

Assignment 1

陈禹凡 12232261

1. Flowchart

```
In [1]: import random
x = 0 #将x, y, z, result初始化定义为数值, 这些都将最后输出
y = 0
z = 0
result = 0
a = random.randint(1,100) #在1-100中随机取一个数, 参考CSDN中代码
b = random.randint(1,100)
c = random.randint(1,100)
#反映流程图的函数
def Print_values(a, b, c):
    if (a > b):
        if (b > c):
            x = a
            y = b
            z = c
        else:
            if (a > c):
                x = a
                y = c
                z = b
            else:
                x = c
                y = a
                z = b
    else:
        if (b > c):
            if (a > c):
                x = b
                y = a
                z = c
            else:
                x = b
                y = c
                z = a
        else:
            x = c
            y = b
            z = a
    result = x + y - 10*z
    print('The initial values a,b,c:')
    print(a,b,c)
    print('The output values x,y,z:')
    print(x,y,z)
    print('The result:')
    print(result)
```

```
In [2]: Print_values(a, b, c) #随机值
```

```
The initial values a,b,c:
73 22 50
The output values x,y,z:
73 50 22
The result:
-97
```

```
In [3]: Print_values(10,5,1)#a = 10, b = 5, c = 1
```

```
The initial values a,b,c:
10 5 1
The output values x,y,z:
10 5 1
The result:
5
```

2. Continuous ceiling function

```
In [4]: from math import *
my_list = [5, 8, 39, 21, 2, 19, 10, 98, 1, 4] #给定一个包含N个正整数的列表
# 定义一个F(x) = F(ceil(x/3)) + 2x函数
def F(x):
    y = []
    y.insert(0, 1)
    for j in range(1, x):
        z = ceil(j/3)-1
        result = y[z]+2*(j+1)
        y.insert(j, result)
    print(x, '的计算结果为:', y[x-1])
# 定义一个将列表中所有数值进行计算的函数
def continuous_ceiling(my_list):
    for x in my_list:
        F(x)
continuous_ceiling(my_list)
```

5 的计算结果为: 15
8 的计算结果为: 23
39 的计算结果为: 113
21 的计算结果为: 61
2 的计算结果为: 5
19 的计算结果为: 55
10 的计算结果为: 27
98 的计算结果为: 293
1 的计算结果为: 1
4 的计算结果为: 9

3. Dice rolling

3.1 Find_number_of_ways

```
In [5]: from itertools import product
#定义一个计算函数
def method(x):
    dice_list = [1, 2, 3, 4, 5, 6] #筛子的可能
    #在dice_list中1-6依次取一个值, 重复10次组成一个元组, 共有6的10次方可能
    #product() 函数通过百度搜索得到, 并查明了其用法
    for e in product(dice_list, repeat=10):
        #将e(tuple元组)数据转化成列表(list)数据
        templ_list = list(e)
        #判断和是否等于x
        if (sum(templ_list) == x):
            print(templ_list)
#当x=11时的情况:
method(11)
```

[1, 1, 1, 1, 1, 1, 1, 1, 1, 2]
[1, 1, 1, 1, 1, 1, 1, 1, 2, 1]
[1, 1, 1, 1, 1, 1, 1, 2, 1, 1]
[1, 1, 1, 1, 1, 1, 2, 1, 1, 1]
[1, 1, 1, 1, 1, 2, 1, 1, 1, 1]
[1, 1, 1, 2, 1, 1, 1, 1, 1, 1]
[1, 1, 2, 1, 1, 1, 1, 1, 1, 1]
[1, 2, 1, 1, 1, 1, 1, 1, 1, 1]
[2, 1, 1, 1, 1, 1, 1, 1, 1, 1]

3.2 Number_of_ways

```
In [6]: def method_2(x):
        z = 0 #统计解的个数
        dice_list = [1, 2, 3, 4, 5, 6]
        #重复method_1步骤, 计算每个x的总解数
        for e in product(dice_list, repeat=10):
            temp2_list = list(e)
            if (sum(temp2_list) == x):
                z = z + 1
        return z
#在10-60范围内计算每个x值的总解数
total = 0#定义两个中间变量, 来计算最多解
med = 0
for i in range(10, 61):
    z = method_2(i)#调用函数中的z值
    print('当x=', i, '时, 有', z, '个解')
    #得出最多解的情况
    if z > total:
        total = z
        med = i
print('当x=', med, '含有最多的', total, '个解')
```

```
当x= 10 时, 有 1 个解
当x= 11 时, 有 10 个解
当x= 12 时, 有 55 个解
当x= 13 时, 有 220 个解
当x= 14 时, 有 715 个解
当x= 15 时, 有 2002 个解
当x= 16 时, 有 4995 个解
当x= 17 时, 有 11340 个解
当x= 18 时, 有 23760 个解
当x= 19 时, 有 46420 个解
当x= 20 时, 有 85228 个解
当x= 21 时, 有 147940 个解
当x= 22 时, 有 243925 个解
当x= 23 时, 有 383470 个解
当x= 24 时, 有 576565 个解
当x= 25 时, 有 831204 个解
当x= 26 时, 有 1151370 个解
当x= 27 时, 有 1535040 个解
当x= 28 时, 有 1972630 个解
当x= 29 时, 有 2446300 个解
当x= 30 时, 有 2930455 个解
当x= 31 时, 有 3393610 个解
当x= 32 时, 有 3801535 个解
当x= 33 时, 有 4121260 个解
当x= 34 时, 有 4325310 个解
当x= 35 时, 有 4395456 个解
当x= 36 时, 有 4325310 个解
当x= 37 时, 有 4121260 个解
当x= 38 时, 有 3801535 个解
当x= 39 时, 有 3393610 个解
当x= 40 时, 有 2930455 个解
当x= 41 时, 有 2446300 个解
当x= 42 时, 有 1972630 个解
当x= 43 时, 有 1535040 个解
当x= 44 时, 有 1151370 个解
当x= 45 时, 有 831204 个解
当x= 46 时, 有 576565 个解
当x= 47 时, 有 383470 个解
当x= 48 时, 有 243925 个解
当x= 49 时, 有 147940 个解
当x= 50 时, 有 85228 个解
当x= 51 时, 有 46420 个解
当x= 52 时, 有 23760 个解
当x= 53 时, 有 11340 个解
当x= 54 时, 有 4995 个解
当x= 55 时, 有 2002 个解
当x= 56 时, 有 715 个解
当x= 57 时, 有 220 个解
当x= 58 时, 有 55 个解
当x= 59 时, 有 10 个解
当x= 60 时, 有 1 个解
当x= 35 含有最多的 4395456 个解
```

4. Dynamic programming

4.1 Random_integer

```
In [7]: import random
#定义一个生成数组的函数
def Random_integer(N):
    #N为数组个数
    Random_integer = []#用于存储数组
    for i in range(0,N):
        a = random.randint(1,10)#随机生成一个数组
        Random_integer.insert(i,a)#插入到列表中
    return Random_integer
Random_integer(5)
```

Out[7]: [4, 9, 7, 7, 10]

4.2 Sum_averages

```
In [8]: import itertools
from math import *
#定义一个计算数组子集平均的值的函数
def Sum_averages(mylist):
    #print ('随机给定的数组:',mylist)#用于验证结果是否正确,但因为4.3需要用到该公式,暂不print
    average = 0#给定一个平均的初始值
    #循环计算每个子集的平均值
    for i in range(0,len(mylist)):
        for j in itertools.combinations(mylist,i+1):#i+1为子集中元素个数
            #combinations函数为自己百度找到的,用于挑选出子集(不包含重复元素)
            temp3_list = list(j) #将挑选出的j元组list存于temp3_list中
            n = len(temp3_list)
            average = average+(sum(temp3_list)/ n) #循环计算每个子集的平均的和
    #print('所有子集的平均的和:',average)#用于验证结果是否正确,但因为4.3需要用到该公式,暂不print
    return average
Sum_averages(Random_integer(3))#随机给定数组的长度
```

随机给定的数组: [2, 9, 8]

所有子集的平均的和: 44.333333333333336

Out[8]: 44.333333333333336

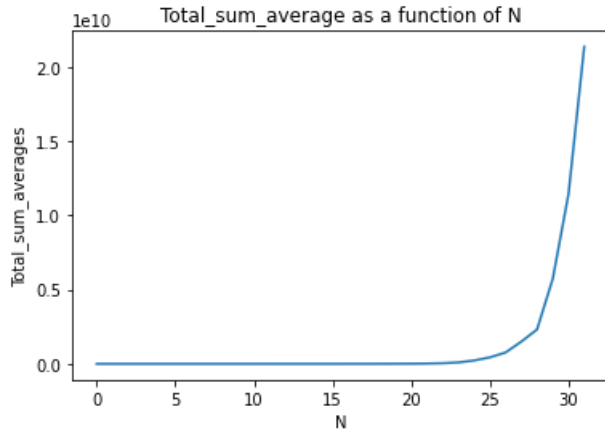
4.3 Total_sum_averages

```
In [9]: Total_sum_averages = [] #用于存储每个数组的平均数
#循环100次计算每次的平均
for i in range(1,101):
    Sum_averages(Random_integer(i))#带入4.2函数
    a = int(Sum_averages(Random_integer(i)))#换成整数用于绘图
    Total_sum_averages.insert(i,a)
    print (Total_sum_averages)#用于看中间过程
print (Total_sum_averages)
```

```
[8]
[8, 16]
[8, 16, 49]
[8, 16, 49, 56]
[8, 16, 49, 56, 217]
[8, 16, 49, 56, 217, 336]
[8, 16, 49, 56, 217, 336, 634]
[8, 16, 49, 56, 217, 336, 634, 1657]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804, 108829]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804, 108829, 194417]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804, 108829, 194417, 299003]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804, 108829, 194417, 299003, 794136]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804, 108829, 194417, 299003, 794136, 1441786]
[8, 16, 49, 56, 217, 336, 634, 1657, 2611, 4194, 11723, 20475, 37804, 108829, 194417, 299003, 794136, 1441786, 2991557]
```

```
In [10]: #因计算量过大，暂时只将输出的数据进行绘制成图
#绘制图
import matplotlib.pyplot as plt
N = range(0, len(Total_sum_averages))
plt.plot(N, Total_sum_averages)
plt.xlabel("N")
plt.ylabel("Total_sum_averages")
plt.title("Total_sum_average as a function of N")
plt.show
```

```
Out[10]: <function matplotlib.pyplot.show(close=None, block=None)>
```



5. Path counting

5.1 Matrix

```
In [11]: import numpy as np
from numpy import random
#定义一个矩阵函数
def arr_Matrix(N,M):
    arr1 = np.random.randint(0,2,size=(N,M)) #在0-1之间随机生成一个N行M列的举证
    arr1[0,0] = 1 #给左上角赋值1
    arr1[N-1,M-1] = 1 #给右下角赋值1
    return arr1
arr_Matrix(5,5)
```

```
Out[11]: array([[1, 1, 0, 1, 0],
                [1, 0, 0, 1, 0],
                [0, 1, 1, 0, 0],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 1]])
```

5.2 Count_path

```
In [12]: #定义一个函数来计算路径，因为有多种路径，因此要将所有路径计算出来，再来求得最大路径
#每当移动时都将为有4种可能分别为：1（左边）1（下边），10，01，00。其中11均可移动最为复杂
def Count_path(x, y, path, N, M):
    k = x + y #从（x，y）处出发往右下角移动，减少循环次数
    for i in range(k, N+M-1):
        #左边和下边均为1的11情况，可以往左边和右边移动：
        if (x != N-1 and y != M-1 and arr[x, y+1] == 1 and arr[x+1, y] == 1): #x代表向下方向，y代表向右方向
            #进行两次循环，将这两次移动计算
            for j in range(0, 2):
                if (j == 0):
                    temp_x = x #首先进行向下方移动的计算，存储原始点地方，不妨碍后续向右方向移动的计算
                    temp_y = y
                    temp_path = path
                    x = x + 1
                    path = path + 1 #向下移动一次并赋值+1
                    Count_path(x, y, path, N, M) #镶嵌函数，进行下一次相同循环（进行下一次移动）
                    x = temp_x #将上述镶嵌循环的x, y, path重新返回，即返回初始点，后续进行向右移动的计算
                    y = temp_y
                    path = temp_path
                if (j == 1): #进行向右移动计算，与上述同理
                    temp_x = x
                    temp_y = y
                    temp_path = path
                    y = y + 1
                    path = path + 1
                    Count_path(x, y, path, N, M)
                    x = temp_x
                    y = temp_y
                    path = temp_path
            #还为到达矩阵边界时，左边为0，下边为1的01情况
            elif (x != N-1 and y != M-1 and arr[x+1, y] == 1 and arr[x, y+1] == 0):
                x = x + 1
                path = path + 1
            #还为到达矩阵边界时，左边为1，下边为0的10情况
            elif (x != N-1 and y != M-1 and arr[x, y+1] == 1 and arr[x+1, y] == 0):
                y = y + 1
                path = path + 1
            #还为到达矩阵边界时，左边为0，下边为0的00情况，堵塞输出
            elif (x != N-1 and y != M-1 and arr[x+1, y] == 0 and arr[x, y+1] == 0):
                break
            #当到达矩阵下方边界时，右边为1，可移动
            elif (x == N-1 and y != M-1 and arr[x, y+1] == 1):
                y = y + 1
                path = path + 1
            #当到达矩阵右方边界时，下方为1，可移动
            elif (y == M-1 and x != N-1 and arr[x+1, y] == 1):
                x = x + 1
                path = path + 1
            #其余情况遇到堵塞输出path
            else:
                break
    my_list.append(path) #存储所有走过的路径值
```

```
In [15]: #验证代码正确，当矩阵中1的含量过多（即11情况很多）会导致计算量过大，计算时间变长
my_list = [] #存储路径
arr = arr_Matrix(10, 8) #第一问的随机生成一个矩阵
print (arr)
Count_path(0, 0, 0, 10, 8)
print(max(my_list)) #最多的路径
```

```
[[1 1 1 1 0 1 1 1]
 [1 0 1 1 0 0 1 0]
 [0 0 1 1 0 1 0 1]
 [1 0 1 1 0 0 1 0]
 [1 1 1 0 0 0 0 0]
 [0 1 1 1 1 1 0 0]
 [0 0 1 0 0 1 0 0]
 [0 0 0 1 1 0 1 1]
 [0 1 1 1 1 1 0 0]
 [0 0 0 1 0 0 0 1]]
```

11

5.3 count paths

```
In [16]: sum = 0 #步数总和
average = 0 #步数平均值
final_list = [] #存储平均值
for i in range(0,1000):
    my_list = []
    arr = arr_Matrix(10,8)
    Count_path(0,0,0,10,8)
    sum = sum + max(my_list)#计算步数总和
    average = sum/(i+1)#计算步数平均值
    final_list.insert(i,average) #存储平均值到final_list
    print(final_list)#查看中间值
```

```
[2.0]
[2.0, 2.5]
[2.0, 2.5, 5.0]
[2.0, 2.5, 5.0, 4.75]
[2.0, 2.5, 5.0, 4.75, 4.6]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5, 3.272727272727273]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5, 3.272727272727273, 4.083333333333333]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5, 3.272727272727273, 4.083333333333333, 3.8461538461538463]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5, 3.272727272727273, 4.083333333333333, 3.8461538461538463, 3.7857142857142856]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5, 3.272727272727273, 4.083333333333333, 3.8461538461538463, 3.7857142857142856, 3.533333333333333]
[2.0, 2.5, 5.0, 4.75, 4.6, 4.5, 4.571428571428571, 4.0, 3.6666666666666665, 3.5, 3.272727272727273, 4.083333333333333, 3.8461538461538463, 3.7857142857142856, 3.533333333333333, 3.9502074688796682]
```

```
In [19]: #循环1000次平均值，在这里因为未循环完，因此输出最后一次循环值
final_list[-1] #循环了241次
```

Out[19]: 3.9502074688796682