Lab Report #1

1. Flowchart

1.1. 实验目的

根据给定流程图，编写函数 Print_values(a, b, c)，实现对输入变量的排序与计算操作。

1.2. 实验原理与思路

根据题目所示流程图要求：

1) 输入三个数 a, b, c；
2) 将这三个数放入列表中，并按从大到小排序；
3) 将排序后的值依次赋给变量 x, y, z；
4) 按公式计算结果：

$$result = x + y - 10z$$

5) 打印中间变量与最终结果。

其中，排序操作使用 Python 内置函数 sorted()，通过设置参数 reverse=True 实现降序排列。

1.3. 实验核心代码

```
def Print_values(a, b, c):
    # 1 把 a, b, c 放入列表中并从大到小排序
    sorted_values = sorted([a, b, c], reverse=True)

    # 2 依次赋值给 x, y, z
    x, y, z = sorted_values

    # 3  计算 x + y - 10z
    result = x + y - 10 * z

    # 4 打印结果
    print(f"x = {x}, y = {y}, z = {z}")
    print(f"Computed result (x + y - 10z) = {result}")
```

1.4. 实验结果

输入：

```
Print_values(5, 15, 10)
```

输出：

```
x = 15, y = 10, z = 5
Computed result (x + y - 10z) = -25
```

通过测试可见，程序可以正确执行排序与公式计算操作，输出结果符合预期。

2. Continuous Ceiling Function

2.1. 实验目的

理解递归（recursion）的调用机制，掌握如何在函数中定义自调用的数学递推关系。

2.2. 实验原理与思路

定义递归函数：

$$F(x) = F\left(\left\lceil \frac{x}{3} \right\rceil\right) + 2x, \qquad F(1) = 1$$

其中 $\left\lceil \frac{x}{3} \right\rceil$ 表示对 $\frac{x}{3}$ 取上整（即不小于该值的最小整数）。

实现思路：

1. 当 x=1 时为递归终止条件；
2. 否则按公式调用自身；
3. 输出若干测试值，观察递归增长规律。

其中，向上取整操作使用 Python 的内置函数 math.ceil()进行；

2.3. 实验核心代码

```
import math
def F(x):
    # 基本情况（递归终止条件）
    if x == 1:
        return 1
    else:
        return F(math.ceil(x / 3)) + 2 * x
for i in [1, 2, 3, 4, 9, 10]:
    print(f"F({i}) = {F(i)}")
```

2.4. 实验结果

输出：

```
F(1) = 1
F(2) = 5
F(3) = 7
F(4) = 13
F(9) = 25
F(10) = 33
```

F(x) 的定义展示了复杂函数可通过递归形式实现连续映射关系。每次递归会将问题规模缩小约 1/3；随着 x 增大，结果呈非线性快速增长；该函数体现了“分解—递归—合并”的递推计算思想。

3. Dice Rolling

3.1. 实验目的

通过编程模拟投掷 10 个骰子的所有可能结果，统计每个可能的总点数（10～60）出现的方式数，进而找出哪一个总点数出现的组合数最多，并分析分布特征。

3.2. 实验原理与思路

若采用暴力枚举（即列出所有 6^{10} 种组合），计算量极大，不可行。因此使用动态规划（Dynamic Programming, DP）进行优化。

1) 状态定义

设 $dp[i][j]$ 表示投掷 i 个骰子时，总点数为 j 的组合数。

2) 状态转移方程

对于每个骰子可能的点数 k=1,2, ..., 6：

$$dp[i][j] = \sum_{k=1}^{6} dp[i-1][j-k]$$

即：要得到和为 j 的所有组合，可以由上一个骰子得到的 j−k 点数累积而来。

3) 边界条件

$$dp[0][0] = 1$$

表示“0 个骰子得到点数 0 的方式只有 1 种”（即不投任何骰子）。

4) 算法复杂度

时间复杂度为 O(n×x×6)，远远优于暴力法 $O(6^n)$。

3.3. 实验核心代码

```
import numpy as np
import matplotlib.pyplot as plt
# ============================
# 3.1 定义函数：计算得到指定和 x 的方式数
# ============================
def Find_number_of_ways(n, x):
    """
    使用动态规划计算：
    投 n 个骰子（每个骰子 1~6 点）时，
    总点数为 x 的组合数。
    """
    # dp[i][j] 表示 i 个骰子得到点数 j 的方式数
    dp = np.zeros((n + 1, x + 1), dtype=int)
    dp[0][0] = 1  # 基础情况：0 个骰子得到和为 0 的方式数为 1
    # 填表
    for i in range(1, n + 1):          # 第 i 个骰子
        for j in range(1, x + 1):      # 目标和
            for k in range(1, 7):      # 当前骰子可能的点数（1~6）
```

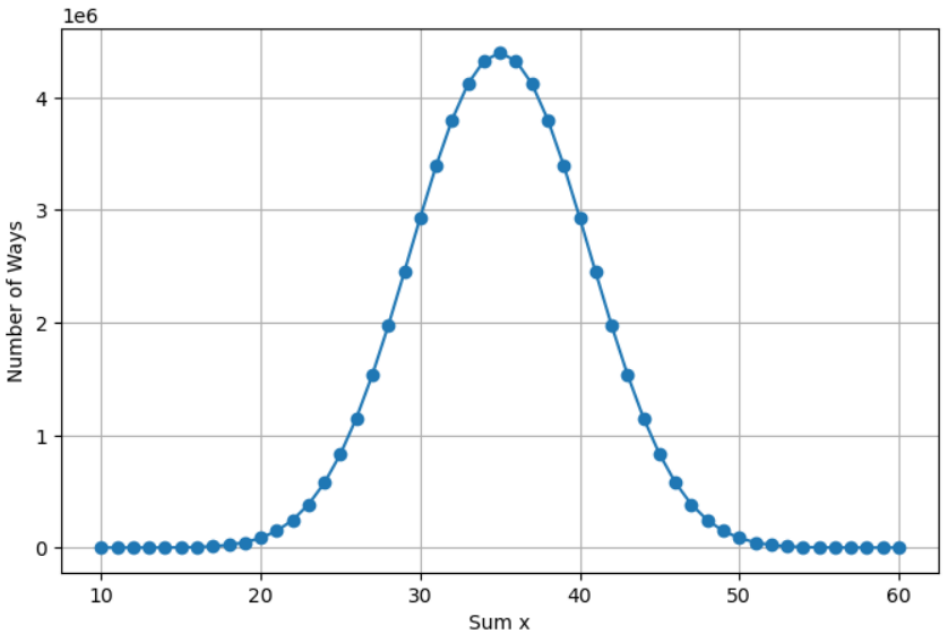
```
19.                 if j - k >= 0:
20.                     dp[i][j] += dp[i - 1][j - k]
21.     return dp[n][x]
```

3.4. 实验结果

输出：

当总和 x = 35 时，出现的组合数最多，为 4395456 种。

```
Group 1          Group 2          Group 3          Group 4          Group 5
x=10: 1          x=11: 10         x=12: 55         x=13: 220        x=14: 715
x=15: 2,002      x=16: 4,995      x=17: 11,340     x=18: 23,760     x=19: 46,420
x=20: 85,228     x=21: 147,940    x=22: 243,925    x=23: 383,470    x=24: 576,565
x=25: 831,204    x=26: 1,151,370  x=27: 1,535,040  x=28: 1,972,630  x=29: 2,446,300
x=30: 2,930,455  x=31: 3,393,610  x=32: 3,801,535  x=33: 4,121,260  x=34: 4,325,310
x=35: 4,395,456  x=36: 4,325,310  x=37: 4,121,260  x=38: 3,801,535  x=39: 3,393,610
x=40: 2,930,455  x=41: 2,446,300  x=42: 1,972,630  x=43: 1,535,040  x=44: 1,151,370
x=45: 831,204    x=46: 576,565    x=47: 383,470    x=48: 243,925    x=49: 147,940
x=50: 85,228     x=51: 46,420     x=52: 23,760     x=53: 11,340     x=54: 4,995
x=55: 2,002      x=56: 715        x=57: 220        x=58: 55         x=59: 10
x=60: 1                    None             None             None             None
```



由结果图可知，曲线呈明显的正态分布，中间值（骰子总和为 35）组合数最多，两端（骰子总和为 10 或 60）组合数极少，因为需要所有骰子都为同一面，组合情况更少。

4. Dynamic programming

4.1. 实验目的

本实验的目标是通过编程实现一个动态规划与组合数学相结合的问题求解过程：

1. 使用 Python 生成随机整数序列；
2. 理解并计算所有非空子集的平均值之和；
3. 掌握从暴力枚举到数学优化的算法提速思想；

4. 通过绘图分析结果的变化趋势。

在上述实验目的的基础上，编写函数 Random_integer()，随机生成长度为 N 的整数数组，整数取值范围为 0–10；编写函数 Sum_averages()，计算数组所有非空子集平均值的总和。令 N 从 1 到 100 递增，记录每个 N 的计算结果并绘图，分析结果变化。

4.2. 实验原理与思路

1) 初始思路（暴力枚举法）

最直接的思路是使用 itertools.combinations(arr, r) 枚举所有大小为 r 的子集，再对每个子集求平均值，进而将所有平均值相加。但是该方法需要遍历 2^{N-1} 个子集，时间复杂度为 $O(2^N)$。当 N>20 时几乎无法计算，N=100 更是完全不可行。

2) 优化思路（动态规划）

设数组为 $A=[a_1,a_2, ..., a_N]$，所有非空子集的平均值之和为：

$$mean(S) = \sum_{S\subseteq A, S\neq\emptyset} \frac{sum(S)}{|S|}$$

对每个元素 a_i 来说，若子集大小为 k，则包含 a_i 的子集数为C_{N-1}^{k-1}；每个大小为 k 的子集对结果的贡献权重为$\frac{1}{k}$，每个 a_i 的总贡献为：

$$ai \times \sum_{k=1}^{N} \frac{C_{N-1}^{k-1}}{k}$$

于是可以得到每个子集均值的总和为：

$$S = \sum_{i=1}^{N}(ai \times \sum_{k=1}^{N} \frac{C_{N-1}^{k-1}}{k}) = (\sum_{i=1}^{N} ai) \times \sum_{k=1}^{N} \frac{C_{N-1}^{k-1}}{k})$$

$$f(N) = \sum_{k=1}^{N} \frac{C_{N-1}^{k-1}}{k})$$

进一步简化式子，得到：

$$S(N) = f(N) \times \sum_{i=1}^{N} ai$$

通过对组合数的递推规律，可以得到$f(N) = f(N-1)$+ 1/N

由此可以得到，算法复杂度从 $O(2^N) \rightarrow O(N)$，可瞬间处理 N=100 的情况。其中组合数的计算可以由 math 内置函数 comb()完成。

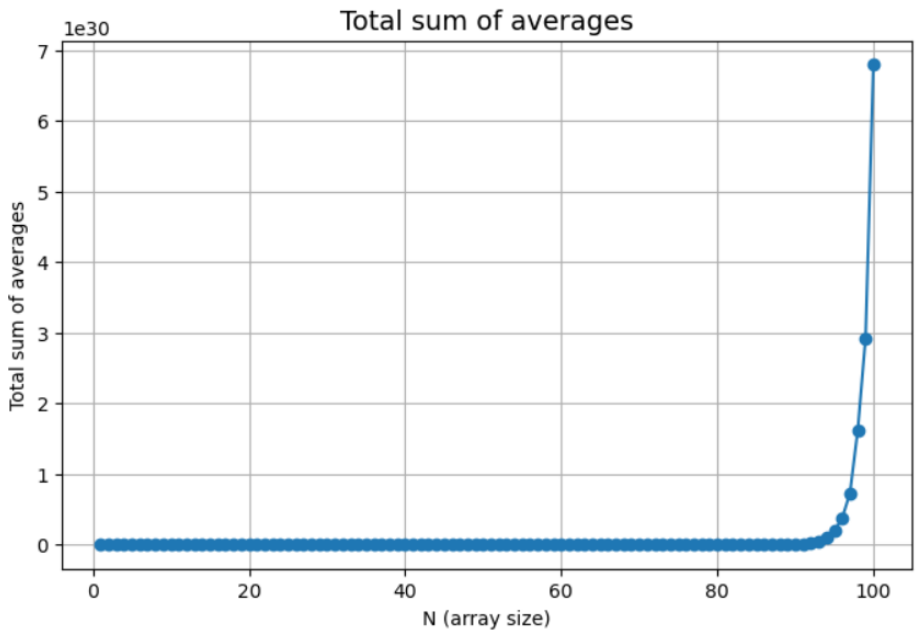
4.3. 实验核心代码

```
1. import numpy as np
2. import math
3. import matplotlib.pyplot as plt
4. # ============================
```

```python
# 4.1 随机整数生成函数
# ============================
def Random_integer(N):
    """随机生成长度为 N 的整数数组，范围 [0,10]"""
    return np.random.randint(0, 11, N)
# ============================
# 4.2 数学优化后的求和函数（O(N)）
# ============================
def Sum_averages(arr):
    """
    使用组合数学优化，避免枚举所有子集。
    返回所有非空子集平均值的总和。
    """
    N = len(arr)
    fN = sum(math.comb(N - 1, k - 1) / k for k in range(1, N + 1))
    return fN * np.sum(arr)
```

4.4. 实验结果

输出：



随着 N 增大，总和快速上升，但计算速度极快；算法复杂度从指数级 (2^N) 降到线性级 (N)。

5. Path counting

5.1. 实验目的

利用动态规划（DP）在路径计数中的核心思想将复杂问题分解为子问题递推求解，实现从左上角到右下角的路径计数算法，并考虑障碍（不可通行的单元格）。通过重复实验（1000

次随机矩阵），统计平均路径数量，分析路径通达性与障碍分布的关系。

5.2. 实验原理与思路

给定一个大小为 N×M 的矩阵 matrix：

- 元素取值为 0 或 1；
- 值为 **1** 的格子表示可通行；
- 值为 **0** 的格子表示障碍或死路；
- 左上角 (0,0) 和右下角 (N-1, M-1) 始终设为 1（确保起点和终点存在）。

要求计算从左上角出发，仅能向右（→） 或 向下（↓） 移动，能到达右下角的不同路径总数。

若没有障碍，网格路径数是组合数问题：

Paths=C_{N+M-2}^{N-1}

但由于障碍的存在，该公式不再适用，需要使用动态规划（Dynamic Programming）递推求解。

定义状态：

$$dp[i][j] = 到达格子(i,j)的路径数量$$

状态转移方程：

$$dp[i][j]\begin{cases}0, & 若matrix[i][j]=0\\ dp[i-1][j]+dp[i][j-1]\end{cases}$$

初始条件：

$$dp[0][0] = 1$$

即最终路径数为：

$$Paths = dp[N-1][M-1]$$

算法复杂度为 O(N×M)

5.3. 实验核心代码

```python
import numpy as np
# ============================
# 5.1 创建随机矩阵函数
# ============================
def Create_matrix(N, M):
    """
    创建一个 N 行 M 列的矩阵。
    - 左上角 (0,0) 和右下角 (N-1,M-1) 填 1。
    - 其他位置随机填 0 或 1。
    """
    matrix = np.random.randint(0, 2, size=(N, M))  # 0 或 1
    matrix[0, 0] = 1           # 起点
    matrix[-1, -1] = 1         # 终点
    return matrix
```

```python
# ============================
# 5.2 计算路径数函数（动态规划）
# ============================
def Count_path(matrix):
    """
    使用动态规划计算从左上角到右下角的路径数量。
    规则：
      - 只能向右或向下移动；
      - 标记为 0 的格子视为障碍；
      - 标记为 1 的格子可通行。
    """
    N, M = matrix.shape
    dp = np.zeros((N, M), dtype=int)
    # 如果起点被封锁，路径数为 0
    if matrix[0, 0] == 0:
        return 0

    # 初始化起点
    dp[0, 0] = 1
    # 填充 DP 矩阵
    for i in range(N):
        for j in range(M):
            if matrix[i, j] == 0:
                dp[i, j] = 0  # 障碍点，无路径
            else:
                if i > 0:
                    dp[i, j] += dp[i - 1, j]  # 来自上方
                if j > 0:
                    dp[i, j] += dp[i, j - 1]  # 来自左侧
    # 返回右下角的路径总数
    return dp[-1, -1]
```

5.4. 实验结果

输出：

在 1000 次随机生成的 10×8 矩阵中，平均路径数量为：0.62

单次运算复杂度仅 O(NM)=80O(NM)=80O(NM)=80，即便重复 1000 次也能快速完成。当矩阵中 1 较多时（通路多），路径数量可能高达数百；当 0 较多时（障碍密集），路径数量往往为趋近于 0。