Decimal to Binary Conversion

```
    """
    Convert a decimal number to binary representation.

    Core Logic:
    1. Divide the decimal number by 2
    2. Store the remainder in an array
    3. Update the number to be the quotient
    4. Repeat until the quotient becomes 0
    5. The binary number is the remainders read in reverse order

    Args:
    decimal_num (int): The decimal number to convert

    Returns:
    str: Binary representation of the decimal number
    """
```

```
def decimal_to_binary(decimal_num):
    if decimal_num == 0:
        return "0"

    binary_digits = []
    num = decimal_num

    while num > 0:
        remainder = num % 2
        binary_digits.append(str(remainder))
        num = num // 2

    # The binary digits are stored in reverse order
    binary_str = ''.join(reversed(binary_digits))
    return binary_str

# Example usage:
print(decimal_to_binary(10))  # Output: "1010"
print(decimal_to_binary(0))   # Output: "0"
```

```
print(decimal_to_binary(1))   # Output: "1"
print(decimal_to_binary(42))  # Output: "101010"
```

The function works by:

1. Handling the special case of 0 input
2. Repeatedly dividing the number by 2 and storing remainders
3. Reversing the remainders at the end to get the correct binary representation
4. Returning the result as a string
5. Here's a Python program to generate Pascal's Triangle, with the **core logic explained as a comment** at the start:
6. ```
   """
   Core Logic:
   Pascal's Triangle is a triangular array where each row starts and ends with 1,
   and every other element is the sum of the two elements directly above it.
   Mathematically:
       triangle[row][col] = triangle[row-1][col-1] + triangle[row-1][col]
   Steps:
   1. Start with the first row as [1].
   2. For each next row:
       - Start with 1.
       - For each position between start and end:
           Add the two numbers from the previous row's adjacent positions.
       - End with 1.
   3. Repeat until the desired number of rows is generated.
   """

   def generate_pascals_triangle(n):
       triangle = []  # To store all rows

       for row in range(n):
           if row == 0:
               triangle.append([1])
           else:
               prev_row = triangle[-1]
               new_row = [1]  # First element is always 1
               for col in range(1, row):
   ```
   ```

```
                # Sum of two numbers directly above
                new_row.append(prev_row[col - 1] + prev_row[col])
            new_row.append(1)  # Last element is always 1
            triangle.append(new_row)

    return triangle

# Example: Display Pascal's Triangle
rows = 5
triangle = generate_pascals_triangle(rows)
for r in triangle:
    print(r)
```

Fibonacci sequence

```
def fibonacci(n):
    """
    Generate the Fibonacci sequence up to n terms.

    Core Logic:
    1. Handle invalid or small values (n <= 0, n == 1, n == 2) as special cases.
    2. Start with the first two terms: 0 and 1.
    3. From the 3rd term onward, each term = sum of the previous two terms.
    4. Append each new term to the sequence until n terms are generated.

    Args:
    n (int): Number of terms to generate.

    Returns:
    list: List containing the first n terms of the Fibonacci sequence.
    """
```

```
    if n <= 0:
        return "Invalid input"
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

    sequence = [0, 1]  # Start with first two terms

    for i in range(2, n):
        next_term = sequence[i - 1] + sequence[i - 2]
        sequence.append(next_term)

    return sequence


# Example usage:
print(fibonacci(1))   # [0]
print(fibonacci(2))   # [0, 1]
print(fibonacci(5))   # [0, 1, 1, 2, 3]
print(fibonacci(10))  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]


def gcd(a, b):
    """
    Compute the Greatest Common Divisor (GCD) of two integers using the Euclidean algorithm.


    Core Logic:

    1. If b is 0, then GCD is a.

    2. Otherwise, recursively compute GCD of b and the remainder of a divided by b.

    3. This repeats until the remainder is 0, at which point the current divisor is the GCD.
```

```
    Args:
    a (int): First integer
    b (int): Second integer

    Returns:
    int: The GCD of a and b
    """
    while b != 0:
        a, b = b, a % b
    return a


def lcm(a, b):
    """
    Compute the Least Common Multiple (LCM) of two integers based on GCD.

    Core Logic:
    1. Calculate GCD of a and b.
    2. Use the relation: LCM(a, b) * GCD(a, b) = |a * b|
    3. Calculate LCM as absolute product divided by GCD.

    Args:
    a (int): First integer
```

```
    b (int): Second integer

    Returns:
    int: The LCM of a and b
    """
    gcd_value = gcd(a, b)
    return abs(a * b) // gcd_value

# Example usage:
print("GCD of 54 and 24:", gcd(54, 24))  # Output: 6
print("LCM of 54 and 24:", lcm(54, 24))  # Output: 216
print("GCD of 7 and 13:", gcd(7, 13))    # Output: 1 (since 7 and 13 are coprime)
print("LCM of 7 and 13:", lcm(7, 13))    # Output: 91
```

Warmup codes for reference :

Below is the input array. Generate the output as expected below. First and last number must not be changed. Other elements must be sum obtained from addition of given array. Find the pattern and complete

Input array : 6 8 9 7 5

Output array : 6 14 17 16 5

```
def transform_array(arr):
    """
```

Transform the given array based on the pattern:

Core Logic:

1. Keep the first and last elements unchanged.

2. Replace each middle element with the sum of itself and the previous element from the INPUT array.

3. Return the transformed array.

Args:

arr (list): The input list of integers.

Returns:

list: The transformed list following the pattern.

"""

```
# If array length is less than or equal to 2, return as is
if len(arr) <= 2:
    return arr[:]

result = [arr[0]]  # First element stays the same

# Process middle elements
for i in range(1, len(arr) - 1):
    result.append(arr[i] + arr[i - 1])
```

```
    result.append(arr[-1])  # Last element stays the same
    return result

# Example usage:
input_array = [6, 8, 9, 7, 5]
output_array = transform_array(input_array)
print(output_array)  # Output: [6, 14, 17, 16, 5]
```

- For a given decimal number divide the number by 2 and store the remainder in an array and do till the quotient becomes 0 or 1

```
def decimal_to_binary_until_one(decimal_num):
    """
    Convert a decimal number to binary representation, stopping when the quotient is 0 or 1.

    Core Logic:
    1. Divide the decimal number by 2.
    2. Store the remainder in an array.
    3. Repeat the division using the quotient as the new number.
    4. Stop when the quotient becomes 0 or 1.
    5. Append the last quotient (if 1) to the array.
    6. Reverse the collected remainders to get the binary number.

    Args:
    decimal_num (int): The decimal number to convert.

    Returns:
    str: Binary representation of the decimal number.
    """

    if decimal_num == 0:
```

```
    return "0"

  remainders = []
  num = decimal_num

  while num > 1:
    remainders.append(str(num % 2))
    num = num // 2

  # Append the last quotient (which will be 0 or 1)
  remainders.append(str(num))

  return binary_str


# Example usage:
print(decimal_to_binary_until_one(10))  # Output: "0101"
print(decimal_to_binary_until_one(0))   # Output: "0"
print(decimal_to_binary_until_one(1))   # Output: "1"
print(decimal_to_binary_until_one(42))  # Output: "010101"
```