**part1：**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 |
| 5 | 1 | 1 | 2 | 2 | 3 | 3+1 | 4+1 | 4+2 | 5+2 | 5+3 | 6+3+1 | 6+4+1 | 7+4+2 | 7+5+2 |
| 10 | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 10+1 | 10+1 | 12+2 | 14+2 |

我发现只考虑0、1、2的情况很容易列出总数，且很难得出结论，而加上5之后就开始不同，因此我以可以使用5作为切入点。

先看amout=0~4元，在不足以使用5的时候与上一行无异，而amount=5时，选择数比上一行多了1，就是直接使用5的情况。amount=6时，还是之比上一行多1，很好理解，就是使用5元+1元的情况。

然而到了7元，时代变了。我第一眼认为也是比上一行多1，但实际上，多了有5元+2元以及5元+1元+1元这两种情况。即：先用7元-5元，剩下的2元不足以使用5元了，要用上一行的graph[2][2]来凑，所以暂时得出公式graph[3][7]=graph[2][7]+graph[2][2]。amount=5~9元时都套用这个式子得到了结果。

然而到了10元，时代又变了。因为除了直接用上述公式，还有一种情况是直接用2张5元钞票，因此要再加1。而11元也是再加1即可。

然而到了12元，时代又双变了。因为用2张5元钞票之后还剩下2元，这有两种组成情况，因此应该是graph[3][12]=graph[2][12]+graph[2][7]+graph[2][2]。

到此，我发现对于某一面额（coins[i]元）这一行，应该先算出要凑的amount中最多包含多少张coins[i]元，即contain\_num=amount/ coins[i]，然后对每个包含不同张coins[i]元情况（包含0、1、2、3张…）都用上一行的数据来凑成。即：

int contain\_num = j / coins[i];//j中包含多少个coins[i]

for (int k=0;k<=contain\_num;k++)

graph[i+1][j] += graph[i][j-k\*coins[i]];

时间复杂度：O(amount\*coins.size())

将空间复杂度优化到M：即数组只用存两行，分别是现在正在算的这一行和这一行的前一行。

**part2：**

（1）

A B C D E F

A: 0 1 0 1 1 1

B: 0 0 0 1 1 0

C: 1 1 0 0 1 0

D: 0 0 1 0 0 0

E: 0 0 0 1 0 1

F: 0 1 1 1 0 0

B必不可能吃鸡。由于B只打得过DE，因此B要吃鸡就整局只能跟DE打。而B开局左边是A右边是C。

先看左边：A只怕C，因此B的左边永远有个A，除非A被C杀了，但那样就只剩B和C俩人，B还是打不过。

再看右边：C一定不能死，如果C没了那么整局没人能限制A，必是A吃鸡。因此B的右边必须保证有个C，但是B又打不过C，C死了也不行不死也不行。

所以B可以直接开始下一局了，芜湖

A吃鸡：C开局被D杀了，然后A乱杀

C吃鸡：打工仔A一路杀掉FED，给C送人头，然后C再干掉B

D吃鸡：A杀了BFE，剩A-C-D-A，然后C杀了A，D杀了C

E吃鸡：A杀了B，C反手杀了A，D反手杀了C，剩D-E-F-D，E屠杀

F吃鸡：C疯狂打工，A杀了B，C杀了A，剩C-D-E-F-C，然后E杀了D，C又杀了E，只剩F和C，F杀了C

(2) 时间复杂度：最坏O(amount3)

空间复杂度：O(amount2)

感觉按我的思路写已经优化不了了，别的思路可能还能优化。

思路：用bool矩阵graph[amount][amount]来记录是否可以主动交手，graph[i][j]=true表示i可以主动和j交手，=false表示i不可以主动和j交手，只能等j来找他，否则双方无法交手。（注意，graph[i][j]和graph[j][i]的值可能不同，这与我们选取的交手方向有关）我选择让每个人都只沿一个方向进行PK，即要么选从小到大的人PK，要么不PK，以此来更新graph矩阵。所以graph的初始化情况：对每个i，除了graph[i][i+1]为true，别的都为false（graph[i][i-1]也为false）。

为了避免对双方能否交手产生漏判或误判（[i][j]和[j][i]中只要一个为true即可交手），我们必须对每个玩家“同步”进行判断，即判断第一位玩家能否主动与自己左边第i人主动交手后，不能立即判断第一位玩家能否与自己左边第i+1人主动交手，而是要继续判断第二位、第三位…第n位玩家能否与自己左边第i人主动交手，然后再判断i+1的情况。因此第一层循环是相隔人数len1(从2到n，n即自己)，第二层循环是遍历每位玩家i（从0到amount-1）。

第三层循环则是根据一位中间玩家（利用距离len2来寻找）来判断i能否主动与j交手。假如在0~len1之间有个玩家k满足graph[i][k]=true且graph[k][j]=true，那么i只差一步就可以主动与j交手了，这一步就是把k干掉，谁干掉k不重要，因此conquer[i][k]=1或conquer[j][k]=1都可。于是这三重循环如下：

for (int len1=2;len1<=amount;len1++)//从各自的下下个人开始

    for (int i=0;i<amount;i++)//从第一个人开始依次判断

        int player=op(i,len1,amount);

        //下面判断i和player是否能交手

        if (graph[i][player]==false)

            for (int len2=1;len2<len1;len2++)

                int temp\_player=op(i,len2,amount);//工具人

                if(graph[i][temp\_player]==true && graph[temp\_player][player]==true&& (conquer[i][temp\_player]==1 || conquer[player][temp\_player]==1) )

                {

                    graph[i][player]=true;

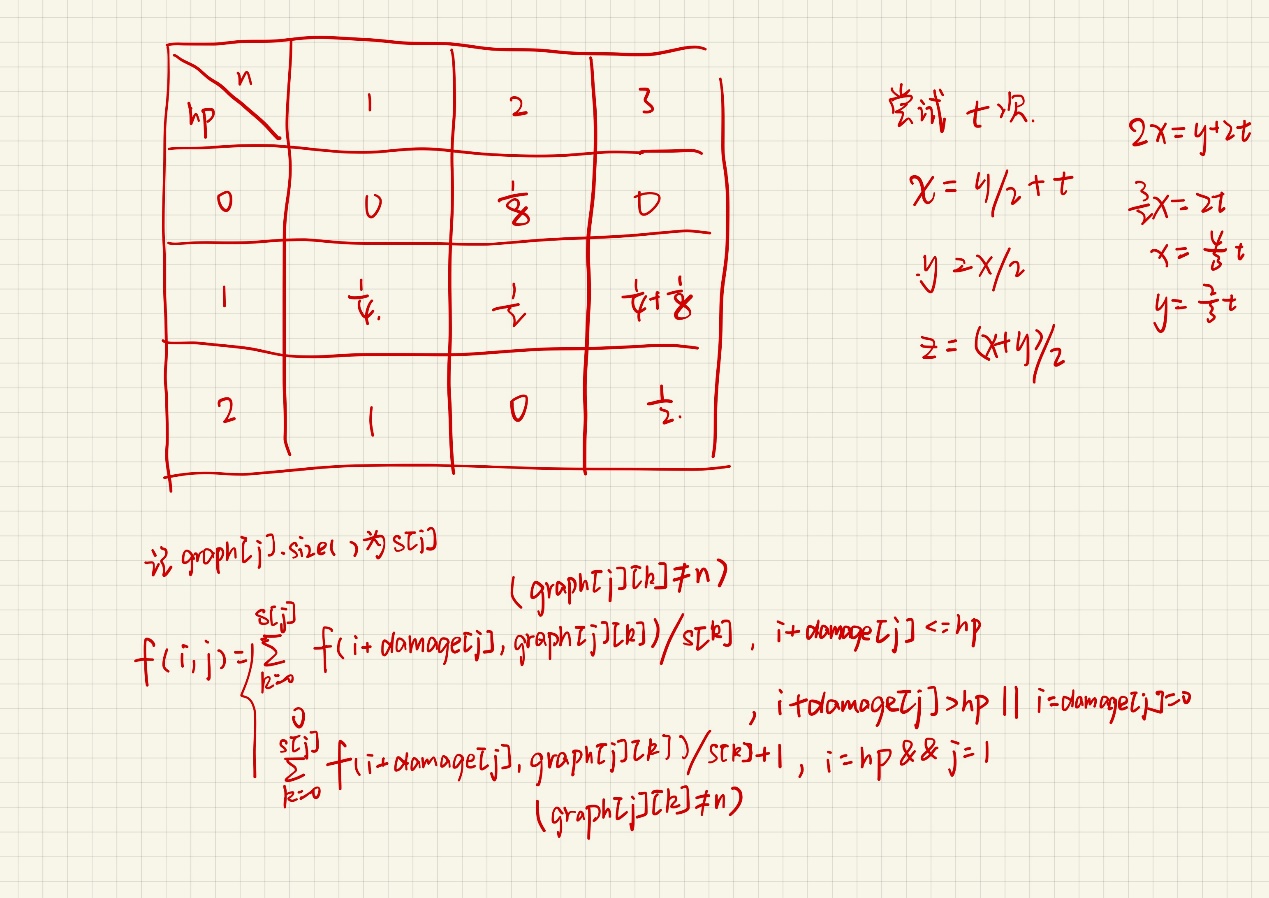
                    break;

                }

那么判断一个人能不能吃鸡，就变成了判断graph[i][i]是否为true。因为他如果能够主动与自己交手，说明他是转了一圈并且干掉了最后一个人才来到自己面前的，所以他有可能吃鸡。

**part3**

1. 用邻接表graph来存每个路口能到达的路口，则状态转移方程：



f为在尝试t次的情况下，在hp=i的情况下到达j路口的次数的期望，于是最后到达n路口的期望是f( ,n)的求和，再除以t。在这里我认为自己能够分清次数和概率，所以我直接设定t=1（也就是说f(hp, 1)一定会>=1）

1. 对于某个特定的hp，所有无陷阱节点构成的方程组，未知数为到达所有无陷阱节点的期望次数（概率）。以part3-case3在hp=2时的方程组为例

方程组：，其中x, y, z分别表示到达1, 2, 5的次数的期望

增广矩阵：

1. 时间复杂度：max{O(hp\*n2), O(hp\*num3)}，其中num为damage为0的路口个数

空间复杂度：O(n2)

上一小问给我的启发是：高斯消元法的系数矩阵a不随hp的变化而变化，只有常数列b是跟hp有关的。而高斯消元法的时间复杂度为O(num3)，其中num为a的行数也即damage为0的路口的个数。而如下图，高斯消元法中对a的处理恰好是三重for循环嵌套，因此我们可以通过提前处理a来使高斯消元法的时间复杂度降为O(num2)，总时间复杂度将降为O(hp\*n2)。

for (k = 0; k < num - 1; k++)

    {

        //求出第K次初等行变换的系数

        for (i = k + 1; i < num; i++)

            c[i] = a[i][k] / a[k][k];

        //第K次的消元计算

        for (i = k + 1; i < num; i++)

        {

            for (j = 0; j < num; j++)

            {

                a[i][j] = a[i][j] - c[i] \* a[k][j];

            }

            b[i] = b[i] - c[i] \* b[k];

        }

    }

不幸的是，我发现对b的处理要跟a和c（工具列）同步进行，因此不能提前处理好a、c然后直接传入，那么怎么办呢…

那就把c做成num\*num的工具矩阵不就好了！所以就只需要b[i]=b[i]-c[k][i]\*b[k]

^ ^