In [1]:
```python
#ese5023 ps1 report ( think myself first and get help from gemini )比大小
a = 5
b = 15
c = 10
#定义函数def
def calculate_and_print(x, y, z):
    result = x + y - 10 * z
    #AI help:重点f，另有：
    #1:  +连接:  print(" 计算: " + str(x) + " + " + str(y) + " - 10 * " + str(z)
    #2:  .format() 方法:  print(" 计算: {} + {} - 10 * {} = {}".format(x, y, z,
    print(f"  计算: {x} + {y} - 10 * {z} = {result}")
def  Print_values(a,b,c):
    #右侧命令框，注意下面if需要在def下缩进一格
    if a>b :
        if b > c:
            #a>b>c; and learn the step after print with AI
            print("a,b,c")
            calculate_and_print(a,b,c)
            #break :code run row by row if this is run,the code would stop
        else:
            if a > c:
                print("a,c,b")
                calculate_and_print(a,c,b)
            else:
                print("c,a,b")
                calculate_and_print(c,a,b)
    #左侧b>a
    else:
         if b > c:
            if a > c:
                #b>a>c
                print("there have nothing")
                calculate_and_print(b,a,c)
            else:
                #b>c>a
                print("there have nothing")
                calculate_and_print(b,c,a)
         else:
            print("c,b,a")
            calculate_and_print(c,b,a)
#print(result)，这步需要函数执行
#print(f"{result}")为什么会错误？
Print_values(a,b,c)
```

```
there have nothing
  计算: 15 + 10 - 10 * 5 = -25
```

In [ ]:
```python
```

In [3]:
```python
#work 2 Continuous celing function ( think myself first and get help from gemini
#ask AI 'what is celing'.=>进一法；Recursion
#导入math，才能使用ceil函数;
#在 Python 中，f(i) 这种写法的意思是“调用一个叫做 f 的函数，并把 i 作为参数传给它
import math
#创建列表
input_list = [1,3,4,6,46,78,35]
#print(max)
#列表是否为空
```

```python
#if input_list = []:
if not input_list :
    print("输入列表为空，程序结束")
else:
    #max_val = max(input_list) 本身是完全正确的。问题出在你之前运行过的代码“污染”
    max_val = max(input_list)
    #f(1)=1
    f_results = {1 : 1}
    print(f"需要输出的max值：{max_val}")
    #算最大里面含有其他比它小的数的答案
    for i in range (2 , max_val + 1):
        f_i_index = math.ceil(i/3)

        process_value = f_results[f_i_index]

        F_i = process_value + 2 * i

        f_results[i] = F_i
    print("最大值以及过程值运算完成")

    F_x = []

    for number in input_list:
        f_x = f_results[number]
        F_x.append(f_x)

    print("结果")
    print(f"输入的列表：{input_list}")
    print(f"计算的结果：{F_x}")
```

```
需要输出的max值：78
最大值以及过程值运算完成
结果
输入的列表：[1, 3, 4, 6, 46, 78, 35]
计算的结果：[1, 7, 13, 17, 141, 233, 107]
```

In [ ]:

In [4]:
```python
#work 3 Dice（骰子） rolling ( think myself first and get help from gemini )
#dp方法 vs 矩阵：dp方法复杂度低，更优雅
def solve_dice_problem():
    #骰子个数以及点数
    n_dice  = 10
    n_score = 6
    #可能的最大数
    max_sum_maybe = n_dice * n_score
    #列表推导式，创建我们需要的二维 DP 表；_可以无需输入数而是直接引用
    dp = [[0] * (max_sum_maybe + 1) for _ in range(n_dice +1)]
    #基本定义计算次数
    dp[0][0] = 1
    #开始遍历所有可能，同时在做题时注意要补充j=>骰子个数
    #all dice number
    for i in range(1 , n_dice +1 ):
        #all dice and all sum maybe score，易出错
        for j in range (1 , max_sum_maybe + 1 ):
            #all single dice score
            for z in range (1 , n_score + 1 ):
                #>= right , => error
                if j >= z :
                    #plus and sum
```

```
                    dp[i][j] += dp[i-1][j-z]

#3.1
    def Find_number_of_ways(x):
        #可能点数要大于dice number,don't forget ':'
        if not (n_dice <= x <= max_sum_maybe):
            return 0
        return dp[n_dice][x]
    #输出检查
    print(f"10个骰子和为 10 的方法数: {Find_number_of_ways(10)}")
    print(f"10个骰子和为 35 的方法数: {Find_number_of_ways(35)}")
    print(f"10个骰子和为 60 的方法数: {Find_number_of_ways(60)}")
    # 测试一个无效输入
    print(f"10个骰子和为 7 的方法数: {Find_number_of_ways(7)}")
#定义函数后应该调用它来运行
solve_dice_problem()
```

```
10个骰子和为 10 的方法数: 1
10个骰子和为 35 的方法数: 4395456
10个骰子和为 60 的方法数: 1
10个骰子和为 7 的方法数: 0
```

In [5]:

```
#work 3 Dice rolling ( think myself first and get help from gemini )

def solve_dice_problem():
    #骰子个数以及点数
    n_dice  = 10
    n_score = 6
    #可能的最大数
    max_sum_maybe = n_dice * n_score
    #列表推导式，创建我们需要的二维 DP 表；_可以无需输入数而是直接引用
    dp = [[0] * (max_sum_maybe + 1) for _ in range(n_dice +1)]
    #基本定义计算次数
    dp[0][0] = 1
    #开始遍历所有可能，同时在做题时注意要补充j=>骰子个数
    #all dice number
    for i in range(1 , n_dice +1 ):
        #all dice and all sum maybe score，易出错
        for j in range (1 , max_sum_maybe + 1 ):
            #all single dice score
            for z in range (1 , n_score + 1 ):
                #>= right , => error
                if j >= z :
                    #plus and sum,notice what function set
                    dp[i][j] += dp[i-1][j-z]

#3.1
    def Find_number_of_ways(x):
        #可能点数要大于dice number,don't forget ':'
        if not (n_dice <= x <= max_sum_maybe):
            return 0
        return dp[n_dice][x]
#3.2
    #定义解题列表
    Number_of_ways = []
    #数学上x =[10,60]，编程右侧+1,notice the step would make list star from 1(it
    all_x_range = range(10 , 60 +1 )
    for x in all_x_range:

        ways = Find_number_of_ways(x)
```

```
            #放置数字在列表最后端，记录外加查找
            Number_of_ways.append(ways)
    #set max_index to find.remember add 10
    max_ways  = max(Number_of_ways)
    max_index = Number_of_ways.index(max_ways)
    #recover the place to i think(start from 10)
    x_max_ways = 10 + max_index
    #check the results
    print(f"x 从 10 到 60 的方法数列表 (Number_of_ways):")
    print(f"\n完整的方法数列表 (从x=10到x=60):\n{Number_of_ways}")
    #print(f"10个骰子和为 {all_x_range} 的方法数: {Find_number_of_ways(x)}"),这
    print(f"  列表中的最大方法数为: {max_ways}")
    print(f"  产生这个最大方法数的 x 值是: {x_max_ways}")
#定义函数后应该调用它来运行
solve_dice_problem()
```

```
x 从 10 到 60 的方法数列表 (Number_of_ways):

完整的方法数列表 (从x=10到x=60):
[1, 10, 55, 220, 715, 2002, 4995, 11340, 23760, 46420, 85228, 147940, 243925, 383
470, 576565, 831204, 1151370, 1535040, 1972630, 2446300, 2930455, 3393610, 380153
5, 4121260, 4325310, 4395456, 4325310, 4121260, 3801535, 3393610, 2930455, 244630
0, 1972630, 1535040, 1151370, 831204, 576565, 383470, 243925, 147940, 85228, 4642
0, 23760, 11340, 4995, 2002, 715, 220, 55, 10, 1]
  列表中的最大方法数为: 4395456
  产生这个最大方法数的 x 值是: 35
```

In [ ]:

In [6]:
```
#work 4 Dynamic programming ( think myself first and get help from gemini )
#4.1
import random

def Random_integer(N):
    #  0 到 10 之间（包含 0 和 10）随机选取的整数
    return [random.randint(0, 10) for _ in range(N)]
    #print(f"{Random_integer(N)}"),return后面的代码会无法运行
#检查运行
list_size = 5
generated_list = Random_integer(list_size)
print(f" {list_size} 个随机数的列表是: {generated_list}")
```

```
5 个随机数的列表是: [1, 6, 8, 9, 6]
```

In [7]:
```
#work 4 Dynamic programming ( think myself first and get help from gemini )
import random

def Random_integer(N):
    #  0 到 10 之间（包含 0 和 10）随机选取的整数
    return [random.randint(0, 10) for _ in range(N)]
#4.2
def Sum_averages(arr):
    N = len(arr)#看列表有多少元素
    if N == 0:
        return 0
    arr_sum = sum(arr)
    #计算2的N次方：pow(2,N)
    pow_2_N = pow(2,N)
    total = arr_sum * (pow_2_N - 1) / N
    return total
```

In [8]:

```python
#work 4 Dynamic programming ( think myself first and get help from gemini )
#4.3
#
if __name__ == "__main__":

    # 初始化一个列表来存储 N 从 1 到 100 的所有 Sum_averages 结果
    Total_sum_averages = []

    # 1 到 100（包含 100）
    N_range = range(1, 101)

    print(" N 从 1 到 100 的： ")

    # 循环 N
    for N in N_range:
        # 4.1: 生成一个随机数组
        random_array = Random_integer(N)

        # 4.2: 计算该数组的子集平均值之和
        current_sum = Sum_averages(random_array)

        # 将结果添加到我们的总列表中
        Total_sum_averages.append(current_sum)

    for i in range(100):
        print(f"  N = {i+1}: {Total_sum_averages[i]:.2f}")
    #文字说明
    print("这是一个指数函数，说明需要其他的方法减少计算量，不然面对太多元素的时候
```

N 从 1 到 100 的：
 N = 1: 7.00
 N = 2: 4.50
 N = 3: 35.00
 N = 4: 63.75
 N = 5: 124.00
 N = 6: 325.50
 N = 7: 635.00
 N = 8: 860.62
 N = 9: 2327.89
 N = 10: 4398.90
 N = 11: 12095.91
 N = 12: 22863.75
 N = 13: 36544.46
 N = 14: 92446.93
 N = 15: 163835.00
 N = 16: 385018.12
 N = 17: 655355.00
 N = 18: 1485477.00
 N = 19: 2317900.42
 N = 20: 5085588.75
 N = 21: 10385890.67
 N = 22: 24784517.73
 N = 23: 43037201.13
 N = 24: 68506961.25
 N = 25: 197300054.28
 N = 26: 322638764.42
 N = 27: 591552204.19
 N = 28: 1485981983.04
 N = 29: 2887995245.38
 N = 30: 6335076755.70
 N = 31: 12469259885.81
 N = 32: 24159191034.38
 N = 33: 43730576099.64
 N = 34: 80341152944.03
 N = 35: 160999916919.66
 N = 36: 240518168572.50
 N = 37: 698338466285.08
 N = 38: 1374389534715.00
 N = 39: 2720586463594.64
 N = 40: 5415094766791.88
 N = 41: 13086870106205.95
 N = 42: 23770394238580.50
 N = 43: 45616947533771.19
 N = 44: 96757023244282.50
 N = 45: 184522484732535.91
 N = 46: 347254454963684.81
 N = 47: 670748880672196.75
 N = 48: 1372190511464443.00
 N = 49: 2573485501354564.50
 N = 50: 5742089524897377.00
 N = 51: 12142057818891038.00
 N = 52: 24163544154545540.00
 N = 53: 46565520675453424.00
 N = 54: 77395193596292960.00
 N = 55: 161146983030275200.00
 N = 56: 319112202167966592.00
 N = 57: 753444316607106176.00
 N = 58: 1470968816222528768.00
 N = 59: 2696663858232964096.00

```
N = 60: 5168931412320697344.00
N = 61: 13041243248831547392.00
N = 62: 24099778547910864896.00
N = 63: 38943126377831276544.00
N = 64: 85027960964754964480.00
N = 65: 188440585614509867008.00
N = 66: 338749300262666305536.00
N = 67: 685007451572975566848.00
N = 68: 1480079936267048714240.00
N = 69: 2557948511554390982656.00
N = 70: 6054748454822152765440.00
N = 71: 12969879776895503958016.00
N = 72: 23021536603989520416768.00
N = 73: 50199402612422529253376.00
N = 74: 80918387841604199120896.00
N = 75: 185368625674243125805056.00
N = 76: 369835859290001663590400.00
N = 77: 836042855772454635175936.00
N = 78: 1677772050939533382909952.00
N = 79: 2899891681227496449114112.00
N = 80: 5712174497679122420989952.00
N = 81: 12477308459232468588822528.00
N = 82: 21406832805859043120775168.00
N = 83: 49755308431368352746700800.00
N = 84: 89805918028515317483831296.00
N = 85: 185691005892807034362920960.00
N = 86: 382357933645557126335037440.00
N = 87: 766597926626435138936897536.00
N = 88: 1417300669977296042056482816.00
N = 89: 2983574589064202750298947584.00
N = 90: 5680769291387355759432957952.00
N = 91: 12433815339635577512808939520.00
N = 92: 25996740824992985772881608704.00
N = 93: 46855364927790739651045621760.00
N = 94: 94188799584777016865330298880.00
N = 95: 208495164511221955850114433024.00
N = 96: 389538465695133016624505815040.00
N = 97: 793915195503762222453348106240.00
N = 98: 1555459027320862979787547738112.00
N = 99: 3175528776329302117026069741568.00
N = 100: 5742457219033879504032039436288.00
```

这是一个指数函数，说明需要其他的方法减少计算量，不然面对太多元素的时候容易让电脑无法快速运行

In [ ]:

In [9]:
```python
#work 5 Path counting ( think myself first and get help from gemini )
#5.1
#get ready to set np
import numpy as np
N = 10  # 行数
M = 8   # 列数
#创建矩阵，先行后列
def create_matrix(N,M):
    matrix = np.random.randint(0 , 2 ,size=(N,M),dtype=int)
    #初始点左上角以及终点必为一
    matrix[0,0] = 1
    matrix[N-1,M-1] = 1
    return matrix
#检查运行
```

```
generated_matrix = create_matrix(N, M)
print(f"生成的矩阵是: \n{generated_matrix}")
```

```
生成的矩阵是:
[[1 0 1 1 0 0 0 0]
 [1 0 0 1 0 1 1 0]
 [1 1 0 1 1 1 1 0]
 [0 0 1 0 1 0 0 1]
 [1 1 1 0 0 0 0 0]
 [1 0 1 0 1 1 1 1]
 [0 1 1 0 1 0 1 0]
 [1 0 1 1 1 0 1 0]
 [1 0 1 1 1 0 1 0]
 [0 0 0 0 1 1 1 1]]
```

In [10]:

```python
#work 5 Path counting ( think myself first and get help from gemini )
import numpy as np
N = 10  # 行数
M = 8  # 列数
#创建矩阵，先行后列
def create_matrix(N,M):
    matrix = np.random.randint(0 , 2 ,size=(N,M),dtype=int)
    #初始点左上角以及终点必为一
    matrix[0,0] = 1
    matrix[N-1,M-1] = 1
    return matrix

#5.2
def Count_path(matrix):
    N,M = matrix.shape
    #creat dynamic matrix
    dp = np.zeros((N,M),dtype = int)
    #at the start
    dp[0, 0] = 1
    #only down
    for i in range(1,N):
        #only 1 can run
        if matrix[i,0] == 1:
            dp[i,0] = dp[i - 1 ,0]    #only down
    for i in range(1,N):
        if matrix[i,0] == 1:
            dp[i,0] = dp[i - 1 ,0]
    #left run
    for j in range(1,M):
        #only 1 can run
        if matrix[0,j] == 1:
            dp[0,j] = dp[0,j - 1]

    #run other area
    for i in range(1,N):
        for j in range(1,M):
            if matrix[i,j] == 1:
                dp[i,j] = dp[i-1,j] + dp[i,j-1]
    return dp[N-1,M-1]
#检查
generated_matrix = create_matrix(N, M)
print(f"生成的 {N}x{M} 矩阵是:\n{generated_matrix}")
total_paths = Count_path(generated_matrix)
print(f"从左上角到右下角的路径总数是: {total_paths}")
```

```
生成的 10x8 矩阵是:
[[1 0 1 1 1 0 0 0]
 [1 0 0 1 0 1 1 0]
 [0 1 1 0 0 1 1 0]
 [0 0 1 0 1 0 1 1]
 [1 0 1 1 0 0 0 1]
 [1 1 0 0 0 0 0 1]
 [1 0 0 1 0 1 0 0]
 [1 1 1 1 0 0 0 0]
 [0 0 1 1 0 0 1 0]
 [0 1 1 0 1 0 0 1]]
从左上角到右下角的路径总数是: 0
```

In [11]:

```python
#work 5 Path counting ( think myself first and get help from gemini )
import numpy as np
N = 10  # 行数
M = 8  # 列数
#创建矩阵，先行后列
def create_matrix(N,M):
    matrix = np.random.randint(0 , 2 ,size=(N,M),dtype=int)
    #初始点左上角以及终点必为一
    matrix[0,0] = 1
    matrix[N-1,M-1] = 1
    return matrix

def Count_path(matrix):
    N,M = matrix.shape
    #creat dynamic matrix
    dp = np.zeros((N,M),dtype = int)
    #at the start
    dp[0, 0] = 1
    #only down
    for i in range(1,N):
        #only 1 can run
        if matrix[i,0] == 1:
            dp[i,0] = dp[i - 1 ,0]    #only down
    for i in range(1,N):
        if matrix[i,0] == 1:
            dp[i,0] = dp[i - 1 ,0]
    #left run
    for j in range(1,M):
        #only 1 can run
        if matrix[0,j] == 1:
            dp[0,j] = dp[0,j - 1]

    #run other area
    for i in range(1,N):
        for j in range(1,M):
            if matrix[i,j] == 1:
                dp[i,j] = dp[i-1,j] + dp[i,j-1]
    return dp[N-1,M-1]
#5.3

if __name__ == "__main__":
    num_runs = 1000  # 模拟次数
    # 初始化总路径数
    total_paths = 0
    print(f"开始 {num_runs} 次路径计算...")
    print(f"矩阵大小: N={N}, M={M}")
    # 循环
```

```
    for i in range(num_runs):
        random_matrix = create_matrix(N, M)
        paths_count = Count_path(random_matrix)
        total_paths += paths_count
    # 计算平均
    mean_paths = total_paths / num_runs
    # 结果
    print(f"累计找到的总路径数: {total_paths}")
    print(f"平均路径数: {mean_paths:.2f}") # 2f : 保留两位小数
```

```
开始 1000 次路径计算...
矩阵大小: N=10, M=8
累计找到的总路径数: 374
平均路径数: 0.37
```

In [ ]: