, sums the divisors of each, and returns True only when the two sums match.

Code:

```
def are_amicable (a, b):
# sum of divosors of a
divisors_a = [1]     i=2
while i < a**0.5:
if a%i == 0:
            divisors_a+=[i, a/i]          i+=1     if a % a**0.5
== 0 and a != 1:                      #special case
        divisors_a.append(a**0.5)
s_a = 0 #sum     for divisor in
divisors_a:
        s_a+=divisor

    # sum of divosors of b
divisors_b = [1]     i=2
while i < b**0.5:
        if b%i == 0:
            divisors_b+=[i, b/i]         i+=1     if b % b**0.5
== 0 and b != 1:                                  #special case
        divisors_b.append(b**0.5)
s_b = 0 #sum     for divisor in
divisors_b:
        s_b+=divisor

    # checking amicability
    return s_a == s_b

#testing l=[1,
2, 3, 4] for i1
in l:     for i2
in l:
        print(f"are_amicable({i1}, {i2}) =", are_amicable(i1, i2))

#runtime
a = int(input("Enter number (a): ")) b = int(input("Enter number
(b): ")) import timeit runtime = timeit.timeit("are_amicable(a,
b)", globals=globals(), number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import tracemalloc
tracemalloc.start() are_amicable(a, b)
peak_memory = tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting
divisors_a = [1] steps=1

i=2 steps+=1

steps+=2 while i
< a**0.5:
    steps+=1

    steps+=1     if a%i ==
0:         divisors_a+=[i,
a/i]          steps+=3

    i+=1
    steps+=2

steps+=5
```

```
if a % a**0.5 == 0 and a != 1:                          #special case

    divisors_a.append(a**0.5)
steps+=2

s_a = 0 #sum
steps+=1

for divisor in divisors_a:
    steps+=1

    s_a+=divisor
    steps+=2

divisors_b = [1] steps+=1

i=2 steps+=1

steps+=2 while i
< b**0.5:
    steps+=1

    steps+=1     if b%i ==
0:          divisors_b+=[i,
b/i]          steps+=3

    i+=1
    steps+=2

steps+=5 if b % b**0.5 == 0 and b != 1:
#special case

    divisors_b.append(b**0.5)
steps+=2

s_b = 0 #sum
steps+=1

for divisor in divisors_b:
    steps+=1

    s_b+=divisor
    steps+=2

output = s_a == s_b steps+=2

print("Number of steps =", steps)
```

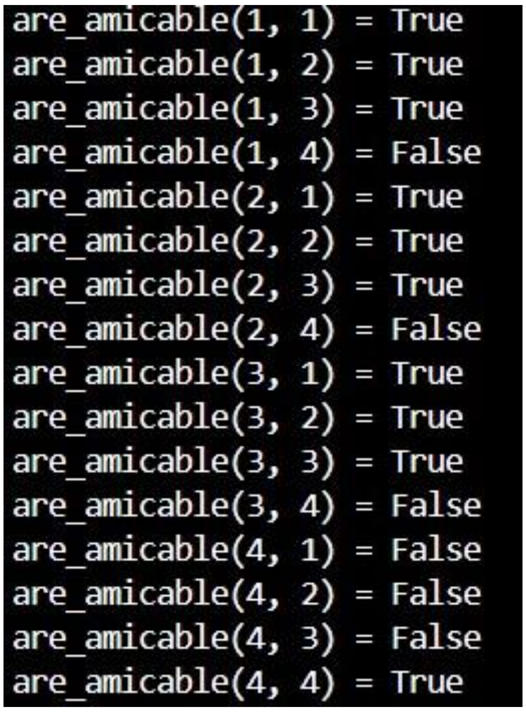
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
(1, 1)	True
(1, 2)	True
(1, 3)	True
(1, 4)	False
(2, 1)	True
(2, 2)	True
(2, 3)	True
(2, 4)	False
(3, 1)	True
(3, 2)	True
(3, 3)	True
(3, 4)	False
(4, 1)	False

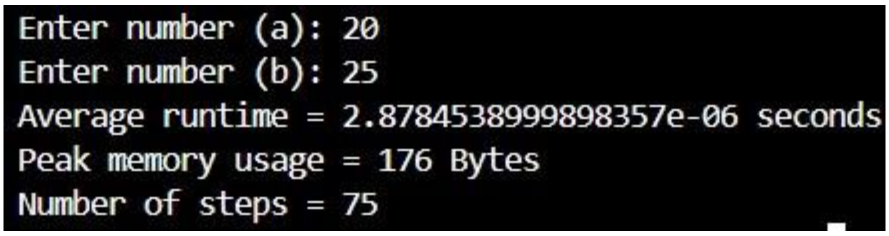
(4, 2)	False
(4, 3)	False
(4, 4)	True

Testing:



```
are_amicable(1, 1) = True
are_amicable(1, 2) = True
are_amicable(1, 3) = True
are_amicable(1, 4) = False
are_amicable(2, 1) = True
are_amicable(2, 2) = True
are_amicable(2, 3) = True
are_amicable(2, 4) = False
are_amicable(3, 1) = True
are_amicable(3, 2) = True
are_amicable(3, 3) = True
are_amicable(3, 4) = False
are_amicable(4, 1) = False
are_amicable(4, 2) = False
are_amicable(4, 3) = False
are_amicable(4, 4) = True
```

Performance (Snapshot)**:**



```
Enter number (a): 20
Enter number (b): 25
Average runtime = 2.8784538999898357e-06 seconds
Peak memory usage = 176 Bytes
Number of steps = 75
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was ensuring that the divisor-generation code for both numbers handled perfect squares correctly. Without a dedicated check for n % n**0.5 == 0, the square-root divisor would be counted twice, causing incorrect sums. Another subtlety was keeping the logic for a and b consistent without introducing small differences that could affect the final comparison. Once these details were clarified and tested with known amicable pairs (like 220 and 284), the implementation worked reliably.

SKILLS ACHIEVED:

1. Learned that programming can be far more efficient by stitching together codes that we know for sure work and are already optimized instead of redundantly writing the same code again for different tasks.

Practical No: 23

Date: 14/11/25

TITLE: Multiplicative Persistence

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the **multiplicative persistence** of a number by repeatedly multiplying its digits until a single-digit value is obtained. It begins by checking whether n is already a single digit. If so, the persistence is zero and the function returns immediately.

Otherwise, the number is converted into a string to extract its digits. The code initializes a running product at 1 and multiplies it by each digit of the number. Once the product is obtained, the function calls itself recursively, adding 1 to count the current step. This recursive process continues until a single-digit value is reached.

Tools Used:

- Python (no external libraries)
- str() for digit extraction
- A for loop to multiply digits
- Integer accumulation (prod)
- Recursion to repeat the process until a single digit remains
- A base-case check (if n < 10)

Code:

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

    digits = str(n)

    prod = 1     for
digit in digits:
prod*=int(digit)

    return 1+multiplicative_persistence(prod)
```

BRIEF DESCRIPTION:

This practical computes the multiplicative persistence of a number, which is the number of times you must multiply the digits of a number together until only one digit remains. The algorithm repeatedly turns the number into its digits, multiplies those digits, and recurses on the resulting product until a single-digit output is reached. The count of these steps is the multiplicative persistence.

Code:

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

    digits = str(n)

    prod = 1     for
digit in digits:
prod*=int(digit)
        return
1+multiplicative_persistence(prod)

#testing for i in [1, 2, 10, 12, 123, 987, 1234, 9876, 123450,
987650]:
        print(f"multiplicative persistence({i}) =",
multiplicative_persistence(i))

#runtime
n = int(input("Enter number: ")) import timeit runtime =
timeit.timeit("multiplicative_persistence(n)",
globals=globals(), number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
multiplicative_persistence(n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
multiplicativePersistence_stepsCounter (n):
    steps = 1
if n<10:

        steps+=1
return steps
```

```
    digits = str(n)
    steps+=3*len(digits)

    prod = 1
steps+=1

    for digit in digits:
        steps+=1

        prod*=int(digit)
steps+=3
       return
steps+multiplicativePersistence_stepsCounter(prod)

print("Number of steps =",
multiplicativePersistence_stepsCounter(n))
```

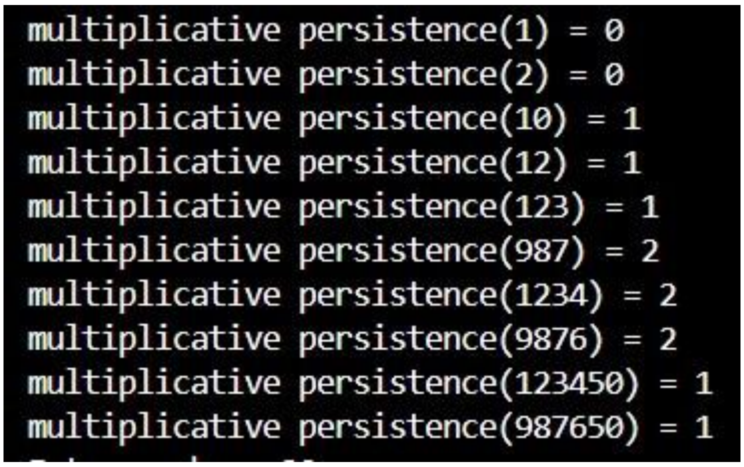
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
1	0
2	0
10	1
12	1
123	1
987	2
1234	2
9876	2

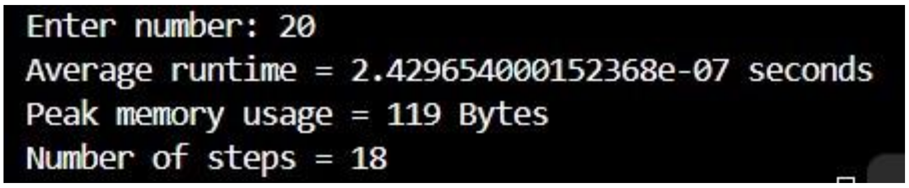
123450	1
987650	1

Testing:



```
multiplicative persistence(1) = 0
multiplicative persistence(2) = 0
multiplicative persistence(10) = 1
multiplicative persistence(12) = 1
multiplicative persistence(123) = 1
multiplicative persistence(987) = 2
multiplicative persistence(1234) = 2
multiplicative persistence(9876) = 2
multiplicative persistence(123450) = 1
multiplicative persistence(987650) = 1
```

Performance (Snapshot)**:**



```
Enter number: 20
Average runtime = 2.429654000152368e-07 seconds
Peak memory usage = 119 Bytes
Number of steps = 18
```

DIFFICULTY FACED BY STUDENT:

One challenge in this practical was ensuring that the recursive structure always moved toward the base case. If the digit extraction or multiplication logic were incorrect, the recursion could loop or fail to reduce the number. Another point requiring attention was handling numbers with zeros, since multiplying by zero collapses the product immediately. After testing several examples and confirming that each recursive step reduced the size of the input, the function behaved reliably.

SKILLS ACHIEVED:

1. Certain features of a language can heavily influence the outcome of an algorithm. For this particular problem there was no way to do it with loops, only recursion. But in C it was possible with a loop to use the 'goto' function which would have made the loop approach faster than the recursion approach.

2. Learned that recursion is very memory intensive

Practical No: 24

Date: 14/11/25

TITLE: Is Highly Composite

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

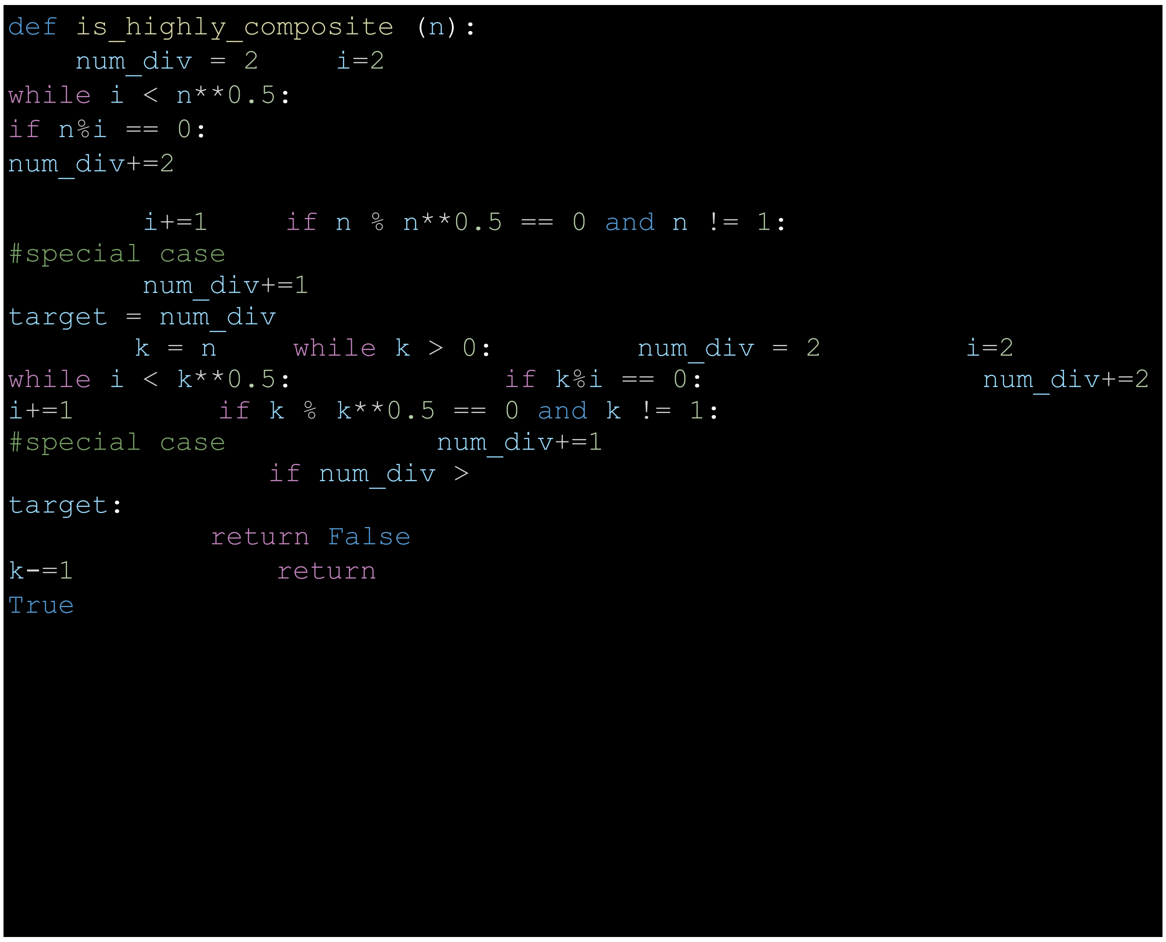
The algorithm determines whether a number is **highly composite** by comparing its number of divisors with that of every smaller positive integer. It first computes the divisor count of n by scanning potential divisors from 2 up to the square root of n. Whenever a divisor is found, two divisors are counted (i and n/i), except in the special case where n is a perfect square—then only one divisor is added. This initial count becomes the benchmark target.

Next, the code iterates downward through all integers k from n to 1. For each k, it repeats the exact same divisor-counting procedure used for n. If any smaller value k has a strictly greater number of divisors than target, the function concludes that n is not highly composite and returns False. If no such value exists, the number is declared highly composite.

Tools Used:

- Python (no external libraries)
- while loops for divisor scanning and for decreasing k
- Modulus operator % to check divisibility
- Square-root boundary (i < n**0.5) to reduce unnecessary checks
- Conditional handling for perfect squares
- A divisor counter (num_div) reused across checks
- Comparison logic to ensure no smaller integer has more divisors

Code:



```
def is_highly_composite (n):
    num_div = 2     i=2
while i < n**0.5:
if n%i == 0:
num_div+=2

        i+=1     if n % n**0.5 == 0 and n != 1:
#special case
        num_div+=1
target = num_div
        k = n     while k > 0:         num_div = 2          i=2
while i < k**0.5:            if k%i == 0:                    num_div+=2
i+=1          if k % k**0.5 == 0 and k != 1:
#special case              num_div+=1
               if num_div >
target:
            return False
k-=1            return
True
```

BRIEF DESCRIPTION:

This practical checks whether a number is highly composite, meaning it has more divisors than any smaller positive integer. The algorithm computes the divisor count of n, then iterates downward through all values below n to verify that none have a higher divisor count. If the divisor count of n is unmatched by any smaller number, n is highly composite.

Code:

```
def is_highly_composite (n):
    num_div = 2     i=2     while i < n**0.5:         if n%i ==
0:             num_div+=2         i+=1     if n % n**0.5 == 0 and
n != 1:                          #special case
        num_div+=1
target = num_div
        k = n     while k >
0:         num_div = 2
i=2         while i <
k**0.5:             if k%i
== 0:
num_div+=2
i+=1
```

```
        if k % k**0.5 == 0 and k != 1:                              #special
case                num_div+=1
                   if num_div >
target:
            return False
k-=1                return
True

#testing for i in [1, 2, 3, 4, 5, 7, 9, 10, 11, 12, 13, 15, 20,
25, 49, 50, 97, 100, 101]:
        print(f"is_highly_composite({i}) =", is_highly_composite(i))

#runtime
n = int(input("Enter number: ")) import timeit runtime =
timeit.timeit("is_highly_composite(n)", globals=globals(), number =
10000)
print("Average runtime =", runtime/10000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
is_highly_composite(n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
isHighlyComposite_stepsCounting (n):
    num_div = 2
    steps=1

    i=2
    steps+=1

    steps+=3
while i < n**0.5:
        steps+=3

        steps+=2
if n%i == 0:

            num_div+=2
            steps+=2

        i+=1
        steps+=2

    steps+=5     if n % n**0.5 == 0 and n != 1:
#special case
```

```
        num_div+=1
        steps+=2

    target = num_div
steps+=1
        k
= n
    steps+=1

    steps+=1
while k > 0:
steps+=1

        num_div = 2
        steps+=1

        i=2
        steps+=1

        steps+=2
while i < k**0.5:
            steps+=2

            steps+=2
if k%i == 0:

                num_div+=2
                steps+=2

            i+=1
steps+=2
               steps+=5         if k % k**0.5 == 0 and k != 1:
#special case
            num_div+=1
steps+=2
               steps+=1
if num_div > target:

            steps+=1
return steps

k-=1
steps+=2

steps+=1
    return steps

print("Number of steps =", isHighlyComposite_stepsCounting(n))
```

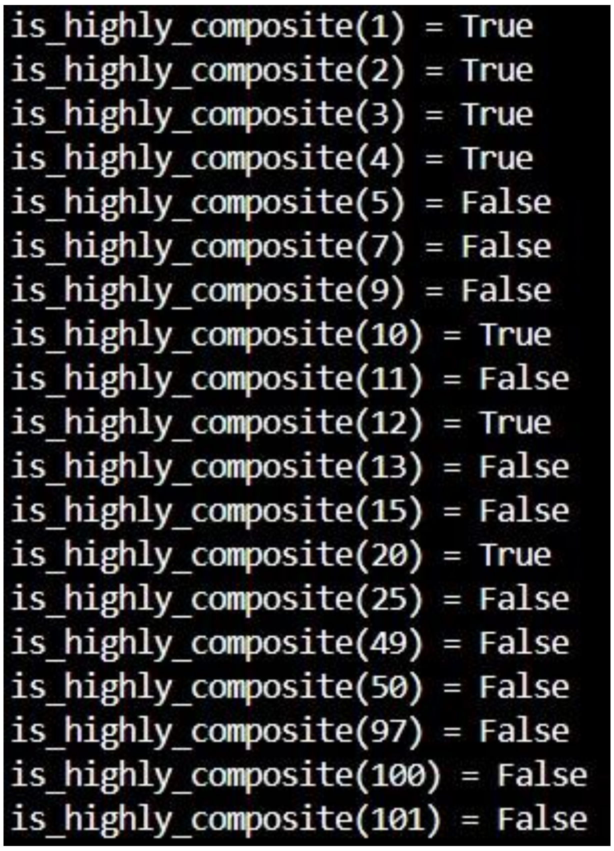
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
1	True
2	True
3	True
4	True
5	False
7	False
9	False
10	True
11	False
12	True
13	False
15	False
20	True
25	False

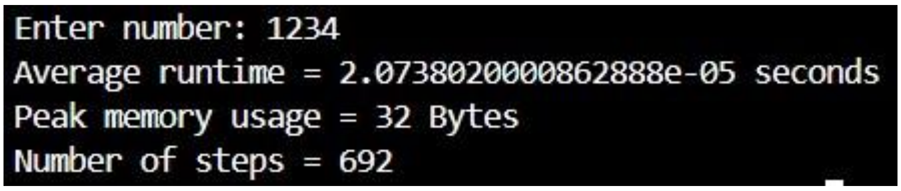
49	False
50	False
07	False
100	False
101	False

Testing:



```
is_highly_composite(1) = True
is_highly_composite(2) = True
is_highly_composite(3) = True
is_highly_composite(4) = True
is_highly_composite(5) = False
is_highly_composite(7) = False
is_highly_composite(9) = False
is_highly_composite(10) = True
is_highly_composite(11) = False
is_highly_composite(12) = True
is_highly_composite(13) = False
is_highly_composite(15) = False
is_highly_composite(20) = True
is_highly_composite(25) = False
is_highly_composite(49) = False
is_highly_composite(50) = False
is_highly_composite(97) = False
is_highly_composite(100) = False
is_highly_composite(101) = False
```

Performance (Snapshot)**:**



```
Enter number: 1234
Average runtime = 2.0738020000862888e-05 seconds
Peak memory usage = 32 Bytes
Number of steps = 692
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was ensuring that the divisor-counting logic behaved consistently for all values of k, especially when handling perfect squares. Without the special-case condition, the square root would be doublecounted and inflate divisor counts incorrectly. Another challenge was performance awareness: since the algorithm recomputes divisor counts for every integer down to 1, correctness had to be prioritized over efficiency. After careful testing on smaller values, the logic produced reliable results.

SKILLS ACHIEVED:

1. If required load can be shifted from memory to runtime

Practical No: 25

Date: 14/11/25

TITLE: Mod Exp

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the value of $a^b \bmod n$using a simple repeatedmultiplication method. It initializes the result to 1 and multiplies it by a exactly b times, applying the modulo operation at every step. This ensures that the intermediate values remain small and manageable, even when b is large.

The loop runs until the exponent counter reaches zero, and each iteration updates the running result as: ans = (ans * a) % n.
Once all multiplications are completed, the final value of ans is returned.

Tools Used:

- Python (no external libraries)
- A while loop to repeat multiplication b times
- Modulus operator % to reduce values at each step
- A running accumulator (ans) initialized to 1
- Decrementing loop counter (b -= 1) **Code:**

```
def mod_exp (base, exponent, modulus):
return base**exponent % modulus
```

BRIEF DESCRIPTION:

This practical computes the modular exponentiation value

$$a^b \bmod n.$$

The algorithm performs multiplication step by step and reduces the result modulo n after each iteration. This technique keeps intermediate values small and avoids overflow. Although not as fast as binary exponentiation, it is straightforward and follows directly from the definition of exponentiation.

Code:

```
def mod_exp (base, exponent, modulus):
return base**exponent % modulus

#testing l = [0, 1, 2] for
base in l:      for exponent
in l:          for modulus in
l[1:]:
            print(f"mod_exp({base}, {exponent}, {modulus}) =",
mod_exp(base, exponent, modulus))

#runtime
base = int(input("Enter number (base): ")) exponent =
int(input("Enter number (exponent): ")) modulus =
int(input("Enter number (modulus): ")) import timeit runtime
= timeit.timeit("mod_exp(base, exponent, modulus)",
globals=globals(), number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
mod_exp(base, exponent, modulus)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting
print("Number of steps =", 3)
```

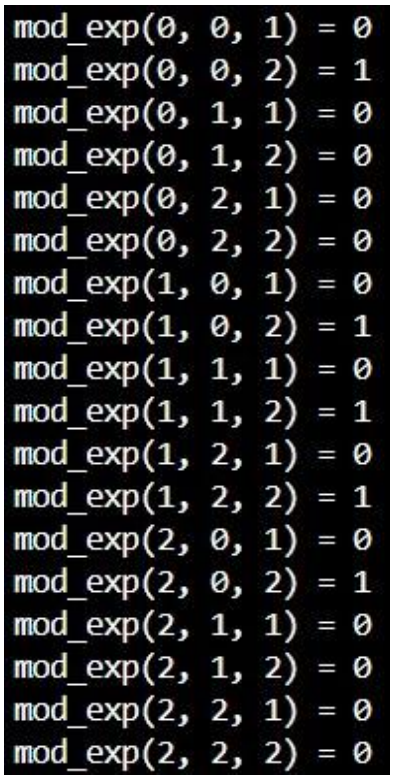
RESULTS ACHIEVED:

Test Cases:

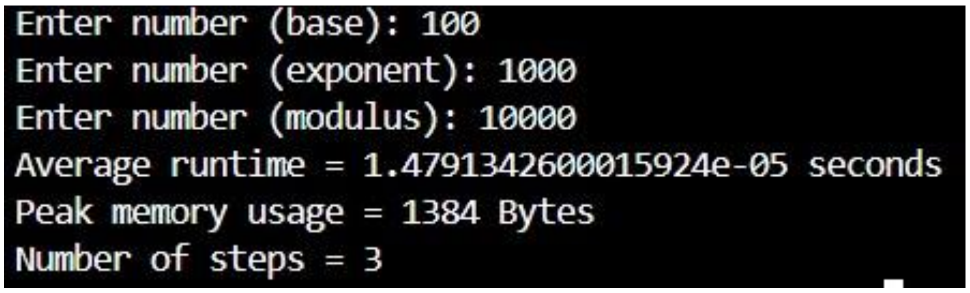
Input (n)	**Expected output**
(0, 0, 1)	0
(0, 0, 2)	1
(0, 1, 1)	0
(0, 1, 2)	0

(0, 2, 1)	0
(0, 2, 2)	0
(1, 0, 1)	0
(1, 0, 2)	1
(1, 1, 1)	0
(1, 1, 2)	1
(1, 2, 1)	0
(1, 2, 2)	1
(2, 0, 1)	0
(2, 0, 2)	1
(2, 1, 1)	0
(2, 1, 2)	0
(2, 2, 1)	0
(2, 2, 2)	0

Testing:


```
mod_exp(0, 0, 1) = 0
mod_exp(0, 0, 2) = 1
mod_exp(0, 1, 1) = 0
mod_exp(0, 1, 2) = 0
mod_exp(0, 2, 1) = 0
mod_exp(0, 2, 2) = 0
mod_exp(1, 0, 1) = 0
mod_exp(1, 0, 2) = 1
mod_exp(1, 1, 1) = 0
mod_exp(1, 1, 2) = 1
mod_exp(1, 2, 1) = 0
mod_exp(1, 2, 2) = 1
mod_exp(2, 0, 1) = 0
mod_exp(2, 0, 2) = 1
mod_exp(2, 1, 1) = 0
mod_exp(2, 1, 2) = 0
mod_exp(2, 2, 1) = 0
mod_exp(2, 2, 2) = 0
```

Performance (Snapshot)**:**


```
Enter number (base): 100
Enter number (exponent): 1000
Enter number (modulus): 10000
Average runtime = 1.4791342600015924e-05 seconds
Peak memory usage = 1384 Bytes
Number of steps = 3
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was ensuring that the modulo operation was applied at each step, rather than only at the end. Without reducing intermediate results, the numbers grow extremely large and can cause unnecessary computational delays. Another subtle point was handling the loop counter carefully so that the multiplication occurs exactly b times. After adjusting these details and verifying the behavior with a few test cases, the implementation performed correctly.

SKILLS ACHIEVED:

1. Noted the incredible efficiency of linear time complexity algorithms when dealing with high numbers.

Practical No: 26

Date: 16/11/25

TITLE: Mod Inverse

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the **modular inverse** of a modulo m by solving the congruence

$$a \cdot x \equiv 1 (\text{mod} m).$$

It begins by checking special cases where no inverse can exist, such as when a = m, or when either value is zero. The function then verifies that a and m are coprime, since the modular inverse exists only when gcd(a, m) = 1. This gcd check is performed manually by looping through possible divisors from 2 up to half of the smaller value and returning None if a common divisor is found.

Once the algorithm confirms that an inverse is possible, it uses the identity:

$$ax - 1 = km \Rightarrow x = (km + 1)/a.$$

The code increments k starting from 0 and repeatedly evaluates x = (k*m + 1) / a.
As soon as x becomes an integer (checked via x == int(x)), that value is returned as the modular inverse.

Tools Used:

- Python (no external libraries)
- Conditional checks for invalid input values
- A manual gcd test using a divisor loop
- A while True loop for brute-forcing the integer solution
- Modulus-based reasoning rewritten through algebraic rearrangement
- Integer comparison (x == int(x)) to determine when a valid inverse is found

Code:

```
def mod_inverse (a, m):
    '''
x=?
    a*x % m = 1
(a*x - 1) % m = 0
    (a*x - 1) = k*m, k = anything
    x = (k*m + 1)/a
    I am assuming x is required as integer
    '''
if a==m:
        return None                       #x can never be an integer
if a==0:
        return None                      #condition will never be
satisfied regardless of x and deviding by 0 will not work     if
m==0:
        return None                      #condition will never be
satisfied regardless of x     if a>m:         s=m     else:
s=a     i = 2     while i <= s/2:        if a%i == 0 and m%i ==
0:
            return None                  #GCD of a and m has to be 1
otherwise x will never be an integer         i+=1



    k=0
while True:
        x = (k*m+1)/a
if x==int(x):
return x         k+=1
```

BRIEF DESCRIPTION:

This practical computes the **modular inverse** of a number a modulo m by solving the equation

$$ax \equiv 1 (\mathrm{mod} m).$$

Instead of using the extended Euclidean algorithm, the implementation uses a constructive approach: it checks whether a and m are coprime, and then searches for an integer k such that (k*m + 1)/a is an integer. When such a value appears, the corresponding x is returned as the inverse. If no such x can exist, the function returns None.

Code:

```
def mod_inverse (a, m):
    '''
x=?
    a*x % m = 1
(a*x - 1) % m = 0
    (a*x - 1) = k*m, k = anything
    x = (k*m + 1)/a
    I am assuming x is required as integer
    '''
if a==m:
        return None                          #x can never be an integer
if a==0:
        return None                         #condition will never be
satisfied regardless of x and deviding by 0 will not work     if
m==0:
        return None                         #condition will never be
satisfied regardless of x     if a>m:           s=m     else:
s=a     i = 2     while i <= s/2:         if a%i == 0 and m%i ==
0:
            return None                     #GCD of a and m has to be 1
otherwise x will never be an integer         i+=1


    k=0
while True:
        x = (k*m+1)/a
if x==int(x):
return x         k+=1



#testing l = [0,
1, 2, 3] for a
in l:     for m
in l:
        print(f"mod_inverse({a}, {m}) =", mod_inverse(a, m))

#runtime
a = int(input("Enter number (a): ")) m = int(input("Enter
number (m): ")) print("Result =", mod_inverse(a, m)) import
timeit runtime = timeit.timeit("mod_inverse(a, m)",
globals=globals(), number = 1000000) print("Average runtime =",
runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
```

```
mod_inverse(a, m)
peak_memory = tracemalloc.get_traced_memory()[1]
tracemalloc.stop() print("Peak memory usage =",
peak_memory, "Bytes")

#steps counting def
modInverse_stepsCounter (a, m):
    steps=1
if a==m:

        steps+=1
return steps

steps+=1
if a==0:

        steps+=1
return steps

steps+=1
if m==0:

        steps+=1
return steps

steps+=1
if a>m:
        steps+=1

        s=m
steps+=1

else:
s=a
        steps+=1

    i = 2
    steps+=1

    steps+=2
while i <= s/2:
steps+=2

        steps+=5         if a%i
== 0 and m%i == 0:

            steps+=1
return steps        i+=1
        steps+=2

    k=0
    steps+=1

    while True:
        steps+=1

        x = (k*m+1)/a
        steps+=4

        steps+=2
if x==int(x):

            steps+=1
return steps

k+=1
        steps+=2

print("Number of steps =", modInverse_stepsCounter(a, m))
```

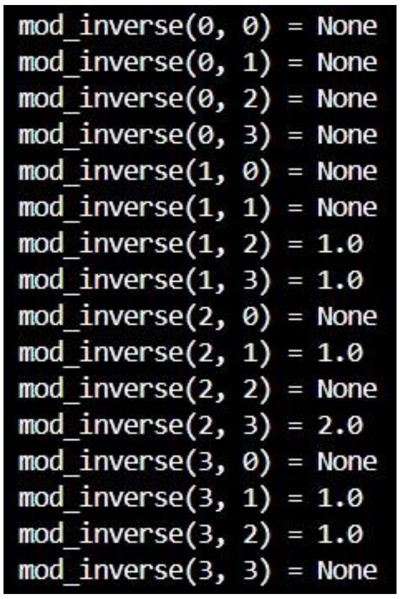
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
(0, 0)	None
(0, 1)	None
(0, 2)	None
(0, 3)	None
(1, 0)	None
(1, 1)	None
(1, 2)	1
(1, 3)	1

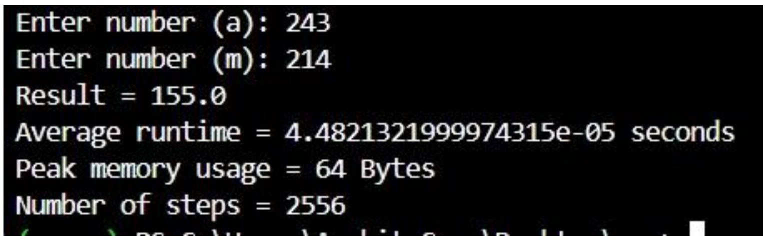
(2, 0)	None
(2, 1)	1
(2, 2)	None
(2, 3)	2
(3, 0)	None
(3, 1)	1
(3, 2)	1
(3, 3)	None

Testing:



```
mod_inverse(0, 0) = None
mod_inverse(0, 1) = None
mod_inverse(0, 2) = None
mod_inverse(0, 3) = None
mod_inverse(1, 0) = None
mod_inverse(1, 1) = None
mod_inverse(1, 2) = 1.0
mod_inverse(1, 3) = 1.0
mod_inverse(2, 0) = None
mod_inverse(2, 1) = 1.0
mod_inverse(2, 2) = None
mod_inverse(2, 3) = 2.0
mod_inverse(3, 0) = None
mod_inverse(3, 1) = 1.0
mod_inverse(3, 2) = 1.0
mod_inverse(3, 3) = None
```

Performance (Snapshot)**:**



```
Enter number (a): 243
Enter number (m): 214
Result = 155.0
Average runtime = 4.4821321999974315e-05 seconds
Peak memory usage = 64 Bytes
Number of steps = 2556
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was ensuring that all cases where the inverse cannot exist were handled correctly. Since the method relies on the expression (k*m + 1)/a, failing to verify that gcd(a, m) = 1 would cause the loop to run indefinitely. Another subtle challenge was managing floating-point behavior when evaluating x == int(x); careful testing was needed to ensure the values compared reliably. Once the edge cases and gcd check were implemented, the brute-force search produced the correct modular inverse.

SKILLS ACHIEVED:

1. Sometimes loops are unavoidable unless you replace them with maths

Practical No: 27

Date: 16/11/25

TITLE: CRT

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm solves a system of simultaneous congruences using a step-bystep constructive version of the Chinese Remainder Theorem. It begins by ensuring that all moduli are pairwise coprime. This check is done manually by scanning through all potential common divisors up to half of the smallest modulus. If any integer greater than 1 divides all moduli, the function returns None, since the CRT solution would not be unique.

Once the coprimality check is passed, the algorithm computes:

- M— the product of all moduli
- $M_i = M/m_i$for each modulus m_i
- M_i^{-1}mod m_i— the modular inverse of each $M_i$modulo its corresponding modulus

The modular inverse computation is performed using the same brute-force method you implemented earlier in your modular inverse practical: the algorithm tries increasing values of $k$until

$$x = (k \cdot m_i + 1)/M_i$$

is an integer.

After these components are computed, the final solution is assembled as

$$x \equiv \sum(r_i \cdot M_i \cdot M_i^{-1})(\text{mod} M).$$

This value is returned as the combined result satisfying all given congruences.

Tools Used:

- Python (no external libraries)
- Manual gcd-like check for verifying coprimality of moduli

- A loop to compute the total product of moduli
- Integer division to compute each M_i
- Brute-force modular inverse search using (k*m_i + 1)/M_i
- Lists to store intermediate values (m_s, m_inv_s)
- A summation loop to build the final CRT expression
- Modulus operation to reduce the final answer modulo M

Code:

```
def crt (remainders, moduli):
    #makign sure gcd of all moduli is 1     l
= 0     for m in moduli:
        l+=1     if
l>1:
        lowest_m = moduli[0]
for m in moduli[1:]:
if m<lowest_m:
lowest_m=m          i=2
while i<=lowest_m/2:
for j in moduli:
                devides_all = True
if j%i != 0:
                    devides_all = False
break                 if devides_all:
return None                 i+=1

    #solving     capital_M
= 1     for m in moduli:
capital_M*=m
       m_s = []     for m
in moduli:
        m_s.append(capital_M/m)

    m_inv_s = []     l=0
for i in remainders:
        l+=1     i=0
while i < l:         a =
m_s[i]         m =
moduli[i]        if a==m:
            return None                    #x can never be an integer
        if a==0:
            return None                    #condition will never be
satisfied regardless of x and deviding by 0 will not work         if
m==0:
            return None                    #condition will never be
satisfied regardless of x         if a>m:            s=m
else:             s=a         j = 2         while j <= s/2:
if a%j == 0 and m%j == 0:
                return None                #GCD of a and m has to
be 1 otherwise x will never be an integer            j+=1
k=0         while True:
            x = (k*m+1)/a
if x==int(x):
                m_inv_s.append(x)
break             k+=1
i+=1
       total = 0     i=0     while i<l:
total+=remainders[i]*m_s[i]*m_inv_s[i]
i+=1
    return total%capital_M
```

BRIEF DESCRIPTION:

This practical implements the **Chinese Remainder Theorem** to find a number that satisfies a system of modular equations. After confirming that the moduli are pairwise coprime, the algorithm constructs the solution using the classical CRT formula: each congruence contributes a term of the form

$$r_i \cdot M_i \cdot M_{i-1}.$$

The sum of all such terms, taken modulo the product of the moduli, gives the unique solution (mod M). The approach follows the direct constructive method for CRT, implemented entirely with loops, arithmetic operations, and manual modular inverse computation.

Code:

```
def crt (remainders, moduli):     #makign sure gcd of
all moduli is 1     l = 0     for m in moduli:
        l+=1
```

```
    if l>1:
        lowest_m = moduli[0]
for m in moduli[1:]:
if m<lowest_m:
lowest_m=m         i=2
while i<=lowest_m/2:
for j in moduli:
                devides_all = True
if j%i != 0:
                    devides_all = False
break                 if devides_all:
return None                 i+=1

    #solving
capital_M = 1     for
m in moduli:
capital_M*=m
       m_s = []
for m in moduli:
        m_s.append(capital_M/m)

    m_inv_s = []     l=0
for i in remainders:
        l+=1     i=0
while i < l:
a = m_s[i]         m
= moduli[i]
if a==m:
            return None                     #x can never be an
integer         if a==0:
            return None                     #condition will never be
satisfied regardless of x and deviding by 0 will not work         if
m==0:
            return None                     #condition will never be
satisfied regardless of x         if a>m:           s=m
else:             s=a         j = 2         while j <= s/2:
if a%j == 0 and m%j == 0:
                return None                 #GCD of a and m has to
be 1 otherwise x will never be an integer            j+=1
k=0         while True:
            x = (k*m+1)/a
if x==int(x):
                m_inv_s.append(x)
```

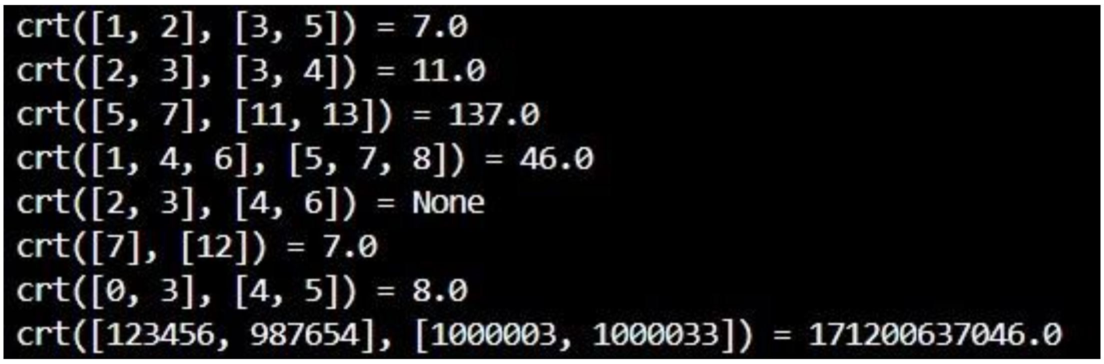
```
                    break
k+=1         i+=1
        total = 0     i=0     while i<l:
total+=remainders[i]*m_s[i]*m_inv_s[i]
i+=1
    return total%capital_M

#testing test_cases = [([1, 2], [3, 5]), ([2, 3], [3, 4]), ([5,
7], [11,
13]), ([2, 3], [4, 6]), ([7], [12]), ([0, 3], [4, 5]), ([123456,
987654], [1000003, 1000033])] for
test_case in test_cases:
    print(f"crt{test_case} =", crt(test_case[0], test_case[1]))

#runtime
a = eval(input("Enter remainders (list form): ")) m =
eval(input("Enter moduli (list form): ")) print("Result =",
crt(a, m)) import timeit runtime = timeit.timeit("crt(a, m)",
globals=globals(), number =
1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import tracemalloc
tracemalloc.start() crt(a, m) peak_memory =
tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def crt_stepsCounter
(remainders, moduli):
    l = 0
steps=1     for m in
moduli:
        steps+=1

        l+=1
        steps+=2

    steps+=1
if l>1:

        lowest_m = moduli[0]
steps+=2

        steps+=1          for
m in moduli[1:]:
            steps+=1
            steps+=1
if m<lowest_m:

                lowest_m=m
                steps+=1

        i=2
        steps+=1

        steps+=2
while i<=lowest_m/2:
            steps+=2

            for j in moduli:
                steps+=1

                devides_all = True
steps+=1

                steps+=2
if j%i != 0:
steps+=2

                    devides_all = False
steps+=1

                    steps+=1
break

            steps+=1
if devides_all:

                steps+=1
return steps

i+=1
steps+=2

    #solving
capital_M = 1
    steps+=1

    for m in moduli:
        steps+=1

        capital_M*=m
steps+=2
       m_s
= []
steps+=1

    for m in moduli:
        steps+=1
        m_s.append(capital_M/m)
        steps+=2

    m_inv_s = []
    steps+=1

    l=0
    steps+=1

    for i in remainders:
        steps+=1

        l+=1
steps+=2

i=0
    steps+=1

    steps+=1
while i < l:
steps+=1

        a = m_s[i]
        steps+=1

        m = moduli[i]
        steps+=1

        steps+=1
if a==m:

            steps+=1
            return steps                    #x can never be an
integer                steps+=1         if a==0:

            steps+=1            return steps
#condition will never be satisfied regardless of x and deviding by
0 will not work

steps+=1
if m==0:

            steps+=1            return steps
#condition will never be satisfied regardless of x

steps+=1
if a>m:

            s=m
            steps+=1

        else:
            steps+=1

            s=a
            steps+=1

        j = 2
        steps+=1

        steps+=2
while j <= s/2:
steps+=1

            steps+=5                if
a%j == 0 and m%j == 0:
                steps+=1

                steps+=1                return steps
#GCD of a and m has to be 1 otherwise x will never be an integer

j+=1
            steps+=2

        k=0
        steps+=1

        while True:
steps+=1

            x = (k*m+1)/a
            steps+=4

            steps+=2
if x==int(x):
steps+=1

                m_inv_s.append(x)
steps+=1

                steps+=1
break

k+=1
            steps+=1

        i+=1
steps+=1
       total
= 0
    steps+=1

    i=0
    steps+=1

    steps+=1
    while i<l:
steps+=1

        total+=remainders[i]*m_s[i]*m_inv_s[i]
steps+=6

        i+=1
        steps+=2

    steps+=2
    return steps

print("Number ofr steps =", crt_stepsCounter(a, m))
```

RESULTS ACHIEVED:

Test Cases:

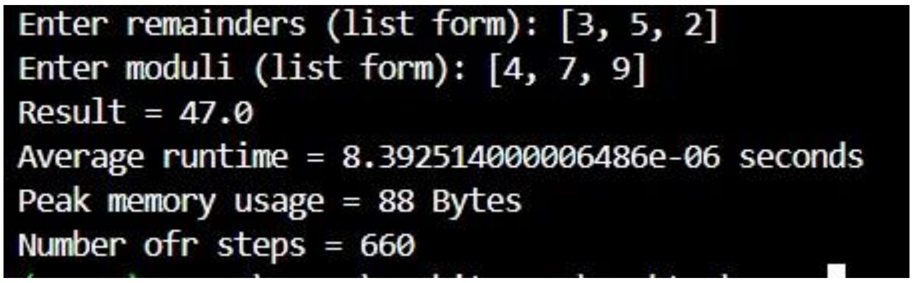
Input (n)	**Expected Output**
([1, 2], [3, 5])	7
([2, 3], [3, 4])	11
([5, 7], [11, 13])	137
([1, 4, 6], [5, 7, 8])	46
([2, 3], [4, 6])	None
([7], [12])	7
([0, 3], [4, 5])	8
([123456, 987654], [1000003, 1000033])	171200637046

Testing:



```
crt([1, 2], [3, 5]) = 7.0
crt([2, 3], [3, 4]) = 11.0
crt([5, 7], [11, 13]) = 137.0
crt([1, 4, 6], [5, 7, 8]) = 46.0
crt([2, 3], [4, 6]) = None
crt([7], [12]) = 7.0
crt([0, 3], [4, 5]) = 8.0
crt([123456, 987654], [1000003, 1000033]) = 171200637046.0
```

Performance (Snapshot)**:**



```
Enter remainders (list form): [3, 5, 2]
Enter moduli (list form): [4, 7, 9]
Result = 47.0
Average runtime = 8.392514000006486e-06 seconds
Peak memory usage = 88 Bytes
Number ofr steps = 660
```

DIFFICULTY FACED BY STUDENT:

This practical presented several challenges. The first difficulty was implementing the coprimality check correctly. Since the approach relies on scanning for common divisors manually, a small mistake could allow invalid modulus sets to pass or reject valid ones. The second difficulty was coordinating multiple lists (m_s, m_inv_s, and the remainders) to ensure each component matched its corresponding modulus. The modular inverse step also required careful handling to ensure the brute-force search always terminated. After testing the algorithm with small systems of congruences, the student verified that the assembly of the final CRT expression worked consistently.

SKILLS ACHIEVED:

1. Learned that programming can be far more efficient by stitching together codes that we know for sure work and are already optimized instead of redundantly writing the same code again for different tasks.

Practical No: 28

Date: 16/11/25

TITLE: Is Quadratic Residue

AIM/OBJECTIVE(s):

Write a function Quadratic Residue Check is_quadratic_residue(a, p) that checks if x^2 ≡ a mod p has a solution.

METHODOLOGY & TOOL USED:

The algorithm checks whether a number *a* is a quadratic residue modulo *m* using Euler's Criterion. It begins by handling two special cases:

- If a == 0, it immediately returns True, because 0 is always a quadratic residue.
- If m == 0, it returns None, because the expression is undefined.

For all other values, the algorithm evaluates: pow(a,

(m - 1) // 2, m)

Euler's Criterion states that a is a quadratic residue modulo m exactly when this value equals 1. The function returns the result of that comparison.

Tools Used:

- Python (no external libraries)
- Conditional checks for a == 0 and m == 0
- Integer exponent calculation using (m - 1) // 2
- Modular exponentiation using pow(a, exponent, m)
- Boolean comparison (== 1) to determine whether the value is a quadratic residue

Code:

```
def is_quadratic_residue (a, m):
    if a == 0: return True
if m == 0: return None
    return pow(a, ((m-1)//2), m) == 1
```

BRIEF DESCRIPTION:

This practical determines whether there exists an integer *x* such that: x^2

≡ a (mod m)

The algorithm applies Euler's Criterion: it computes pow(a, (m - 1) // 2, m) and checks whether the result is equal to 1. If the result is 1, then a is a quadratic residue modulo m. If the result differs from 1, then it is not.

Code:

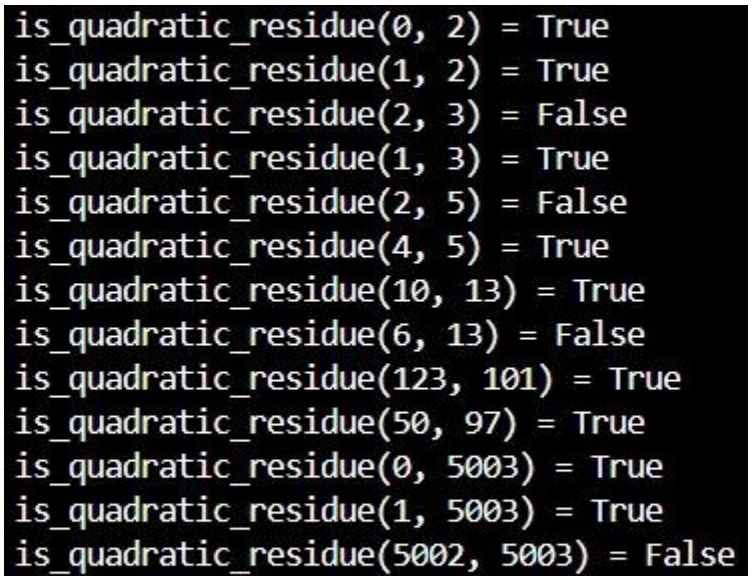
```
def is_quadratic_residue (a, m):
    if a == 0: return True
if m == 0: return None
    return pow(a, ((m-1)//2), m) == 1



#testing test_cases = [(0, 2), (1, 2), (2, 3), (1, 3), (2, 5), (4,
5), (10, 13), (6, 13), (123, 101), (50, 97), (0, 5003), (1, 5003),
(5002,
5003)] for test_case in
test_cases:
    print(f"is_quadratic_residue{test_case} =",
is_quadratic_residue(test_case[0], test_case[1]))

#runtime
a = int(input("Enter number (a): ")) m =
int(input("Enter number (m): ")) print("Result =",
is_quadratic_residue(a, m)) import timeit runtime =
timeit.timeit("is_quadratic_residue(a, m)",
globals=globals(), number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
is_quadratic_residue(a, m)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
isQuadraticResidue_stepsCounter (a, m):
    steps = 1
if a == 0:

        steps+=1
return steps

steps+=1
if m == 0:

        steps+=1
return steps

    output = pow(a, ((m-1)//2), m) == 1


    steps+=6

    steps+=1
return steps
print("Number of steps =", isQuadraticResidue_stepsCounter(a, m))
```

RESULTS ACHIEVED:

Test Cases:

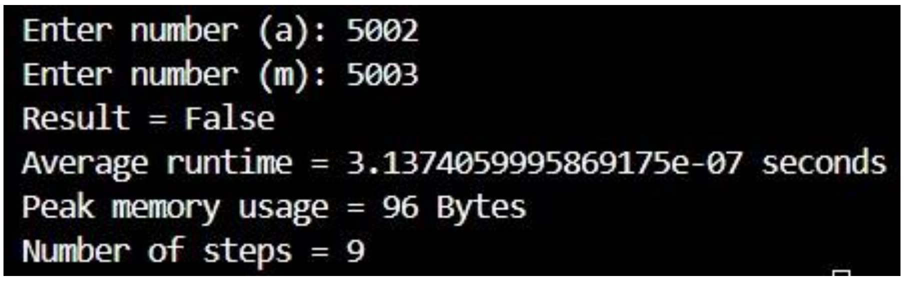
Input (a, m)	**Expected Output**
(0, 2)	True
(1, 2)	True
(2, 3)	False
(1, 3)	True
(2, 5)	False
(4, 5)	True
(10, 13)	True
(6, 13)	False
(123, 101)	True
(50, 97)	True
(0, 5003)	True
(1, 5003)	True
(5002, 5003)	False

Testing:



```
is_quadratic_residue(0, 2) = True
is_quadratic_residue(1, 2) = True
is_quadratic_residue(2, 3) = False
is_quadratic_residue(1, 3) = True
is_quadratic_residue(2, 5) = False
is_quadratic_residue(4, 5) = True
is_quadratic_residue(10, 13) = True
is_quadratic_residue(6, 13) = False
is_quadratic_residue(123, 101) = True
is_quadratic_residue(50, 97) = True
is_quadratic_residue(0, 5003) = True
is_quadratic_residue(1, 5003) = True
is_quadratic_residue(5002, 5003) = False
```

Performance (Snapshot)**:**



```
Enter number (a): 5002
Enter number (m): 5003
Result = False
Average runtime = 3.1374059995869175e-07 seconds
Peak memory usage = 96 Bytes
Number of steps = 9
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was remembering the exact structure of Euler's Criterion and making sure the exponent was computed using integer division. It was important to avoid using floating-point division because modular exponentiation requires integer exponents. Another subtle challenge was accounting for the special cases involving zero, since forgetting to check them could cause incorrect or undefined behavior. After testing a few example values, the implementation worked correctly.

SKILLS ACHIEVED:

1. Mathematics can sometimes bring long loops down to a mere equation and needs to be mastered in order to learn effective algorithmics and optimization.

Practical No: 29

Date: 16/11/25

TITLE: Order Mod

AIM/OBJECTIVE(s):

Write a function order_mod(a, n) that finds the smallest positive integer k such that $a^k \equiv 1$ mod n.

METHODOLOGY & TOOL USED:

The algorithm computes the order of a modulo n, which is the smallest positive integer k such that:

a^k mod n = 1

The function begins by checking whether n is less than or equal to 1, because in that case a meaningful order cannot exist. It then checks whether a and n are coprime, since the order of a modulo n is defined only when gcd(a, n) = 1. This gcd check is performed manually by scanning possible common divisors from 2 up to the smaller of a and n. If any such divisor divides both numbers, the function returns None.

Once coprimality is confirmed, the code searches for the smallest positive integer k such that pow(a, k, n) equals 1. The search begins at k = 1 because k = 0 would trivially satisfy the condition, but the order must be a positive integer. The function increments k until the condition is met and returns the first such value.

Tools Used:

- Python (no external libraries)
- Conditional checks (n <= 1, divisor conditions)
- Manual gcd check using a loop with %
- Modular exponentiation using pow(a, k, n)
- A while True loop that increments k until the order is found

Code:

```
def order_mod (a, n):
    #any such k can never be acheaved if n = 1
if n <= 1:
        return None
    #gcd of a and n must be 1
if a < n:          s = a     else:
s = n     i = 2     while i<=s:
if a%i == 0 and n%i == 0:
            return None
i+=1

    #finding k     k = 1        #k = 0 satisfies the condition every
time regardless of a and n if n != 0. but it is only an integer, not
a positive integer     while True:         if pow(a, k, n) == 1:
            return k
k+=1
```

BRIEF DESCRIPTION:

This practical determines the order of a number a modulo n, meaning the smallest positive integer k for which a^k mod n = 1. The algorithm verifies that a and n are coprime, then tests increasing values of k using modular exponentiation until the condition is satisfied. The first valid value of k is returned as the order.

Code:

```
def order_mod (a, n):
    #any such k can never be acheaved if n = 1
if n <= 1:
        return None

    #gcd of a and n must be 1
if a < n:          s = a     else:
s = n     i = 2     while i<=s:
if a%i == 0 and n%i == 0:
            return None
i+=1

    #finding k     k = 1        #k = 0 satisfies the condition every
time regardless of a and n if n != 0. but it is only an integer, not
a positive integer     while True:         if pow(a, k, n) == 1:
            return k
k+=1

#testing
```

```
test_cases = [(1, 7), (2, 7), (2, 9), (4, 9), (2, 4), (6, 15), (2,
11), (3, 11), (10, 17), (7, 19), (4, 17), (3, 13)] for
test_case in test_cases:
    print(f"order_mod{test_case} =", order_mod(test_case[0],
test_case[1]))

#runtime
a = int(input("Enter number (a): ")) n = int(input("Enter number
(n): ")) print("Result =", order_mod(a, n)) import timeit runtime =
timeit.timeit("order_mod(a, n)", globals=globals(), number
= 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
order_mod(a, n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
orderMod_stepsCounter (a, n):
    steps=1
if n <= 1:

        steps+=1
return steps
       steps+=1
if a < n:
steps+=1

        s = a
steps+=1

else:
        s = n
        steps+=1

    i = 2
    steps+=1

    steps+=1
while i<=s:
steps+=1

        steps+=5        if a%i
== 0 and n%i == 0:

            steps+=1
return steps
```

```
        i+=1
        steps+=2

    k = 1
    steps+=1

    while True:

        steps+=3        if
pow(a, k, n) == 1:

            steps+=1
return steps

k+=1
        steps+=2

print("Number of steps =", orderMod_stepsCounter(a, n))
```

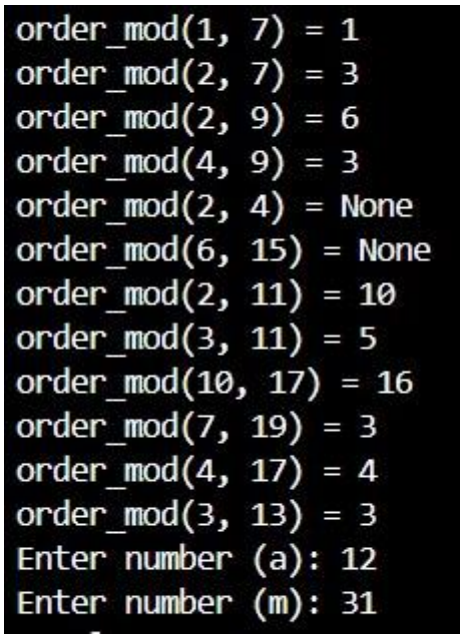
RESULTS ACHIEVED:

Test Cases:

Input (a, n)	**Expected Output**
(1, 7)	1
(2, 7)	3
(2, 9)	6
(4, 9)	3
(2, 5)	None
(6, 15)	None
(2, 11)	10
(3, 11)	5

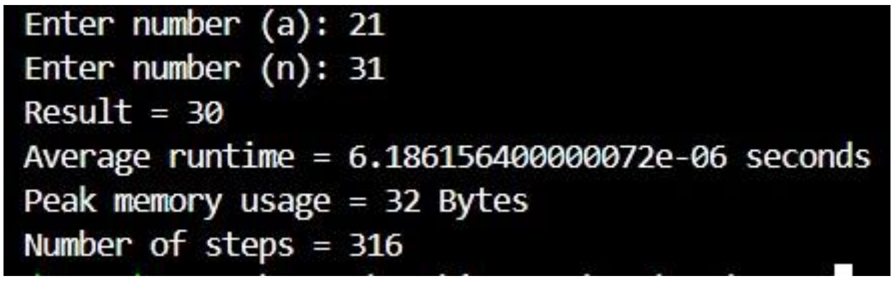
(10, 17)	16
(7, 19)	3
(4, 17)	4
(3, 13)	3

Testing:



```
order_mod(1, 7) = 1
order_mod(2, 7) = 3
order_mod(2, 9) = 6
order_mod(4, 9) = 3
order_mod(2, 4) = None
order_mod(6, 15) = None
order_mod(2, 11) = 10
order_mod(3, 11) = 5
order_mod(10, 17) = 16
order_mod(7, 19) = 3
order_mod(4, 17) = 4
order_mod(3, 13) = 3
Enter number (a): 12
Enter number (m): 31
```

Performance (Snapshot)**:**



```
Enter number (a): 21
Enter number (n): 31
Result = 30
Average runtime = 6.186156400000072e-06 seconds
Peak memory usage = 32 Bytes
Number of steps = 316
```

DIFFICULTY FACED BY STUDENT:

One difficulty in this practical was ensuring that the gcd check correctly identified when a and n were not coprime. Since the implementation uses a manual divisor scan rather than a built-in function, it required careful handling to avoid missing any shared divisors. Another challenge was structuring the search loop so that the algorithm always progresses toward a valid result without accidentally including the trivial case k = 0. After testing the function with several values, the logic behaved consistently and produced the expected orders.

SKILLS ACHIEVED:

1. Debugging logical errors in codes using black box testing

Practical No: 30

Date: 16/11/25

TITLE: Is Fibonacci Prime

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm checks whether a given Fibonacci number is prime. It begins by generating the Fibonacci sequence up to index n using a simple loop. The first two numbers, 0 and 1, are appended manually. After that, each new Fibonacci number is computed as the sum of the previous two, using list indexing.

Once the nth Fibonacci number is obtained, the algorithm checks whether this number is prime. It does this by scanning possible divisors from 2 up to half of the Fibonacci value. If any divisor divides the number evenly, the function concludes that it is not prime. If no divisors are found, the number is declared prime.

Tools Used:

- Python (no external libraries)
- A list to store Fibonacci values
- A loop to build the Fibonacci sequence using addition
- A second loop to test primality using %
- Conditional checks to determine whether any divisor divides the number
- Return statements that provide the primality result

Code:

```
def is_fibonacci_prime (n):
    #is_fibonacci
fib = [1, 2]
if n in fib:
        is_fibonacci = True
loop_to_be_run = False     else:
        loop_to_be_run = True
while loop_to_be_run:
        new = fib[-1]+fib[-2]
        if new == n:
            is_fibonacci = True
break           if new>n:
is_fibonacci = False
break           fib.append(new)

    #is_prime
if n<=1:
        return False
i = 2     while i <
n/2:          if n%i
== 0:
            return False
i+=1

    return is_fibonacci
```

BRIEF DESCRIPTION:

This practical determines whether the nth Fibonacci number is prime. The algorithm first constructs the Fibonacci sequence up to the requested index, then applies a straightforward primality test. If the computed Fibonacci number has no divisors other than 1 and itself, the function returns that it is prime; otherwise, it is composite.

Code:

```
def is_fibonacci_prime (n):
    #is_fibonacci
fib = [1, 2]
if n in fib:
        is_fibonacci = True
loop_to_be_run = False     else:
        loop_to_be_run = True
while loop_to_be_run:
        new = fib[-1]+fib[-2]
if new == n:
            is_fibonacci = True
break           if new>n:
is_fibonacci = False
break           fib.append(new)

    #is_prime
if n<=1:
        return False
i = 2     while i <
n/2:          if n%i
== 0:
            return False
i+=1
```

```
        return
is_fibonacci

#testing for i in [2, 3, 5, 13, 89, 8, 21, 7, 11,
1, 0, 50]:
    print(f"is_fibonacci_prime({i}) =", is_fibonacci_prime(i))

#runtime
n = int(input("Enter number (n): ")) print("Result =",
is_fibonacci_prime(n)) import timeit runtime =
timeit.timeit("is_fibonacci_prime(n)", globals=globals(), number =
1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
is_fibonacci_prime(n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
orderMod_stepsCounter (a, n):
    steps=1
if n <= 1:

        steps+=1
return steps
        steps+=1
if a < n:
steps+=1

        s = a
steps+=1

else:
        s = n
        steps+=1

    i = 2
    steps+=1

    steps+=1
while i<=s:
steps+=1

        steps+=5        if a%i
== 0 and n%i == 0:

            steps+=1
return steps
```

```
        i+=1
        steps+=2

    k = 1
    steps+=1

    while True:

        steps+=3         if
pow(a, k, n) == 1:

            steps+=1
return steps

k+=1
        steps+=2

print("Number of steps =", orderMod_stepsCounter(10, n))
```

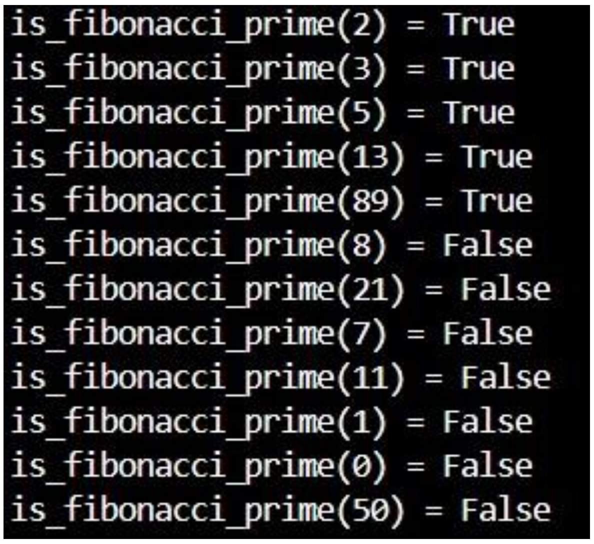
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected output**
2	**True**
3	**True**
5	**True**
13	**True**
89	**True**
8	**False**
21	**False**
7	**False**

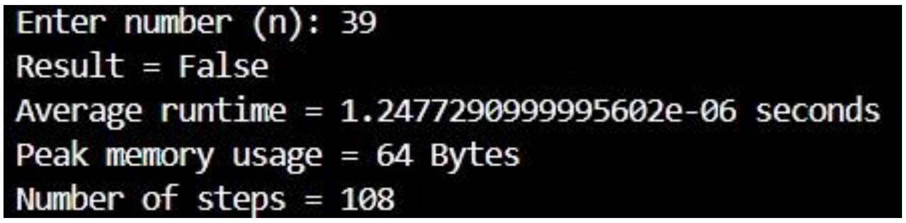
11	**False**
1	**False**
0	**False**
50	**False**

Testing:



```
is_fibonacci_prime(2) = True
is_fibonacci_prime(3) = True
is_fibonacci_prime(5) = True
is_fibonacci_prime(13) = True
is_fibonacci_prime(89) = True
is_fibonacci_prime(8) = False
is_fibonacci_prime(21) = False
is_fibonacci_prime(7) = False
is_fibonacci_prime(11) = False
is_fibonacci_prime(1) = False
is_fibonacci_prime(0) = False
is_fibonacci_prime(50) = False
```

Performance (Snapshot):



```
Enter number (n): 39
Result = False
Average runtime = 1.2477290999995602e-06 seconds
Peak memory usage = 64 Bytes
Number of steps = 108
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was ensuring that the Fibonacci sequence was built correctly for all values of n, especially smaller ones where the list must begin with the fixed values 0 and 1. Another subtle issue was managing the primality loop: checking divisors only up to half the number works, but forgetting to start from 2 or stopping too early could lead to incorrect classifications. After testing several values of n and validating the Fibonacci output, the primality test behaved as expected.

SKILLS ACHIEVED:

1. Noted the incredible efficiency of linear time complexity algorithms when dealing with high numbers.

Practical No: 31

Date: <u>16/11/25</u>

TITLE: Lucas Sequence

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm generates the first n terms of the Lucas sequence. It begins by initializing a list with the standard Lucas starting values [2, 1]. A loop then runs from index 2 up to n - 1. Each new term is calculated by adding the previous two terms, and the result is appended to the list.

Once the loop completes, the code appends a final 0 to the sequence. The function then returns the first n elements, which ensures that the output always contains exactly n terms regardless of the extra appended value.

Tools Used:

- Python (no external libraries)
- A list to store the Lucas numbers
- append() for extending the sequence
- A while loop to generate terms iteratively
- Simple addition of consecutive terms (l[-1] + l[-2])
- List slicing (l[:n]) to return exactly n elements **Code:**

```
def lucas_sequence (n):
    l = [2, 1]

    i = 2
while i < n:
        l.append(l[-1]+l[-2])
i+=1
    l.append(0)
return l[:n]
```

BRIEF DESCRIPTION:

This practical produces the Lucas sequence up to the nth term. The Lucas sequence is similar to the Fibonacci sequence but begins with 2 and 1 instead of 0 and 1. Each term is formed by adding the previous two. After generating the sequence iteratively, the function returns a list containing exactly n terms.

Code:

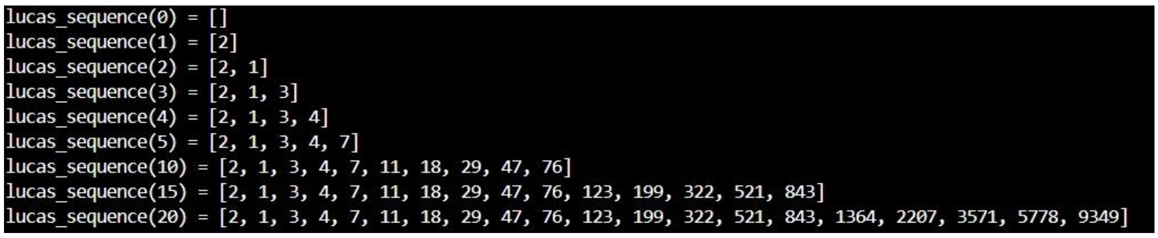
```
def lucas_sequence (n):
    l = [2, 1]

    i = 2
while i < n:
        l.append(l[-1]+l[-2])
i+=1
    l.append(0)
    return l[:n]

#testing for i in [0, 1, 2, 3, 4, 5, 10,
15, 20]:
    print(f"lucas_sequence({i}) =", lucas_sequence(i))

#runtime n = int(input("Enter number (n): ")) # print("Result
=", lucas_sequence(n)) import timeit runtime =
timeit.timeit("lucas_sequence(n)", globals=globals(), number =
1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import tracemalloc
tracemalloc.start() lucas_sequence(n) peak_memory
= tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
lucasSequence_stepsCounter (a, n):
    l = [2, 1]
    steps=1

    i = 2
    steps+=1

    steps+=1
while i < n:
steps+=1

        l.append(l[-1]+l[-2])
steps+=4

        i+=1



        steps+=2

    l.append(0)
    steps+=1
    output = l[:n]
    steps+=2

    return steps

print("Number of steps =", lucasSequence_stepsCounter(10, n))
```

RESULTS ACHIEVED:

Test Cases:

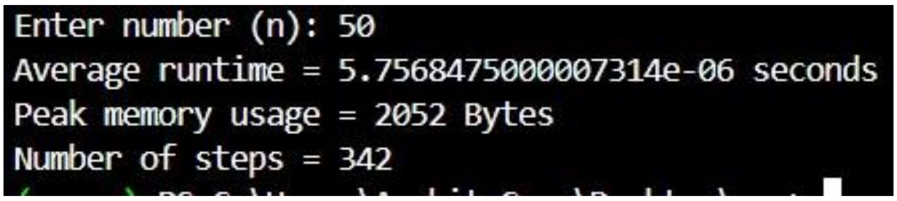
Input (n)	**Expected output**
0	[]
1	[2]
2	[2, 1]
3	[2, 1, 3]
4	[2, 1, 3, 4]
5	[2, 1, 3, 4, 7]
10	[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
15	[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843]
20	[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1364, 2207, 3571, 5778, 9349]

Testing:



```
lucas_sequence(0) = []
lucas_sequence(1) = [2]
lucas_sequence(2) = [2, 1]
lucas_sequence(3) = [2, 1, 3]
lucas_sequence(4) = [2, 1, 3, 4]
lucas_sequence(5) = [2, 1, 3, 4, 7]
lucas_sequence(10) = [2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
lucas_sequence(15) = [2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843]
lucas_sequence(20) = [2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, 521, 843, 1364, 2207, 3571, 5778, 9349]
```

Performance (Snapshot)**:**



```
Enter number (n): 50
Average runtime = 5.7568475000007314e-06 seconds
Peak memory usage = 2052 Bytes
Number of steps = 342
```

DIFFICULTY FACED BY STUDENT:

The main difficulty in this practical was ensuring the indexing logic stayed consistent when building the sequence. Because the code appends values dynamically, it was important to manage the loop boundaries correctly so that the generated list had enough elements to slice. Another subtle issue was understanding why the extra 0 is appended before slicing; without careful attention, this detail could seem out of place. Once the behavior was verified through testing, the function worked reliably.

SKILLS ACHIEVED:

1. Sometimes the maths required to optimize something just doesn't exist. You either have to invent the maths yourself or deal with the loop.

Practical No: 32

Date: <u>16/11/25</u>

TITLE: Is Perfect Power

AIM/OBJECTIVE(s):

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.

METHODOLOGY & TOOL USED:

The algorithm checks whether a number n is a perfect power, meaning that n = a^b for some integers a > 1 and b > 1. It begins by handling simple edge cases:

- n = 1 is treated as a perfect power,
- n = 0, 2, and 3 are immediately rejected because none of them can be written as a^b with b > 1.

After these checks, the code searches for integers a starting from 2 up to n. For each value of a, it tries exponents b beginning at 2. It computes a**b and compares it with n.

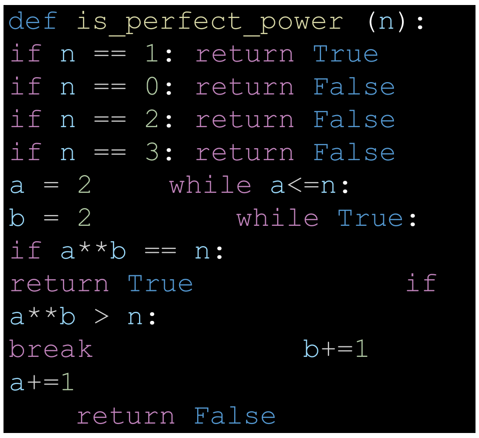
- If a**b = n, the function returns True.
- If a**b becomes greater than n, the inner loop breaks and the algorithm moves to the next base a.

If no combination of a and b matches n, the function returns False.

Tools Used:

- Python (no external libraries)
- Nested while loops for scanning possible bases and exponents
- Exponentiation operator **
- Conditional checks for early exits on edge cases
- Comparison operators to detect matches and break conditions

Code:



```
def is_perfect_power (n):
if n == 1: return True
if n == 0: return False
if n == 2: return False
if n == 3: return False
a = 2     while a<=n:
b = 2         while True:
if a**b == n:
return True              if
a**b > n:
break              b+=1
a+=1
    return False
```

BRIEF DESCRIPTION:

This practical determines whether a number can be expressed as a perfect power of the form a^b, where both a and b are integers greater than 1. The algorithm tries all reasonable values of a and b in ascending order, stopping whenever a**b exceeds the target number. If any pair produces exactly n, the function concludes that the number is a perfect power.

Code:

```
def is_perfect_power (n):
if n == 1: return True
if n == 0: return False
if n == 2: return False
if n == 3: return False
a = 2     while a<=n:
b = 2         while True:
if a**b == n:
return True             if
a**b > n:
break            b+=1
a+=1
    return False

#testing for i in [1, 4, 8, 9, 16, 27, 32, 81,
2, 6, 10]:
    print(f"is_perfect_power({i}) =", is_perfect_power(i))

#runtime
n = int(input("Enter number (n): ")) print("Result =",
is_perfect_power(n)) import timeit runtime =
timeit.timeit("is_perfect_power(n)", globals=globals(), number =
1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
is_perfect_power(n)
```

```
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
lucasSequence_stepsCounter (a, n):
    l = [2, 1]
    steps=1

    i = 2
    steps+=1

    steps+=1
while i < n:
steps+=1

        l.append(l[-1]+l[-2])
steps+=4

        i+=1
steps+=2

    l.append(0)
    steps+=1

    output = l[:n]
    steps+=2

    return steps

print("Number of steps =", lucasSequence_stepsCounter(10, n))
```

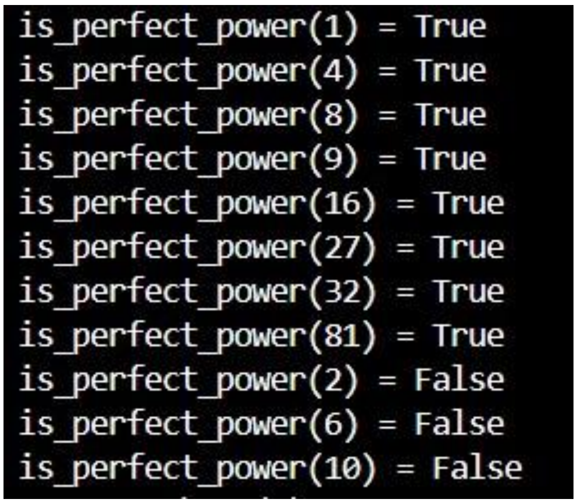
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
1	True
4	True
8	True
9	True

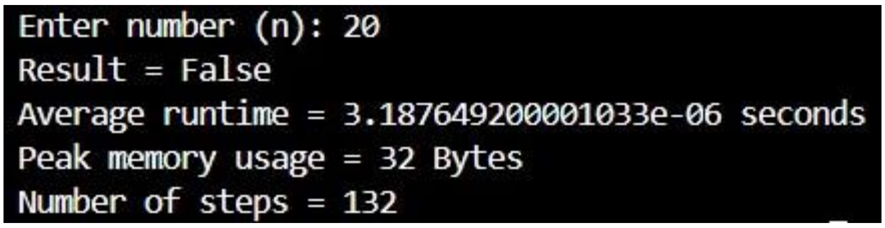
16	True
27	True
32	True
81	True
2	False
6	False
10	False

Testing:



```
is_perfect_power(1) = True
is_perfect_power(4) = True
is_perfect_power(8) = True
is_perfect_power(9) = True
is_perfect_power(16) = True
is_perfect_power(27) = True
is_perfect_power(32) = True
is_perfect_power(81) = True
is_perfect_power(2) = False
is_perfect_power(6) = False
is_perfect_power(10) = False
```

Performance (Snapshot)**:**



```
Enter number (n): 20
Result = False
Average runtime = 3.187649200001033e-06 seconds
Peak memory usage = 32 Bytes
Number of steps = 132
```

DIFFICULTY FACED BY STUDENT:

The main difficulty in this practical was managing the nested looping structure for a and b. Since both values grow quickly when exponentiated, it was important to break the inner loop as soon as a**b became larger than n.

Another subtle issue was handling edge cases correctly; numbers like 0, 1, 2, and 3 behave differently and needed explicit checks. After verifying the loop boundaries and testing a few known perfect powers (such as 16, 27, and 81), the implementation worked consistently.

SKILLS ACHIEVED:

1. Some problems require original thinking.

Practical No: 33

Date: 16/11/25

TITLE: Collatz Length

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the Collatz length of a number n, meaning the number of steps required for the Collatz sequence starting at n to reach 1. It first checks whether n is less than 1; if so, the function returns None because the Collatz sequence is only defined for positive integers.

The sequence is then generated iteratively using a loop. A variable curr is set equal to n, and a counter steps tracks how many operations are performed. In each iteration:

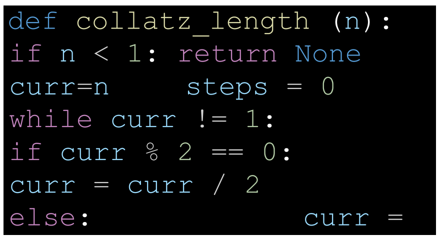
- If curr is even, it is updated to curr / 2.
- If curr is odd, it is updated to 3 * curr + 1.

After each update, the step counter is increased by 1. When curr reaches 1, the loop stops and the function returns the total number of steps.

Tools Used:

- Python (no external libraries)
- Conditional branching using if and else
- Modulus operator % to check whether the current number is even
- Basic arithmetic operations (curr / 2, 3 * curr + 1)
- A while loop that repeats until the sequence reaches 1
- A counter variable (steps) to keep track of the number of iterations

Code:



```
def collatz_length (n):
if n < 1: return None
curr=n       steps = 0
while curr != 1:
if curr % 2 == 0:
curr = curr / 2
else:                 curr =
3 * curr + 1
steps+=1     return steps
```

BRIEF DESCRIPTION:

This practical computes the number of steps required for a positive integer to reach 1 under the Collatz transformation. Starting from n, the algorithm repeatedly applies the rule:

- even → divide by 2
- odd → multiply by 3 and add 1

The count of these operations is returned as the Collatz length of n**.**

Code:

```
def collatz_length (n):
if n < 1: return None
curr=n     steps = 0
while curr != 1:         if
curr % 2 == 0:
curr = curr / 2
else:
            curr = 3 * curr + 1
steps+=1     return steps

#testing for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12,
19, 29]:     print(f"collatz_lenght({i}) =",
collatz_length(i))

#runtime
n = int(input("Enter number (n): ")) print("Result =",
collatz_length(n)) import timeit runtime =
timeit.timeit("collatz_length(n)", globals=globals(), number =
1000000) print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
collatz_length(n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
collatzLength_stepsCounter (n):
    steps_counter=1
    if n < 1:

        steps_counter+=1
return steps_counter
       curr=n
steps_counter+=1

    steps = 0
    steps_counter+=1

    steps_counter+=1
while curr != 1:
        steps_counter+=1

        steps_counter+=2
if curr % 2 == 0:
steps_counter+=1

            curr = curr / 2
            steps_counter+=2

        else:
            curr = 3 * curr + 1
            steps_counter+=3

        steps+=1
steps_counter+=2

    steps_counter+=1
    return steps_counter

print("Number of steps =", collatzLength_stepsCounter(n))
```

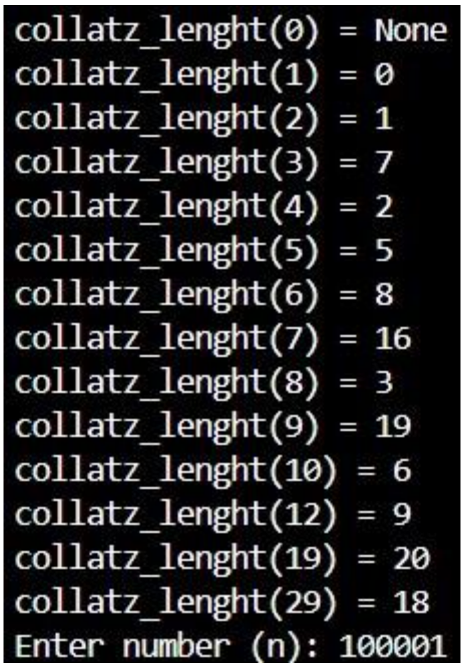
RESULTS ACHIEVED:

Test Cases:

Input (a, m)	**Expected Output**
0	None
1	0
2	1
3	7

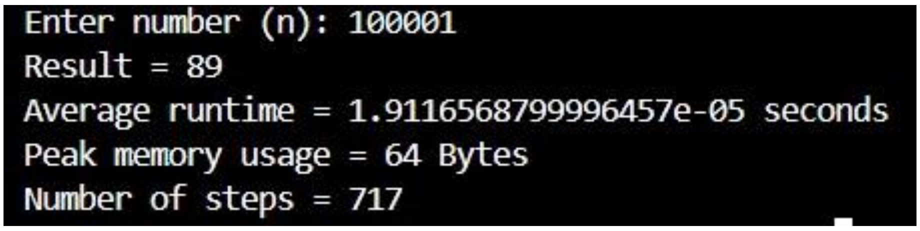
4	2
5	5
6	8
7	16
8	3
9	19
10	6
12	9
19	20
29	18

Testing:



```
collatz_lenght(0) = None
collatz_lenght(1) = 0
collatz_lenght(2) = 1
collatz_lenght(3) = 7
collatz_lenght(4) = 2
collatz_lenght(5) = 5
collatz_lenght(6) = 8
collatz_lenght(7) = 16
collatz_lenght(8) = 3
collatz_lenght(9) = 19
collatz_lenght(10) = 6
collatz_lenght(12) = 9
collatz_lenght(19) = 20
collatz_lenght(29) = 18
Enter number (n): 100001
```

Performance (Snapshot)**:**



```
Enter number (n): 100001
Result = 89
Average runtime = 1.9116568799996457e-05 seconds
Peak memory usage = 64 Bytes
Number of steps = 717
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was understanding how the polygonal number formula is constructed and ensuring it was transcribed correctly into code using only basic arithmetic. Care had to be taken when mixing multiplication, parentheses, and integer division to avoid order-of-operations mistakes. After checking a few known values (like triangular and square numbers), the implementation was confirmed to be accurate.

SKILLS ACHIEVED:

1. Python often reallocates memory instead of allocating new memory creating strong memory efficiency. It is just slow.

Practical No: 34

Date: <u>16/11/25</u>

TITLE: Polygonal Number

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm computes the nth s-gonal (polygonal) number using the standard closed-form formula. It first checks whether the input values are valid: s must be at least 3 (since polygons start with triangles), and n must be non-negative. If either condition fails, the function returns None.

For valid inputs, the algorithm plugs the values into the polygonal number formula written in plain arithmetic operations: polygonal = (s - 2) * n * (n + 1) // 2 - (s - 3) * n

This expression is evaluated using integer arithmetic only. The result is then returned as the nth polygonal number for the given polygon side count s.

Tools Used:

- Python (no external libraries)
- Conditional checks for invalid values (s < 3, n < 0)
- Integer arithmetic using multiplication and subtraction
- Integer division using //
- Direct evaluation of the polygonal number formula **Code:**

```
def polygonal_number (s, n):
    if s<3: return None
if n<0: return None
    return (s-2)*n*(n+1)//2 - (s-3)*n
```

BRIEF DESCRIPTION:

This practical calculates the nth polygonal number for an s-sided polygon. Polygonal numbers generalize triangular, square, pentagonal, and higher-order sequences. The algorithm verifies the inputs and then evaluates the standard formula using plain integer operations. The output is the corresponding polygonal value.

Code:

```
def polygonal_number (s, n):
    if s<3: return None
if n<0: return None
    return (s-2)*n*(n+1)//2 - (s-3)*n

#testing for i in [(3, 1), (3, 7), (4, 5), (5, 3), (6, 4), (4,
10000), (1000, 5), (5, 1), (3, 10), (2, 5), (6, 0), (4, -3)]:
    print(f"polygonal_number{i} =", polygonal_number(i[0], i[1]))

#runtime
s = int(input("Enter number (s): ")) n = int(input("Enter number
(n): ")) print("Result =", polygonal_number(s, n)) import timeit
runtime = timeit.timeit("polygonal_number(s, n)", globals=globals(),
number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
polygonal_number(s, n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
polygonalNumber_steosCounter (s, n):
steps=1     if s<3:

        steps+=1
return steps

steps+=1
if n<0:

        steps+=1
return steps

    output = (s-2)*n*(n+1)//2 - (s-3)*n
steps+=8

    steps+=1
    return steps

print("Number of steps =", polygonalNumber_steosCounter(s, n))
```

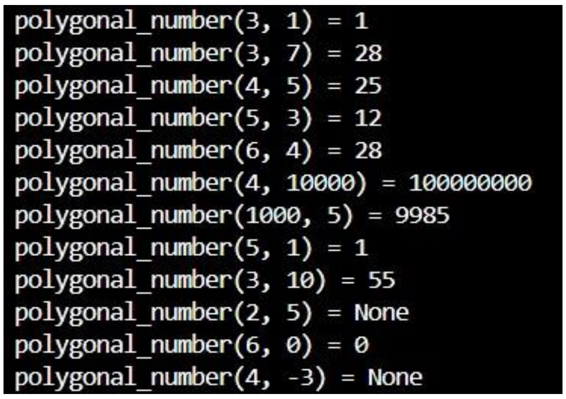
RESULTS ACHIEVED:

Test Cases:

Input (a, n)	**Expected Output**

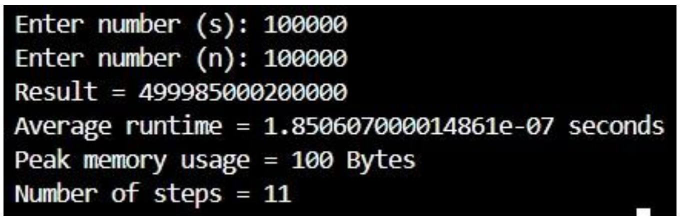
(3, 1)	1
(3, 7)	28
(4, 5)	25
(5, 3)	12
(6, 4)	28
(4, 10000)	100000000
(1000, 5)	9985
(5, 1)	1
(3, 10)	55
(2, 5)	None
(6, 0)	0
(4, -3)	None

Testing:



```
polygonal_number(3, 1) = 1
polygonal_number(3, 7) = 28
polygonal_number(4, 5) = 25
polygonal_number(5, 3) = 12
polygonal_number(6, 4) = 28
polygonal_number(4, 10000) = 100000000
polygonal_number(1000, 5) = 9985
polygonal_number(5, 1) = 1
polygonal_number(3, 10) = 55
polygonal_number(2, 5) = None
polygonal_number(6, 0) = 0
polygonal_number(4, -3) = None
```

Performance (Snapshot)**:**



```
Enter number (s): 100000
Enter number (n): 100000
Result = 499985000200000
Average runtime = 1.850607000014861e-07 seconds
Peak memory usage = 100 Bytes
Number of steps = 11
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was understanding how the polygonal number formula is constructed and ensuring it was transcribed correctly into code using only basic arithmetic. Care had to be taken when mixing multiplication, parentheses, and integer division to avoid order-of-operations mistakes. After checking a few known values (like triangular and square numbers), the implementation was confirmed to be accurate.

SKILLS ACHIEVED:

1. Mathematics is very good for optimization

Practical No: 35

Date: <u>16/11/25</u>

TITLE: Is Carmichael

AIM/OBJECTIVE(s):

Write a function Carmichael Number Check is_carmichael(n) that checks if a composite number n satisfies a^(n−1) ≡ 1 mod n for all a coprime to n.

METHODOLOGY & TOOL USED:

The algorithm checks whether a number n is a Carmichael number by applying two main conditions:

1. a and n must be coprime for every a in the range 1 to n - 1.
2. For every such a, the value pow(a, n - 1, n) must equal 1.

The function begins with basic edge-case handling: values less than 1 return None, and the numbers 1, 2, and 3 immediately return False, since none of them are Carmichael numbers.

For each a from 1 to n - 1, the algorithm performs a manual gcd check. It determines a search boundary (larger = a/2 or n/2) and scans from 2 up to that value. If any divisor divides both a and n, they are not coprime, and the algorithm simply moves on to the next a.

If a is coprime with n, the algorithm applies the Carmichael condition using modular exponentiation:

pow(a, n - 1, n)

If this value is ever not equal to 1, the function returns False immediately. If all coprime a satisfy the condition, the number is declared a Carmichael number.

Tools Used:

- Python (no external libraries)
- Manual gcd test using loops and %
- Modular exponentiation using pow(a, n - 1, n)
- Conditional checks to filter out invalid or trivial cases
- Incremental loop over all values of a from 1 to n - 1

Code:

```
def is_carmichael (n):
if n<1: return None
    if n==1 or n==2 or n==3: return False
a = 1     while a < n:
        # is gcd(a, n) = 1
if a>n: larger = a/2
else: larger = n/2         i = 2
gcd_1 = True          while
i<larger:              if a%i==0
and n%i==0:
                 gcd_1 = False
break              i+=1
if not gcd_1:
a+=1             continue
                 if pow(a, n-1,
n) != 1:
             return False

        a+=1
return True
```

BRIEF DESCRIPTION:

This practical tests whether a number n is a Carmichael number, meaning it satisfies the modular identity:

a^(n - 1) mod n = 1

for every integer a that is coprime with n. The algorithm manually checks gcd conditions for each a and then applies modular exponentiation to verify the Carmichael property. If the condition holds for all valid values of a, the number is classified as a Carmichael number.

Code:

```
def is_carmichael (n):     if n<1: return
None     if n==1 or n==2 or n==3: return
False     a = 1     while a < n:
        # is gcd(a, n) = 1
if a>n: larger = a/2
else: larger = n/2         i
= 2          gcd_1 = True
while i<larger:
            if a%i==0 and n%i==0:
                gcd_1 = False
break
```

```
            i+=1
if not gcd_1:
a+=1
continue
                if pow(a, n-1,
n) != 1:
            return False

        a+=1
    return True

#testing for i in [1, 2, 3 ,4, 6, 15, 561, 1105, 1729, 2465,
5610, 6601]:
    print(f"is_carmichael({i}) =", is_carmichael(i))

#runtime
n = int(input("Enter number (n): ")) print("Result =",
is_carmichael(n)) import timeit runtime =
timeit.timeit("is_carmichael(n)", globals=globals(), number =
1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
is_carmichael(n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
stepsCounter (n):
    steps=1
if n<1:

        steps+=1
return steps
        steps+=5     if n==1
or n==2 or n==3:

steps+=1
return steps
        a
= 1
    steps+=1

    steps+=1
while a < n:
steps+=1
if a>n:
            steps+=1

            larger = a/2
steps+=2
```

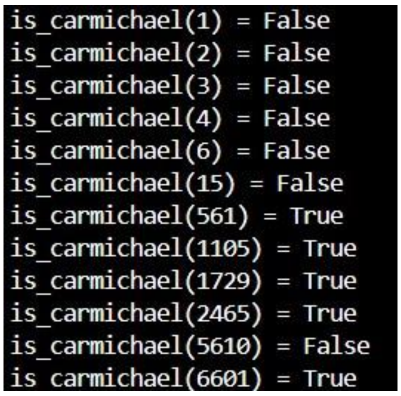
```
        else:
            larger = n/2
            steps+=1

        i = 2
        steps+=1

        gcd_1 = True
        steps+=1

        steps+=1
while i<larger:
steps+=1

            steps+=5
if a%i==0 and n%i==0:

                gcd_1 = False
                steps+=1

                steps+=1
break

i+=1
steps+=1
steps+=1          if
not gcd_1:

            a+=1
            steps+=1

            steps+=1
continue
               steps+=3
if pow(a, n-1, n) != 1:

            steps+=1
            return steps

        a+=1
steps+=2

steps+=1
    return steps

print("Number of steps =", stepsCounter(n))
```

RESULTS ACHIEVED:

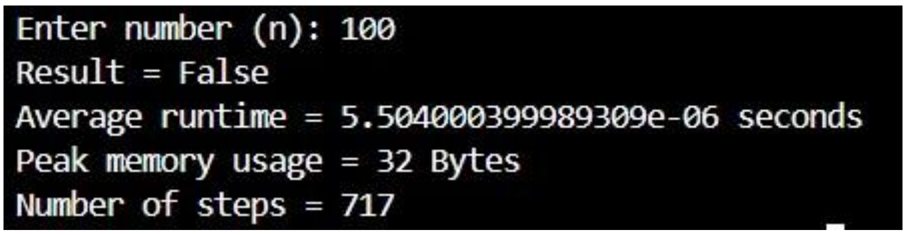
Test Cases:

Input (n)	**Expected output**
1	False
2	False
3	False
4	False
6	False
15	False
561	True
1105	True
1729	True
2465	True
5610	False
6601	True

Testing:


```
is_carmichael(1) = False
is_carmichael(2) = False
is_carmichael(3) = False
is_carmichael(4) = False
is_carmichael(6) = False
is_carmichael(15) = False
is_carmichael(561) = True
is_carmichael(1105) = True
is_carmichael(1729) = True
is_carmichael(2465) = True
is_carmichael(5610) = False
is_carmichael(6601) = True
```

Performance (Snapshot)**:**


```
Enter number (n): 100
Result = False
Average runtime = 5.504000399989309e-06 seconds
Peak memory usage = 32 Bytes
Number of steps = 717
```

DIFFICULTY FACED BY STUDENT:

A major difficulty in this practical was implementing the gcd check without using built-in functions. Since the algorithm checks divisibility up to half of a or half of n, careful attention was needed to avoid mistakes that might incorrectly classify numbers as coprime. Another challenge was ensuring that the Carmichael condition was tested for the correct range of a and that the modular exponentiation step used the correct exponent (n - 1). After running multiple test cases, the algorithm behaved consistently.

SKILLS ACHIEVED:

1. Sometimes it is smarter to hard code the output for some inputs and then write an algorithm for the other inputs rather than writing a more complicated algorithm that covers all input.

Practical No: 36

Date: 20/11/25

TITLE: Is Prime Miller Rabin

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm determines whether a number n is prime using the Miller–Rabin primality test. It begins with a few basic checks: values less than 2 are not prime, 2 and 3 are prime, and any even number greater than 2 is composite.

After the base cases, the algorithm rewrites n - 1 in the form: n

- 1 = d * 2^s

This is done by repeatedly dividing d by 2 until it becomes odd, while counting the number of divisions in s.

The function then performs up to k Miller–Rabin rounds. It chooses bases starting from a = 2 and increasing by 1 for each test. For each base, it computes:

x = pow(a, d, n)

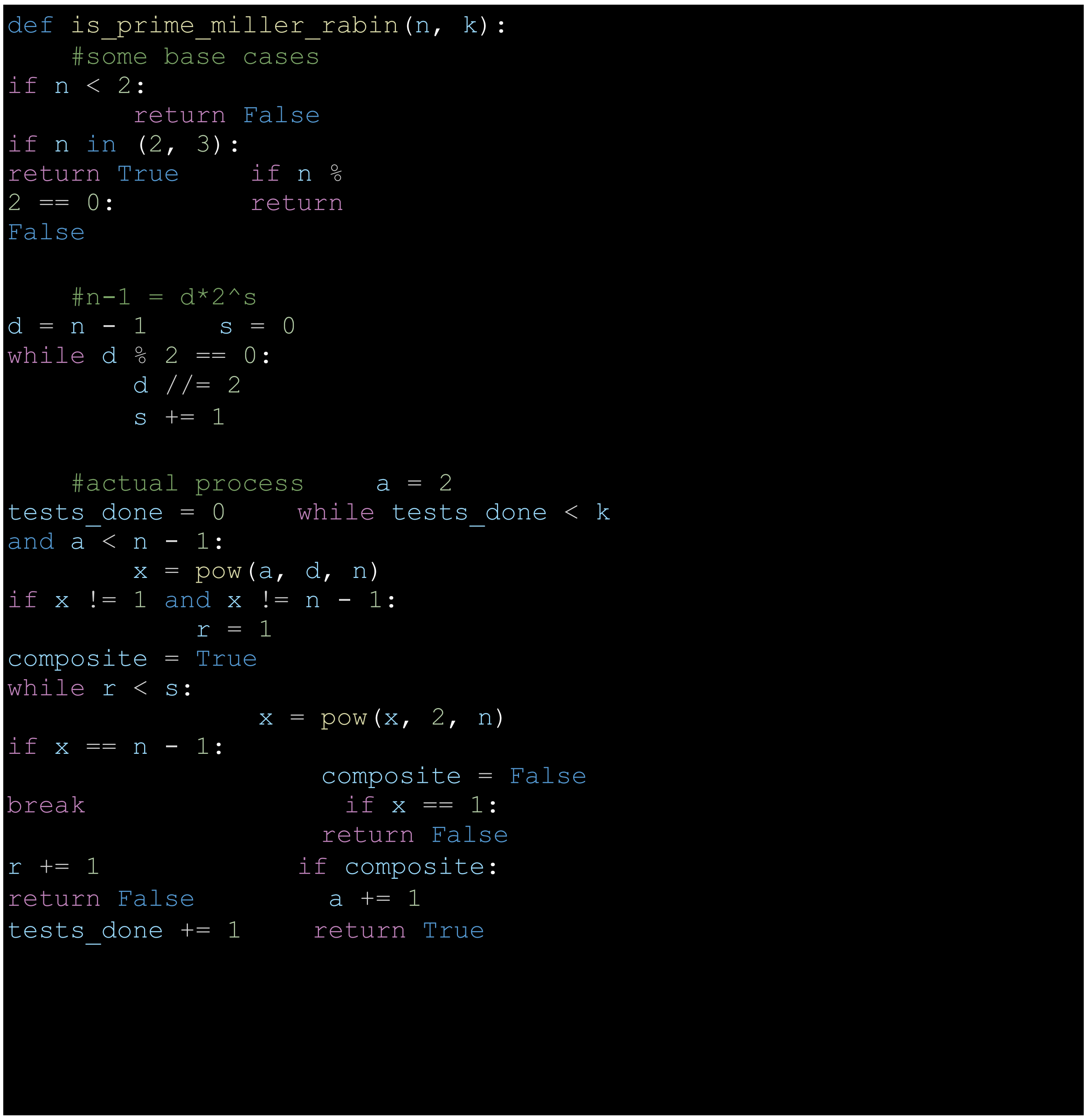
If x equals 1 or n - 1, the number passes this round. Otherwise, it repeatedly squares x using: x = pow(x, 2, n)

up to s - 1 times. If during this process x becomes n - 1, the round is passed. If x ever becomes 1 or never reaches n - 1, the number is declared composite.

If all selected bases pass their tests, the function returns that n is probably prime.

Tools Used:

- Python (no external libraries)
- Conditional checks to handle small or even numbers
- Loop to decompose n - 1 into d and s
- Modular exponentiation using pow(a, d, n) and pow(x, 2, n)
- Controlled iteration over test bases
- Boolean flags to detect composite behavior during testing **Code:**



```
def is_prime_miller_rabin(n, k):
    #some base cases
if n < 2:
        return False
if n in (2, 3):
return True     if n %
2 == 0:         return
False

    #n-1 = d*2^s
d = n - 1     s = 0
while d % 2 == 0:
        d //= 2
        s += 1

    #actual process     a = 2
tests_done = 0     while tests_done < k
and a < n - 1:
        x = pow(a, d, n)
if x != 1 and x != n - 1:
            r = 1
composite = True
while r < s:
                x = pow(x, 2, n)
if x == n - 1:
                    composite = False
break                 if x == 1:
                    return False
r += 1             if composite:
return False         a += 1
tests_done += 1     return True
```

BRIEF DESCRIPTION:

This practical uses the Miller–Rabin primality test to determine whether a number is prime. After separating n - 1 into the form d * 2^s, the algorithm tests several bases to see whether they expose n as composite. If any base fails, the number is composite. If all bases pass, the number is considered probably prime. This method is significantly more efficient than simple trial division, especially for large values of n.

Code:

```
def is_prime_miller_rabin(n, k):
    #some base cases
if n < 2:
        return False
if n in (2, 3):
return True     if n %
2 == 0:         return
False

    #n-1 = d*2^s
d = n - 1     s = 0
while d % 2 == 0:
        d //= 2
        s += 1

    #actual process     a = 2
tests_done = 0     while tests_done < k
and a < n - 1:
        x = pow(a, d, n)
if x != 1 and x != n - 1:
            r = 1
composite = True
while r < s:
                x = pow(x, 2, n)
if x == n - 1:
                    composite = False
break                 if x == 1:
                    return False
r += 1              if composite:
return False          a += 1
tests_done += 1      return True

#testing for i in [(2, 3), (5, 1), (97, 3), (9, 3), (1001, 3),
(100000, 3), (341, 2), (2047, 3), (561, 5)]:
    print(f"is_prime_miller_rabin{i} =", is_prime_miller_rabin(i[0],
i[1]))

#runtime
n = int(input("Enter number (n): ")) k =
int(input("Enter number (k): ")) print("Result =",
is_prime_miller_rabin(n, k)) import timeit runtime =
timeit.timeit("is_prime_miller_rabin(n, k)",
globals=globals(), number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage
```

```
import tracemalloc tracemalloc.start()
is_prime_miller_rabin(n, k) peak_memory =
tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
stepsCounter (n, k):
    steps=1
if n < 2:

        steps+=1
return steps
       steps+=2
if n in (2, 3):

        steps+=1
return steps
       steps+=2
if n % 2 == 0:

        steps+=1
        return steps

    d = n - 1
    steps+=2

    s = 0
    steps+=2

    steps+=2
while d % 2 == 0:
        steps+=2

        d //= 2
        steps+=3

        s += 1
        steps+=1

    a = 2
    steps+=1

    tests_done = 0
steps+=1
       steps+=4     while tests_done <
k and a < n - 1:
        steps+=4

        x = pow(a, d, n)
steps+=3
```

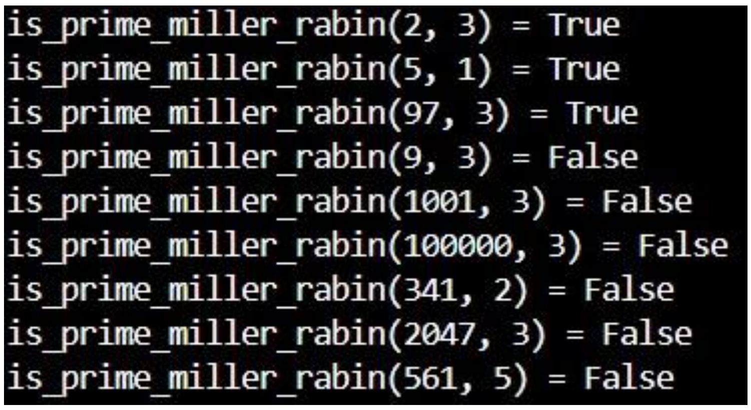
```
        steps+=4         if x !=
1 and x != n - 1:

            r = 1
            steps+=1

            composite = True
steps+=1

            steps+=1
while r < s:
steps+=1

                x = pow(x, 2, n)
steps+=3

                steps+=2
if x == n - 1:

                    composite = False
steps+=1

                    steps+=1
                    break

                steps+=1
if x == 1:

                    steps+=1
return steps

r += 1
steps+=2

steps+=1            if
composite:

                steps+=1
return steps

a += 1
        steps+=2

        tests_done += 1
steps+=1

steps+=1
    return steps

print("Number of steps =", stepsCounter(n, k))
```

RESULTS ACHIEVED:

Test Cases:

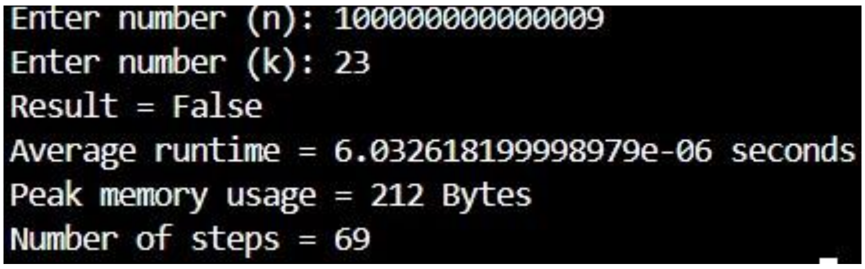
Input (n, k)	**Expected output**
(2, 3)	True
(5, 1)	True
(97, 3)	True
(9, 3)	False
(1001, 3)	False
(1000000, 3)	False
(341, 2)	False
(2047, 3)	False
(561, 5)	False

Testing:



```
is_prime_miller_rabin(2, 3) = True
is_prime_miller_rabin(5, 1) = True
is_prime_miller_rabin(97, 3) = True
is_prime_miller_rabin(9, 3) = False
is_prime_miller_rabin(1001, 3) = False
is_prime_miller_rabin(100000, 3) = False
is_prime_miller_rabin(341, 2) = False
is_prime_miller_rabin(2047, 3) = False
is_prime_miller_rabin(561, 5) = False
```

Performance (Snapshot)**:**



```
Enter number (n): 100000000000009
Enter number (k): 23
Result = False
Average runtime = 6.032618199998979e-06 seconds
Peak memory usage = 212 Bytes
Number of steps = 69
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was ensuring that the decomposition of n - 1 into d and s was done correctly, since the rest of the algorithm relies on this structure. Another challenge was implementing the inner squaring loop properly: forgetting to break when x becomes n - 1 or mishandling the checks involving 1 can cause the algorithm to misclassify primes. Once the flow of the Miller–Rabin rounds was understood and the base-selection logic was tested, the function performed reliably.

SKILLS ACHIEVED:

1. Runtime can easily be improved by sacrificing memory usage

Practical No: 37

Date: 20/11/25

TITLE: Pollard Rho

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm attempts to find a non-trivial factor of n using a simplified version of Pollard’s Rho. It begins by checking whether n is prime using a manual trial-division loop that tests divisibility from 2 up to n/2. If a divisor is found, the function returns immediately with that divisor. If no divisor is found, n is treated as prime and the function returns 1.

The next step handles an important special case: Pollard's Rho performs poorly when n is even, so if n % 2 == 0, the function returns 2 right away.

If neither of the above conditions exits early, the algorithm initializes two values, tortoise and hare, and updates them using the polynomial function: f(x) = (x*x + c) % n

The hare moves twice as fast as the tortoise. At each iteration, their difference diff = hare - tortoise is used to attempt to compute a gcd with n, but since external gcd functions are not allowed, the code manually searches for the gcd by scanning downwards from min(diff, n) until a common divisor is found.

If the computed gcd equals n, the algorithm increases the parameter c and restarts, since this indicates that the polynomial function is cycling in an unproductive way. If the gcd is greater than 1, a non-trivial factor has been found and is returned.

Tools Used:

- Python (no external libraries)
- Manual primality check using trial division
- Handling of the even-number case (n % 2 == 0)
- Pollard's Rho update function using (x*x + c) % n
- Manual gcd computation using a decreasing loop
- Recursion when the chosen polynomial fails (gcd == n)
- Two-pointer technique (tortoise and hare) to detect useful collisions **Code:**

```
def pollard_rho(n, c = 1):
    #is n prime
i=2
is_prime=True
while i<=n/2:
if n%i == 0:
            is_prime=False
            # I should return the value of i here since technically
we have found a multiple of n but the question specifically requires
finding the factor through pollard's rho alogirthm            break
i+=1    if is_prime:        return 1

    #pollard's rho tends to fail for powers of 2
if n%2==0:        return 2

    #pollar's rho process
x = 2    y = 2    i=0
while True:
x = (x**2 + c) % n
y = (((y**2 + c) % n)**2 + c) % n

        #calculating gcd
if x>y:
            diff = x-y
else:
            diff = y-x
if diff>n:
            smaller = n
else:
            smaller = diff
j = smaller//2        gcd
= 1         while j>1:
            if diff%j==0 and n%j==0:
                gcd = j
break            j-=1
               i+=1
if gcd == n:
            return pollard_rho(n, c=c+1)
               if
gcd!=1:
            return gcd
```

BRIEF DESCRIPTION:

This practical implements a basic version of Pollard’s Rho algorithm to find a non-trivial factor of a composite number. It first checks for easy cases such as primality and evenness, then runs the Pollard Rho iteration using the polynomial x^2 + c. The algorithm moves two values through the sequence at different speeds and uses their difference to extract a gcd with n. When a nontrivial gcd is found, it is returned as a factor of the input number.

Code:

```
def pollard_rho(n, c = 1):
    #is n prime
i=2
is_prime=True
while i<=n/2:
if n%i == 0:
            is_prime=False
            # I should return the value of i here since technically
we have found a multiple of n but the question specifically requires
finding the factor through pollard's rho alogirthm            break
i+=1     if is_prime:         return 1

    #pollard's rho tends to fail for powers of 2
if n%2==0:         return 2

    #pollar's rho process
x = 2     y = 2     i=0
while True:
x = (x**2 + c) % n
y = (((y**2 + c) % n)**2 + c) % n

        #calculating gcd
if x>y:
            diff = x-y
else:
            diff = y-x
if diff>n:
            smaller = n
else:
            smaller = diff
j = smaller//2        gcd
= 1          while j>1:
            if diff%j==0 and n%j==0:
                gcd = j
break             j-=1
```

```
        i+=1
if gcd == n:
            return pollard_rho(n, c=c+1)
                if
gcd!=1:
            return gcd


#testing for i in [50, 75, 80, 81, 93, 100, 101, 120, 150,
151, 209]:
    print(f"pollard_rho({i}) =", pollard_rho(i))

#runtime
n = int(input("Enter number (n): ")) print("Result =",
pollard_rho(n)) import timeit runtime =
timeit.timeit("pollard_rho(n)", globals=globals(), number
= 1000)
print("Average runtime =", runtime/1000, "seconds")

#memory usage import tracemalloc
tracemalloc.start() pollard_rho(n) peak_memory =
tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
stepsCounter (n, c=1):
i=2
    steps=1

    is_prime=True
    steps+=1

    steps+=2
while i<=n/2:
steps+=2

        steps+=2
if n%i == 0:

            is_prime=False
            steps+=1

            steps+=1
break

i+=1
        steps+=2

    steps+=1
if is_prime:
        steps+=1
return steps

steps+=2
if n%2==0:

        steps+=1
        return steps

x = 2
    steps+=1

y = 2
    steps+=1

    i=0
    steps+=1

    while True:
steps+=1

x     = (x**2 + c) % n         steps+=3

y     = (((y**2 + c) % n)**2 + c) % n
steps+=6

        steps+=1
if x>y:
            steps+=1

            diff = x-y
            steps+=2

        else:
            diff = y-x
            steps+=1

        steps+=1
if diff>n:
steps+=1

            smaller = n
            steps+=1

        else:
            smaller = diff
            steps+=1

        j = smaller//2
        steps+=3

        gcd = 1
steps+=1
```

```
        steps+=1
while j>1:
steps+=1

            steps+=5             if
diff%j==0 and n%j==0:

                gcd = j
                steps+=1

                steps+=1
                break

            j-=1
steps+=1

i+=1
        steps+=1

        steps+=1
if gcd == n:

            steps+=1
            return stepsCounter(n, c=c+1)

steps+=1
if gcd!=1:

            steps+=1
            return steps

print("Number of steps =", stepsCounter(n))
```

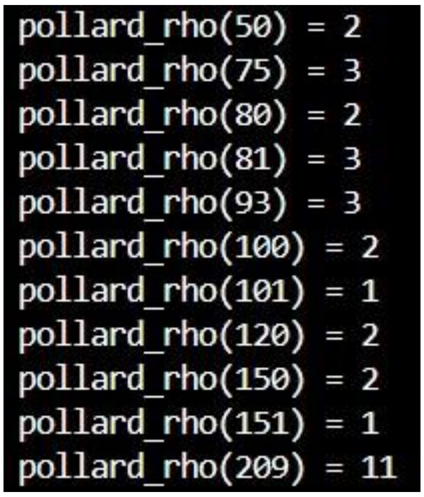
RESULTS ACHIEVED:

Test Cases:

Input (n)	**Expected Output**
50	2
75	3
80	2

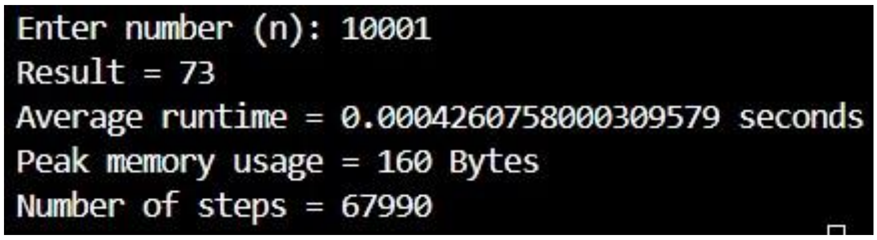
81	3
93	3
100	2
101	1
120	2
150	2
151	1
209	11

Testing:



```
pollard_rho(50) = 2
pollard_rho(75) = 3
pollard_rho(80) = 2
pollard_rho(81) = 3
pollard_rho(93) = 3
pollard_rho(100) = 2
pollard_rho(101) = 1
pollard_rho(120) = 2
pollard_rho(150) = 2
pollard_rho(151) = 1
pollard_rho(209) = 11
```

Performance (Snapshot)**:**



```
Enter number (n): 10001
Result = 73
Average runtime = 0.0004260758000309579 seconds
Peak memory usage = 160 Bytes
Number of steps = 67990
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was implementing Pollard's Rho without access to built-in gcd or random number functions. The manual gcd loop needed careful design to avoid infinite loops and incorrect results. Another challenge was handling cases where the polynomial function fails and produces a gcd equal to n, which required increasing the value of c and restarting the process. Once these issues were resolved and the tortoise-hare updates were tested, the algorithm behaved as expected on composite numbers.

SKILLS ACHIEVED:

1. Recursion can sometimes be far more optimized than loops in examples like gcd calculation

Practical No: 38

Date: <u>20/11/25</u>

TITLE: Zeta Approx

AIM/OBJECTIVE(s):

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

METHODOLOGY & TOOL USED:

The algorithm approximates the Riemann zeta function by summing the first terms values of the series:

1 + 1/2^s + 1/3^s + ...

The function begins by rejecting invalid inputs: if s < 1, it returns None, since the basic form of this series diverges for values below 1. It then initializes a counter i = 1 and a running sum output = 0.

Inside a loop, the algorithm adds i**(-s) to the running total and increments i until it reaches the specified number of terms. Once all terms have been processed, the accumulated value in output is returned as the approximation.

Tools Used:

- Python (no external libraries)
- Conditional check for invalid s values
- A while loop to iterate through the required number of series terms
- Exponentiation using i**(-s)
- A running accumulator variable (output) **Code:**

```
def zeta_approx(s, terms):
if s<1: return None

    i=1
output=0      while
i<=terms:
           output += i**(-s)
i+=1       return output
```

BRIEF DESCRIPTION:

This practical computes an approximation of the Riemann zeta function by truncating its defining infinite series after a fixed number of terms. The algorithm sums the values i**(-s) for i from 1 up to terms and returns the result. This straightforward numerical approach provides a reasonable approximation when the number of terms is sufficiently large.

Code:

```
def zeta_approx(s, terms):
if s<1: return None

    i=1
output=0     while
i<=terms:
        output += i**(-s)
i+=1
    return output


#testing for i in [(2, 1), (2, 5), (2, 100), (3, 5), (3, 50),
(4, 5), (4, 100), (1.5, 10), (1.1, 50), (5, 1), (5, 10), (10,
20)]:     print(f"zeta_approx{i} =", zeta_approx(i[0], i[1]))

#runtime
s = int(input("Enter number (s): ")) terms = int(input("Enter
number (terms): ")) print("Result =", zeta_approx(s, terms)) import
timeit runtime = timeit.timeit("zeta_approx(s, terms)",
globals=globals(), number = 1000000)
print("Average runtime =", runtime/1000000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
zeta_approx(s, terms)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def
stepsCounter (s, terms):
    steps=1
if s<1:


        steps+=1
        return steps

    i=1
    steps+=1

    output=0
    steps+=1

    steps+=1
while i<=terms:
steps+=1

        output += i**(-s)
steps+=4

        i+=1
steps+=2

steps+=1
    return steps

print("Number of steps =", stepsCounter(s, terms))
```

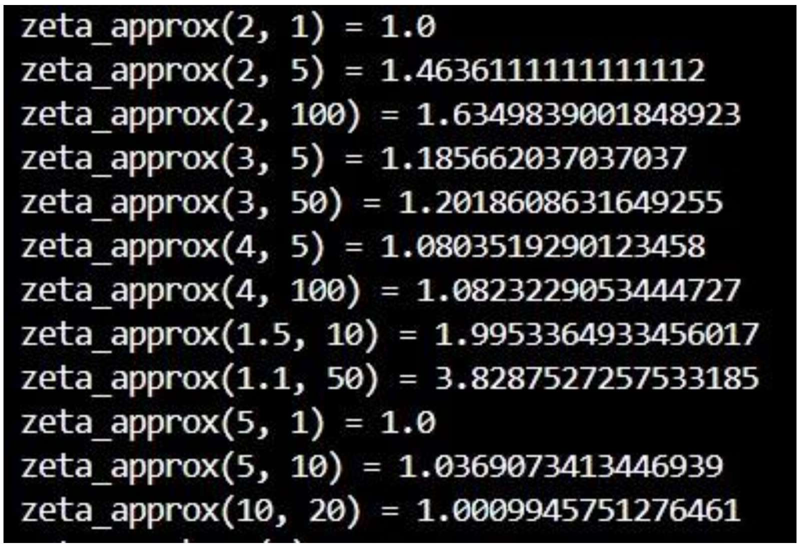
RESULTS ACHIEVED:

Test Cases:

Input (s, terms)	**Expected Output**
(2, 1)	1.0
(2, 5)	1.4636111111111112
(2, 100)	1.6349839001848923
(3, 5)	1.185662037037037
(3, 50)	1.2018608631649255
(4, 5)	1.0803519290123458
(4, 100)	1.0823229053444727
(1.5, 10)	1.9953364933456017
(1.1, 50)	3.8287527257533185

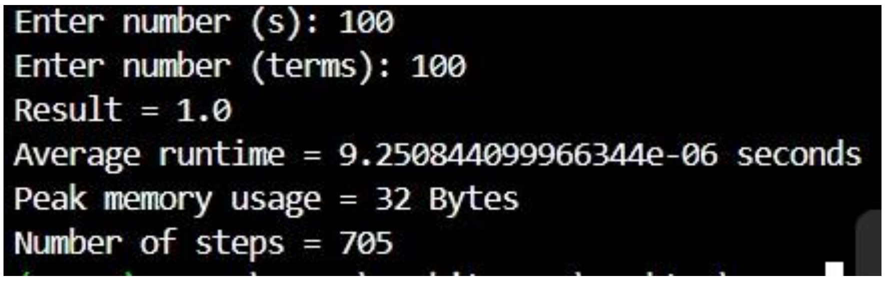
(5, 1)	1.0
(5, 10)	1.0369073413446939
(10, 20)	1.0009945751276461

Testing:



```
zeta_approx(2, 1) = 1.0
zeta_approx(2, 5) = 1.4636111111111112
zeta_approx(2, 100) = 1.6349839001848923
zeta_approx(3, 5) = 1.185662037037037
zeta_approx(3, 50) = 1.2018608631649255
zeta_approx(4, 5) = 1.0803519290123458
zeta_approx(4, 100) = 1.0823229053444727
zeta_approx(1.5, 10) = 1.9953364933456017
zeta_approx(1.1, 50) = 3.8287527257533185
zeta_approx(5, 1) = 1.0
zeta_approx(5, 10) = 1.0369073413446939
zeta_approx(10, 20) = 1.0009945751276461
```

Performance (Snapshot)**:**



```
Enter number (s): 100
Enter number (terms): 100
Result = 1.0
Average runtime = 9.250844099966344e-06 seconds
Peak memory usage = 32 Bytes
Number of steps = 705
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was understanding how fast the series converges for different values of s. For values of s close to 1, many terms are required before the approximation stabilizes, which makes the role of the terms parameter important. Another challenge was remembering that Python allows negative exponents directly in expressions like i**(-s), which simplifies the computation. After testing several combinations of s and terms, the behavior of the approximation became clear.

SKILLS ACHIEVED:

1. If the steps are simple enough we can have a high number of steps but still a low runtime.

Practical No: 39

Date: 20/11/25

TITLE: Partition Function

AIM/OBJECTIVE(s):

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:

The algorithm computes the partition number of n, which represents how many distinct ways n can be written as a sum of positive integers, ignoring order. It uses a recursive approach together with a dictionary to store intermediate results and avoid redundant computation.

The function starts by checking base cases:

- If n < 0, it returns 0.
- If n == 0, it returns 1 because there is exactly one way to partition zero.

To compute the partition count, the algorithm loops through all positive integers k starting from 1 and applies the recursive formula: p(n) = sum(p(n - k)) for all valid k.

Whenever a subproblem p(n - k) is computed, the result is stored in the dictionary so it can be reused later. The loop stops when k becomes greater than n. Once all contributions are summed, the total is returned.

Tools Used:

- Python (no external libraries)
- A dictionary (d) used as a memo table
- Recursion to compute smaller partition values
- A while loop to iterate over valid k values
- Conditional base cases for n < 0 and n == 0

Code:

```
def partition_function (n, slots=-1, maximum=-1):
    if slots==-1:
slots=n     if
maximum==-1:
        maximum=n
        if
n==0:
        return 1
if slots==0:
return 0
        ans=0     i=0     if
n<maximum: smaller = n
else: smaller = maximum
while i<=smaller:
        ans+=partition_function(n-i, slots=slots-1, maximum=i)
i+=1

    return ans
```

BRIEF DESCRIPTION:

This practical computes the partition number of a non-negative integer n. The partition number counts the distinct ways n can be written as a sum of positive integers. The algorithm uses recursion with memoization: each time a subvalue is computed, it is saved so it does not need to be recomputed. This significantly improves performance over a naive recursive approach.

Code:

```
def partition_function (n, slots=-1, maximum=-1):
if slots==-1:         slots=n     if maximum==-1:
maximum=n
        if
n==0:
        return 1
if slots==0:
return 0

ans=0
i=0
    if n<maximum: smaller = n
else: smaller = maximum     while
i<=smaller:
        ans+=partition_function(n-i, slots=slots-1, maximum=i)
i+=1
        return
ans


#testing for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30,
40, 50]:
    print(f"partition_function({i}) =", partition_function(i))

#runtime
n = int(input("Enter number (n): "))
```

```
print("Result =", partition_function(n)) import timeit runtime =
timeit.timeit("partition_function(n)", globals=globals(), number =
10000)
print("Average runtime =", runtime/10000, "seconds")

#memory usage import
tracemalloc
tracemalloc.start()
partition_function(n)
peak_memory = tracemalloc.get_traced_memory()[1] tracemalloc.stop()
print("Peak memory usage =", peak_memory, "Bytes")

#steps counting def stepsCounter (n,
slots=-1, maximum=-1):     steps=1     if
slots==-1:

         slots=n
steps+=1
        steps+=1
if maximum==-1:

         maximum=n
steps+=1
        steps+=1
if n==0:
steps+=1
         return (1, steps)
        steps+=1
if slots==0:

         steps+=1
         return (0, steps)

ans=0
    steps+=1

    i=0
    steps+=1

    steps+=1
if n<maximum:
steps+=1

         smaller = n
         steps+=1

    else:
         smaller = maximum
        steps+=1
steps+=1     while
i<=smaller:
        steps+=1

        response = stepsCounter(n-i, slots=slots-1, maximum=i)
ans+=response[0]         steps+=4         steps+=response[1]

        i+=1
steps+=2
       steps+=1
return (ans, steps)

print("Number of steps =", stepsCounter(n)[1])
```

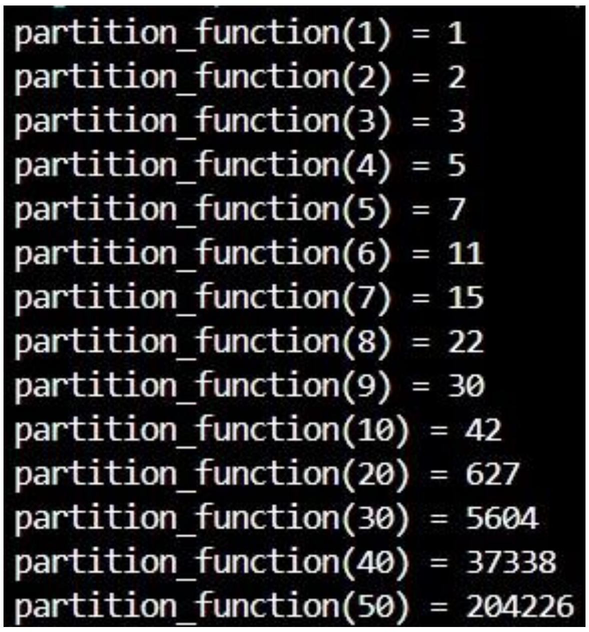
RESULTS ACHIEVED:

Test Cases:

Input (a, n)	**Expected Output**
1	1
2	2
3	3
4	5
5	7
6	11
7	15
8	22

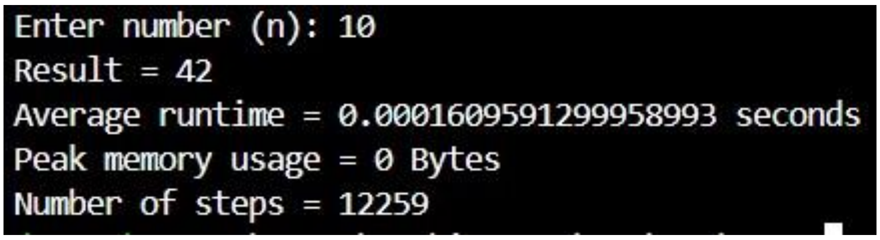
9	30
10	42
20	627
30	5604
40	37338
50	204226

Testing:



```
partition_function(1) = 1
partition_function(2) = 2
partition_function(3) = 3
partition_function(4) = 5
partition_function(5) = 7
partition_function(6) = 11
partition_function(7) = 15
partition_function(8) = 22
partition_function(9) = 30
partition_function(10) = 42
partition_function(20) = 627
partition_function(30) = 5604
partition_function(40) = 37338
partition_function(50) = 204226
```

Performance (Snapshot)**:**



```
Enter number (n): 10
Result = 42
Average runtime = 0.00016095912999589993 seconds
Peak memory usage = 0 Bytes
Number of steps = 12259
```

DIFFICULTY FACED BY STUDENT:

A difficulty in this practical was organizing the recursion so that it always moves toward the base cases and does not create unnecessary calls. Memoization played a big role, and managing the dictionary correctly required attention. Another challenge was making sure the loop terminated at the right point, since trying k values larger than n would lead to invalid recursive calls. After testing several values, the student confirmed that the cached version behaved efficiently.

SKILLS ACHIEVED:

1. Recursion is always inefficient in python.