1. Flowchart

```
import numpy as np  # 导入 numpy 库，后期需调用 random 函数
def Print_values(a, b, c):  #定义一个 Print_values 的函数，该函数有三个变量
    if a > b:  # 第一层判断 a > b ，如果为真则进入本段后续判断
        if b > c:  #开始第二层判断 b > c，如果 b > c 为真，对应流程图右下输出 a, b, c，即计算 a + b - 10*c。若为假进入 else
            print(a + b - 10 * c)
        else:  # 第二层判断 b <= c
            if a > c:  # 开始第三层判断 a > c，如果 a > c 为真，对应流程图中右下输出 a, c, b，即计算 a + c - 10*b
                print(a + c - 10 * b)
            else:  # 第三层判断 a <= c，如果 a > c 为假，对应流程图中左下输出 c, a, b，即计算 c + a - 10*b
                print(c + a - 10 * b)
    else:  # 同为第一层判断 a <= b
        if b > c:  # 开始第二层判断判断 b > c?
            if a > c:  # 开始第三层判断如果 b > c 为真，则判断 a > c，如果 a > c 为真，对应流程图左分支中输出 a, c, b，即计算 a + c - 10*b
                print(a + c - 10 * b)
            else:  # 同为第三层判断 a <= c，如果 a > c 为假，对应流程图左分支中输出 c, a, b，即计算 c + a - 10*b
                print(c + a - 10 * b)
        else:  # 同为第二层判断 b <= c，如果 b > c 为假，对应流程图最左下输出 c, b, a，即计算 c + b - 10*a
            print(c + b - 10 * a)
# 使用 numpy 生成随机值测试函数
# 生成三个随机浮点数，范围在 0 到 20 之间，便于观察
random_a = np.random.uniform(0, 20)
random_b = np.random.uniform(0, 20)
random_c = np.random.uniform(0, 20)
print("随机生成的 a, b, c 值：")
print(f"a = {random_a}, b = {random_b}, c = {random_c}")
print("对应输出结果：")
Print_values(random_a, random_b, random_c)
# 指定值测试：a=5, b=15, c=10
Out:随机生成的 a, b, c 值：
a = 11.866521899851989, b = 16.90950234774206, c = 12.813446961892952
对应输出结果：
-144.41505461567567

print("\n 当 a=5, b=15, c=10 时：")
Print_values(5, 15, 10)  # 输出应为 10 + 5 - 10*15 = -135
Out:当 a=5, b=15, c=10 时：-135
```

2. Continuous celing function

```
import numpy as np  # 导入 numpy 库，用于数组操作和数学计算。本题借助 Google 搜索得到了 ceil 函数的含义，由 Grok 告诉我本体实质是递归函数
def F(x):  #定义 F(x)为连续取整递归函数，并且限定 F(1)=1
    if x == 1:  # 限定递归起点 F(1) = 1
        return 1
    else:
        return F(np.ceil(x / 3).astype(int)) + 2 * x  # 这一步在 Grok 的帮助下，注意到 numpy.ceil 返回浮点数，需转换为整数
def list_input(input_numbers): #考虑到题目会同时输入一串列表，所以要定义一个列表函数来容纳这些输入项
    list = []  # 初始化一个空列表，用于存储每个 x 的 F(x)计算结果
    for x in input_numbers:  # 遍历输入列表 numbers 中的每个元素 x
        list.append(F(x))  # 对当前 x 调用 F(x)，并将结果添加到 list 列表末尾
    return list  # 返回包含所有 F(x)值的列表
N = 5
random_input_numbers = np.random.randint(1, 21, size=N)  # 生成 5 个 1 到 20 的随机整数
print("随机生成的列表：")
print(random_input_numbers)
print("对应 F(x) 值：")
print(list_input(random_input_numbers))
```

Out:随机生成的列表：
[3 17 10 5 19]
对应 F(x) 值：
[np.int32(7), np.int64(51), np.int64(33), np.int64(15), np.int64(59)]

3. Dice rolling

```
import numpy as np #导入 numpy 库，本代码参考了知乎 Python 入门案例（七）：模拟掷骰子
def Find_number_of_ways(): #定义一个函数
    total_times = 10000 #投掷一千次骰子
    roll1_arr = np.random.randint(1, 7, size=total_times) #记录骰子的结果，骰子六面，随机范围为 1-7（不含 7）
    roll2_arr = np.random.randint(1, 7, size=total_times)
    roll3_arr = np.random.randint(1, 7, size=total_times)
    roll4_arr = np.random.randint(1, 7, size=total_times)
    roll5_arr = np.random.randint(1, 7, size=total_times)
    roll6_arr = np.random.randint(1, 7, size=total_times)
    roll7_arr = np.random.randint(1, 7, size=total_times)
    roll8_arr = np.random.randint(1, 7, size=total_times)
    roll9_arr = np.random.randint(1, 7, size=total_times)
    roll10_arr = np.random.randint(1, 7, size=total_times)
    total_arr = roll1_arr + roll2_arr + roll3_arr + roll4_arr + roll5_arr + roll6_arr + roll7_arr + roll8_arr + roll9_arr + roll10_arr
    #计算总和 total
    Number_of_ways, x = np.histogram(total_arr, bins=range(10, 61)) #调用 histogram 函数进行频数和频率对应，频数范围为 10-61（不含 61）
    max_ways = max(Number_of_ways)  #找到最大频率
    max_x_index = np.argmax(Number_of_ways)  #找到最大频率的索引，google 搜索到 np.argmax 用于返回一个 numpy 数组中最大值的索引值
    max_x = x[max_x_index]  #找到最大索引值在 x 中的对应量，对应的 x 值（取左边界），找到 Number_of_ways 中最大值对应的 x
    print("Number_of_ways:",Number_of_ways) #输出频率
    print("x:",x) #输出频数
    print("最大频率",max_ways)
    print("对应的 x",max_x)
if __name__ == "__main__":  # 调用主函数
    Find_number_of_ways()
Out:Number_of_ways: [   0    0    0    0    0    1    1    2    5    9    8   27   42   55 104
158 182 256
 318 366 466 544 635 680 720 703 728 723 624 514 491 437 329 285 180 153
  99   56   40   29    9   14    5    2    0    0    0    0    0]
x: [10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
 58 59 60]
最大频率 728
对应的 x 36
```

4. Dynamic programming

```
import numpy as np    #导入 numpy 库，用于随机数生成和数组操作
def Random_integer(N): #定义 Random_integer 函数
    return np.random.randint(0, 11, size=N)    #返回 N 个 0 到 10（包含 0，不含 11）的数构成的数组
def Sum_averages(arr): #定义 Sum_averages 函数用来计算数组的所有非空子集平均值的和
    n = len(arr) #计算出 arr 数组的长度并赋值给 n
    total_sum = 0.0 #定义一个起始为 0 的总和变量 total_sum
    for i in range(1, 1 << n):    #1<<n 表示 2^n，同门石三俊师兄告诉可以这么写来表示 2^n，排除空子集 (i=0)，只有 2^n-1 个
        arr_sum = 0 #考虑到要求平均值引入两个变量，初始化当前子集的元素之和
        arr_count = 0 #初始化当前子集的元素个数
        for j in range(n): #嵌套 loop，因为子集的元素个数是动态变化的，所以需要引入 j 来推进，这部分参考了 Grok 给出的循环建议
            if i & (1 << j):    #  如果第 j 位为 1，包含 arr[j]
                arr_sum += arr[j]
                arr_count += 1
        total_sum += arr_sum / arr_count
    return total_sum
N_numbers = range(1, 101)    # N 从 1 到 100
Total_sum_averages = []    #  存储每个 N 的总和平均值
for N in N_numbers:
    arr = Random_integer(N)
    total_avg = Sum_averages(arr)
    Total_sum_averages.append(total_avg) #将每个 total_avg 添加到 Total_sum_averages 数组里
    print(f"N = {N}, Total_avg = {total_avg}")    #  调试输出
print(Total_sum_averages)
```

Out:N = 1, Total_avg = 7.0
N = 2, Total_avg = 4.5
N = 3, Total_avg = 49.0
N = 4, Total_avg = 112.5
N = 5, Total_avg = 204.60000000000002
N = 6, Total_avg = 357.0000000000001
N = 7, Total_avg = 435.42857142857144
N = 8, Total_avg = 828.7499999999998
N = 9, Total_avg = 2895.666666666666
N = 10, Total_avg = 4501.2

计算量到后期呈现指数级上涨，在此只计算了前十个输出结果。

5. Path counting

```
import numpy as np   #导入 numpy 库，用于矩阵操作和随机数生成
def create_matrix(N, M): #创建一个 N 行 M 列的矩阵
    matrix = np.random.randint(0, 2, size=(N, M))   #先随机填充 0 或 1，再进行改单元格操作
    matrix[0, 0] = 1   #进行矩阵操作左上角赋值为 1
    matrix[N-1, M-1] = 1   #进行矩阵操作右下角赋值为 1
    return matrix #返回矩阵
def Count_path(matrix): #创建路径函数
    N, M = matrix.shape #使用 shape 函数获取矩阵的行和列
    dp = np.zeros((N, M), dtype=int) # 创建动态规划数组，初始化为 0，此处参考了 CSDN 系统性动态规划分类总结（一）：dp 数组定义与递推公式
    if matrix[0, 0] == 1: #判断起点是否，有可能存在[0,0]附近全是阻塞 0
        dp[0, 0] = 1
        #由于要求给定的单元格只能向右或向下移动，所以需要单独对第一行和第一列做一次 for 遍历
    for j in range(1, M): #填充第一行（只能向右）
        if matrix[0, j] == 1 and dp[0, j-1] > 0: #确保当前格可通行，确保前一格有路径
            dp[0, j] = dp[0, j-1]
    for i in range(1, N): #填充第一列（只能向下）
        if matrix[i, 0] == 1 and dp[i-1, 0] > 0: #确保当前格可通行，确保前一格有路径
            dp[i, 0] = dp[i-1, 0]
    for i in range(1, N): # 填充其余单元格
        for j in range(1, M):
            if matrix[i, j] == 1:
                dp[i, j] = dp[i-1, j] + dp[i, j-1]   # 路径数为上方和左方可通行的路径之和
    return dp[N-1, M-1]
N, M = 10, 8   #设置矩阵大小
total_paths = 0   #存储 1000 次运行的路径总数
for _ in range(1000):
    matrix = create_matrix(N, M)
    paths = Count_path(matrix)
    total_paths += paths
average_paths = total_paths / 1000 #计算平均值
print(matrix)
print(f"1000 次运行中路径总数的平均值：{average_paths}")
```

Out： [[1 1 1 1 0 1 1 0]
 [1 1 1 0 1 1 0 0]
 [0 1 1 1 1 0 0 0]
 [1 0 0 0 1 1 1 0]
 [1 1 0 1 0 1 1 0]
 [0 1 0 1 0 0 0 1]
 [0 0 1 0 1 1 0 1]

[0 0 1 0 1 1 1 1]
[0 1 1 0 0 0 1 0]
[0 1 0 1 1 0 0 1]]
1000 次运行中路径总数的平均值：0.417