Basic Math Problem

Formula for Farenheit to celsius - $C = (F - 32) \times \frac{5}{9}$

Forula for celsius to Farenheit - $F = C \times \frac{9}{5} + 32$

```
def convert_temperature(temperature, from_unit, to_unit):
    if from_unit == 'C' and to_unit == 'F':
        return temperature * 9/5 + 32
    elif from_unit == 'F' and to_unit == 'C':
        return (temperature - 32) * 5/9
    else:
        raise ValueError("Invalid unit conversion, use 'C' and 'F' only.")

# Example usage
celsius = 25
fahrenheit = convert_temperature(celsius, 'C', 'F')
print(f"{celsius}°C is {fahrenheit}°F")

fahrenheit = 77
celsius = convert_temperature(fahrenheit, 'F', 'C')
print(f"{fahrenheit}°F is {celsius}°C")
```

```
25°C is 77.0°F
77°F is 25.0°C
```

```
def calculate_total_cost(cart):
    total_cost=0
    for item in cart:
        total_cost+=item['price']* item['quantity']

    return total_cost
```

```
## Example cart data

cart=[
    {'name':'Apple','price':0.5,'quantity':4},
    {'name':'Banana','price':0.3,'quantity':6},
    {'name':'Orange','price':0.7,'quantity':3}

]

## calling the function
total_cost=calculate_total_cost(cart)
print(total_cost)
```

```
5.8999999999999995
```

```
import re

def is_valid_email(email):
    # Regular expression pattern for validating an email
    pattern = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'

    # Match the input email with the pattern
    if re.match(pattern, email):
        return True
    else:
        return False

# Example usage
email_to_test = "example@example.com"
if is_valid_email(email_to_test):
    print(f"{email_to_test} is a valid email address.")
else:
    print(f"{email_to_test} is not a valid email address.")
```

```
example@example.com is a valid email address.
```

Regex Breakdown: - ^ asserts the start of the string.

- [a-zA-Z0-9_.+-]+ matches one or more characters that can be a letter (lowercase or uppercase), a number, or one of _.+-.

- @ matches the '@' symbol, which separates the local part and the domain part of an email address.
- [a-zA-Z0-9-]+ matches one or more characters that can be a letter, a number, or a hyphen, representing the domain.
- \. escapes the dot, which is a special character in regex that needs to be matched literally in the domain.
- [a-zA-Z0-9-.]+ matches one or more characters that can be a letter, a number, a dot (for subdomains), or a hyphen.
- $ asserts the end of the string.

```
# Method 1
def sum_of_n_numbers(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total

n = 10  # Example input
print("Sum of first", n, "numbers is:", sum_of_n_numbers(n))

# Method 2
def sum_of_n_numbers(n):
    total = n * (n + 1) // 2  # Using integer division
    return total

n = 10  # Example input
print("Sum of first", n, "numbers is:", sum_of_n_numbers(n))
```

```
Sum of first 10 numbers is: 55
Sum of first 10 numbers is: 55
```

```
def is_even(number):
    return number % 2 == 0

# Example usage:
number = 10
if is_even(number):
    print(f"{number} is even.")
else:
```

```
    print(f"{number} is odd.")
```

```
10 is even.
```

```
def is_prime(n):
    """Check if a number is prime."""
    # Prime numbers are greater than 1
    if n <= 1:
        return False
    # Check divisibility from 2 to the square root of n
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Example usage:
number = 29
if is_prime(number):
    print(f"{number} is a prime number.")
else:
    print(f"{number} is not a prime number.")
```

```
29 is a prime number.
```

```
[i for i in range(50) if is_prime(i)]
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

```
def is_perfect_square(n):
    if n < 0:
        return False  # Negative numbers cannot be perfect squares

    # Start from 0 because 0 * 0 is a perfect square (0)
    i = 0

    # Loop to find an integer i such that i * i == n
    while i * i <= n:
        if i * i == n:
            return True  # Found an integer i such that i^2 == n
```

```
        i += 1  # Move to the next integer

    return False  # If loop ends without finding, n is not a perfect square

# Example usage:
print(is_perfect_square(16))  # True, because 4*4 is 16
print(is_perfect_square(14))  # False, because 14 is not a perfect square
```

```
True
False
```

```
### 1. GCD (Greatest Common Divisor) using the Euclidean Algorithm

# The Euclidean Algorithm for finding the GCD of two numbers is:
# 1. If `b == 0`, then `GCD(a, b) = a`.
# 2. Otherwise, set `a = b` and `b = a % b` and repeat until `b` becomes `0`.
# See hcf is gcd
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def lcm(a, b):
    return abs(a * b) // gcd(a, b)

# Example usage
print(lcm(48, 18))  # Output: 144
```

```
144
```

```
def sum_of_elements(int_list):
    total_sum = 0
    for number in int_list:
        total_sum += number
    return total_sum

# Example usage:
numbers = [1, 2, 3, 4, 5]
result = sum_of_elements(numbers)
print("The sum of elements is:", result)
```

```
The sum of elements is: 15
```

```python
def find_largest_element(int_list):
    # Check if the list is empty and handle accordingly
    if not int_list:
        raise ValueError("The list is empty and has no largest element.")

    # Initialize the largest number with the first element of the list
    largest_number = int_list[0]

    # Iterate through the list starting from the second element
    for number in int_list[1:]:
        if number > largest_number:
            largest_number = number

    return largest_number

# Example usage:
numbers = [3, 7, 2, 9, 5]
largest = find_largest_element(numbers)
print("The largest element is:", largest)
```

```
The largest element is: 9
```

```python
def count_odd_even(lst):
    odd_count = 0
    even_count = 0

    for number in lst:
        if number % 2 == 0:
            even_count += 1
        else:
            odd_count += 1

    return odd_count, even_count

# Example usage
numbers = [10, 21, 4, 45, 66, 93, 1, 0, -2, -5]
odd_count, even_count = count_odd_even(numbers)
print(f"Odd count: {odd_count}")
print(f"Even count: {even_count}")
```

```
Odd count: 5
Even count: 5
```

```python
def max_consecutive_difference(lst):
    # Initialize a variable to store the maximum difference found
    max_diff = 0

    # Loop through the list to compute the difference of consecutive elements
    for i in range(1, len(lst)):
        # Calculate the absolute difference between consecutive elements
        diff = abs(lst[i] - lst[i - 1])

        # Update max_diff if the current difference is larger
        if diff > max_diff:
            max_diff = diff

    return max_diff

# Example usage:
lst = [3, 8, 15, 7, 2, 10]
result = max_consecutive_difference(lst)
print(f"The maximum difference between two consecutive elements is: {result}")
```

```
The maximum difference between two consecutive elements is: 8
```

```python
def merge_lists_to_dictionary(keys, values):
    """
    Merges two lists into a dictionary where elements of the first
    list are keys and elements of the second list are values.

    Parameters:
    keys (list): The list of keys.
    values (list): The list of values.

    Returns:
    dict: A dictionary formed by merging the two lists.
    """
    # Zip the two lists to pair each key with the corresponding value
    # for i in zip(keys, values):
    #     print(i)
    merged_dict = dict(zip(keys, values))
```

```
    return merged_dict

# Example usage:
keys = ['a', 'b', 'c']
values = [1, 2, 3]
result = merge_lists_to_dictionary(keys, values)
print(result)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

```
{'a': 1, 'b': 2, 'c': 3}
```

```
def merge_lists_to_dictionary(keys, values):
    """
    Merges two lists into a dictionary where elements of the first list
    are keys and elements of the second list are values. If the listsb
    are of unequal lengths, the resulting dictionary will only include pairs
    up to the length of the shorter list.

    Parameters:
    keys (list): The list of keys.
    values (list): The list of values.

    Returns:
    dict: A dictionary created by merging the two lists.
    """
    # Create the dictionary using dictionary comprehension with zip
    merged_dict = {key: value for key, value in zip(keys, values)}
    return merged_dict

# Example usage:
keys = ['apple', 'banana', 'cherry']
values = [1, 2, 3]
result = merge_lists_to_dictionary(keys, values)
print(result)  # Output: {'apple': 1, 'banana': 2, 'cherry': 3}
```

```
{'apple': 1, 'banana': 2, 'cherry': 3}
```

```
def merge_three_dictionaries(dict1, dict2, dict3):
    """
    Merges three dictionaries into one.
```

```
    Parameters:
    dict1 (dict): The first dictionary to be merged.
    dict2 (dict): The second dictionary to be merged.
    dict3 (dict): The third dictionary to be merged.

    Returns:
    dict: A dictionary containing all key-value pairs from the
    three input dictionaries.
    If there are duplicate keys, the value from the last
    dictionary with that key will be used.
    """
    # Use dictionary unpacking to merge the dictionaries
    merged_dict = {**dict1, **dict2, **dict3}
    return merged_dict

# Example usage:
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict3 = {'d': 5, 'a': 6}

result = merge_three_dictionaries(dict1, dict2, dict3)
print(result)  # Output: {'a': 6, 'b': 3, 'c': 4, 'd': 5}
```

```
{'a': 6, 'b': 3, 'c': 4, 'd': 5}
```

```
def merge_dicts_with_overlapping_keys(dicts):
    """
    Merges multiple dictionaries into one, summing values for overlapping keys.

    Parameters:
    dicts (list): A list of dictionaries where keys might overlap.

    Returns:
    dict: A dictionary where values for overlapping keys are summed.
    """
    merged_dict = {}

    # Iterate over each dictionary in the list
    for dictionary in dicts:
        # Iterate over each key-value pair in the dictionary
        for key, value in dictionary.items():
```

```
            # Sum values for overlapping keys
            if key in merged_dict:
                merged_dict[key] += value
            else:
                merged_dict[key] = value

    return merged_dict

# Example usage:
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 3, 'c': 4, 'd': 5}
dict3 = {'c': 5, 'd': 6, 'e': 7}

dicts = [dict1, dict2, dict3]
result = merge_dicts_with_overlapping_keys(dicts)
print(result)  # Output: {'a': 1, 'b': 5, 'c': 12, 'd': 11, 'e': 7}
```

```
{'a': 1, 'b': 5, 'c': 12, 'd': 11, 'e': 7}
```

```
def count_occurrences(lst):
    # Create an empty dictionary to store counts of each element
    counts = {}
    for item in lst:
        # Use the `get` method to return the current count of the
        # item, or 0 if the item has not been seen before
        counts[item] = counts.get(item, 0) + 1
    return counts

# Example
lst = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
print(count_occurrences(lst))
# Output: {'apple': 3, 'banana': 2, 'orange': 1}
```

```
{'apple': 3, 'banana': 2, 'orange': 1}
```

```
def group_by_parity(lst):
    # Create a dictionary with lists to hold even and odd numbers
    grouped = {'even': [], 'odd': []}
    for num in lst:
        # Check if the number is even or odd and append it
```

```
        # to the respective list
        if num % 2 == 0:
            grouped['even'].append(num)
        else:
            grouped['odd'].append(num)
    return grouped

# Example
lst = [1, 2, 3, 4, 5, 6]
print(group_by_parity(lst))
# Output: {'even': [2, 4, 6], 'odd': [1, 3, 5]}
```

```
{'even': [2, 4, 6], 'odd': [1, 3, 5]}
```

```
def most_frequent(lst):
    # Create a dictionary to count occurrences of each element
    counts = {}
    for item in lst:
        counts[item] = counts.get(item, 0) + 1
    # Use `max` with a custom key to find the element with the maximum count
    return max(counts, key=counts.get)

# Example
lst = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple']
print(most_frequent(lst))
# Output: 'apple'
```

```
apple
```

```
def map_lists_to_dict(keys, values):
    # Use `zip` to pair elements of `keys` and `values`,
    # then convert to a dictionary
    return dict(zip(keys, values))

# Example
keys = ['name', 'age', 'city']
values = ['Alice', 25, 'New York']
print(map_lists_to_dict(keys, values))
# Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

```python
def invert_dict(d):
    # Use dictionary comprehension to swap keys and values
    return {v: k for k, v in d.items()}

# Example
d = {'a': 1, 'b': 2, 'c': 3}
print(invert_dict(d))
# Output: {1: 'a', 2: 'b', 3: 'c'}
```

```
{1: 'a', 2: 'b', 3: 'c'}
```

```python
def find_common_elements(lst1, lst2):
    # Convert the first list to a dictionary to allow fast membership testing
    dict1 = {item: True for item in lst1}
    # Iterate through the second list and collect common elements
    return [item for item in lst2 if item in dict1]

# Example
lst1 = [1, 2, 3, 4]
lst2 = [3, 4, 5, 6]
print(find_common_elements(lst1, lst2))
# Output: [3, 4]
```

```
[3, 4]
```

```python
def merge_dicts(d1, d2):
    # Copy the first dictionary to avoid altering it
    merged = d1.copy()
    for key, value in d2.items():
        # Sum the value from the second dictionary to the value in
        # the merged dictionary
        merged[key] = merged.get(key, 0) + value
    return merged

# Example
d1 = {'a': 1, 'b': 2, 'c': 3}
d2 = {'b': 3, 'c': 4, 'd': 5}
print(merge_dicts(d1, d2))
# Output: {'a': 1, 'b': 5, 'c': 7, 'd': 5}
```

```
{'a': 1, 'b': 5, 'c': 7, 'd': 5}
```

```
def first_unique(lst):
    # Dictionary to count occurrences of each element
    counts = {}

    for num in lst:
        # Increment the count of the element
        counts[num] = counts.get(num, 0) + 1

    for num in lst:
        # Return the first element with a count of 1
        if counts[num] == 1:
            return num

    # Return None if no unique element is found
    return None

# Example
lst = [4, 5, 1, 2, 1, 4, 5]
print(first_unique(lst))
# Output: 2
```

```
2
```

```
def flatten(nested_list):
    # List to store the flattened elements
    flat_list = []

    for sublist in nested_list:
        # Extend the flat_list by adding elements from each sublist
        flat_list.extend(sublist)

    return flat_list

# Example
nested_list = [[1, 2], [3, 4], [5]]
print(flatten(nested_list))
# Output: [1, 2, 3, 4, 5]
```

```
[1, 2, 3, 4, 5]
```

```
def merge_dict_list(dict_list):
    # Dictionary to hold the merged result
    merged = {}

    for d in dict_list:
        for key, value in d.items():
            # Add the value to the existing value in the dictionary or
            # initialize it
            merged[key] = merged.get(key, 0) + value

    return merged

# Example
dict_list = [{'a': 1, 'b': 2}, {'a': 2, 'c': 3}, {'b': 3, 'c': 1}]
print(merge_dict_list(dict_list))
# Output: {'a': 3, 'b': 5, 'c': 4}
```

```
{'a': 3, 'b': 5, 'c': 4}
```

```
def intersect(lst1, lst2):
    # Dictionary to count occurrences of each element in the first list
    counts = {}

    for num in lst1:
        counts[num] = counts.get(num, 0) + 1

    # List to hold the result of intersection
    result = []

    for num in lst2:
        # If the element is in the dictionary and count is more than 0,
        # add it to the result
        if counts.get(num, 0) > 0:
            result.append(num)
            counts[num] -= 1  # Decrement the count

    return result

# Example
lst1 = [1, 2, 2, 1]
lst2 = [2, 2]
```

```
print(intersect(lst1, lst2))
# Output: [2, 2]
```

```
[2, 2]
```

```
def count_pairs(pairs):
    # Dictionary to hold the frequency of each tuple
    counts = {}

    for pair in pairs:
        # Increment the count of occurrences for each pair
        counts[pair] = counts.get(pair, 0) + 1

    return counts

# Example
pairs = [(1, 2), (2, 3), (1, 2), (3, 4), (2, 3)]
print(count_pairs(pairs))
# Output: {(1, 2): 2, (2, 3): 2, (3, 4): 1}
```

```
{(1, 2): 2, (2, 3): 2, (3, 4): 1}
```

```
def dot_product(d1, d2):
    product = 0
    for key in d1:
        if key in d2:
            product += d1[key] * d2[key]
    return product

# Example
d1 = {'a': 1, 'b': 2, 'c': 3}
d2 = {'a': 4, 'b': 5, 'd': 6}
print(dot_product(d1, d2))
# Output: 14 (because 1*4 + 2*5 = 14)
```

```
14
```

```python
# An Armstrong number (also known as a narcissistic number) is a number
# that is equal to the sum of its own digits each raised to the power of the
# number of digits. For example, the number 153 is an Armstrong number
# because it has 3 digits, and:
# [ 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153 ]

def is_armstrong(number):
    # Convert the number to a string to easily iterate over digits
    digits = str(number)
    # Calculate the number of digits
    power = len(digits)
    # Calculate the sum of each digit raised to the
    # power of the number of digits
    total = sum(int(digit) ** power for digit in digits)
    # Check if the calculated sum equals the original number
    return total == number

# Example usage
number = 153
if is_armstrong(number):
    print(f"{number} is an Armstrong number.")
else:
    print(f"{number} is not an Armstrong number.")

# Additional examples
numbers_to_test = [153, 9474, 9475]
for number in numbers_to_test:
    if is_armstrong(number):
        print(f"{number} is an Armstrong number.")
    else:
        print(f"{number} is not an Armstrong number.")
```

```
153 is an Armstrong number.
153 is an Armstrong number.
9474 is an Armstrong number.
9475 is not an Armstrong number.
```

```python
def reverse_number(n):
    # To handle negative numbers
    negative = n < 0
    if negative:
```

```
        n = -n

    reversed_num = 0
    while n > 0:
        # Get the last digit
        digit = n % 10
        # Append the digit to the reversed number
        reversed_num = reversed_num * 10 + digit
        # Remove the last digit from n
        n //= 10

    # Restore negative sign if the original number was negative
    if negative:
        reversed_num = -reversed_num

    return reversed_num

# Example usage
number = 12345
reversed_number = reverse_number(number)
print("Original number:", number)
print("Reversed number:", reversed_number)
```

```
Original number: 12345
Reversed number: 54321
```

```
def factorial_iterative(n):
    if n < 0:
        return "Undefined for negative numbers"

    result = 1
    for i in range(1, n + 1):
        result *= i  # Multiply result by the current number

    return result

# Sample input and output for iterative approach
number = 5
factorial_result = factorial_iterative(number)
print(f"Iterative: Factorial of {number} is {factorial_result}")
```

```
Iterative: Factorial of 5 is 120
```

```python
def factorial_recursive(n):
    if n < 0:
        return "Undefined for negative numbers"
    elif n == 0 or n == 1:
        return 1  # Base case: factorial(0) and factorial(1) are both 1

    else:
        return n * factorial_recursive(n - 1)  # Recursive case

# Sample input and output for recursive approach
number = 5
factorial_result = factorial_recursive(number)
print(f"Recursive: Factorial of {number} is {factorial_result}")
```

```
Recursive: Factorial of 5 is 120
```

```python
def fibonacci_iterative(n):
    # Handles the first two Fibonacci numbers
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    # Initialize the first two numbers in the sequence
    a, b = 0, 1

    # Iterate until we reach the nth Fibonacci number
    for i in range(2, n + 1):
        a, b = b, a + b
        # b is now the current Fibonacci number

    # Return the nth Fibonacci number
    return b

# Sample input and output
n = 10
print(f"Fibonacci Iterative ({n}): {fibonacci_iterative(n)}")
```

```
Fibonacci Iterative (10): 55
```

```python
def fibonacci_recursive(n):
    # Base cases: return 0 or 1 for the first two Fibonacci numbers
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    # Recursive call to calculate the nth Fibonacci number
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Sample input and output
n = 10
print(f"Fibonacci Recursive ({n}): {fibonacci_recursive(n)}")
```

```
Fibonacci Recursive (10): 55
```

The formula used here, known as **Binet's formula**, is a closed-form expression for calculating Fibonacci numbers without recursion:

$F(n) = \frac{\phi^n - \psi^n}{\sqrt{5}}$

where: - ($\phi = \frac{1+\sqrt{5}}{2}$) is the **golden ratio** (approximately 1.618). - ($\psi = \frac{1-\sqrt{5}}{2}$) is the **conjugate of the golden ratio** (approximately -0.618).

Since (psi^n) approaches zero as n grows (because it's less than 1 in magnitude), we can approximate the nth Fibonacci number using only the *phi* term:

$F(n) \approx \frac{\phi^n}{\sqrt{5}}$

This approximation provides an efficient way to compute Fibonacci numbers, especially for large (n), and the result is rounded to the nearest integer to match the actual sequence values.

```python
import math

def fibonacci_golden_ratio(n):
    phi = (1 + math.sqrt(5)) / 2  # Calculate the golden ratio
    # Calculate the Fibonacci number using the approximation
    fibonacci_approx = (phi**n) / math.sqrt(5)
    # Round the result to get the nearest integer
    return round(fibonacci_approx)

# Sample input and output
```

```
n = 10
approximated_fib = fibonacci_golden_ratio(n)
print(f"Approximated Fibonacci number for n={n}: {approximated_fib}")
```

```
Approximated Fibonacci number for n=10: 55
```

```
def fibonacci_golden_ratio_approx(n):
    phi_approx = 1.618  # Use the approximate golden ratio
    # Calculate the Fibonacci number using the approximation
    fibonacci_approx = (phi_approx**n) / math.sqrt(5)
    # Round the result to get the nearest integer
    return round(fibonacci_approx)

# Sample input and output
n = 10
approximated_fib_approx = fibonacci_golden_ratio_approx(n)
print(f"Fibonacci number for n={n} {approximated_fib_approx}")
```

```
Fibonacci number for n=10 55
```

```
def is_palindrome(s):
    # Convert to lowercase and remove non-alphanumeric characters
    clean_s = ''.join(char.lower() for char in s if char.isalnum())
    # Check if the string reads the same forward and backward
    return clean_s == clean_s[::-1]

# Example usage
string = "A man, a plan, a canal, Panama"
if is_palindrome(string):
    print(f'"{string}" is a palindrome.')
else:
    print(f'"{string}" is not a palindrome.')
```

```
"A man, a plan, a canal, Panama" is a palindrome.
```

```
def is_palindrome(s):
    # Convert to lowercase and remove non-alphanumeric characters
    clean_s = ''.join(char.lower() for char in s if char.isalnum())
```

```
    # Initialize two pointers
    left, right = 0, len(clean_s) - 1

    # Move the pointers towards each other
    while left < right:
        if clean_s[left] != clean_s[right]:
            return False
        left += 1
        right -= 1

    return True

# Example usage
string = "A man, a plan, a canal, Panama"
if is_palindrome(string):
    print(f'"{string}" is a palindrome.')
else:
    print(f'"{string}" is not a palindrome.')
```

```
"A man, a plan, a canal, Panama" is a palindrome.
```

```
def generate_pascals_triangle(n):
    # Initialize the first row of the triangle
    triangle = [[1]]

    for i in range(1, n):
        # Start each row with a 1
        row = [1]
        # Retrieve the previous row from the triangle
        previous_row = triangle[i - 1]

        # Compute the values in the middle of the row
        for j in range(1, i):
            row.append(previous_row[j - 1] + previous_row[j])

        # End each row with a 1
        row.append(1)

        # Append the computed row to the triangle
        triangle.append(row)
```

```
    return triangle

# Example usage
n = 5
triangle = generate_pascals_triangle(n)
for row in triangle:
    print(row)
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

Code Breakdown

1. **Initialize the Triangle:**

   ```
   triangle = [[1]]
   ```

   We start by initializing the `triangle` variable with the first row, which is `[1]`. Pascal's Triangle always starts with a single 1 at the top.

2. **Outer Loop for Rows:**

   ```
   for i in range(1, n):
   ```

   We start a loop that goes from 1 up to (n - 1) (since we already have the first row, we start with `i=1`). This loop will create each new row of the triangle.

3. **Initialize Each Row:**

   ```
   row = [1]
   ```

   Each row in Pascal's Triangle starts with a 1, so we initialize `row` with `[1]`.

4. **Retrieve the Previous Row:**

   ```
   previous_row = triangle[i - 1]
   ```

   To calculate the values in the current row, we use the previous row (`triangle[i - 1]`).

5. **Inner Loop for Middle Values:**

   ```
   for j in range(1, i):
       row.append(previous_row[j - 1] + previous_row[j])
   ```

This loop calculates each middle element in the current row. Each element is the sum of two elements directly above it in the previous row. Specifically, `previous_row[j - 1]` and `previous_row[j]` are the two elements from the previous row that contribute to the new element in the current row.

6. **End Each Row with a 1:**

```
row.append(1)
```

After calculating the middle values, we add a 1 to the end of the row, as each row in Pascal's Triangle ends with a 1.

7. **Append Row to Triangle:**

```
triangle.append(row)
```

We add the newly created `row` to the `triangle`.

8. **Return the Triangle:**

```
return triangle
```

After constructing all rows up to `n`, we return the complete triangle.

Example Walkthrough with (n = 5)

Let's see what each row looks like as we go through the function with `n=5`.

Step-by-Step Rows:

- **Initial State:** `triangle = [[1]]`
- **Iteration 1 (`i = 1`):**
  - `row = [1]`
  - `previous_row = [1]`
  - No middle values (since there's only one element in `previous_row`)
  - `row.append(1)` results in `row = [1, 1]`
  - `triangle = [[1], [1, 1]]`
- **Iteration 2 (`i = 2`):**
  - `row = [1]`
  - `previous_row = [1, 1]`
  - Middle values:
    - `row.append(1 + 1)` → `row = [1, 2]`

- `row.append(1)` results in `row = [1, 2, 1]`
- `triangle = [[1], [1, 1], [1, 2, 1]]`

- **Iteration 3 (`i = 3`):**
  - `row = [1]`
  - `previous_row = [1, 2, 1]`
  - Middle values:
    - `row.append(1 + 2)` → `row = [1, 3]`
    - `row.append(2 + 1)` → `row = [1, 3, 3]`
  - `row.append(1)` results in `row = [1, 3, 3, 1]`
  - `triangle = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]`

- **Iteration 4 (`i = 4`):**
  - `row = [1]`
  - `previous_row = [1, 3, 3, 1]`
  - Middle values:
    - `row.append(1 + 3)` → `row = [1, 4]`
    - `row.append(3 + 3)` → `row = [1, 4, 6]`
    - `row.append(3 + 1)` → `row = [1, 4, 6, 4]`
  - `row.append(1)` results in `row = [1, 4, 6, 4, 1]`
  - `triangle = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]`

Final Output

The generated Pascal's Triangle for (n = 5) is:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

Each row follows the rules of Pascal's Triangle, where each element is the sum of the two elements directly above it in the previous row.

```python
class Solution:
    def generate(self, numRows: int) -> list[list[int]]:
        # This method generates the entire Pascal's Triangle
        # up to the specified number of rows.
        return [self.pascals(i) for i in range(numRows)]
```

```
    def pascals(self, rows: int) -> list[int]:
        # This method calculates a specific row of Pascal's Triangle.
        ans = 1  # Initializing the first binomial coefficient as 1
        final = [1]  # Start each row with 1
        for i in range(rows):
            # Calculate the next element in the row using the relationship
            # between consecutive binomial coefficients:
            # ans = ans * (rows - i) // (i + 1)
            # This efficiently calculates C(rows, i+1) from C(rows, i)
            ans = ans * (rows - i) // (i + 1)
            final.append(ans)  # Append the computed coefficient to the row
        return final  # Return the completed row
```

```
solution = Solution()
numRows = 5
triangle = solution.generate(numRows)
for row in triangle:
    print(row)
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
```

This solution generates Pascal's Triangle up to a given number of rows by breaking down the task into two methods:

1. `generate` method: Generates the entire Pascal's Triangle up to the specified number of rows.
2. `pascals` method: Computes a specific row of Pascal's Triangle using binomial coefficients.

Let's go through the code step-by-step.

```
class Solution:
    def generate(self, numRows: int) -> list[list[int]]:
        # This method generates the entire Pascal's Triangle
        # up to the specified number of rows.
        return [self.pascals(i) for i in range(numRows)]
```

`generate` Method

- **Purpose**: To generate the complete Pascal's Triangle up to the specified number of rows (`numRows`).
- **Parameters**:
  - `numRows` (int): The number of rows to generate in Pascal's Triangle.
- **Process**:
  - It uses a list comprehension to call the `pascals` method for each row index `i` from `0` to `numRows - 1`.
  - `self.pascals(i)` computes the row at index `i` (0-based index).
  - This method returns a list of lists, where each inner list represents a row in Pascal's Triangle.

Example of `generate` Method

Suppose `numRows = 5`, `generate` will call `pascals` for rows `0` through `4` and collect each result into a list:

```
solution = Solution()
print(solution.generate(5))
# Output:
# [
#   [1],
#   [1, 1],
#   [1, 2, 1],
#   [1, 3, 3, 1],
#   [1, 4, 6, 4, 1]
# ]
```

```
    def pascals(self, rows: int) -> list[int]:
        # This method calculates a specific row of Pascal's Triangle.

        ans = 1  # Initializing the first binomial coefficient as 1
        final = [1]  # Start each row with 1
        for i in range(rows):
            # Calculate the next element in the row using the relationship
            # between consecutive binomial coefficients:
```

```
            # ans = ans * (rows - i) // (i + 1)
            # This efficiently calculates C(rows, i+1) from C(rows, i)
            ans = ans * (rows - i) // (i + 1)
            final.append(ans)  # Append the computed coefficient to the row
        return final  # Return the completed row
```

`pascals` Method

- **Purpose**: To calculate a specific row in Pascal's Triangle.
- **Parameters**:
  - `rows` (int): The 0-based row index to compute.
- **Process**:
  1. **Initialization**:
     - `ans` is initialized to `1`, representing the first binomial coefficient (C(rows, 0) = 1).
     - `final` starts with `[1]` because each row in Pascal's Triangle begins with `1`.
  2. **Loop**:
     - For each `i` from `0` to `rows - 1`, we calculate the next binomial coefficient in the row using the relationship:
       $C(\text{rows}, i+1) = \frac{C(\text{rows}, i) \times (\text{rows} - i)}{i+1}$
     - This formula efficiently calculates each element in the row without recalculating from scratch.
     - The computed `ans` is appended to `final`, gradually building up the row.
  3. **Return**:
     - Once all elements are computed, `final` (representing the specific row of Pascal's Triangle) is returned.

Example of `pascals` Method

For `rows = 4`: 1. `final = [1]` 2. Iteration 1 (`i = 0`): `ans = 1 * (4 - 0) // (0 + 1) = 4` → `final = [1, 4]` 3. Iteration 2 (`i = 1`): `ans = 4 * (4 - 1) // (1 + 1) = 6` → `final = [1, 4, 6]` 4. Iteration 3 (`i = 2`): `ans = 6 * (4 - 2) // (2 + 1) = 4` → `final = [1, 4, 6, 4]` 5. Iteration 4 (`i = 3`): `ans = 4 * (4 - 3) // (3 + 1) = 1` → `final = [1, 4, 6, 4, 1]`

The row `[1, 4, 6, 4, 1]` is returned for `rows = 4`.

Overall Output of `generate`

Using the example from above, calling `generate(5)` produces the first five rows of Pascal's Triangle.