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($amount^3$)

空间复杂度：O($amount^2$)

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<u>第一位</u>玩家能否主动与自己<u>左边第 i 人</u>主动交手后，<u>不能立即判断第一位</u>玩家能否与自己<u>左边第 i+1</u> 人主动交手，而是要<u>继续判断第二位、第三位…第 n 位玩家</u>能否与自己<u>左边第 i 人</u>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][pl
ayer]==true&& (conquer[i][temp_player]==1 || conquer[player][temp_playe
r]==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：

记 graph[j].size( ) 为 s[j]

$$f(i,j)=\begin{cases}\sum_{k=0}^{s[j]} f(i+damage[j], graph[j][k])/s[k] \quad (graph[j][k]\neq n), & i+damage[j]<=hp \\ 0, & i+damage[j]>hp \;||\; i=damage[j]=0 \\ \sum_{k=0}^{s[j]} f(i+damage[j], graph[j][k])/s[k]+1 \quad (graph[j][k]\neq n), & i=hp \;\&\&\; j=1\end{cases}$$

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）

(2) 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：$\begin{cases} x=\frac{y}{3}+\frac{f(2,3)}{4} \\ y=\frac{x}{2}+\frac{f(2,3)}{4}+\frac{f(2,4)}{3} \\ z=\frac{f(2,3)}{4}+\frac{f(2,4)}{3}\end{cases}$，其中 x, y, z 分别表示到达 1, 2, 5 的次数的期望

增广矩阵：$\begin{array}{cccc} 1 & -\frac{1}{3} & 0 & \frac{f(2,3)}{4} \\ -\frac{1}{2} & 1 & 0 & \frac{f(2,3)}{4}+\frac{f(2,4)}{3} \\ 0 & 0 & 1 & \frac{f(2,3)}{4}+\frac{f(2,4)}{3}\end{array}$

(3) 时间复杂度：$\max\{O(hp*n^2), O(hp*num^3)\}$，其中 num 为 damage 为 0 的路口个数

空间复杂度：$O(n^2)$

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 $O(num^3)$，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 $O(num^2)$，总时间复杂度将降为 $O(hp*n^2)$。

```
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]

^ ^