1：由于a<=b且b>c的话是没有任何指向的，不做任何操作，所以无输出结果

```python
def print_values(a, b, c):

    # 首先判断：a是否大于b
    if a > b:
        # 当a > b时，判断b是否大于c
        if b > c:
            # a > b 且 b > c → 取x=a, y=b, z=c计算
            x, y, z = a, b, c
            result = x + y - 10 * z
            print(f"计算结果：{result}")
        else:
            # 当a > b且b <= c时，判断a是否大于c
            if a > c:
                # a > b 且 b <= c 且 a > c → 取x=a, y=c, z=b计算
                x, y, z = a, c, b
                result = x + y - 10 * z
                print(f"计算结果：{result}")
            else:
                # a > b 且 b <= c 且 a <= c → 取x=c, y=a, z=b计算
                x, y, z = c, a, b
                result = x + y - 10 * z
                print(f"计算结果：{result}")
    else:
        # 当a <= b时，判断b是否大于c
        if b > c:
            # a <= b 且 b > c 为无指向，不做任何操作
            pass
        else:
            # a <= b 且 b <= c → 取x=c, y=b, z=a计算
            x, y, z = c, b, a
            result = x + y - 10 * z
            print(f"计算结果：{result}")
            print_values(5, 15, 10)  # 运行后无任何结果
```

2：取列表[1,2,3,4,5]进行测试，得到结果为[1,5,7,13,15]

[13]:
```python
import functools
@functools.lru_cache(maxsize=None)
def continuous_ceiling(x):
    if x == 1:
        return 1  # 初始F(1) = 1
    else:
        # 计算 ceil(x/3)其实就是计算 (x + 2) //
        ceil_x3 = (x + 2) // 3
        return continuous_ceiling(ceil_x3) + 2 * x
def process_list(num_list):
    return [continuous_ceiling(x) for x in num_list]
# 进行测试
test_list = [1, 2, 3, 4, 5]
result = process_list(test_list)
print(f"输入列表：{test_list}")
print(f"对应F(x)结果：{result}")
```

```
输入列表：[1, 2, 3, 4, 5]
对应F(x)结果：[1, 5, 7, 13, 15]
```

3-1：取总和数x=10进行测试，得到结果方法数为1

[1]:
```python
def Find_number_of_ways(x):
    # dp[骰子数][总和]，骰子数范围0-10，总和范围0-60
    dp = [[0] * 61 for _ in range(11)]
    dp[0][0] = 1  #0个骰子总和为0的方法数是1
    for dice_count in range(1, 11):  # 骰子个数从1到10
        # 每个骰子至少1，所以dice_count个骰子的总和范围是[dice_count, 6*dice_count]
        for total in range(dice_count, 6 * dice_count + 1):
            for face in range(1, 7):  # 每个骰子的点数是1-6
                # 前dice_count-1个骰子的总和需为total - face，且需满足最小总和（dice_count-1）
                if total - face >= (dice_count - 1):
                    dp[dice_count][total] += dp[dice_count - 1][total - face]

    # 仅当x在10-60之间时返回有效结果，否则返回0
    return dp[10][x] if 10 <= x <= 60 else 0
    # 取x=10（10个骰子都掷出1）的方法数应为1
print("x=10的方法数:", Find_number_of_ways(10))
```

```
x=10的方法数： 1
```

3-2：总和数x=35时，其方法数最大为4395456

```python
#3-2
def solve_3_2():
    Number_of_ways = []
    max_ways = 0
    max_x = 0
    for x in range(10, 61):
        ways = Find_number_of_ways(x)
        Number_of_ways.append(ways)
        # 更新最大方法数和对应的x
        if ways > max_ways:
            max_ways = ways
            max_x = x
    return Number_of_ways, max_x
    # 测试统计并找到最大方法数的x
number_of_ways_list, max_x = solve_3_2()
print("方法数列表（x从10到60）:", number_of_ways_list)
print("最大方法数对应的x:", max_x)
```

```
方法数列表（x从10到60）: [1, 10, 55, 220, 715, 2002, 4995, 11340, 23760, 46420, 85228, 147940, 243925, 383470, 576565, 831204, 1151370, 1535040, 1972630,
2446300, 2930455, 3393610, 3801535, 4121260, 4325310, 4395456, 4325310, 4121260, 3801535, 3393610, 2930455, 2446300, 1972630, 1535040, 1151370, 831204,
576565, 383470, 243925, 147940, 85228, 46420, 23760, 11340, 4995, 2002, 715, 220, 55, 10, 1]
最大方法数对应的x: 35
```

4：输出结果如下图所示，当N>80时开始呈现指数型暴涨
在子集平均值计算中参考了统计基础：样本平均值公式和组合数学中的子集遍历策略以及从GeeksforGeeks - 子集相关算法优化中学习到了算法的优化

```python
#4. Dynamic programming
# 4-1 生成随机整数数组
import random
def Random_integer(N):
    return [random.randint(0, 10) for _ in range(N)]
```

```python
# 4-2 计算所有子集的平均值之和
def Sum_averages(arr):
    n = len(arr)
    if n == 0:
        return 0.0
    sum_arr = sum(arr)
    return sum_arr * (2 ** n - 1) / n
```

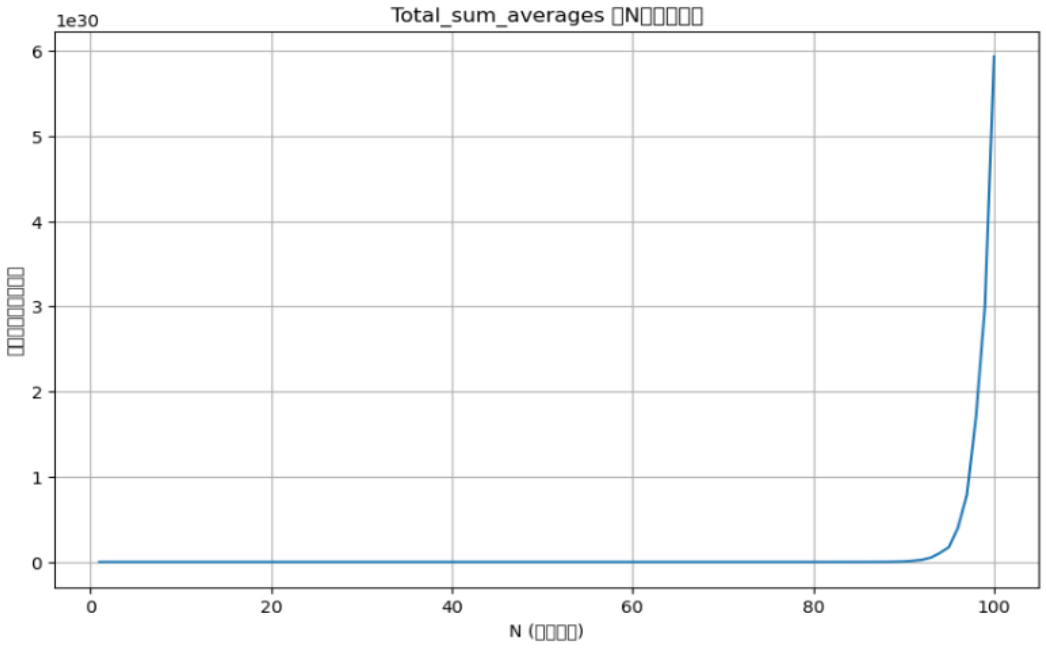
```python
# 4-3 循环计算并绘图
import matplotlib.pyplot as plt

def solve_4_3():
    Total_sum_averages = []
    for N in range(1, 101):
        arr = Random_integer(N)
        total = Sum_averages(arr)
        Total_sum_averages.append(total)

    # 绘制趋势图
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, 101), Total_sum_averages)
    plt.xlabel('N (数组长度)')
    plt.ylabel('所有子集平均值之和')
    plt.title('Total_sum_averages 随N的变化趋势')
    plt.grid(True)
    plt.show()

    return Total_sum_averages

# 测试并查看结果
solve_4_3()
```



5：1000次模拟的平均路径数： 0.142

在Real Python - random 模块进阶中的“Generating Random Integers” 部分，结合二维列表场景学习如何控制随机范围与固定值。
在这道经典的路径计数题LeetCode - 62. 不同路径（经典 DP 入门题）的学习中，我们的路径计数问题是这道题的 “带障碍版”，题解中详细讲解了 “dp[i][j] 表示到 (i,j) 的路径数”“状态转移方程 dp[i][j] = dp[i-1][j] + dp[i][j-1]”，还有边界初始化逻辑（第一行只能右移，路径数恒为 1），完全可迁移。
接着是GeeksforGeeks - 带障碍的路径计数直接对应 “含障碍的路径问题”，代码用 Python 实现，和所使用的 Count_path 函数逻辑高度一致（包括第一行/列的障碍处理、状态转移时的条件判断），以及“算法思路” 部分的分步讲解，特别是“如果当前格子是障碍，dp[i][j] = 0”的逻辑。
NumPy 官方快速入门学习中，简洁讲解“如何用 NumPy 处理数值列表”，比如把 Python 列表 total_paths 转成 NumPy 数组（np.array(total_paths)），再用 np.mean() 求平均，和所使用代码中“np.mean(total_paths)”的用法完全对应。特别是“Basic Operations”部分的“Reduction”小节，重点看了np.mean() 的示例，理解 “为什么用 NumPy 求平均比 Python 内置 sum(total_paths)/len(total_paths) 更高效（尤其数据量大时）”。

[ ]:
```python
#5. Path counting
#5-1
import random

def create_matrix(N, M):
    """
    创建N行M列的矩阵，左上角和右下角固定为1，其余位置随机填充0或1
    """
    matrix = [[0 for _ in range(M)] for _ in range(N)]
    # 固定角落为1
    matrix[0][0] = 1
    matrix[N-1][M-1] = 1
    # 随机填充其余位置
    for i in range(N):
        for j in range(M):
            if (i, j) not in [(0, 0), (N-1, M-1)]:
                matrix[i][j] = random.randint(0, 1)
    return matrix
```

[ ]:
```python
#5-2
def Count_path(matrix):
    #计算从左上角到右下角的路径数（仅可右移或下移，0为障碍）
    #动态规划思路：dp[i][j]表示到(i,j)的路径数
    N, M = len(matrix), len(matrix[0])
    dp = [[0 for _ in range(M)] for _ in range(N)]
    # 初始化左上角
    dp[0][0] = 1 if matrix[0][0] == 1 else 0
    # 处理第一行（只能从左边来）
    for j in range(1, M):
        dp[0][j] = dp[0][j-1] if matrix[0][j] == 1 else 0
    # 处理第一列（只能从上面来）
    for i in range(1, N):
        dp[i][0] = dp[i-1][0] if matrix[i][0] == 1 else 0

    # 处理其余位置（路径数=上方+左方，如果当前格可走的话）
    for i in range(1, N):
        for j in range(1, M):
            dp[i][j] = dp[i-1][j] + dp[i][j-1] if matrix[i][j] == 1 else 0

    return dp[N-1][M-1]
```

[*]:
```python
#5-3
import numpy as np

def solve_5_3():
    #模拟1000次N=10、M=8的矩阵，计算平均路径数
    N, M = 10, 8
    total_paths = []
    for _ in range(1000):
        matrix = create_matrix(N, M)
        paths = Count_path(matrix)
        total_paths.append(paths)
    mean_paths = np.mean(total_paths)
    return mean_paths

# 执行并输出结果
print("1000次模拟的平均路径数：", solve_5_3())
```