Date: 06-05-2025

> *1. WAP to `count digit` of a Number.*

```
public static int countDigit(int n)
{
    int count=0;
    while(n!=0)
    {
        count++;
        n/=10;
    }
    return count;
}
```

Method Tracing: `countDigit(int n)`

Purpose

This method counts the number of digits in a given integer `n`.

Initial Setup

- `count` is initialized to `0`
- Input `n` is processed in a loop until it becomes `0`

Tracing Steps

Example 1: `n = 12345`

Iteration	n (before)	Condition (n != 0)	count++	n /= 10 (after)	count (after)
1	12345	true	1	1234	1
2	1234	true	2	123	2
3	123	true	3	12	3
4	12	true	4	1	4
5	1	true	5	0	5
6	0	false (loop ends)	-	-	5
Final Return Value: `5`					

Example 2: `n = 0`

Iteration	n (before)	Condition (n != 0)	count++	n /= 10 (after)	count (after)
1	0	false (loop ends)	-	-	0

Final Return Value: `0`

Example 3: `n = -789` **(Negative Number)**

Iteration	n (before)	Condition (n != 0)	count++	n /= 10 (after)	count (after)
1	-789	true	1	-78	1
2	-78	true	2	-7	2
3	-7	true	3	0	3
4	0	false (loop ends)	-	-	3

Final Return Value: `3`

Key Observations

1. The loop continues until `n` becomes `0`
2. Each iteration:
   - Increments `count` by 1
   - Divides `n` by 10 (integer division)
3. Works for negative numbers (treats them the same as positives)
4. Returns `0` when input is `0` (edge case)
5. Time Complexity: $O(\log_{10} n)$ (number of digits in `n`)

> *2. WAP to find `digital sum` of a digit.*

```
public static int digitSum(int n)
{           int sum =0;
    while(n!=0)
    {
        sum+=n%10;
        n/=10;
    }
    return sum;
}
```

Method Tracing: `digitSum(int n)`

Purpose

This method calculates the sum of all digits in a given integer `n` .

Initial Setup

- `sum` is initialized to `0`
- Input `n` is processed in a loop until it becomes `0`

Tracing Steps

Example 1: `n = 12345`

Iteration	n (before)	Condition (n != 0)	n%10 (digit)	sum += digit	n /= 10 (after)	sum (after)

1	12345	true	5	0 + 5 = 5	1234	5
2	1234	true	4	5 + 4 = 9	123	9
3	123	true	3	9 + 3 = 12	12	12
4	12	true	2	12 + 2 = 14	1	14
5	1	true	1	14 + 1 = 15	0	15
6	0	false (loop ends)	-	-	-	15

Final Return Value: `15`

Example 2: `n = 0`

Iteration	n (before)	Condition (n != 0)	n%10 (digit)	sum += digit	n /= 10 (after)	sum (after)
1	0	false (loop ends)	-	-	-	0

Final Return Value: `0`

Example 3: `n = -789` **(Negative Number)**

Iteration	n (before)	Condition (n != 0)	n%10 (digit)	sum += digit	n /= 10 (after)	sum (after)
1	-789	true	-9	0 + (-9) = -9	-78	-9
2	-78	true	-8	-9 + (-8) = -17	-7	-17
3	-7	true	-7	-17 + (-7) = -24	0	-24
4	0	false (loop ends)	-	-	-	-24

Final Return Value: `-24`
(Note: For negative numbers, the sum will also be negative)

Key Observations

1. The loop continues until `n` becomes `0`
2. Each iteration:
   - Extracts the last digit using `n%10`
   - Adds the digit to `sum`
   - Removes the last digit using `n /= 10`
3. Handles negative numbers (digits contribute negatively to the sum)
4. Returns `0` when input is `0` (edge case)
5. Time Complexity: $O(\log_{10} n)$ (number of digits in `n`)

> *3. WAP to reverse a Digit of Number.*

```java
public static int reverseDigit(int n)
{
    int revNum=0;
    while(n!=0)
    {
        revNum=revNum*10+n%10;
        n/=10;
    }
    return revNum;
}
```

Method Tracing: `reverseDigit(int n)`

Purpose

This method reverses the digits of a given integer `n` (e.g., 1234 → 4321).

Initial Setup

- `revNum` is initialized to `0`
- Input `n` is processed in a loop until it becomes `0`

Tracing Steps

Example 1: `n = 1234`

Iteration	n (before)	Condition (n != 0)	n%10 (digit)	revNum = revNum*10 + digit	n /= 10 (after)	revNum (after)
1	1234	true	4	0*10 + 4 = 4	123	4
2	123	true	3	4*10 + 3 = 43	12	43
3	12	true	2	43*10 + 2 = 432	1	432
4	1	true	1	432*10 + 1 = 4321	0	4321
5	0	false (loop ends)	-	-	-	4321

Final Return Value: `4321`

Example 2: `n = 100`

Iteration	n (before)	Condition (n != 0)	n%10 (digit)	revNum = revNum*10 + digit	n /= 10 (after)	revNum (after)
1	100	true	0	0*10 + 0 = 0	10	0
2	10	true	0	0*10 + 0 = 0	1	0
3	1	true	1	0*10 + 1 = 1	0	1

4	0	false (loop ends)	-	-	-	1

Final Return Value: `1`
(Note: Leading zeros in the original number are dropped in the reversal)

Example 3: `n = -123` **(Negative Number)**

Iteration	n (before)	Condition (n != 0)	n%10 (digit)	revNum = revNum*10 + digit	n /= 10 (after)	revNum (after)
1	-123	true	-3	0*10 + (-3) = -3	-12	-3
2	-12	true	-2	-3*10 + (-2) = -32	-1	-32
3	-1	true	-1	-32*10 + (-1) = -321	0	-321
4	0	false (loop ends)	-	-	-	-321

Final Return Value: `-321`
(Preserves the negative sign while reversing digits)

Key Observations

1. **Digit Extraction**: `n%10` gets the last digit
2. **Number Construction**: `revNum*10 + digit` appends the digit
3. **Termination**: Loop exits when `n` becomes `0`
4. **Handling Negatives**: Maintains sign while reversing digits
5. **Leading Zeros**: Drops leading zeros from original number
6. **Time Complexity**: $O(\log_{10} n)$ (number of digits in `n`)

Special Note

- **Overflow Risk**: For large reversed numbers (e.g., reversing 2147483647 gives 7463847412 which exceeds `Integer.MAX_VALUE`), the result may be incorrect due to integer overflow. This implementation doesn't handle overflow cases.

Date: 07-05-2025

> *4. WAP to find factorial of a number.*

```
public static int factorial(int n)
{
    int fact=1;
    for(int i=2;i<=n;i++)
        fact*=i;
    return fact;

}
```

Method Tracing: `factorial(int n)`

Purpose

This method calculates the factorial of a non-negative integer `n` iteratively.

Mathematical Definition

n! = n × (n-1) × (n-2) × ... × 2 × 1
0! = 1 (by definition)

Initial Setup

- `fact` initialized to `1`
- Loop runs from `i = 2` to `i = n`

Tracing Steps

Example 1: `n = 5`

Iteration	i Value	fact (before)	Operation (fact *= i)	fact (after)
1	2	1	1 × 2 = 2	2
2	3	2	2 × 3 = 6	6
3	4	6	6 × 4 = 24	24
4	5	24	24 × 5 = 120	120

Final Return Value: `120`
(5! = 5 × 4 × 3 × 2 × 1 = 120)

Example 2: `n = 0` **(Edge Case)**

The loop condition `i <= n` (2 <= 0) is false immediately.

Final Return Value: `1`
(0! = 1 by definition)

Example 3: `n = 1` **(Edge Case)**

The loop condition `i <= n` (2 <= 1) is false immediately.

Final Return Value: `1`
(1! = 1)

Key Characteristics

1. **Loop Initialization**: Starts from `i = 2` because:
   - 0! and 1! both equal 1 (handled by initialization)
   - Multiplying by 1 is unnecessary
2. **Termination Condition**: Loop continues while `i <= n`
3. **Efficiency**:
   - Time Complexity: O(n)
   - Space Complexity: O(1) (constant space)

> *5. WAP to calculate Power of a number.*

```
public static int power(int n,int pow)
{
    int res=1;
    for(int i=1;i<=pow;i++)
        res*=n;
    return res;
}
```

Method Tracing: `power(int n, int pow)`

Purpose

This method calculates `n` raised to the power `pow` using iteration.

Mathematical Definition

n^pow = n × n × ... × n (pow times)

Initial Setup

- `res` initialized to `1`
- Loop runs from `i = 1` to `i = pow`

Tracing Steps

Example 1: `n = 2`, `pow = 3`

Iteration	i Value	res (before)	Operation (res *= n)	res (after)
1	1	1	1 × 2 = 2	2
2	2	2	2 × 2 = 4	4
3	3	4	4 × 2 = 8	8

Final Return Value: `8`
($2^3 = 2 \times 2 \times 2 = 8$)

Example 2: `n = 5`, `pow = 0` **(Edge Case)**

The loop condition `i <= pow` (1 <= 0) is false immediately.

Final Return Value: `1`
(Any number to the power 0 is 1 by definition)

Example 3: `n = 3`, `pow = 1` **(Edge Case)**

Iteration	i Value	res (before)	Operation (res *= n)	res (after)
1	1	1	1 × 3 = 3	3

Final Return Value: `3`
($3^1 = 3$)

Key Characteristics

1. **Loop Initialization**: Starts from `i = 1` (inclusive) to `pow` (inclusive)
2. **Base Case Handling**: Returns 1 when pow=0 due to initialization
3. **Efficiency**:
   - Time Complexity: O(pow)
   - Space Complexity: O(1) (constant space)

> *6. WAP to Check number is a perfect number or not.*

```
public static boolean  isPerfectNumber(int n)
{
    int sum = 0;
    for(int i=1;i<=n/2;i++){
        if(n%i==0)
            sum+=i;
    }
    return sum==n;
}
```

Method Tracing: `checkPerfectNumber(int n)`

Purpose

This method checks if a number is a perfect number (a positive integer that equals the sum of its proper divisors).

Mathematical Definition

A perfect number equals the sum of its proper positive divisors (excluding itself).

Initial Setup

- `sum` initialized to `0`
- Loop runs from `i = 1` to `i = n/2`

Tracing Steps

Example 1: `n = 6` (Perfect Number)

Iteration	i Value	n%i	Condition (n%i==0)	sum (before)	sum (after)
1	1	0	true	0	0+1=1
2	2	0	true	1	1+2=3
3	3	0	true	3	3+3=6

Final Comparison: `6 == 6`
Return Value: `true`

Example 2: `n = 28` (Perfect Number)

Iteration	i Value	n%i	Condition	sum (before)	sum (after)

1	1	0	true	0	1
2	2	0	true	1	3
3	4	0	true	3	7
4	7	0	true	7	14
5	14	0	true	14	28

Final Comparison: `28 == 28`
Return Value: `true`

Example 3: `n = 12` **(Not Perfect)**

Iteration	i Value	n%i	Condition	sum (before)	sum (after)
1	1	0	true	0	1
2	2	0	true	1	3
3	3	0	true	3	6
4	4	0	true	6	10
5	5	2	false	10	10
6	6	0	true	10	16

Final Comparison: `16 != 12`
Return Value: `false`

Key Characteristics

1. **Loop Optimization**: Only checks up to `n/2` since no divisor > n/2 exists
2. **Proper Divisors**: Only sums divisors less than `n` (excludes `n` itself)
3. **Efficiency**:
   - Time Complexity: O(n)
   - Space Complexity: O(1)

> *7. WAP to check weather a number is prime or not.*

```
public static boolean  isPrimeNumber(int n)
{
    if (n <= 1)
        return false;
    if (n == 2)
        return true;
    if (n % 2 == 0)
        return false;
    for (int i = 3; i <= Math.sqrt(n); i += 2)
    {
        if (n % i == 0)
            return false;
```

```
        }
        return true;
    }
```

Method Tracing: `checkPrimeNumber(int n)`

Purpose

This method checks if a number is prime (has exactly two distinct positive divisors: 1 and itself).

Mathematical Definition

A prime number is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.

Initial Checks

1. Numbers ≤ 1 → Not prime
2. 2 → Only even prime
3. Even numbers > 2 → Not prime

Tracing Steps

Example 1: `n = 7` (Prime)

1. `7 > 1` → Continue
2. `7 != 2` → Continue
3. `7 % 2 != 0` → Continue
4. Loop: `i` from 3 to √7 (≈2.645)
   - `i=3` : 3 > 2.645 → Loop exits
5. No divisors found → **Returns** `true`

Example 2: `n = 9` (Not Prime)

1. `9 > 1` → Continue
2. `9 != 2` → Continue
3. `9 % 2 != 0` → Continue
4. Loop: `i` from 3 to √9 (3)
   - `i=3` : 9 % 3 == 0 → **Returns** `false`

Example 3: `n = 2` (Edge Case)

1. `2 > 1` → Continue
2. `2 == 2` → **Returns** `true`

Example 4: `n = 1` (Edge Case)

1. `1 <= 1` → **Returns** `false`

Key Characteristics

1. **Early Eliminations**:
   - All numbers ≤ 1: Not prime
   - Even numbers > 2: Not prime
2. **Loop Optimization**:

- Only checks odd divisors (i += 2)
- Only checks up to √n (largest possible factor)

3. **Efficiency**:

   - Time Complexity: O(√n)
   - Space Complexity: O(1)

> *8. WAP to check a year is leap year.*

```
public static boolean  isLeapYear(int year)
{
    return year%4==0 && (year%400==0 || year%100!=0);
}
```

Method Tracing: `checkLeapYear(int year)`

Purpose

This method determines if a given year is a leap year according to the Gregorian calendar rules.

Leap Year Rules

1. **Divisible by 4**: Potential leap year
2. **Exception**: If divisible by 100 → Not leap year unless...
3. **Exception to Exception**: Also divisible by 400 → Leap year

Method Logic

```
return year%4==0 && (year%400==0 || year%100!=0);
```

Tracing Steps

Example 1: `year = 2000` (Leap Year)

1. `2000 % 4 == 0` → true
2. `2000 % 400 == 0` → true
   - Short-circuit evaluation: skips `%100` check
3. **Returns** `true`

Example 2: `year = 1900` (Not Leap Year)

1. `1900 % 4 == 0` → true
2. `1900 % 400 == 0` → false
3. `1900 % 100 != 0` → false
4. **Returns** `false`

Example 3: `year = 2024` (Leap Year)

1. `2024 % 4 == 0` → true
2. `2024 % 400 == 0` → false
3. `2024 % 100 != 0` → true
4. **Returns** `true`

Example 4: `year = 2023` **(Not Leap Year)**

1. `2023 % 4 == 0` → false
   - Short-circuit: skips remaining checks
2. **Returns** `false`

Key Characteristics

1. **Logical Structure**:
   - `year%4==0` : First gate (most common case)
   - `year%400==0` : Exception handler (centuries)
   - `year%100!=0` : Regular leap year confirmation
2. **Short-Circuit Evaluation**:
   - Stops evaluating if first condition (`%4`) fails
   - Within parentheses, stops if `%400` succeeds
3. **Efficiency**:
   - Time Complexity: O(1) (constant time)
   - Space Complexity: O(1)

Date: 08-05-2025

> *9. WAP to check a number is Strong Number or Not.*

```
public static boolean isStrong(int n)
{
    int sum=0;
    int temp=n;
    while(n!=0){
        int rem = n%10;
        sum+=factorial(rem);
        n/=10;
    }
    return sum==temp;

}
```

Method Tracing: `checkStrong(int n)`

Purpose

This method checks if a number is a "Strong number" - a special number where the sum of factorials of its digits equals the number itself.

Mathematical Definition

A number is strong if: sum of factorials of each digit = original number

Components

1. Extracts each digit

2. Calculates factorial of each digit
3. Sums the factorials
4. Compares sum to original number

Tracing Steps

Example 1: `n = 145` **(Strong Number)**

Step	Variable	Value/Operation
Init	sum	0
Init	temp	145
Loop 1	rem = 145%10	5
	sum += factorial(5)	0 + 120 = 120
	n /= 10	14
Loop 2	rem = 14%10	4
	sum += factorial(4)	120 + 24 = 144
	n /= 10	1
Loop 3	rem = 1%10	1
	sum += factorial(1)	144 + 1 = 145
	n /= 10	0
Comparison	sum == temp	145 == 145
Result		**true**

Example 2: `n = 40585` **(Strong Number)**

Step	Variable	Value/Operation
Init	sum	0
Init	temp	40585
Loop 1	rem = 40585%10	5
	sum += factorial(5)	0 + 120 = 120
	n /= 10	4058
Loop 2	rem = 4058%10	8
	sum += factorial(8)	120 + 40320 = 40440
	n /= 10	405
Loop 3	rem = 405%10	5
	sum += factorial(5)	40440 + 120 = 40560

	n /= 10	40
Loop 4	rem = 40%10	0
	sum += factorial(0)	40560 + 1 = 40561
	n /= 10	4
Loop 5	rem = 4%10	4
	sum += factorial(4)	40561 + 24 = 40585
	n /= 10	0
Comparison	sum == temp	40585 == 40585
Result		**true**

Example 3: `n = 123` **(Not Strong)**

Step	Variable	Value/Operation
Init	sum	0
Init	temp	123
Loop 1	rem = 123%10	3
	sum += factorial(3)	0 + 6 = 6
	n /= 10	12
Loop 2	rem = 12%10	2
	sum += factorial(2)	6 + 2 = 8
	n /= 10	1
Loop 3	rem = 1%10	1
	sum += factorial(1)	8 + 1 = 9
	n /= 10	0
Comparison	sum == temp	9 == 123
Result		**false**

Key Characteristics

1. **Digit Extraction**:
    - Uses `n%10` to get last digit
    - Uses `n/10` to remove last digit
2. **Factorial Calculation**:
    - Assumes existence of `factorial()` method

- Pre-calculated factorials (0-9):

  ```
  0! = 1, 1! = 1, 2! = 2, 3! = 6
  4! = 24, 5! = 120, 6! = 720
  7! = 5040, 8! = 40320, 9! = 362880
  ```

3. **Termination**:
   - Loop continues until `n` becomes 0
   - Preserves original number in `temp`

> *10. WAP to check number is neon number or not.*

```java
    public static boolean isNeon(int n)
    {
        int squre=n*n;
        int sum =0;
        while(squre!=0){
            sum+=squre%10;
            squre/=10;
        }
        return sum==n;
    }
```

Method Tracing: `checkNeon(int n)`

Purpose

This method checks if a number is a "Neon number" - a number where the sum of digits of its square equals the number itself.

Mathematical Definition

A number is neon if: sum of digits of (n^2) = n

Components

1. Calculates square of the number
2. Sums digits of the square
3. Compares sum to original number

Tracing Steps

Example 1: `n = 9` (Neon Number)

Step	Variable	Value/Operation
Init	square	9*9 = 81
Init	sum	0
Loop 1	square%10	81%10 = 1

	sum += 1	0 + 1 = 1
	square /= 10	81/10 = 8
Loop 2	8%10	8
	sum += 8	1 + 8 = 9
	square /= 10	8/10 = 0
Comparison	sum == n	9 == 9
Result		**true**

Example 2: `n = 1` **(Neon Number)**

Step	Variable	Value/Operation
Init	square	1*1 = 1
Init	sum	0
Loop 1	1%10	1
	sum += 1	0 + 1 = 1
	square /= 10	1/10 = 0
Comparison	sum == n	1 == 1
Result		**true**

Example 3: `n = 3` **(Not Neon)**

Step	Variable	Value/Operation
Init	square	3*3 = 9
Init	sum	0
Loop 1	9%10	9
	sum += 9	0 + 9 = 9
	square /= 10	9/10 = 0
Comparison	sum == n	9 == 3
Result		**false**

Key Characteristics

1. **Square Calculation**:
   - Computes `n*n` upfront
   - Works for both positive and negative (though negatives would fail comparison)
2. **Digit Summation**:

- Uses modulo 10 to extract last digit
- Uses integer division by 10 to remove last digit

3. **Termination**:

   - Loop continues until square becomes 0
   - Final comparison is exact equality check

> *11. WAP to check a number is happy number or not.*

```
public static void isHappy(int n)
{
    while(n!=1 && n!=4){
        int sum=0;
        while(n!=0){
            int digit = n % 10;
            sum += digit * digit;
            n /= 10;
        }
        n=sum;
    }
    return n==1;
}
```

Method Tracing: `happyNumber(int n)`

Purpose

This method determines if a number is a "Happy Number" - a number that eventually reaches 1 when replaced by the sum of the squares of its digits repeatedly.

Mathematical Definition

A happy number is defined by the process:

1. Start with any positive integer
2. Replace the number by the sum of the squares of its digits
3. Repeat until the number equals 1 (happy) or loops endlessly in a cycle (unhappy)

Key Insight

Unhappy numbers will eventually reach the cycle 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4

Tracing Steps

Example 1: `n = 19` (Happy Number)

Outer Loop	n Value	Inner Loop Operations	sum	New n
1	19	$9^2 + 1^2 = 81 + 1$	82	82
2	82	$2^2 + 8^2 = 4 + 64$	68	68
3	68	$8^2 + 6^2 = 64 + 36$	100	100

4	100	$0^2 + 0^2 + 1^2 = 0+0+1$	1	1
Termination	n=1 → Happy Number			

Example 2: `n = 4` **(Unhappy Number)**

- Immediately terminates (n=4 is in the unhappy cycle)
- **Result**: Unhappy number

Example 3: `n = 2` **(Unhappy Number)**

Outer Loop	n Value	Inner Loop Operations	sum	New n
1	2	$2^2 = 4$	4	4
Termination	n=4 → Unhappy Number			

Key Characteristics

1. **Termination Conditions**:
    - Stops when n becomes 1 (happy) or 4 (unhappy cycle starter)
2. **Digit Processing**:
    - Extracts each digit using `n%10`
    - Removes digit using `n/10`
    - Squares each digit and sums them
3. **Cycle Detection**:
    - Uses 4 as proxy for detecting the unhappy cycle
    - All unhappy numbers eventually reach 4

> *12. WAP to check Armstrong Number or not.*

```
`  public static boolean IsArmStrong(int n)
{
        int count = countDigit(n);
        int sum=0;
        int temp=n;
        while (n!=0) {
            sum+=power(n%10, count);
            n/=10;
        }
        return sum==temp;
    }
```

Method Tracing: `armstrongNumber(int n)`

Purpose

This method checks if a number is an Armstrong number (also called narcissistic number) - a number that equals the sum of its own digits each raised to the power of the number of digits.

Mathematical Definition

An n-digit number is Armstrong if: $\text{digit}_1^n + \text{digit}_2^n + \ldots + \text{digit}_k^n$ = original number

Components

1. Counts digits (`countDigit`)
2. Calculates power of each digit (`power`)
3. Sums the powered digits
4. Compares sum to original number

Tracing Steps

Example 1: `n = 153` (Armstrong Number)

Step	Variable	Value/Operation
Init	count	countDigit(153) → 3
Init	sum	0
Init	temp	153
Loop 1	n%10	153%10 = 3
	power(3,3)	27
	sum += 27	0 + 27 = 27
	n /= 10	15
Loop 2	15%10	5
	power(5,3)	125
	sum += 125	27 + 125 = 152
	n /= 10	1
Loop 3	1%10	1
	power(1,3)	1
	sum += 1	152 + 1 = 153
	n /= 10	0
Comparison	sum == temp	153 == 153
Result		**true**

Example 2: `n = 370` (Armstrong Number)

Step	Variable	Value/Operation
Init	count	3
Init	sum	0

Init	temp	370
Loop 1	370%10	0
	power(0,3)	0
	sum += 0	0
	n /= 10	37
Loop 2	37%10	7
	power(7,3)	343
	sum += 343	0 + 343 = 343
	n /= 10	3
Loop 3	3%10	3
	power(3,3)	27
	sum += 27	343 + 27 = 370
	n /= 10	0
Comparison	370 == 370	
Result		**true**

Example 3: `n = 123` **(Not Armstrong)**

Step	Variable	Value/Operation
Init	count	3
Init	sum	0
Init	temp	123
Loop 1	123%10	3
	power(3,3)	27
	sum += 27	0 + 27 = 27
	n /= 10	12
Loop 2	12%10	2
	power(2,3)	8
	sum += 8	27 + 8 = 35
	n /= 10	1
Loop 3	1%10	1
	power(1,3)	1

	sum += 1	35 + 1 = 36
	n /= 10	0
Comparison	36 == 123	
Result		**false**

Key Characteristics

1. **Digit Counting**:
   - Must count digits before destruction of `n`
   - Uses helper method `countDigit`
2. **Power Calculation**:
   - Each digit raised to digit count power
   - Uses helper method `power`
3. **Termination**:
   - Continues until all digits processed (n=0)
   - Preserves original number in `temp`